Yang Xiang
Alfredo Cuzzocrea
Michael Hobbs
Wanlei Zhou (Eds.)

LNCS 7016

Algorithms and Architectures for Parallel Processing

11th International Conference, ICA3PP 2011
Melbourne, Australia, October 2011
Proceedings, Part I

Springer

Lecture Notes in Computer Science 7016

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Yang Xiang Alfredo Cuzzocrea
Michael Hobbs Wanlei Zhou (Eds.)

Algorithms and Architectures for Parallel Processing

11th International Conference, ICA3PP 2011
Melbourne, Australia, October 24-26, 2011
Proceedings, Part I

Springer

Volume Editors

Yang Xiang
Wanlei Zhou
Deakin University, School of Information Technology
Melbourne Burwood Campus, 221 Burwood Highway
Burwood, VIC 3125, Australia
E-mail: {yang, wanlei}@deakin.edu.au

Alfredo Cuzzocrea
ICAR-CNR and University of Calabria
Via P. Bucci 41 C, 87036 Rende (CS), Italy
E-mail: cuzzocrea@si.deis.unical.it

Michael Hobbs
Deakin University, School of Information Technology
Geelong Waurn Ponds Campus, Pigdons Road
Geelong, VIC 3217, Australia
E-mail: mick@deakin.edu.au

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24649-4 e-ISBN 978-3-642-24650-0
DOI 10.1007/978-3-642-24650-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011937820

CR Subject Classification (1998): F.2, H.4, D.2, I.2, G.2, H.3

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

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

Printed on acid-free paper

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

Message from the ICA3PP 2011 Program Chairs

A warm welcome to the 11th International Conference on Algorithms and Ar chitectures for Parallel Processing (ICA3PP 2011) and to Melbourne, Australia.

ICA3PP 2011 is the 11th in this series of conferences that started in 1995 and is devoted to algorithms and architectures for parallel processing. ICA3PP is now recognized as the main regular event focusing on the many dimensions of parallel algorithms and architectures, encompassing fundamental theoretical approaches, practical experimental results, and commercial components and systems. As applications of computing systems have permeated every aspects of daily life, the power of computing systems has become increasingly critical. On top of these motivations, ICA3PP 2011 provides a widely-known forum for researchers and practitioners from countries around the world to exchange ideas for improving the computation power of computing systems.

In response to the ICA3PP 2011 call for papers, we received 85 submissions from 33 different countries. These papers were evaluated on the basis of their originality, significance, correctness, relevance, and technical quality. Each paper was reviewed by at least three members of the Program Committee. Based on these evaluations, 24 regular papers and 17 short papers were selected for presentation at the conference, representing an acceptance rate of 28.2% for regular papers and 20% for short papers.

We would like to thank the Program Committee members and additional reviewers from all around the world for their efforts in reviewing the large number of papers. We are grateful to all the associated Conference/Workshop Chairs for their dedication and professionalism. We would like to extend our sincere thanks to the ICA3PP Steering Committee Chairs, Prof. Wanlei Zhou and Prof. Yi Pan, and to the General Chairs, Prof. Andrzej Goscinski and Prof. Peter Brezany. They provided us with invaluable guidance throughout the process of paper selection and program organization. We thank Georgi Cahill, the Conference Secretary, for her professional organization. We also thank Yu Wang and Sheng Wen for their help on completing the final proceedings.

Last but not least, we would also like to take this opportunity to thank all the authors for their submissions to ICA3PP 2011 and the associated symposium/workshops. Many of you have travelled some distance to participate in the conference.

Welcome to Melbourne and enjoy!

October 2011

Yang Xiang
Alfredo Cuzzocrea
Michael Hobbs

Message from the ICA3PP 2011 General Chairs

Welcome to the beautiful and 'World's Most Livable City' – Melbourne. We are privileged and delighted to welcome you to the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2011).

Following the traditions of the previous successful ICA3PP conferences held in Hangzhou, Brisbane, Singapore, Melbourne, Hong Kong, Beijing, Cyprus, Taipei and Busan, this year ICA3PP 2011 is held in Melbourne, Australia. The objective of ICA3PP 2011 is to bring together researchers and practitioners from academia, industry and government to advance the theories and technologies in parallel and distributed computing. ICA3PP 2011 focuses on two broad areas of parallel and distributed computing, i.e., architectures, algorithms and networks, and systems and applications. The conference of ICA3PP 2011 is organized by Deakin University, Australia.

In addition to the ICA3PP 2011 main conference, one symposium and three workshops are being held together with ICA3PP 2011. They are:

1. 2011 International Symposium on Advances of Distributed Computing and Networking (ADCN 2011)
2. The 4th IEEE International Workshop on Internet and Distributed Computing Systems (IDCS 2011)
3. The 1st IEEE International Workshop on Parallel Architectures for Bioinformatics Systems (HardBio 2011)
4. The 3rd International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2011)

We sincerely thank the many people who have helped in organizing ICA3PP 2011 and the associated symposium/workshops. We would like to thank the Program Chairs, Yang Xiang, Alfredo Cuzzocrea and Michael Hobbs, for their leadership in providing the excellent technical program.

We wish you a very enjoyable and rewarding experience at ICA3PP 2011 in Melbourne!

October 2011

Andrzej Goscinski
Peter Brezany

ICA3PP 2011 Committees

General Chairs

Andrzej Goscinski	Deakin University, Australia
Peter Brezany	University of Vienna, Austria

Program Chairs

Yang Xiang	Deakin University, Australia
Alfredo Cuzzocrea	ICAR-CNR and University of Calabria, Italy
Michael Hobbs	Deakin University, Australia

Steering Committee Chairs

Wanlei Zhou	Deakin University, Australia
Yi Pan	Georgia State University, USA

Workshop Chairs

Wen Tao Zhu	Chinese Academy of Sciences, China
Muhammad Khurram Khan	King Saud University, Saudi Arabia

Publicity Chairs

Ali Shahrabi	Glasgow Caledonian University, UK
Haixin Duan	Tsinghua University, China

Publication Chairs

Meikang Qiu	University of Kentucky, USA

Program Committee

Bechini Alessio	University of Pisa, Italy
Giuseppe Amato	ISTI-CNR, Italy
Cosimo Anglano	Università del Piemonte Orientale, Italy
Novella Bartolini	Univ. of Rome La Sapienza, Italy
Ladjel Bellatreche	ENSMA, France
Ateet Bhalla	NRI Institute of Information Science and Technology, India
Angelo Brayner	University of Fortaleza, Brazil
Massimo Cafaro	University of Salento, Italy

Jiannong Cao	Hong Kong Polytechnic University, Hong Kong
Andre Carvalho	Universidade de Sao Paulo, Brazil
Tania Cerquitelli	Politecnico di Torino, Italy
Ruay-Shiung Chang	National Dong Hwa University, Taiwan
Yue-Shan Chang	National Taipei University, Taiwan
Tzung-Shi Chen	National University of Tainan, Taiwan
Zizhong Chen	Colorado School of Mines, USA
Carmela Comito	University of Calabria, Italy
Raphaël Couturier	University of Franche Comté, France
Gennaro Della Vecchia	ICAR-CNR, Italy
Der-Rong Din	National Changhua University of Education, Taiwan
Susan Donohue	The College of New Jersey, USA
Shantanu Dutt	University of Illinois at Chicago, USA
Todd Eavis	Concordia University, Canada
Giuditta Franco	University of Verona, Italy
Karl Fuerlinger	University of California, USA
Jerry Gao	San Jose State University, USA
Jinzhu Gao	University of the Pacific, USA
Jose Daniel Garcia	University Carlos III of Madrid, Spain
Irene Garrigos	University of Alicante, Spain
Alex Gerbessiotis	New Jersey Institute of Technology, USA
Harald Gjermundrod	University of Nicosia, Cyprus
Houcine Hassan	Univ. Politécnica de Valencia, Spain
Pilar Herero	Univ. Politécnica de Madrid, Spain
Ching-Hsien Hsu	Chung Hua University, Taiwan
Tsung-Chuan Huang	National Sun Yat-sen University, Taiwan
Yo-Ping Huang	National Taipei University of Technology, Taiwan
George Karypis	University of Minnesota, USA
Muhammad Khurram Khan	King Saud University, Saudi Arabia
Soo-Kyun Kim	PaiChai University, Korea
Changhoon Lee	Hanshin University, Korea
Deok-Gyu Lee	ETRI, Korea
Laurent Lefevre	INRIA, France
Daniele Lezzi	Barcelona Supercomputing Center, Spain
Keqin Li	State University of New York at New Paltz, USA
Keqin Li	SAP Research, France
Keqiu Li	Dalian University of Technology, China
Kai Lin	Dalian University of Technology, China
Pangfeng Liu	National Taiwan University, Taiwan
Alberto Marchetti-Spaccamela	Sapienza U. of Rome, Italy
Tomas Margalef	Universitat Autonoma de Barcelona, Spain
Amiya Nayak	University of Ottawa, Canada

Leonardo B. Oliveira	Unicamp, Brazil
Marion Oswald	Hungarian Academy of Sciences, Hungary
Deng Pan	Florida International University, USA
Apostolos Papadopoulos	Aristotle Univ. of Thessaloniki, Greece
Dana Petcu	West University of Timisoara, Romania
Rubem Pereira	Liverpool John Moores University, UK
Kleanthis Psarris	The University of Texas at San Antonio, USA
Pedro Pereira Rodrigues	University of Porto, Portugal
Casiano Rodriguez-Leon	Universidad de La Laguna, Spain
Marcel C. Rosu	IBM, USA
Giovanni Maria Sacco	Universitá di Torino, Italy
Erich Schikuta	University of Vienna, Austria
Martin Schulz	Lawrence Livermore National Laboratory, USA
Seetharami Seelam	IBM T.J. Watson Research Center, USA
Edwin Sha	University of Texas at Dallas, USA
Rahul Shah	Louisiana State University, USA
Giandomenico Spezzano	ICAR-CNR, Italy
Peter Strazdins	The Australian National University, Australia
Domenico Talia	Universitá della Calabria, Italy
Uwe Tangen	Ruhr-Universität Bochum, Germany
Jichiang Tsai	National Chung Hsing University, Taiwan
Chen Wang	CSIRO ICT Centre, Australia
Cho-Li Wang	The University of Hong Kong, Hong Kong
Xiaofang Wang	Villanova University, USA
Qishi Wu	University of Memphis, USA
Fatos Xhafa	Polytechnic University of Catalonia, Spain
Zheng Yan	Nokia Research Center, Finland
Chao-Tung Yang	Tunghai University, Taiwan
Zhiwen Yu	Northwestern Polytechnical University, China
Eiko Yoneki	University of Cambridge Computer Laboratory, UK
Sotirios G. Ziavras	NJIT, USA
Roger Zimmermann	National University of Singapore, Singapore

ICA3PP 2011 Additional Reviewers

Atif, Muhammad	Gouvea, Conrado P.L.
Cai, Jie	Guazzone, Marco
Canonico, Massimo	Jin, Chao
Chan, Philip	Khan, Bilal
Ding, Chong	Macias, Mario
Dionysiou, Ioanna	Miranda-Valladares, Gara
Eldefrawy, Mohamed	Mochetti, Karina
Estévez, José Ignacio	Mou, Duxing
Figueiredo, Thomaz	Printista, Marcela

Rodríguez Martínez, Diego
Ruj, Sushmita
Segredo Gonzalez, Eduardo Manuel
Segura, Carlos
Song, Huaguang
Tiskin, Alexander
Tsai, Pei-Wei
Vlad, Ioan
Zhu, Kai
Zola, Matteo

Table of Contents – Part I

ICA3PP 2011 Keynote

ICA3PP 2011 Regular Papers

ICA3PP 2011 Short Papers

Table of Contents – Part II

ADCN 2011 Papers

IDCS 2011 Papers

M2A2 2011 Papers

HardBio 2011 Papers

Keynote: Assertion Based Parallel Debugging

David Abramson

Monash eScience and Grid Engineering Lab, Faculty of Information Technology,
Monash University,
Australia
davida@csse.monash.edu.au

Abstract. Programming languages have advanced tremendously over the years, but program debuggers have hardly changed. Sequential debuggers do little more than allow a user to control the flow of a program and examine its state. Parallel ones support the same operations on multiple processes, which are adequate with a small number of processors, but become unwieldy and ineffective on very large machines. Typical scientific codes have enormous multi-dimensional data structures and it is impractical to expect a user to view the data using traditional display techniques. In this seminar I will discuss the use of debug-time assertions, and show that these can be used to debug parallel programs. The techniques reduce the debugging complexity because they reason about the state of large arrays without requiring the user to know the expected value of every element. Assertions can be expensive to evaluate, but their performance can be improved by running them in parallel. I demonstrate the system with a case study finding errors in a parallel version of the Shallow Water Equations, and evaluate the performance of the tool on a 4,096 cores Cray XE6.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, p. 1, 2011.

Secure and Energy-Efficient Data Aggregation with Malicious Aggregator Identification in Wireless Sensor Networks*

Hongjuan Li[1], Keqiu Li[1,**], Wenyu Qu[2], and Ivan Stojmenovic[3]

[1] School of Computer Science and Technology,
Dalian University of Technology, Dalian, 116024, China
keqiu@dlut.edu.cn
[2] School of Information Science and Technology,
Dalian Maritime University, Dalian, 116026, China
[3] SITE, University of Ottawa, Ontario K1N 6N5, Canada

Abstract. Data aggregation in wireless sensor networks is employed to reduce the communication overhead and prolong the network lifetime. However, an adversary may compromise some sensor nodes, and use them to forge false values as the aggregation result. Previous secure data aggregation schemes have tackled this problem from different angles. The goal of those algorithms is to ensure that the Base Station (BS) does not accept any forged aggregation results. But none of them have tried to detect the nodes that inject into the network bogus aggregation results. Moreover, most of them usually have a communication overhead that is (at best) logarithmic per node. In this paper, we propose a secure and energy-efficient data aggregation scheme that can detect the malicious nodes with a constant per node communication overhead. In our solution, all aggregation results are signed with the private keys of the aggregators so that they cannot be altered by others. Nodes on each link additionally use their pairwise shared key for secure communications. Each node receives the aggregation results from its parent (sent by the parent of its parent) and its siblings (via its parent node), and verifies the aggregation result of the parent node. Theoretical analysis on energy consumption and communication overhead accords with our comparison based simulation study over random data aggregation trees.

1 Introduction

Wireless sensor networks (WSNs) are becoming increasingly popular to provide solutions to many security-critical applications such as wildfire tracking, military surveillance, and homeland security [1]. As thousands of sensor nodes collectively monitor an area, there is high redundancy in the raw data. Data aggregation [2,3,4,5,6] is an essential paradigm to eliminate data redundancy and reduce energy consumption. During

* This work is supported by National Natural Science Foundation of China under Grant nos. 90718030, 90818002, 60903154, and 60973117, and New Century Excellent Talents in University (NCET) of Ministry of Education of China.

** Corresponding author.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 2–13, 2011.

a typical data aggregation process, sensor nodes are organized into a hierarchical tree rooted at the base station. However, data aggregation is challengeable in some applications due to the fact that the sensor nodes are vulnerable to physical tampering, which may lead to the failure of data aggregation. The sensor nodes are often deployed in hostile and unattended environments, and are not made tamper-proof due to cost considerations. So they might be captured by an adversary, which may arbitrarily tamper with the data to achieve its own purpose.

To meet this challenge, some work has been done [9,10,11,12,13,14] in the area of secure data aggregation. For example, Chan et al. [9] put forward a secure hierarchical in-network aggregation scheme that provides favorable and impressive security properties. This scheme can verify whether or not tampering has occurred on the path between a leaf and the root [9]. Nevertheless, it cannot pinpoint the exact node where the tampering has happened in case of tampering. To the best of our knowledge, none of the existing work is able to identify the nodes that tamper the intermediate aggregation results.

To overcome this deficiency, we present a secure and energy-efficient data aggregation scheme termed MAI to effectively locate the malicious aggregators. Each node verifies its parent's aggregation by recalculating the aggregation result according to the results obtained from its siblings. If an inconsistency occurs, the parent node is flagged as a malicious node; otherwise, it is a normal one. Another characteristic of the scheme is that the aggregation and verification can be executed interactively. A node's result can be further aggregated only after it passes the verification. This can avoid unnecessary wrong data transmissions and further reduce the energy consumption. Moreover, the verification procedure is a localized one, which results in a low communication overhead.

The rest of the paper is organized as follows: In Section 2, we overview some related work on secure data aggregation. Section 3 describes our system model. In Section 4, we give a detailed description on the proposed MAI. Theoretical analysis and discussion are also presented in this section to further explain our scheme. Section 5 reports the simulation results. Finally, we summarize our work and conclude the paper in Section 6.

2 Related Work

Data aggregation has the benefit to achieve bandwidth and energy efficiency. There has been extensive research [17,18,19] on data aggregation in various application scenarios. However, these aggregation schemes have been designed without security in mind. Recently, secure data aggregation is a hot research problem in some applications. Basically, there are two types of aggregation models, i.e., the single-aggregator model and the multiple-aggregator model.

The authors in [10,11] investigated secure data aggregation for the single-aggregator model. The secure information aggregation (SIA) protocol presented by Przydatek et al. [10] was the first one to propose the aggregate-commit-prove framework. Du et al. [11] proposed a scheme using multiple witness nodes as additional aggregators to verify the integrity of the aggregated result. As for the single-aggregator model, the corresponding schemes do not provide per-hop aggregation.

The multiple-aggregator model employs more than one aggregator. Hu et al. [14] presented a secure aggregation protocol that is resilient to single aggregator compromising. However, this protocol cannot deal with the situation where there exist two consecutive colluding compromised aggregator nodes. Yang et al. [12] proposed SDAP, which utilizes a novel probabilistic grouping technique to probe the suspicious groups. Due to the statistical nature, SDAP may not be able to detect the attacks that slightly change the intermediate aggregation results.

In the privacy-preservation domain, Castelluccia et al. [15] proposed a new homomorphic encryption scheme in which the aggregation is carried out by aggregating the encrypted data without decrypting them, resulting in a higher level privacy. He et at. [16] proposed two privacy-preserving data aggregation schemes CPDA and SMART for additive aggregation functions.

3 System Model

We model a wireless sensor network as a graph consisting of a set of n resource-limited sensor nodes $U = \{u_1, u_2, \ldots, u_n\}$, each of which has an unique identifier ID_{u_i}. In addition, a resource-enhanced BS R is deployed to connect the sensor network to the outside infrastructure, e.g. the Internet. We assume that a topological tree rooted at R is constructed to perform the data aggregation. There are three types of nodes in the sensor network: leaf nodes, intermediate nodes, and the base station. The leaf nodes are collecting sensor readings. An intermediate node acts as an aggregator, aggregating the data transmitted from its child nodes and forwarding the aggregation result to its patent node. The base station is the node where the final result is aggregated. An example of such an aggregation tree is shown in Figure 1. One method for constructing such an aggregation tree can be found in TAG [6].

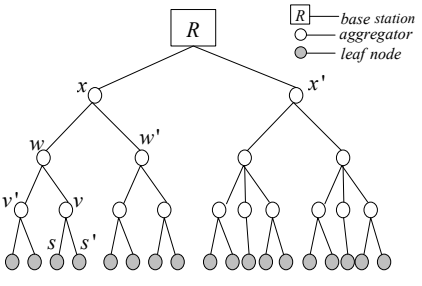


Fig. 1. An example aggregation tree

Our scheme assumes that the network utilizes an identity-based public key cryptosystem, which is also used in [8]. Each sensor node $u \in U$ is deployed with a private key, K_u^{-1}, and other nodes can calculate u's public key based on its ID, i.e. , $K_u = f(ID_u)$. Traditionally, it is assumed that public key systems exceed the memory and computational capacity of the sensor nodes. However, public key cryptography

on new sensor hardware may not be as prohibitive as that is traditionally assumed [8]. We further assume that the sensor nodes have the ability to perform symmetric-key encryption and decryption as well as to compute a collision-resistant cryptographic hash function.

In this paper, we focus on defending against the attacks tampering with the intermediate aggregation results to make the BS accept a false value. The goal of our design is to localize the exact aggregator(s) that performs the malicious tampering. In this paper, we do not consider the value changing attack where a compromised node forges a false reading on its own behalf. As indicated in [12,14], the impact of such an attack is usually limited. Besides, such a compromised node is more likely a faulty sensor node. Some other studies have targeted the identification of faulty sensors [7,20,21].

4 Secure and Energy-Efficient Data Aggregation with Malicious Aggregator Identification

In this section, we present a secure and energy-efficient data aggregation with malicious aggregator identification (MAI). For simplicity, we describe our scheme for the SUM aggregation function. However, our design supports various other aggregation functions such as MAX/MIN, MEAN, COUNT, and so on. We apply our scheme on the aggregation tree shown in Figure 1.

4.1 Aggregation Commitment

Before describing the details of the proposed scheme, we first introduce the format of the packets transmitted during the aggregation. The packet has the following format:

$$< id, count, value, signature >$$

where id is the node's ID, $count$ is the number of leaves in the subtree rooted at this node, $value$ is the aggregation result computed over all the leaves in the subtree, and $signature$ is a commitment computed by the node using its private key. We call the $signature$ a proof. If an adversary compromises an aggregator and wants to send an invalid aggregation result, it has to forge the proof on the invalid result.

The packet for node u_i can be inductively expressed as:

$$< u_i, C_i, V_i, S_i >$$

where $S_i = \{H(u_i \| C_i \| V_i)\} K_{u_i}^{-1}$ and $H(u_i \| C_i \| V_i)$ is a cryptographic hash function over the packet value.

If u_i is a leaf node, then $C_i = 1$ and $V_i = r_{u_i}$, where r_{u_i} is the data collected by node u_i. If u_i is an intermediate node having child nodes v_j $(j = 1, 2, ...k)$ with packets $< v_j, C_j, V_j, S_j >$, then $C_i = \sum_{j=1}^{k} C_j$, $V_i = \sum_{j=1}^{k} V_j$.

The pairwise key shared between u_i and its parent node is used to encrypt the packet. This encryption in practice provides not only confidentiality but also authentication. Using encryption saves the bandwidth that will otherwise be used for an additional message authentication code (MAC) [12].

Since there exits three types of nodes in the sensor network, we will respectively introduce the aggregation process executed on each type of the nodes.

1) Leaf node aggregation: Data aggregation starts from the leaf nodes toward the BS. Since a leaf node does not need to do aggregation, the value in its packet is just the sensor reading. For example, in the case of Figure 1, the leaf node s sends to its parent v the following packet:

$$s \rightarrow v :< s, 1, r_s, \{H(s\|1\|r_s)\}K_s^{-1} >$$

where $\|$ denotes the stream concatenation and r_s is the sensor reading by node s. This packet is encrypted using the pairwise key shared between s and v.

2) Intermediate node aggregation: When an intermediate node receives an aggregated report from one of its children, it verifies the signature of the report and keeps a copy locally (used by the aggregation verification phase) before further aggregation is performed. More specifically, an intermediate node first decrypts the report using its pairwise key shared with its child node. It then performs some simple checking on the validity of the report. The value of each item should fall in a certain range, and the verification signature should be matched with that of the report. The signature of the report is verifiable because the intermediate node can calculate the public keys of its child nodes using their IDs. If the report does not pass this checking, the packet will be discarded; otherwise, the readings of all the reports received from its children will be aggregated. A new count is calculated as the sum of the count values in all the received reports. Furthermore, a new signature is calculated and attached to the end. For the example shown in Figure 1, node w is the parent of node v. The packet that v sends to w is shown as follows:

$$v \rightarrow w < v, 2, Agg_v, \{H(v\|2\|Agg_v)\}K_v^{-1} >$$

where "2" is the count value summed over the count values of s and s', and Agg_v is the aggregation value over r_s and $r_{s'}$. Similarly, node w sends a packet to its parent x in such a format:

$$w \rightarrow x :< w, 4, Agg_w, \{H(w\|4\|Agg_w)\}K_w^{-1} >$$

Each of these packets is encrypted with the pairwise key shared between the corresponding sender and its parent.

3) BS aggregation: After the BS receives the aggregated data from all its children, it decrypts them, verifies their signatures and stores them locally. Then it computes the

final aggregation result just like a regular intermediate node does. As such, the final aggregation result in the BS for the example shown in Figure 1 is as follows:

$$BS :< R, 18, Agg_R, \{H(R\|18\|Agg_R)\}K_R^{-1} >$$

4.2 Aggregation Verification

Before we present the details of our verification procedure, a high level overview of the process is introduced as follows. First, each sensor node gets the values of all its siblings (called sibling values) and the aggregation result of its parent node. Then it independently verifies whether or not its parent's aggregation result equals the recalculated one based on its own value and the received sibling values. If not, an alarm is raised (for example, using broadcast) to warn the entire network that the parent node is malicious, and the malicious node can be evicted from the network through a certain method. If no alarm is raised, all the aggregation operations are correct, and the final aggregation result can be accepted by the BS.

In what follows, we will present the detailed design of the proposed scheme.

1) Dissemination of the sibling packets: To enable verification, each sensor node must get the values of its siblings in order to recalculate the aggregated value of its parent. Thus, each parent node is required to distribute the copies of the sibling packets to all child nodes. Upon receiving the sibling packets, each node verifies their signatures, which are employed to ensure that the parent node cannot tamper with the packets of its child nodes because it does not know the private keys of its children.

2) Dissemination of parent packets: To determine whether the aggregation operation is correct or not, the child nodes need to know the original aggregation result obtained by its parent node. However, a malicious parent node may tamper with the aggregation result in the aggregation phase, but send a correct result to its child nodes in the verification phase so that it can avoid being detected. In our scheme, the grandparent nodes are involved, which prevent the parent nodes from transmitting different values. Actually, it is the grandparent nodes that send the parent nodes' aggregated values to the child nodes.

As shown in the example (Figure 1), w is the grandparent node, v is the parent node, and s is the child node. The packet w receives from v is shown as follows:

$$v \rightarrow w < v, 2, Agg_v, \{H(v\|2\|Agg_v)\}K_v^{-1} >$$

This packet should be sent to the child node s in the verification phase. First, w encrypts the signature of v using its own private key. In other words, the signature of w in this packet is calculated over $v's$ signature.

$$w \rightarrow v < v, 2, Agg_v, \{\{H(v\|2\|Agg_v)\}K_v^{-1}\}K_w^{-1} >$$

v verifies the signature and then sends the packet to s and s'.

The reason for the second signature involving two private keys is to make sure that neither the grandparent node nor the parent node can tamper with the packet, so that the packet must be the original one obtained in the aggregation phase.

3) Verification of the parent's aggregation: After each sensor node gets its sibling values and its parent value, it can verify the parent's aggregation if all the packets pass the verifications on their signatures.

As the sibling values provide all the necessary data to perform the aggregation, each sensor node runs the same aggregation process as its parent to derive the aggregation result, and compares it against the previously received one from its grandparent. Only when all the verification succeeds, the BS accepts the aggregation result.

4.3 Theoretical Analysis on Communication Overhead

In this section, we analyze the communication overhead of our scheme and compare it with the Secure Hierarchical In-network Aggregation scheme (SHIA for short) proposed by Chan et al. [9]. SHIA is selected for comparison because it is the most related and it represents the state-of-the-art.

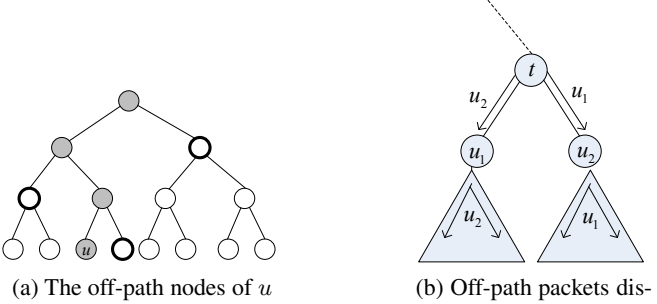
Since both schemes perform similar aggregation operations, we only compare the communication overhead in the verification phase. To accurately measure the overhead, we use the metric $packet * hop$, because the communication overhead is proportional to the transmission distance as each packet needs to travel several hops to arrive at the destination node. Therefore we sum up all the traveled hops for each packet as the communication overhead in the whole network.

Before we present our analytical results, we give two definitions defining the communication overhead and the off-path nodes:

Definition 1. *Suppose there exist a set of packets* $\{p_j | j = 1, 2 \dots z\}$ *used for verification purpose. If the packet* p_j *needs to travel* h_j *hops, then the communication overhead is calculated by* $\sum_{j=1}^{z} h_j$.

Definition 2. *[9] The set of off-path nodes for a node* u *in a tree is the set of all the siblings of each of the nodes on the path from* u *to the root of the tree (the path is inclusive of* u*).*

Figure 2a shows an example of the off-path nodes for node u. The off-path nodes for node u are highlighted in bold. The path from u to the root is shaded as grey.



(a) The off-path nodes of u

(b) Off-path packets dissemination

Fig. 2. Off-path nodes and off-path packets dissemination

We assume that the aggregation tree is a complete tree with a height of h and a degree of d; hence, we have $n = \sum_{i=1}^{h} d^i$. Note that our aggregation tree is rooted at the BS, and we assume that the height of the BS is 0.

In SHIA, the communication overhead consists of two parts: the dissemination of the root value, and the dissemination of the off-path values. The root value will be sent to the entire sensor network using authenticated broadcasts, which incurs a communication overhead of n as there are n sensor nodes in the network. Hence, the communication overhead in this phase can be computed as $\sum_{i=1}^{h} d^i$.

With the knowledge of the root value, each leaf node must receive all its off-path values to enable the verification. As described in [9], the process of dissemination of the off-path values is as follows: Assume that an intermediate node t in the aggregation tree has two children u_1 and u_2. To disseminate the off-path values, t sends the packets aggregated at u_1 to u_2, and vice-versa. Node t also sends any packet received from its parent to both children. See Figure 2b for an illustration of the process. Once a node has received all the packets of its off-path nodes, it can proceed to the verification step.

In SHIA, the packets of every senor node will be sent to its sibling nodes and forwarded along the trees rooted at the sibling nodes until they reach the leaf nodes. Therefore the communication overhead can be calculated by $\sum_{i=1}^{h}(d^i \cdot \sum_{j=0}^{h-i} d^j)$.

Thus the total communication overhead needed in the verification phase of SHIA is:

$$\begin{aligned} & \sum_{i=1}^{h} d^i + \sum_{i=1}^{h}(d^i \cdot \sum_{j=0}^{h-i} d^j) \qquad (1) \\ = & \frac{(h+1)d^{h+2} - (h+2)d^{h+1} - d^2 + 2d}{(1-d)^2} \\ = & \Theta(n \log n)^1 \end{aligned}$$

In our scheme MAI, the communication overhead for the verification process also consists of two parts: the dissemination of the sibling values and the dissemination of the parent value.

To derive the parent aggregation result, each child node needs to get its sibling values, which indicates that each node needs to receive $(d-1)$ packets in the phase of disseminating sibling values. Since there are n nodes in the tree, the communication overhead for this step can be calculated as $n(d-1)$.

To compare the derived aggregation result in an intermediate node with the one computed at the aggregation phase, the parent value should be disseminated to its child nodes. As every intermediate node has d children, and the parent value is sent from its grandparent, the dissemination of each parent value involves $(d+1)$ communication overhead. Since there are $(n - d^h)$ parent nodes, the communication overhead of disseminating the parent values can be computed by $(d+1)(n-d^h)$.

[1] h can be approximated by $\log n$.

Therefore, the total communication overhead for the verification process in our MAI is calculated as follows:

$$\begin{aligned} & n(d-1)+(d+1)(n-d^h) \\ &= 2nd-d^{h+1}-d^h \\ &= \frac{d^{h+2}+d^h-2d^2}{d-1} \\ &= \Theta(n) \end{aligned} \tag{2}$$

From equations (2) and (1), we can easily see that the overhead of MAI is less than that of SHIA. This is because the verification procedure in our scheme is a localized one, while SHIA involves the whole network for verification. Moreover, the advantage of our scheme will be more obvious with the increase of the tree height.

4.4 Discussion

The verification in our scheme is a localized procedure. We can accurately identify malicious nodes by limiting the commit-and-verify scope to every parent-children connection. Once there is malicious tampering at any intermediate node, we can immediately find the inconsistency between the committed aggregate and the reconstructed aggregate.

Our scheme also ensures that all the involved data are the original data. This is because every report is sent only once from the original source and a signature is attached to each report. The signature is computed using the private key that is only known to the source, such that the report cannot be forged when it is kept at other nodes.

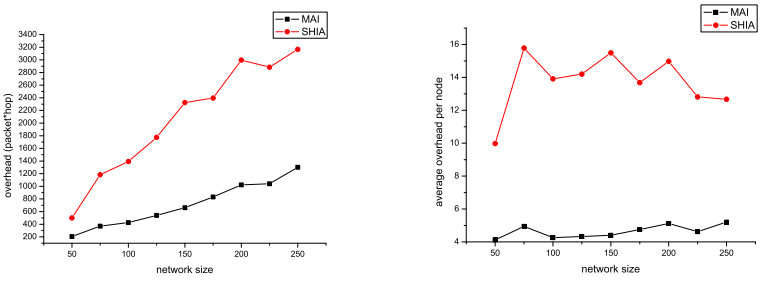
MAI consists of two phases: the aggregation commitment phase and the aggregation verification phase. Actually, the verification phase does not need to wait for the completion of the aggregation phase. These two phases can be executed interactively. After each grandparent node receives a packet from its child node, it may not execute the aggregation immediately. Instead, it asks its grand child nodes to do the verification on the received packet first. Only if the verification succeeds, the grand parent node will accept the packet and do further aggregation; otherwise, the aggregation will stop. If the verification fails, it is an indication that the received packet is forged and the sending node is malicious. Such a false report, if undetected, would be forwarded to the higher level, which can cause not only the deviation in the final aggregation result but also the wastage of energy consumption. In our scheme, we detect such a false report immediately after it is sent out. In this way, we can decrease the damage of the malicious nodes and save energy.

MAI assumes that only the leaf nodes collect sensor readings. Extending our scheme to support the data collection at intermediate nodes results in another problem. The aggregation result at each intermediate node will be based on the data of its child nodes and its own data. We need to get the sensor reading collected by an intermediate node to recalculate the aggregation result in the verification phase. However, the intermediate node may forge a false reading of its own. Such a node is more likely a faulty sensor, which can be detected via various existing techniques [7, 20, 21]. We can employ an existing scheme to verify whether or not a node forges a false data as its own reading.

If not, the data aggregation and verification proceed; otherwise, the node is signaled as malicious.

5 Simulation Evaluation

The previous analytical results are applicable to a balanced tree. To evaluate the performance for more general cases, we conduct simulation study using the NS-2 simulator to compare MAI with SHIA. In our experiments, the nodes are randomly distributed over an area. The network size n varies from 50 nodes to 250 nodes. For each simulated topology, we adjust the communication range so that all the sensor nodes are included in the aggregation tree. In our study, we consider an energy model that sets 0.2818W for sending or receiving a data packet per unit of time, and 100J of total available battery power per node. The data rate is 1 $Mbps$. We compare the communication overhead and the energy consumption of MAI with those of SHIA and the results are reported in the following subsection.



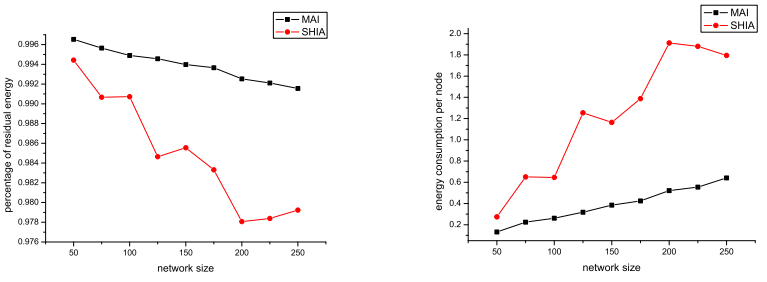
(a) Network communication overhead (b) Average communication overhead

Fig. 3. Communication overhead

Figure 3a shows the communication overhead of MAI and that of SHIA under different network scales. We use $packet * hop$ as the metric. As it can be seen from Figure 3a, the overhead of MAI is much lower than that of SHIA. To further explore the independence of the performance on the size of the aggregation tree, we report the average communication overhead per node in Figure 3b. As shown in this figure, MAI outperforms SHIA in terms of the average amount of communications. And MAI exhibits a little variance when n ranges from 50 to 250. The communication overhead is closely related to the network topology. In the simulations, the nodes are randomly distributed in the area. That's why the overhead increases with the increase of the network size, but still fluctuates at some points.

Figures 4a and 4b illustrate the energy consumption under different network scales. The percentage of the residual energy in the network with respect to the network size is shown in Figure 4a, from which we can conclude that the SHIA scheme consumes energy at a much faster pace. Figure 4b reports the average energy consumption per node.

The results indicate that our scheme is more energy-efficient. This is because data transmissions contribute the major portion of the power consumption for sensor nodes, and the communication overhead of SHIA is higher than that of MAI as discussed before.



(a) Network residual energy (b) Average energy consumption

Fig. 4. Energy consumption

In summary, the theoretical and simulation results both indicate that our proposed MAI is more efficient and effective than SHIA, as it can identify the malicious aggregators with a much less communication overhead.

6 Conclusions

In this paper, we propose a secure and energy-efficient data aggregation scheme with malicious aggregator identification in wireless sensor networks. The goal of our proposed scheme is to make sure that not only the BS does not accept forged aggregation results, but also the malicious aggregators tampering with the intermediate results can be identified. The adversarial aggregators, after detected, can be evicted from the network, hence reducing the damage of malicious aggregators. Theoretical analysis and extensive simulations have been conducted to evaluate our scheme. The results indicate that our proposed scheme is more secure and energy-efficient than SHIA, a state-of-the-art secure hierarchical in-network aggregation scheme proposed in [9].

References

1. Culler, D., Estrin, D., Srivastava, M.: Overview of Sensor Networks. IEEE Computer 37(8), 41–49 (2004)
2. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next century challenges: scalable coordination in sensor networks. In: Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom), pp. 263–270 (1999)
3. Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., Ganesan, D.: Building efficient wireless sensor networks with low-level naming. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), pp. 146–159 (2001)
4. Krishnamachari, B., Estrin, D., Wicker, S.: The impact of data aggregation in wireless sensor networks. In: Proceedings of the International Conference on Distributed Computing Systems (ICDCS) Workshops, pp. 575–578 (2002)

5. Yu, Y., Krishnamachari, B., Prasanna, V.K.: Energy-latency tradeoffs for data gathering in wireless sensor networks. In: Proceedings of the IEEE Computer and Communications Societies, INFOCOM (2004)
6. Madden, S., Franklin, M.J., Hellerstein, J.M.: TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In: Proceedings of the Symposium on Operating Systems Design and Implementation, OSDI (2002)
7. Ding, M., Chen, D., Xing, K., Cheng, X.: Localized Faulty-Tolerant Event Boundary Detection in Sensor Networks. In: Proceedings of the IEEE Computer and Communications Societies, INFOCOM (2005)
8. Parno, B., Perrig, A., Gligor, V.: Distributed Detection of Node Replication Attacks in Sensor Networks. In: Proceedings of the IEEE Symposim on Security and Privacy (SP), pp. 49–63 (2005)
9. Chan, H., Perrig, A., Song, D.: Secure hierarchical in-network aggregation in sensor networks. In: Proceedings of the ACM Conference on Computer and Communication Security, CCS (2006)
10. Przydatek, B., Song, D., Perrig, A.: SIA: Secure information aggregation in sensor network. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, Sensys (2003)
11. Du, W., Deng, J., Han, Y., Varshney, P.K.: A witness-based approach for data fusion assurance in wireless sensor networks. In: Proceedings of the IEEE Global Telecommunications Conference, GLOBECOM (2003)
12. Yang, Y., Wang, X., Zhu, S., Cao, G.: SDAP: A Secure Hop-by-Hop Data Aggregation Protocol for Sensor Networks. In: Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc (2006)
13. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
14. Hu, L., Evans, D.: Secure Aggregation for Wireless Networks. In: Proceedings of the 2003 Symposium on Applications and the Internet Workshops, SAINTW (2003)
15. Castelluccia, C., Mykletun, E., Tsudik, G.: Efficient Aggregation of Encrypted Data in Wireless Sensor Networks. In: Proceedings of the International Conference on Mobile and Ubiquitous Systems, Mobiquitous (2005)
16. He, W., Liu, X., Nguyen, H., Nahrstedt, K., Abdelzaher, T.: PDA: Privacy-preserving Data Aggregation in Wireless Sensor Networks. In: Proceedings of the IEEE Computer and Communications Societies, INFOCOM (2007)
17. Itanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: a scalable and robust communication paradigm for sensor networks. In: Proceedings of the ACM International Conference on Mobile Computing and Networking, MobiCom (2000)
18. Intanagonwiwat, C., Estrin, D., Govindan, R., Heidemann, J.: Impact of Network Density on Data Aggregation in Wireless Sensor Networks. In: Proceedings of the International Conference on Distributed Computing Systems, ICDCS (2002)
19. Tang, X., Xu, J.: Extending network lifetime for precision constrained data aggregation in wireless sensor networks. In: Proceedings of the IEEE Computer and Communications Societies, INFOCOM (2006)
20. Ding, M., Chen, D., Xing, K., Cheng, X.: Localized Fault-Tolerant Event Boundary Detection in Sensor Networks. In: Proceedings of the IEEE Computer and Communications Societies (INFOCOM), March 13-17, pp. 902–913 (2005)
21. Liu, F., Cheng, X., Chen, D.: Insider Attacker Detection in Wireless Sensor Networks. In: Proceedings of the IEEE Computer and Communications Societies (INFOCOM), May 6-12, pp. 1937–1945 (2007)

Dynamic Data Race Detection for Correlated Variables

Ali Jannesari, Markus Westphal-Furuya, and Walter F. Tichy

Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
{jannesari,westphal,tichy}@kit.edu
http://www.kit.edu

Abstract. In parallel programs concurrency bugs are often caused by unsynchronized accesses to shared memory locations, which are called *data races*. In order to support programmers in writing correct parallel programs, it is therefore highly desired to have tools on hand that automatically detect such data races. Today, most of these tools only consider unsynchronized read and write operations on a single memory location. Concurrency bugs that involve multiple accesses on a set of correlated variables may be completely missed. Tools may overwhelm programmers with data races on various memory locations, without noticing that the locations are correlated. In this paper, we propose a novel approach to data race detection that automatically infers sets of correlated variables and logical operations by analyzing data and control dependencies. For data race detection itself, we combine a modified version of the lockset algorithm with happens-before analysis providing the first *hybrid, dynamic race detector for correlated variables*. We implemented our approach on top of the Valgrind, a framework for dynamic binary instrumentation. Our evaluation confirmed that we can catch data races missed by existing detectors and provide additional information for correct bug fixing.

Keywords: data race detection, parallel programs, dynamic analysis, correlated variables.

1 Introduction

As multi-core processors have become more and more ubiquitous in recent years, programmers are faced with the challenge of writing parallel programs to leverage this computing power. Yet, writing parallel programs is inherently harder than sequential ones: Among other difficulties, concurrency related bugs, such as deadlocks, atomicity and order violations [1], tend to appear randomly and are troublesome to reproduce and fix – especially if several variables are involved. How can we support programmers in this tedious work and improve existing tools?

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 14–26, 2011.

1.1 Problem Description

The 'traditional' definition of data races does not cover concurrency bugs that involve more than a single memory location or multiple read/write-operations: Consider the function *scaleVector* shown in Figure 1, where every access to the shared tuple (x, y) is protected by lock m. Although *scaleVector* is clearly intended as an atomic operation on (x, y), another thread could change x or y during the computation of max. If the other thread also protects x and y with Lock m, no data race is detected, yet *scaleVector* obviously suffers from an atomicity violation.

An extensive study of concurrency bugs in [1] has revealed that a significant number (34%) of the examined non-deadlock bugs fall into this category and are therefore not adequately addressed by existing tools. In this paper, we present a new approach to data race detection that will help close this gap. Roughly speaking, we adapt the lockset algorithm and the happens-before analysis to build a *dynamic hybrid data race detector for correlated variables and logical operations*. First and foremost, we must extend the definition of data races to capture scenarios like the one we just described. Therefore, two aspects of data races need reconsideration:

Data: shared vector (x, y); local variables a, b, max; lock m

Function *scaleVector*
lock(m) ;
$(a, b) \leftarrow (x, y)$;
unlock(m) ;
$max \leftarrow a$ **if** $a > b$ **else** b
lock(m) ;
$(x, y) \leftarrow (\frac{x}{max}, \frac{y}{max})$;
unlock(m) ;

Fig. 1. Function with concurrency bug

Spatial Aspect: Instead of single memory locations, we must monitor sets of correlated variables that share a semantic *consistency property*. We call such sets *correlated sets*. In the example above $s = \{x, y\}$ is one such correlated set.
Temporal Aspect: A logical operation on a correlated set that preserves its consistency property may consist of several elementary reads or writes. We call such operations *computational units*. In our example, the computational unit $u = scaleVector$ operates on s. Using these terms, we come up with the following new definition of extended data races: Accesses of two parallel computational units u_1 and u_2 to the same correlated set s are called *extended data race*, if s is modified, and u_1 and u_2 are not synchronized in a manner that enforces mutual exclusion or a specific order. Our work will be based on this definition.

2 Related Work

There is a lot of prior work dealing with data race detection for single memory locations [2,6,4]. However, the problem of dealing with concurrency bugs involving multiple, correlated variables has only been addressed by few authors. We briefly describe three such publications: The papers [7,8] are tailored towards object oriented environments, in particular Java. It is assumed that annotated

fields of a class (per instance) form an *atomic set*, while methods of the same class are *units of work* on these sets. MUVI [10] detects correlated variables by applying data mining techniques during static analysis. Although the authors' main focus is to detect inconsistent update bugs, a basic variant of the dynamic lockset algorithm is developed. The Serializability Violation Detector (SVD) [11] uses dynamically derived control and data dependencies to detect computational units on the fly. When computational units terminate, information about them is discarded and correlations are built "from scratch" – that is, no persistent information, as with correlated sets or atomic sets, is stored.

Looking at the related works, it seems clear that concepts similar to correlated sets and computational units are necessary for the detection of multi-variable concurrency bugs. However, the methods to infer such constructs vary significantly, as do criteria for actual bug detection. While using serializability can prevent benign races in some cases [7], it is inherently dependent on a concrete schedule. Also, order violation bugs may be overlooked: An example is the *use after initialization* pattern, when thread t_1 writes an initial value to v, while thread t_2 reads v – these operations are obviously serializable, but can still lead to program crashes when executed in the wrong order, e.g. if v is a pointer type. Therefore, it is promising and desirable to bring the benefits of hybrid race detection to the domain of multi-variable concurrency bugs.

3 Race Detection for Correlated Variables

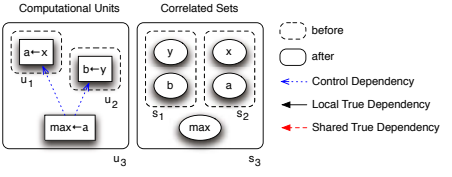
3.1 Inferring Correlated Sets and Computational Units

A prerequisite for detecting extended data races is to dynamically infer correlated sets and computational units. In related work, we have seen several solutions to this problem. Since we aim to develop a method without user intervention, we do not rely on source annotations. Instead, we infer correlated sets and computational units automatically. Our approach is therefore based on the *region hypothesis* [11] for computational units: (a) All operations of a computational unit are related through either true data dependencies (read after write) or control dependencies. (b) Computational units follow the 'read compute write' pattern: A program state is first read from shared memory, the new state is computed using thread exclusive memory and finally written back to shared memory. Therefore, there are no true data dependencies on *shared* memory locations *within* computational units. Additionally, we infer correlated sets using the same heuristic: *All memory locations read or written within a computational unit, form a correlated set.* Based on these criteria, both computational units and correlated sets can be computed fully automatically using the following online algorithm:

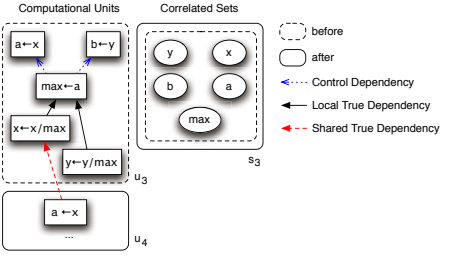
1. Initially, each dynamic operation (instruction) forms its own computational unit and each memory location its own correlated set.
2. When an operation op_1 is executed:
   - We *merge* the computational units of all dynamic operations that op_1 depends on through a control dependency.

– We *merge* the computational units of all dynamic operations that op_1 depends on through a true data dependency[1]. However, if op_1 is true data dependent on op_2 through a *shared* memory location, op_2's computational unit is *not* merged, but instead marked as *closed*.
– We *merge* the correlated sets of all memory locations that op_1 reads, and the correlated sets of all memory locations that op_1 is control dependent on. The merged correlated set is also assigned to the variable written by op_1 (eventually overwriting its old correlated set).

In Figure 2 this algorithm is applied to the function *scaleVector*. Subfigure (a) shows the situation before and after executing $op = max \leftarrow a$: First, the assignments $a \leftarrow x$ and $b \leftarrow y$ form their own computational units u_1 and u_2, and two correlated sets $s_1 = \{a, x\}$ and $s_2 = \{b, y\}$ could be inferred during u_1 and u_2, respectively. After executing op, because of op's control dependencies, all operations are merged into a single computational unit u_3 and all memory locations to a single correlated set s_3. In Subfigure (b), we can see the situation before and after executing *scaleVector* a second time: All operations within this function are related through either control dependencies or true data dependencies; therefore *scaleVector* is recognized as computational unit u_3. Furthermore, when executed a second time, a shared true dependency on x is observed, ending u_3 and starting u_4.



(a) Merging computational units and correlated sets due to control dependency



(b) Ending a computational unit due to shared true dependency

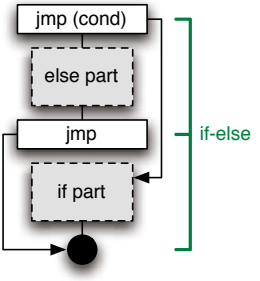
Fig. 2. Region hypothesis applied to the function *scaleVector* of Figure 1

As one can see, it is possible for correlated sets and computational units to contain both shared and exclusive parts alike. While it is mostly the shared parts, which finally matter for data race detection, we must also track thread exclusive computations and memory locations for two main reasons: First, because resources that are now considered exclusive may become shared later on. Second, because correlations between shared resources are often established through exclusive intermediate values – as we've seen in the example above.

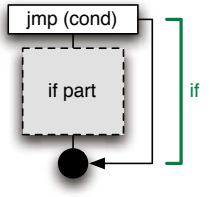
For the *scaleVector* function, the region hypothesis obviously led to correct results. However, because of its heuristic nature, this must not always be the case: In fact, experiments in [11] showed that the region hypothesis holds on the most common paths of 14 examined atomic regions but fails on some rare paths. One

[1] An operation op_1 has a true data dependency on an operation op_2, if op_1 reads a value that was last written by op_2.

common source of errors are shared true dependencies within atomic regions. In these cases, the region hypothesis cuts computational units too early. This limitation could be mitigated by exploiting information about program structure: Shared true dependencies are allowed within computational units, if both operations occur within the same function body (similar to the criterion used for units of work in [8] and [7]). On the other hand, an 'early `if`' could cause the whole program to be interpreted as a single computational unit. We therefore limited the influence of control dependencies on merging to function scope. In [11] control dependencies were completely ignored. One final aspect that has yet to be clarified is how exactly one can detect control dependencies during dynamic analysis. To do so, we use the idea of *reconvergence points* introduced in [12]. When encountering a conditional jump, the jump target is probed to determine the type of control flow construct: For example, if the target is preceded by an unconditional forward jump, we have encountered an `if-else` construct; the reconvergence point is the target of the unconditional jump. On the other hand, if there is *no* jump, we have encountered an `if` construct. Figure 3 illustrates these two cases. However, in contrast to [12] and [11] that are limited to `if` and `if-else` constructs, we're also able to identify loops. This is possible, because of using our loop detection patterns introduced in [3,4]. Furthermore we support non-local jumps caused by `break`, `continue` or `return`.



(a) `if-else` construct



(b) `if` construct

Fig. 3. Finding the reconvergence point (black filled circle)

3.2 Adapting the Lockset Algorithm

As mentioned, the original lockset algorithm checks if every access to a shared resource obeys a certain locking discipline that ensures mutual exclusion. Let us briefly review the original algorithm before we discuss our extensions. The lockset algorithm enforces that every shared memory location v is protected by a non empty set of locks in the sense that all of these locks are held whenever a thread accesses v. Since it is at first unclear which memory location is protected by which locks, we dynamically gather this information during the program's execution: For each Thread t we store L_t, the set of all locks currently held by t. We call L_t the *lockset* of t. Furthermore, we maintain a *candidate set* of locks C_v for each memory location v. Initially, C_v is assumed to consist of all locks and than successively refined on each access to v.

Obviously, this approach considers neither the spatial nor the temporal aspects which we earlier captured in form of correlated sets and computational

units. Therefore, to extend this algorithm for our needs, we must somehow substitute L_t and C_v with equivalents for computational units and correlated sets. Let us look at how we can redefine locksets first. Generally spoken, a computational unit u consists of three parts (see Figure 4):

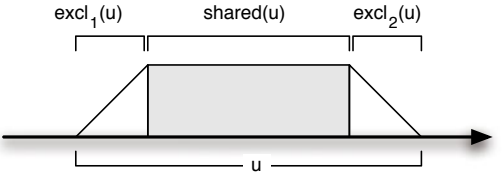


Fig. 4. Different parts of a computational unit u

- $excl_1(u)$: u accesses only exclusive variables
- $shared(u)$: u accesses shared and exclusive variables alike
- $excl_2(u)$: u accesses only exclusive variables

Each of these parts can also be empty.

Based on this observation we define:

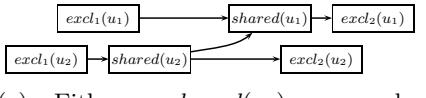
$$L_u := \begin{cases} held_u, & shared(u) \neq \emptyset \\ all\_locks, & \text{otherwise} \end{cases}$$

where

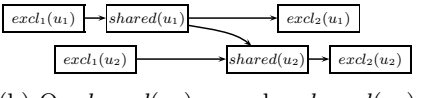
$$held_u := \{\text{Lock } m \mid m \text{ held throughout } shared(u)\}$$

This means: L_u consists of the locks held throughout $shared(u)$, if $shared(u)$ is not empty (denoted by $held_u$ above); otherwise L_u equals the set of all locks. However, because of our tool's intended dynamic nature, we cannot know in advance exactly when $shared(u)$ starts and ends – actually computing L_u is therefore a problem itself, which we discuss in section 3.3. For the moment, we assume that we have all required knowledge available in advance. To complete the adapted lockset algorithm, we can now simply replace L_t with L_u and C_v with C_s (C_s denotes the candidate set of a correlated set s and is computed analogously to C_v).

For the discussion of our locking policy, we assume that a correlated set s is accessed by u_1 and u_2 in parallel, with $L_{u_1} \cap L_{u_2} \neq \emptyset$. Obviously $shared(u_1)$ and $shared(u_2)$ cannot overlap, so that only the following two general cases of interleaving are possible:

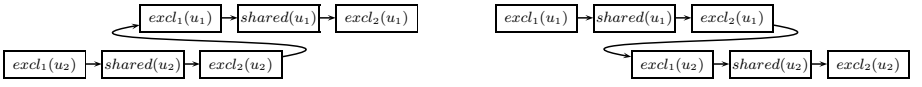


(a) Either $shared(u_2)$ precedes $shared(u_1)$



(b) Or $shared(u_1)$ precedes $shared(u_2)$

In the diagrams above, arrows denote the temporal ordering between individual parts (we will get to know this ordering as happens-before relation in section 3.4). Now, since all exclusive parts solely operate on exclusive variables and do not interfere with parallel computations, the first case is always equivalent to the left side while the second is always equivalent to the right side:



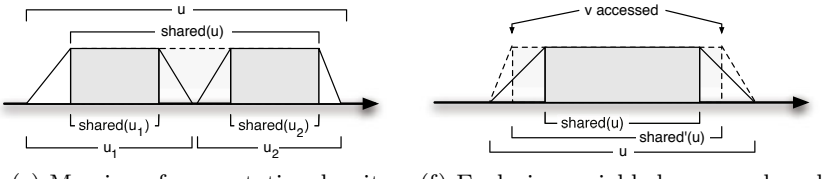
Therefore, our initial assumption implies that all possible interleavings of u_1 and u_2 must be equivalent to either $u_2 \rightarrow u_1$ or $u_1 \rightarrow u_2$ and are thus *serializable*. If we generalize this observation for s with $C_s \neq \emptyset$ and an arbitrary number of u_i accessing s, all u_i are serializable. This guarantees that there is no data race on s.

Note that there are other possibilities to define L_u. In particular, we could have defined L_u to consist of all locks that are held from the very beginning to the very end of u. However, this definition would yield many false positives, since $excl_1(u)$ and $excl_2(u)$ do not need to be protected by locks.

3.3 Calculating a Computational Unit's Lockset

In the previous section, we assumed that we are endowed with sufficient a priori knowledge to compute L_u. That is, knowledge about which memory accesses constitute $shared(u)$. In dynamic program analysis, however, $shared(u)$ can repeatedly change for various reasons:

1. After the last access to a shared memory location, we must assume that all further exclusive read/write-operations are part of $excl_2(u)$. This assumption must be revised, if another access to a shared memory location follows.
2. Upon merging two computational units u_1 and u_2 to u, their shared parts must be combined, yielding $shared(u)$. $shared(u)$ may now contain accesses that are neither part of $shared(u_1)$ nor of $shared(u_2)$ (Figure 5(a)).
3. A variable v that was formerly considered exclusive, may later turn out to be actually shared. The shared part of the computational unit that accessed v earlier, must then be extended accordingly (Figure 5(b)).



(e) Merging of computational units (f) Exclusive variable becomes shared

Fig. 5. Reasons for $shared(u)$ to change

We can solve these problems by introducing a new concept called *lock vector*, which is inspired by vector clocks. For a thread t, the lock vector function $\mathbf{l}_t$ is defined as follows:

$$\mathbf{l}_t : \mathit{Lock} \to \mathbb{N} \times \mathbb{N},\ m \mapsto (\mathbf{l}_t(m)_{\text{acq}}, \mathbf{l}_t(m)_{\text{rel}}).$$

$\mathbf{l}_t$ maps a lock m to its number of acquisitions $\mathbf{l}_t(m)_{\text{acq}}$ and releases $\mathbf{l}_t(m)_{\text{rel}}$ by thread t so far. Note that $\mathbf{l}_t(m)_{\text{acq}}$ is in general *not* equal to the number of calls to $m.lock()$ by t: For Example, in the case of a recursive lock, $\mathbf{l}_t(m)_{\text{acq}}$ isn't further increased, if t already is in possession of m. Furthermore, we store two local copies of the lock vector $\mathbf{l}_t$ at the beginning and end of $\mathit{shared}(u)$, called $\mathbf{l}_{\text{fst},u}$ and $\mathbf{l}_{\text{lst},u}$. Then, we can compute L_u as follows:

$$L_u = \{\text{Lock } m \mid \overbrace{\mathbf{l}_{\text{fst},u}(m)_{\text{acq}} - \mathbf{l}_{\text{fst},u}(m)_{\text{rel}} = 1}^{m \text{ held at the beginning of } \mathit{shared}(u)} \wedge \underbrace{\mathbf{l}_{\text{lst},u}(m)_{\text{rel}} - \mathbf{l}_{\text{fst},u}(m)_{\text{rel}} = 0}_{m \text{ not released until the end of } \mathit{shard}(u)} \}$$

If another shared variable is accessed by u, we can easily update $\mathbf{l}_{\text{lst},u}$ and L_u; if two computational units u_1 and u_2 are merged to u, we set

$$\mathbf{l}_{\text{fst},u} = \min(\mathbf{l}_{\text{fst},u_1}, \mathbf{l}_{\text{fst},u_2}) \quad \text{and} \quad \mathbf{l}_{\text{lst},u} = \max(\mathbf{l}_{\text{lst},u_1}, \mathbf{l}_{\text{lst},u_2})$$

using element-wise comparison and recompute L_u. The problems caused by the above points 1) and 2) are therefore solved, yet problem 3) still remains. To solve it as well, we must store additional copies $\mathbf{l}_{\text{fst},v}$ and $\mathbf{l}_{\text{lst},v}$ of $\mathbf{l}_t$ for an exclusive variable v the first and last time it is accessed by u. If v later becomes shared, u's lock vectors are then updated as follows:

$$\mathbf{l}_{\text{fst},u} = \min(\mathbf{l}_{\text{fst},v}\ , \mathbf{l}_{\text{fst},u}) \quad \text{and} \quad \mathbf{l}_{\text{lst},u} = \max(\mathbf{l}_{\text{lst},v}\ , \mathbf{l}_{\text{lst},u})$$

and L_u is recomputed.

This concludes our description of the adapted lockset algorithm. Before we continue to explain how to integrate temporal ordering, we will exemplarily apply it to the function *scaleVector* in Figure 1. The result is shown in Table 1: Initially, there are neither acquisitions nor releases of lock m, while u is in $excl_1$ and L_u, therefore, contains all locks by definition. As we encounter the

Table 1. Lockset algorithm applied to *scaleVector* (new values are marked bold)

statement executed	$\mathbf{l}_t(m)$	part of u	$\mathbf{l}_{\text{fst},u}(m)$	$\mathbf{l}_{\text{lst},u}(m)$	L_u	C_s
initially	$(0,0)$	$excl_1$	–	–	*all*	*all*
lock(m)	$\mathbf{(1,0)}$	$excl_1$	–	–	*all*	*all*
$(a,b) \leftarrow (x,y)$	$(1,0)$	$\boldsymbol{shared}$	$\mathbf{(1,0)}$	$\mathbf{(1,0)}$	$\mathbf{\{m\}}$	$\mathbf{\{m\}}$
unlock(m)	$\mathbf{(1,1)}$	*shared*	$(1,0)$	$(1,0)$	$\{m\}$	$\{m\}$
$max \leftarrow a$ **if** $a > b$ **else** b	$(1,1)$	$\boldsymbol{excl_2}$	$(1,0)$	$(1,0)$	$\{m\}$	$\{m\}$
lock(m)	$\mathbf{(2,1)}$	$excl_2$	$(1,0)$	$(1,0)$	$\{m\}$	$\{m\}$
$(x,\ y) \leftarrow (\frac{x}{max}, \frac{y}{max})$	$(2,1)$	$\boldsymbol{shared}$	$(1,0)$	$\mathbf{(2,1)}$	$\mathbf{\emptyset}$	$\mathbf{\emptyset}$

first acquisition of m, we increase $\mathbf{l}_t$ for m to $(1,0)$. With the assignment in the third row of Table 1, u accesses the shared resources x and y: u switches to its *shared* part and makes local copies of $\mathbf{l}_t$. When writing to max, we can assume that u has reached $excl_2$. However, this assumption must be revised on the next access to x and y. Since $\mathbf{l}_t(m)$ has changed to $(2,1)$ in between, $\mathbf{l}_{\mathrm{lst},u}(m)$ also takes this new value, causing L_u and finally C_s to become empty. The detection of an extended data race will be reported at this point.

3.4 Happens-Before Analysis – Hybrid Race Detector

A pure lockset based race detector fails to recognize synchronizations like signal/wait, fork/join or barriers, and will therefore produce many false positives. As a hybrid race detector, our approach therefore combines the lockset algorithm with the *happens-before relation* $\rightarrow_{\mathrm{hb}}$ that tracks the temporal and causal ordering of events. For two such events e_1 and e_2 we define:

$$e_1 \rightarrow_{\mathrm{hb}} e_2 \;:\Leftrightarrow\; e_1 \text{ precedes } e_2 \text{ temporally/causally}$$
$$e_1 \| e_2 \;:\Leftrightarrow\; e_1 \not\rightarrow_{\mathrm{hb}} e_2 \wedge e_2 \not\rightarrow_{\mathrm{hb}} e_1$$

The $\rightarrow_{\mathrm{hb}}$ relation itself is implemented by *vector clocks*: A global timestamp vector, called $\mathbf{v}$, is tracked using thread local event counters that are exchanged on synchronization events [14,13]. For two events e_1 and e_2 that take place at times $\mathbf{v}_1$ and $\mathbf{v}_2$, we then have $e_1 \rightarrow_{\mathrm{hb}} e_2 \Leftrightarrow \mathbf{v}_1 < \mathbf{v}_2$ with element-wise comparison.

To combine the lockset algorithm and $\rightarrow_{\mathrm{hb}}$, we use a state machine based on our race detector Helgrind$^+$ [5,4,3]. This state machine can distinguish between parallel and ordered accesses to a correlated set s as follows: Whenever s is accessed, a copy of $\mathbf{v}$, called $\mathbf{v}_s$, is stored with s. Any subsequent access to s at time $\mathbf{v}'$ is then parallel to the first one, iff $\mathbf{v}_s \not< \mathbf{v}'$.

4 Evaluation

Our implementation borrows from Helgrind$^+$ [4], which in turn is based on the Valgrind framework [15,16]. The general principle behind Valgrind is called *disassemble-instrument-resynthesize*. Helgrind$^+$, among other things, provides an implementation of *shadow memory*. Our implementation uses shadow values to store references to point the correlated set and a collection of computational units. In experimental evaluation, we look at complex examples taken from real-world applications to check if computational units, correlated sets, and extended data races are detected. We compare our results to our enhanced race detector, Helgrind$^+$ [3,4], which serves as representative for tools that do not natively support multi-variable race detection.

4.1 Detecting Extended Data Races

The results of our evaluation focuses on the detection of correlations and extended data races shown in Table 2. The simplified codes taken from real applications are indicated as Tests 1 to 8 in the table.

Table 2. Detected correlated sets, computational units and data races

Test-No.	Description	Expected	Helgrind$^+$	Our approach CUs & CSets	Our approach Race
1	ScaleVector without locks	Race	Race	✓	Race
2	ScaleVector with interrupted locks	Race	× *No race*	✓	Race
3	Normalize with consistent locking	No Race	No race	✓	No race
4	Different locks	Race	× *No race*	✓	Race
5	AppendBuffer without locks	Race	× *Multiple races*	✓	Race
6	Swapping correlated variables with locks	No race	No race	✓	No race
7	Swapping uncorrelated variables with locks	No race	No race	×	× *Race*
8	Independent calculations	–	–	×	–

Tests 1 and 2 simply represent the function *scaleVector* from Figure 1. For test 1, the locks were completely omitted, whereas in test 2 the locks were kept as in Figure 1. While Helgrind$^+$ is able to detect the locking violation in test 1, it fails in test 2: In this test, atomicity is violated by releasing and re-acquiring locks during a logical operation. However, Helgrind$^+$ is not able to detect this kind of bug, since each single access to a shared variable is properly protected by locks. The new approach detects the race and passes in test 2.

Test 3 represents another arithmetic function for vector normalization (Figure 6). It has a more complicated dependency graph, because it uses the `sqrt()` function. Still, all correlations are detected. The normalization itself is consistently protected by a single lock, so it is data race free.

```
1 normalize() {
2  lock(&m);
3  float len = sqrt(a*a + b*b);
4  a = a/len; b = b/len;
5  unlock(&m);
6 }
```

Fig. 6. Test 3: Vector normalization

A similar scenario is represented by test 4, shown in Figure 7. The two shared variables `mCont` and `mLen` are related through the string `s`, yet protected by different locks. Again, a typical race detector cannot detect this data race. In contrast, our approach correctly infers the correlated set {`s`, `mCont`, `mLen`} and detects its empty lockset. This test also shows that our method handles function calls reliably: When `f()` is called, `s` is copied from `append()`'s stack frame to `f()`'s stack frame,

```
1  append(char *s) {
2   lock(&m1);
3   mCont = f(mCont,s);
4   unlock(&m1);
5  }
6  reflow(char *s) {
7   lock(&m2);
8   mLen = g(mLen,s);
9   unlock(&m2);
10 }
```

Fig. 7. Test 4: Using different locks for correlated variables

making the two copies dependent. When `f()` ends, the return value is stored in a machine register that depends on the original `s` and `mCont`. This register value is finally moved to `mCont` establishing the correlation of `mCont` and `s`. Likewise, `mLen` is included in this correlated set.

Test 5 is shown in Figure 8; it is part of Apache's `log_config` module, which contains a data race (now fixed). `outCnt` and `outBuf` implement a buffer for status messages. The correlation between those two variables is established through `len`, so that `bufferAppend()` is correctly identified as a single computational unit. Since there are no locks to protect the shared resources, traditional race detectors report *multiple* data races on `outBuf` and `outCnt`. In contrast, our approach reports only a *single* data race on $s = \{\texttt{outCnt}, \texttt{outBuf}\}$. This can make it easier for the programmer to identify the root cause of the detected race and to apply a correct bug fix. Just reporting several seemingly unrelated data races on the other hand may mislead the programmer to wrap every access to a shared variable in locks, but overlook their correlation. This can be a serious problem, especially if the static distance between these accesses is bigger than in our example.

Figure 8 also shows a simple implementation of the `strlen()` function. We include it to demonstrate how control dependencies are tracked by our implementation: The incrementation of `ctr` depends on the outcome of the loop condition `str[ctr]`, leading to the correlation between `ctr` and `str`.

```
1  void bufferAppend(char *str) {
2     int len = strlen(str);
3     if (len+outCnt >= BUFSIZE) {
4        // flush buffer!
5        outCnt = 0;
6     }
7     for (int i = 0; i < len; i++) {
8        outBuf[outCnt+i] = str[i];
9     }
10    outCnt = outCnt+len;
11 }
12 int strlen(char *str) {
13    int ctr = 0;
14    while(str[ctr]) { ctr++; }
15    return ctr;
16 }
```

Fig. 8. Test 5: Appending to a buffer without locks

In tests 6 and 7, two shared variables `a` and `b` are swapped using a temporary variable:

```
tmp = a; a = b; b = tmp;
```

The outcome depends on the previous state of the two variables `a` and `b`: Swapping is correctly identified as a logical operation, when `a` and `b` were already correlated. But for independent values our approach fails for the same reason as in test 4 of the

```
1 static OBJ *list_head;
2 OBJ *dequeue_and_fill(int a, int b) {
3   OBJ *head = list_head;
4   head->a = a, head->b = b;
5   list_head = head->next;
6   return head;
7 }
```

Fig. 9. Test 8: Independent operations within a atomic region

previous section: First, `tmp` inherits `a`'s correlated set, while `a`'s own correlated set is overwritten with the one of *b*. Then *b*'s set is overwritten by `tmp`'s. Effectively, `a` and `b` have now swapped their correlated sets as well. When the variables are accessed for the next time, protected by the same locks as before swapping their values, the locksets then become empty and false positives are reported.

Finally, in test 8 [11], shown in Figure 9, a data structure consisting of semantically correlated variables is initialized, but the initialization values are independent. Inferring of correlated sets and computational units fails in such cases. This special case could be solved by considering address calculations for dependency analysis: `head->a` and `head->b` are computed by adding two fixed offsets to `head`. However, tracking address dependencies could cause over-estimation of correlated sets, since `struct`-members must not automatically be related: Think, for example, of a data structure for counting incoming and outgoing data packets.

5 Conclusion and Future Work

Traditional approaches to data race detection fail in cases where several correlated variables are involved. Based on our definitions of *extended* data races, computational units, and correlated sets, we have demonstrated how to modify the lockset algorithm and the happens-before analysis for such cases. For our implementation, we have opted for inferring correlated sets and computational units fully automatically. We made use of the region hypothesis and proposed improvements based on the program structure, i.e. allowing shared true dependencies within function scope and limiting the effect of control dependencies to function scope.

The evaluation showed that our enhanced race detection approach is able to detect synchronization operations reliably. In contrast to previous approaches, it also works for the case of correlated variables and logical operations. Even if extended data races manifest as multiple single variable data races, our approach is still able to provide further informations that helps identify the problem's root cause.

Some technical improvements are necessary for the current implementation of our approach to integrate it into our race detector Helgrind$^+$ and make it more practical and usable.

We have also seen that in few cases inferring correlated sets and computational units fails. Note that one of our approach's feature is its orthogonality between race detection and finding correlated sets and computational units: We can switch to other methods for the latter, without the need to alter the former. This property will make it easier to further improve the region hypothesis or use completely different ways to infer correlated sets in our implementation. For example, it can be worthwhile to require the user to specify at least correlated sets by annotations.

Alternatively, we could exploit new parallel programming paradigms that are currently gaining focus: i.e. `Task`s and `Operation`s that are being dispatched to execution queues, or `Future`s naturally encapsulate concepts similar to computational units. We leave exploring such possibilities for future work.

References

1. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 329–339. ACM, New York (2008)
2. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
3. Jannesari, A., Tichy, W.: Identifying ad-hoc synchronization for enhanced race detection. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–10 (19-23, 2010)
4. Jannesari, A., Bao, K., Pankratius, V., Tichy, W.F.: Helgrind+: An efficient dynamic race detector. In: International on Parallel and Distributed Processing Symposium, pp. 1–13 (2009)
5. Jannesari, A., Tichy, W.F.: On-the-fly race detection in multi-threaded programs. In: PADTAD 2008: Proceedings of the 6th Workshop on Parallel and Distributed Systems, pp. 1–10. ACM, New York (2008)
6. Yu, Y., Rodeheffer, T., Chen, W.: Racetrack: efficient detection of data race conditions via adaptive tracking. SIGOPS Oper. Syst. Rev. 39(5), 221–234 (2005)
7. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 231–240. ACM, New York (2008)
8. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 334–345. ACM, New York (2006)
9. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
10. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: SOSP 2007: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 103–116. ACM, New York (2007)
11. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. 40(6), 1–14 (2005)
12. Collins, J.D., Tullsen, D.M., Wang, H.: Control flow optimization via dynamic reconvergence prediction. In: MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 129–140. IEEE Computer Society, Washington, DC, USA (2004)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
14. Fidge, J.: Timestamps in Message Passing Systems that Preserve the Partial Ordering. In: Proc. 11th Australian Computer Science Conf., pp. 55–66 (1988)
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)
16. Nethercote, N., Seward, J.: Valgrind: A program supervision framework (2003), http://valgrind.org/
17. Butenhof, D.R.: Programming with POSIX Threads. ser. Professional Computing Series. Addison-Wesley, Reading (1997)

Improving the Parallel Schnorr-Euchner LLL Algorithm

Werner Backes and Susanne Wetzel

Stevens Institute of Technology
Castle Point on Hudson, Hoboken, NJ 07030 USA
{wbackes,swetzel}@cs.stevens.edu

Abstract. This paper introduces a number of modifications that allow for significant improvements of parallel LLL reduction. Experiments show that these modifications result in an increase of the speed-up by a factor of more than 1.35 for SVP challenge type lattice bases in comparing the new algorithm with the state-of-the-art parallel LLL algorithm.

1 Introduction

In the course of the last twenty-five years, lattice basis reduction has evolved as a main tool in modern cryptanalysis [9,10]. At the same time, lattice theory has gained increased importance in the context of developing new cryptographic primitives [12,11,4].

The goal of lattice basis reduction is to determine a nearly-orthogonal, short basis of a lattice. The first polynomial-time lattice basis reduction algorithm for lattices of arbitrary size was introduced by Lenstra, Lenstra, and Lovász [8]. A break-through in enabling lattice basis reduction in practice is due to Schnorr and Euchner [13] who proposed a variant of the original LLL algorithm. In the past twenty years there were only a few efforts that focused on parallelizing lattice basis reduction. Main results include parallelization efforts for the original LLL algorithm [7] and work that is tailored for a vector computer architecture [6]. Recently, Backes and Wetzel [2] introduced the first parallelization of the LLL algorithm that can make efficient use of modern multi-processor, multi-core systems. The algorithm proposed in [2] builds on an advanced Schnorr-Euchner algorithm which in practice performs on par with the current implementation of the L^2 algorithm [9].

This paper introduces an improvement to the parallel advanced Schnorr-Euchner LLL reduction algorithm that significantly advances the state-of-the-art in parallel LLL reduction [2]. In particular, based on the algorithm in [2], this paper develops a new approach which dynamically determines the number of active threads participating in the computation at runtime. These dynamic adjustments are geared to reduce the synchronization overhead and waiting times caused by the frequent—yet necessary—use of barriers and locks to ensure the correctness of the computations.

In addition, this paper proposes the use of sequences of reduction parameters instead of a single reduction parameter. This paper shows that this approach

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 27–39, 2011.

proves to be beneficial for parallel advanced Schnorr-Euchner LLL reduction, despite the fact that in a majority of test cases it only has a marginal effect on the performance of the sequential advanced Schnorr-Euchner algorithm.

Experiments show that the new algorithm performs particularly well for SVP challenge type lattice bases for which the parallel advanced Schnorr-Euchner LLL in [2] is experiencing scaling problems. Specifically, the improvements introduced in this paper result—for the 8-thread version—in an increase of the speed-up by a factor of more than 1.35 compared to the parallel advanced Schnorr-Euchner LLL in [2] when reducing SVP challenge type lattice bases. The improvement of the speed-up is even larger for SVP challenge type lattice bases of smaller dimension. In addition, the 8-thread version of the newly improved algorithm achieves a speed-up in reducing knapsack type lattice bases of close to 4.5.

2 Preliminaries

A *lattice* $L = \left\{\sum_{i=1}^{k} x_i \underline{b}_i | x_i \in \mathbb{Z}, 1 \leq i \leq k\right\} \subset \mathbb{R}^n$ is an additive discrete subgroup of $\mathbb{R}^n$. The linear independent vectors $\underline{b}_1, \ldots, \underline{b}_k \in \mathbb{R}^n$ $(k \leq n)$ form a basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ of dimension k of the lattice L. The basis of a lattice is not unique. Specifically, a basis B may be transformed into another basis B' of the same lattice L by applying a *unimodular transformation* U, i.e., $B' = BU$ with $U \in \mathbb{Z}^{n \times k}$ and $|\det U| = 1$. Unimodular transformations include swaps (exchange of two base vectors) and translations (adding an integral multiple of one base vector to another one). The *Gram-Schmidt orthogonalization* $B^* = (\underline{b}_1^*, \ldots, \underline{b}_k^*)$ of a lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ is computed as $\underline{b}_1^* = \underline{b}_1$, $\underline{b}_i^* = \underline{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \underline{b}_j^*$ for $2 \leq i \leq k$ with $\mu_{i,j} = \frac{\langle \underline{b}_i, \underline{b}_j^* \rangle}{\|\underline{b}_j^*\|}$ for $1 \leq j < i \leq k$ where $\langle ., . \rangle$ defines the scalar product of two vectors. It is important to note that B^* is not necessarily a basis of the lattice L, nor is a vector $\underline{b}_i^*$ of the orthogonalization B^* necessarily in L. The goal of lattice basis reduction is to find a lattice basis B' for the lattice $L(B)$ such that the vectors in B' are as short and as orthogonal to each other as possible. The most well-known and most-widely used lattice basis reduction method is the LLL reduction method [8]:

Definition 1. *For a lattice $L \subseteq \mathbb{R}^n$ with basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$, corresponding Gram-Schmidt orthogonalization $B^* = (\underline{b}_1^*, \ldots, \underline{b}_k^*) \in \mathbb{R}^{n \times k}$ and coefficients $\mu_{i,j}$ $(1 \leq j < i \leq k)$, the basis B is LLL-reduced if* (1) $|\mu_{i,j}| \leq \frac{1}{2}$ *with* $1 \leq j < i \leq k$ *and* (2) $\|\underline{b}_i^* + \mu_{i,i-1}\underline{b}_{i-1}^*\|^2 \geq y\|\underline{b}_{i-1}^*\|^2$ *for* $1 < i \leq k$.
The reduction parameter y may be chosen arbitrarily from $\left(\frac{1}{4}, 1\right)$. In practice, an LLL-reduced basis can efficiently be computed using the Schnorr-Euchner algorithm [13,1].

The original Schnorr-Euchner LLL algorithm [13] has significantly evolved over time (see Algorithm 1). The advanced Schnorr-Euchner LLL (referred to as aSE-LLL) was used by Backes and Wetzel in [2] as a starting point for the development of the parallel advanced Schnorr-Euchner LLL algorithm (referred to as parallel aSE-LLL). In contrast to the original LLL algorithm [8], the original and advanced Schnorr-Euchner algorithms use floating-point approximations of vectors and the basis (APPROX_BASIS and APPROX_VECTOR in Lines `(1)` and `(24)`)

thus making the LLL reduction practical. To increase the stability, the original and advanced Schnorr-Euchner algorithms include suitable measures in the form of correction steps (see [13] for details). These corrections include the computation of exact scalar products (Line (6)) as part of the Gram-Schmidt orthogonalization and a step-back (Line (28)) in case of a large μ_{ij} used as part of the size-reduction (Line (19)). The original and advanced Schnorr-Euchner algorithms use an exact data type for the actual operations on the lattice basis (Line (18)). (In Algorithm 1, p denotes the bit precision of the data type used to approximate the lattice basis.)

Algorithm 1. Advanced Schnorr-Euchner LLL

INPUT: Lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$
OUTPUT: LLL-reduced lattice basis B

```
(1)  APPROX_BASIS(B', B)
(2)  while (i ≤ k) do
       /* scalar products */
(3)    R_ii = ‖b'_i‖
(4)    for (1 ≤ j < i) do
(5)      if (|⟨b'_i, b'_j⟩| < 2^(-p/2) ‖b'_i‖ ‖b'_j‖) then
(6)        R_ij = APPROX_VALUE(⟨b_i, b_j⟩)
(7)      else
(8)        R_ij = ⟨b'_i, b'_j⟩
       /* orthogonalization */
(9)    μ_ii = 1, S_1 = R_ii
(10)   for (1 ≤ j < i) do
(11)     R_ij = R_ij − Σ_{m=1}^{j-1} R_im μ_jm
(12)     μ_ij = R_ij / R_jj
(13)     R_ii = R_ii − R_ij μ_ij
(14)     S_{j+1} = R_ii
       /* size-reduction/μ-update */
(15)   for (i > j ≥ 1) do
(16)     if (|μ_{i,j}| > 1/2) then
(17)       F_r = true
(18)       b_i = b_i − ⌈μ_ij⌋ b_j
(19)     if (|μ_ij| > 2^(p/2)) then
(20)       F_c = true
(21)       for (1 ≤ m ≤ j) do
(22)         μ_im = μ_im − ⌈μ_ij⌋ μ_jm
(23)   if (F_r = true) then
(24)     APPROX_VECTOR(b'_i, b_i)
       /* recompute orth. */
(25)   if (F_c = false ∧ F_r = true) then
(26)     RECOMPUTE_R_ij() [1]
(27)   F_r = false
       /* check LLL condition */
(28)   if (F_c = true) then
(29)     i = max(i − 1, 2)
(30)     F_c = false
(31)   else
(32)     i' = i
(33)     while ((i > 1) ∧
                (y R_{i-1,i-1} > S_{i-1})) do
(34)       SWAP(b_i, b_{i-1})
(35)       SWAP(b'_i, b'_{i-1})
(36)       i = i − 1
(37)     if (i ≠ i') then
(38)       if (i = 1) then
(39)         R_11 = ‖b'_1‖
(40)         i = 2
(41)     else
(42)       i = i + 1
```

In contrast to the original Schnorr-Euchner algorithm, aSE-LLL incorporates a number of improvements—based on the work by Nguyen and Stehle [9], the NTL implementation of LLL [15], as well as some additional (mostly technical) optimizations. These improvements increase the stability of the reduction while significantly decreasing the overall running time. Specifically, the improvements include modifications to the computation of the Gram-Schmidt orthogonalization and coefficients (Algorithm 1, Lines (3) - (14)) in order to increase the accuracy of the computations. In addition, the improvements include a modified checking of the so-called LLL condition (Condition (2), Definition 1). Unlike in the original Schnorr-Euchner algorithm, the aSE-LLL condition check (Algorithm 1, Lines (28) - (42)) allows the algorithm to perform a sequence of vector swap operations without the need to recompute the Gram-Schmidt orthogonalization in between those swaps.

Figures 1 and 2 compare the performance of aSE-LLL to L^2 (using fpLLL version 3.1.1) for knapsack and SVP challenge type lattice basis (see Section 6 for details on these types of lattice bases). fpLLL includes a proved and a heuristic

[1] RECOMPUTE_R$_{ij}$ in Line (26) of Algorithm 1 performs the same operations as shown in Lines (3) - (12).

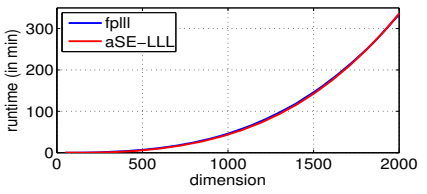


Fig. 1. aSE-LLL compared to fpLLL for knapsack type lattice basis

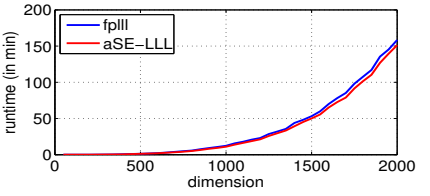


Fig. 2. aSE-LLL compared to fpLLL for SVP challenge type lattice basis

variant of the L^2 algorithm. The proved variant automatically determines the necessary precision for the approximation based on the input whereas the heuristic variant allows the user to set the precision manually. In practice, the heuristic variant works well with a smaller precision than the one that would have been determined by the proved variant and as such is generally faster. In order to allow for a fair comparison with aSE-LLL, the tests have therefore been performed based on the heuristic variant of L^2. The precision for the approximation of the lattice bases and the computation of the Gram-Schmidt orthogonalization was set to $p = 128$ bit for knapsack type and to $p = 256$ bit for SVP challenge type lattice bases (for both aSE-LLL and the heuristic variant of L^2). The results clearly show that in practice the performance of aSE-LLL is on par with the performance of the L^2 algorithm. The parallel LLL algorithm in [2] and the improvements in this paper build on aSE-LLL. It is important to note that—with suitable modifications—it might be possible to adopt the methods of [2] as well as the improvements introduced in this paper for the fpLLL implementation of the L^2 algorithm—possibly leading to an efficient parallel implementation of L^2.

3 Parallel LLL Reduction

In [2], Backes and Wetzel introduced a parallel LLL algorithm that builds on aSE-LLL (Algorithm 1). Their parallel algorithm relies on a PRAM type model where all threads have instant access to shared memory. The shared memory is used to synchronize the parallel computation efforts and to exchange data in between threads. They were the first to find a solution that deals with the dependencies in the main while-loop of the aSE-LLL algorithm which up to then made it impossible to efficiently parallelize the algorithm.

Algorithm 2. Parallel Advanced Schnorr-Euchner LLL Algorithm (simplified)

INPUT: Lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$
OUTPUT: LLL-reduced lattice basis B

```
(1)  APPROX_BASIS(B', B)
(2)  while (i ≤ k) do
(3)    SCALARPRODUCTS(R, μ)
(4)    ORTHOGONALIZATION(R, μ)
(5)    f = μ-UPDATE(μ)
(6)    SIZEREDUCTION(B, f)
(7)    if (F_c = false ∧ F_r = true) then
(8)      RECOMPUTE_R_ij()
(9)    F_r = false
(10)   if (F_c = true) then
(11)     i = max(i − 1, 2)
(12)     F_c = false
(13)   else
(14)     i' = i
(15)     while ((i > 1) ∧
                (y R_{i−1,i−1} > S_{i−1})) do
(16)       SWAP(b_i, b_{i−1})
(17)       SWAP(b'_i, b'_{i−1})
(18)       i = i − 1
(19)     if (i ≠ i') then
(20)       if (i = 1) then
(21)         R_11 = ||b'_1||
(22)         i = 2
(23)       else
(24)         i = i + 1
```

The structure of the parallel aSE-LLL (see Algorithm 2) is very similar to the structure of the sequential aSE-LLL (see Algorithm 1). This is due to the

fact that so far it has been impossible to exploit parallelism at a higher level, i.e., the main while-loop of the (advanced) Schnorr-Euchner LLL algorithm. Backes and Wetzel developed the only viable alternative known to date in efficiently parallelizing all components within the main while-loop of the (advanced) Schnorr-Euchner LLL.

In their solution, Backes and Wetzel replace the original scalar product, orthogonalization, and size-reduction/μ-update components of Algorithm 1 with especially designed parallel counterparts of these components. Algorithm 2 shows a simplified version of the parallel aSE-LLL in [2]. In the parallel algorithm, the update of the μ_{ij} (which corresponds to Lines `(24)` - `(25)` in Algorithm 1) was taken out of the size reduction part and instead was implemented as separate routine. In comparison to the sequential algorithm, the work in [2] uses an additional vector $\underline{f}$ to store the factors necessary for the updates of the exact lattice basis (see Line `(16)`, Algorithm 1). In the parallel reduction algorithm in [2], the routines SCALARPRODUCTS, ORTHOGONALIZATION, μ-UPDATE, and SIZEREDUCTION were separated by barriers in order to ensure the correctness of the respective computations.

The results in [2] showed that the speed-up[2] that can be achieved with this parallel aSE-LLL algorithm does not only depend on the type of the lattice basis to be reduced, but also depends on the architecture of the machine the algorithm is running on. In particular, the bandwidth and the latency for memory access has a profound effect on the speed-up that the algorithm can achieve. The great impact of these factors is due to the fact that all threads of the algorithm frequently access (read and write) data structures in shared memory. It is important to note that on modern CPUs (with integrated memory controllers and fast interconnects in between caches) one can therefore achieve a significantly larger speed-up than on CPUs lacking these features.

4 Motivation and Approach

The work in [2] showed this new algorithm to be very efficient and well-suited for knapsack type lattices, random lattices (as defined by Goldstein and Mayer [5]), and unimodular lattice bases. In further analyzing the algorithm in [2], we discovered scaling issues with a newly developed class of lattice bases—referred to as SVP challenge type lattice bases. The speed-up that the parallel algorithm of [2] achieves for these SVP challenge type bases is significantly lower than the speed-up achieved for the lattice bases tested in [2]. The SVP challenge type lattice bases are derived from lattice bases generated by researchers of the TU Darmstadt as part of their SVP challenge [14]. In order to generate SVP challenge type lattice bases, the generator program for the original SVP challenge bases was modified to limit the size of the lattice basis entries.[3]

[2] The speed-up is defined as the quotient of the running times for the sequential and the parallel version of the algorithm.

[3] Smaller entries allow for large-scale experiments in higher dimensions within a reasonable amount of time.

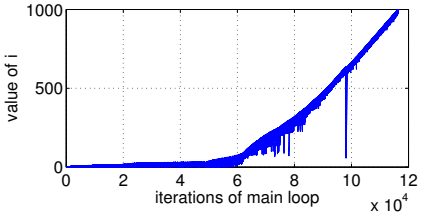


Fig. 3. Value of i for knapsack type lattice basis

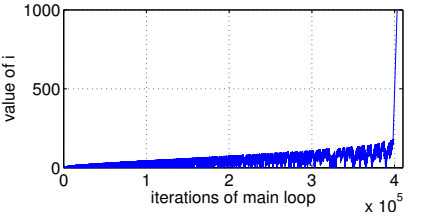


Fig. 4. Value of i for a SVP challenge type lattice basis

In analyzing the value of variable i of the main while-loop of the parallel aSE-LLL (Line `(2)`, Algorithm 2), we found evidence for the cause of the gap between the speed-up for, e.g., knapsack type and SVP challenge type lattice bases. Recall that the loop variable i denotes the basis vector on which the aSE-LLL algorithm is currently working, using the already LLL reduced basis vectors 1 to $i-1$. The amount of computation within a particular iteration of the main while-loop depends on the value of variable i. That is, the larger i, the larger is the amount of computation performed (e.g., more scalar products and a larger portion of the orthogonalization is computed). Consequently, a larger i allows for a significantly larger amount of computations to be parallelized. Thus, in general a large percentage of iterations with large i results in the parallel aSE-LLL algorithm to achieve a larger speed-up. Figures 3 and 4 show the value of i for every iteration of the main loop of Algorithm 2. The value of loop variable i increases much earlier (for specific iterations) for knapsack type lattice bases (see Figure 3) than for the SVP challenge type lattice bases (see Figure 4). Consequently, these figures provide evidence for a correlation between the overall progression of i and the speed-up of the parallel aSE-LLL algorithm in [2].

The analysis of the loop variable values suggests that one should concentrate efforts to achieve further improvements on optimizing the parallel computations for small values of i without sacrificing the superior performance that is already in place for larger values of i. It is in this context that we introduce a new approach that allows us to significantly improve the speed-up for lattice bases which prove to be problematic for the algorithm in [2].

5 Improved Parallel LLL Reduction

Our work includes two main directions: First, we propose improvements to the main components of the parallel aSE-LLL in [2] (Sections 5.1 – 5.3). These components have been modified to take the value of loop variable i into account and optimize the parallel computations for small i. In particular, our modifications are tailored to increase the overall efficiency of the parallel algorithm and minimize the overhead caused by the use of barriers and locks. Even small improvements can lead to a significant overall performance improvement since the main components are executed repeatedly within the main while-loop. Second, we propose the use of a sequence of reduction parameters instead of a single reduction parameter (Section 5.4).

5.1 Scalar Product Part

In each iteration of the main while-loop, the aSE-LLL has to compute i scalar products using the approximate representation of the lattice basis. Schnorr and Euchner introduced a heuristic (see [13] for details) to determine if the accuracy of the result is sufficient. This condition might require the algorithm to recompute the scalar product at a significantly higher cost using the exact representations of the lattice basis. It cannot be predicted at what point or how many exact scalar products have to be computed.

The work in [2] implemented a work stealing approach in which the scalar product computation is divided into smaller slices of scalar product computations. This approach allows Backes and Wetzel to balance the overall scalar product work load recognizing that this not necessarily translates into an equal number of scalar products computed by each thread. One of the shortcomings in [2] is the use of a fixed size for the slice of the parallel scalar product computations. We improve on this in that we dynamically (within certain limits) decide on the size of the distinct slices dependent on the value of loop variable i. The parameter sp_{max} is used to limit the maximum size of a computation slice. Each thread participating in the parallel computation executes the following instructions as part of our improved parallel scalar product computation:[4]

Scalar Product – Thread$_t$

```
(1)  s_sp = ( ⌈i/n⌉ > sp_max ) ? sp_max : ⌈i/n⌉
(2)  s = s_sp · (t − 1), e = s_sp · t
(3)  while (s ≤ i) do
(4)    e = (e > i) ? i : e
(5)    for (s ≤ j < e) do
(6)      if (|⟨b'_i, b'_j⟩| < 2^(p/2) ‖b'_i‖ ‖b'_j‖) then
(7)        R_ij = APPROX_VALUE(⟨b_i, b_j⟩)
(9)      else
(10)       R_ij = ⟨b'_i, b'_j⟩
(11)   MUTEX_LOCK(l_1)
(12)   s = sl
(13)   sl = sl + s_sp
(14)   MUTEX_UNLOCK(l_1)
(15)   e = e + s_sp
```

Our small adjustment to the scalar product computation in [2] leads to a better utilization of all threads which results in an improvement in cases with small values for loop variable i without sacrificing the performance for larger values. The lock-protected operations including the **MUTEX_LOCK** and **MUTEX_UNLOCK** (see Lines `(11)` - `(14)`) can be replaced by an atomic fetch-and-add operation.[5]

5.2 Orthogonalization Part

The work in [2] resolved the main challenge in parallelizing the computation of the Gram-Schmidt orthogonalization and coefficients μ_{ij} by presenting a new method to compute the helper variables R_{ij} (see Line `(9)`, Algorithm 1) in parallel. A transformation of the computation of R_{ij} allows for the parallel computation of an introduced helper variable r_l (see [2] for details) which is used to compute the R_{ij}.

[4] In outlining our multi-threaded programs, we distinguish between variables that are local for every thread and variables that are shared among all threads. Local variables are highlighted by the use of a different font (e.g., `local` vs. *shared*).

[5] Thanks to Philippas Tsigas, Chalmers University for suggesting this alternative.

```
(1) for (1 ≤ j < i) do
(2)   for (1 ≤ m < j) do
(3)     R_ij = R_ij − R_im μ_jm
(4)   μ_ij = R_ij / R_jj
(5)   R_ii = R_ii − R_ij μ_ij
(6)   S_j+1 = R_ii
```

$\Longrightarrow$

```
(1) for (1 ≤ j < i) do
(2)   for (j ≤ l < i) do
(3)     r_l = r_l + R_i,j−1 μ_l,j−1
(4)   R_ij = R_ij − r_j
(5)   μ_ij = R_ij / R_jj
(6)   R_ii = R_ii − R_ij μ_ij
(7)   S_j+1 = R_ii
```

Furthermore, [2] introduces a setup utilizing one control thread and several compute threads for the parallel computation of the orthogonalization.[6] The control thread first pre-computes a part of the orthogonalization (see Lines `(4) - (13)`, Orthogonalization – Thread$_1$) in order to then enable the distribution of the computations of r_l among all threads (control and compute). The function COMPUTE_SPLIT_VALUES determines so-called split values that are used to assign a portion of the parallel computations to each one of the (control and compute) threads. Using a barrier in Line `(16)` of the control and Line `(9)` of the compute threads ensures that the pre-computations and split values have been computed. In the remainder of the control thread (Lines `(18) - (20)`) and compute threads (Lines `(11) - (13)`), the values for r_l are computed in parallel. The size of the pre-computations determines the amount of parallel work and the number of barriers that are necessary to ensure the correctness of the computation. The work in [2] discusses how one can find a balance in maximizing the parallel work while limiting the use of barriers.

As discussed previously, in experimenting with the parallel algorithm in [2], we discovered that small values for the loop variable i cause problems for the parallel computation of the orthogonalization introduced in [2]. Our analysis revealed that for small i there are few operations that need to be performed by each one of the threads which causes a significant increase in the waiting time at the necessary barriers. In contrast to [2], where all threads perform computations at all times, we modify this approach by limiting the number of threads that participate in the computation of the orthogonalization. That is, we dynamically determine the number of active threads based on the current value of i and parameter t_{skip} using ACTIVE_THREAD. In addition, we allow for a dynamic adjustment of the amount of pre-computations determining s_o in INIT_SPLIT_SIZE and UPDATE_SPLIT_SIZE.

Orthogonalization – Thread$_1$ (control)

```
(1)  t_a = ACTIVE_THREAD(i, t_skip)
(2)  s_o = INIT_SPLIT_SIZE(s_o^min, s_o^max, i)
(3)  j = 0
(4)  while (j < i) do
(5)    s = j, m = 0
(6)    while (m < s_o ∧ j < i) do
(7)      for (s ≤ l < j) do
(8)        r_j = r_j − R_il μ_jl
(9)      R_ij = R_ij − r_j
(10)     μ_ij = R_ij / R_jj
(11)     R_ii = R_ii − R_ij μ_ij
(12)     S_j+1 = R_ii
(13)     m = m + 1
(14)     j = j + 1
(15)   COMPUTE_SPLIT_VALUES_1(split, t_a, s_o)
(16)   BARRIER_WAIT(b_1)
(17)   UPDATE_SPLIT_SIZE(s_o)
(18)   for (j ≤ l < split_1) do
(19)     for (s ≤ m < j) do
(20)       r_l = r_l − R_im μ_lm
```

Orthogonalization – Thread$_t$

```
(1)  t_a = ACTIVE_THREAD(i, t_skip)
(2)  s_o = INIT_SPLIT_SIZE(s_o^min, s_o^max, i)
(3)  if (t_a ≤ t) then
(4)    e = 0
(5)    while (e < i) do
(6)      s = e, e = e + s_o
(7)      if (e > i) then
(8)        e = i
(9)      BARRIER_WAIT(b_1)
(10)     UPDATE_SPLIT_SIZE(s_o)
(11)     for (split_t ≤ l < split_t+1) do
(12)       for (s ≤ m < e) do
(13)         r_l = r_l − R_im μ_lm
```

[6] In the control/compute thread setup one thread is the dedicated control thread. The remaining threads execute the instructions intended for compute threads.

The amount of parallel computations and number of barriers that have to be used depend on the actual value of s_o. The parameters s_o^{min} and s_o^{max} define a range for the value of s_o. Our optimizations lead to a reduction in the overall overhead for small values of loop variable i.

5.3 μ-Update and Size-Reduction Part

The operations that are necessary to update the Gram-Schmidt coefficients μ_{ij} after a single size reduction step are similar to the computations performed as part of the orthogonalization. The work in [2] therefore used the respective techniques to efficiently parallelize the update computations. Specifically, [2] introduced a transformation of the size-reduction in Algorithm 1 (see Lines `(15) - (24)`) to allow the treatment of the update of the Gram-Schmidt coefficients as a separate subroutine.

μ-Update

```
(1)  for (i > j ≥ 1) do
(2)    f_j = μ_ij
(3)    if (|μ_ij| > 1/2) then
(4)      F_r = true
(5)      if (|μ_ij| > 2^(p/2)) then
(6)        F_c = true
(7)      for (1 ≤ m ≤ j) do
(8)        μ_im = μ_im − ⌈μ_ij⌋ μ_jm
```

Size-Reduction

```
(10) for (i > j ≥ 1) do
(11)   if (|f_j| > 1/2) then
(12)     b_i = b_i − ⌈f_j⌋ b_j
(13) od
(14) if (F_r = true) then
(15)   APPROX_VECTOR(b'_i, b_i)
```

The work in [2] implemented a control and compute threads setup similar to the one for the orthogonalization. Correspondingly, our improvements developed for the parallel orthogonalization computation (see Section 5.2) can easily be adapted to also improve the parallel update of the μ_{ij} values. The parameters s_μ^{min} and s_μ^{max} define a range for the size s_μ of the pre-computations. The newly optimized parallel μ-Update looks as follows:

μ-Update – Thread$_1$ (control)

```
(1)  t_a = ACTIVE_THREAD(i, t_skip)
(2)  s_μ = INIT_SPLIT_SIZE(s_μ^min, s_μ^max, i)
(3)  j = i − 1, j_e = 0
(4)  while (j ≥ 1) do
(5)    j_s = j, m = 0
(6)    while (m < s_μ ∧ j ≥ j_e) do
(7)      for (j_s ≥ l > j) do
(8)        μ_ij = μ_ij − ⌈f_l⌋ μ_lj
(9)      if (|μ_ij| > 1/2) then
(10)       f_j = μ_ij
(11)       F_r = true
(12)       if (|μ_ij| > 2^(p/2)) then
(13)         F_c = true
(14)       μ_ij = μ_ij − ⌈μ_ij⌋
(15)       m = m + 1
(16)     else
(17)       f_j = 0
(18)     j = j − 1
(19)   j_c = j
(20)   COMPUTE_SPLIT_VALUES_2(split, t_a, s_μ)
(21)   BARRIER_WAIT(b_2)
(22)   UPDATE_SPLIT_SIZE(s_μ)
(23)   j_e = split_T
(24)   for (j_s ≥ m > j) do
(25)     if (f_m ≠ 0) then
(26)       for (split_T ≤ l < split_{T+1}) do
(27)         μ_il = μ_il − ⌈f_m⌋ μ_ml
```

μ-Update – Thread$_t$

```
(1)  t_a = ACTIVE_THREAD(i, t_skip)
(2)  s_μ = INIT_SPLIT_SIZE(s_μ^min, s_μ^max, i)
(3)  if (t_a ≤ t) then
(4)    j = i − 1
(5)    while (j ≥ 1) do
(6)      BARRIER_WAIT(b_2)
(7)      UPDATE_SPLIT_SIZE(s_μ)
(8)      j = j_c
(9)      for (j_s ≥ m > j_c) do
(10)       if (f_m ≠ 0) then
(11)         for (split_t ≤ l < split_{t+1}) do
(12)           μ_il = μ_il − ⌈f_m⌋ μ_ml
```

Aside from the μ-Update, the size-reduction consists only of vector operations. The parallel aSE-LLL algorithm in [2] divided the vectors into a number of slices of similar size. The number of slices in [2] equals the number of threads used. The amount of work for each one of these vector slices therefore depends on the bit length of the vector entries that occur in the computation.

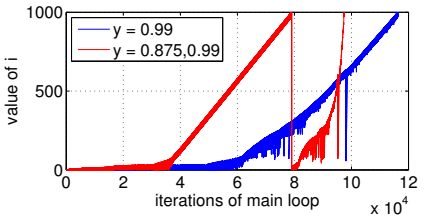


Fig. 5. Value of i for knapsack type lattice basis

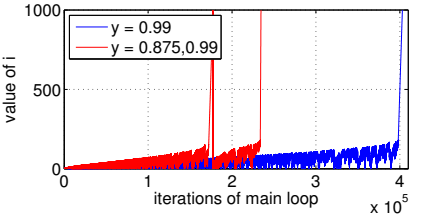


Fig. 6. Value of i for a SVP challenge type lattice basis

Our analysis showed that an optimal balance cannot be reached by that technique in cases where the size of the entries is not balanced throughout the lattice basis. This potential imbalance caused by the size of vector entries is similar to the effect that the possible computation of the exact scalar products has on the balance of the overall parallel scalar product computation.[7] We therefore use a similar approach as in the case of the computation of scalar products in that we choose the size of the vector slices based on the value of the main while-loop variable i and the parameter max_{sr}. The modified size-reduction looks as follows:

Size-Reduction $-$ **Thread**$_t$

```
(1)  if (F_r = true) then
(2)    s_sr = INIT_SPLIT_SIZE(i, max_sr)
(3)    v_s = INIT_VECT_START(t)
(4)    v_e = v_s + s_sr
(5)    while (v_s < n) do
(6)      for (i − 1 ≥ j ≥ 1) do
(7)        if (|f_j| > 1/2) then
(8)          for (v_s ≤ l ≤ v_e) do
(9)            b_il = b_il − ⌈f_j⌋ b_jl
(10)         for (v_s ≤ l ≤ v_e) do
(11)           b'_il = APPROX_VALUE(b_il)
(12)       MUTEX_LOCK(l_1)
(13)       v_s = sv
(14)       sv = sv + s_sr
(15)       MUTEX_UNLOCK(l_1)
(16)       v_e = v_e + s_sr
(17)   od
(18)   F_r = false
```

5.4 Using Sequences of Reduction Parameters

The performance and the quality of the LLL algorithm can be adjusted by means of the reduction parameter y. While this parameter may be chosen from the interval $(\frac{1}{4}, 1)$, most applications use the value $y = 0.99$. In [1], the use of sequences of reduction parameters instead of a single reduction parameter was evaluated for the aSE-LLL algorithm. For the majority of test cases using the sequence $0.75, 0.875, 0.99$ instead of $y = 0.99$ did not result in a significant performance improvement.

For the parallel aSE-LLL, the number of iterations and the value of the loop variable of the main while-loop is of greater importance than in the sequential algorithm. We therefore revisited the heuristic of using a sequence of reduction parameters for the newly improved parallel aSE-LLL. Figures 5 and 6 show the value of the loop variable i for every iteration of the main while-loop for reduction parameter $y = 0.99$ in comparison to the sequence of reduction parameters $0.875, 0.99$. For SVP challenge type lattice bases (see Figure 6), the value of the loop variable i in case of using a sequence of reduction parameters is generally larger than in case of using a single reduction parameter $y = 0.99$. As motivated

[7] In practice, the introduced imbalance is minor compared to the effect of exact scalar products.

earlier, larger values for the loop variable i are beneficial for the parallel aSE-LLL algorithm as they translate into more work that needs to be performed in each iteration of the loop which in turn translates into an increased amount of work that can be parallelized.

It is important to note, that the use of a sequence of reduction parameters results in a significant decrease in the number of iterations. However, this decrease in the total number of iterations does not translate into a significant performance improvement in the sequential algorithm [1]. The amount of work performed within a single iteration of the main while-loop is therefore on average larger than in case of using a single reduction parameter.

For knapsack type lattice bases (see Figure 5), the decrease in the number of iterations is not as significant as in the case of SVP challenge type lattice bases. Therefore, it is to be expected that this heuristic has less of an impact on improving the speed-up for knapsack type than for SVP challenge type lattice bases.

6 Experiments

6.1 Setup

The experiments in this paper were performed using three different types of lattice bases in order to show that our newly improved parallel aSE-LLL reduction algorithm outperforms the parallel algorithm introduced in [2]. Specifically, we evaluated the performance of our newly developed algorithm and the algorithm introduced in [2] for knapsack type lattice bases as well as for two new types of lattice bases, SVP challenge type[8] and cyclic type lattice bases [3].

The experiments were performed on Sun X4150 servers with two quad-core Intel Xeon processors (2.83 GHz) and 8 GB of main memory running Debian Linux. It is important to note, that these CPUs do not have a fast interconnect nor do they have an integrated memory controller.[9] We compiled all programs with GCC 4.4.5 using the same optimization flags. For our implementation of the sequential and parallel aSE-LLL algorithms we used GMP 5.0.1 as long integer arithmetic and MPRF 3.0.0 as multi-precision floating-point arithmetic (for the approximation of the lattice basis). The experiments were performed using MPFR with bit precision $p = 128$ for knapsack and cyclic type lattice bases and $p = 256$ for SVP challenge type lattice bases. Based on this setup, we conducted experiments with the advanced sequential Schnorr-Euchner algorithm (Algorithm 1) as well as the 4-thread and 8-thread versions of the parallel aSE-LLL in [2] and our newly improved implementation of the parallel aSE-LLL algorithm. Based on our hardware setup, the following set of parameters (for balancing the parallel computations) was chosen for the experiments with all types of lattice bases: $sp_{max} = 20$, $s_o^{min} = s_\mu^{min} = 20$, $s_o^{max} = s_\mu^{max} = 60$, $t_{skip} = 40$, and $max_{sr} = 32$ (and accordingly for the parallel aSE-LLL in [2]).

[8] Refer to [14] for details on SVP challenge lattice bases from which the SVP challenge type lattice bases are derived.

[9] The absolute speed-ups achieved by this type of CPU are therefore lower than the speed-ups achieved in [2].

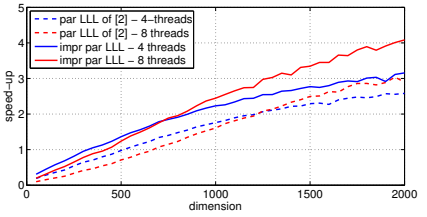


Fig. 7. Speed-up of parallel aSE-LLL for SVP challenge type lattice bases

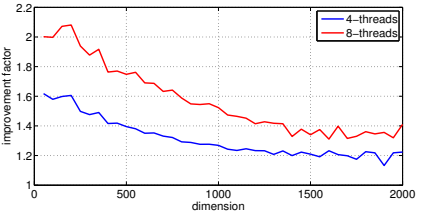


Fig. 8. Improvement factor for SVP challenge type lattice bases

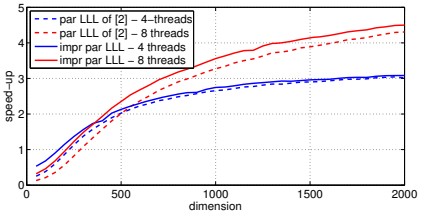


Fig. 9. Speed-up of parallel aSE-LLL for knapsack type lattice bases

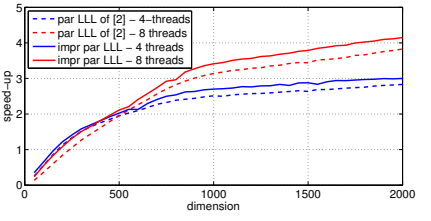


Fig. 10. Speed-up of parallel aSE-LLL for cyclic type lattice bases

6.2 Results

Figure 7 shows the speed-up that the parallel aSE-LLL in [2] and our newly improved parallel aSE-LLL achieve for SVP challenge type lattice bases. As discussed earlier (Section 4), the parallel aSE-LLL in [2] performs poorly on SVP challenge type lattice bases. The modifications introduced in Section 5 (as part of our new parallel aSE-LLL) result in a significant increase in the speed-up.

Figure 8 shows the improvement factor that can be achieved by the new parallel aSE-LLL for SVP challenge type lattice bases. The improvement factor is defined as the quotient of the speed-up for the new improved parallel aSE-LLL and the parallel aSE-LLL of [2]. The improvement factor is larger for smaller dimensions than for larger dimensions, as the modifications were geared towards improving the performance for small values of the loop variable i. The 4-thread version of the algorithm achieves an improvement factor of more than 1.2 and the 8-thread version achieves an improvement factor of more than 1.35.

For knapsack type lattice bases (see Figure 9) and for cyclic type lattice bases (see Figure 10) the parallel aSE-LLL of [2] performs better than for the SVP challenge type lattice bases. However, even for these types of lattice bases our new parallel aSE-LLL algorithm results in an increase for the speed-up. For example, it achieves an improvement factor of close to 1.1 for the speed-up when reducing cyclic type lattice bases.

7 Future Work

Future work includes developing heuristics to automatically determine the parameters which are responsible for the balancing of the parallel computations

(depending both on the hardware used and the type of lattice basis to be reduced). Another line of work is to explore new methods for aSE-LLL reduction that can efficiently utilize a many-core system.

Acknowledgment. This work was supported by NSF Award DUE 1027452.

References

1. Backes, W., Wetzel, S.: Heuristics on Lattice Basis Reduction in Practice. ACM Journal on Experimental Algorithms, 7 (2002)
2. Backes, W., Wetzel, S.: Parallel Lattice Basis Reduction Using a Multi-threaded Schnorr-Euchner LLL Algorithm. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 960–973. Springer, Heidelberg (2009)
3. Coster, M., Joux, A., LaMacchia, B., Odlyzko, A., Schnorr, C., Stern, J.: Improved Low-Density Subset Sum Algorithm. Journal of Computational Complexity 2, 111–128 (1992)
4. Gentry, C.: Toward Basing Fully Homomorphic Encryption on Worst-Case Hardness. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 116–137. Springer, Heidelberg (2010)
5. Goldstein, A., Mayer, A.: On the Equidistribution of Hecke Points. Forum Mathematicum 15, 165–189 (2003)
6. Heckler, C., Thiele, L.: A Parallel Lattice Basis Reduction for Mesh-Connected Processor Arrays and Parallel Complexity. In: Proceedings of Symposium on Parallel and Distributed Processing (SPDP 1993), pp. 400–407. IEEE, Los Alamitos (1993)
7. Joux, A.: A Fast Parallel Lattice Basis Reduction Algorithm. In: Proceedings of the Second Gauss Symposium, pp. 1–15. deGruyter, Berlag (1993)
8. Lenstra, A., Lenstra, H., Lovász, L.: Factoring Polynomials with Rational Coefficients. Math. Ann. 261, 515–534 (1982)
9. Nguyen, P., Stehlé, D.: Floating-Point LLL Revisited. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 215–233. Springer, Heidelberg (2005)
10. Nguyen, P., Stehlé, D.: An LLL Algorithm with Quadratic Complexity. SIAM J. Comput. 39(3), 874–903 (2009)
11. Peikert, C.: Public-key Cryptosystems from the Worst-Case Shortest Vector Problem (Extended Abstract). In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC 2009), pp. 333–342. ACM, New York (2009)
12. Regev, O.: Lattice-based Cryptography. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 131–141. Springer, Heidelberg (2006)
13. Schnorr, C., Euchner, M.: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In: Budach, L. (ed.) FCT 1991. LNCS, vol. 529, pp. 68–85. Springer, Heidelberg (1991)
14. SVP Challenge TU Darmstadt (July 2011), http://www.latticechallenge.org/svp-challenge/
15. NTL - Homepage (July 2011), http://www.shoup.net/ntl/

Distributed Mining of Constrained Frequent Sets from Uncertain Data

Alfredo Cuzzocrea[1] and Carson K. Leung[2]

[1] ICAR-CNR and University of Calabria, Italy
cuzzocrea@si.deis.unical.it
[2] Department of Computer Science, University of Manitoba, Canada
kleung@cs.umanitoba.ca

Abstract. With the advance in technology, sensor networks have been widely used in many application areas such as environmental surveillance. Sensors distributed in these networks serve as good sources for data. This calls for *distributed data mining*, which searches for implicit, previously unknown, and potentially useful patterns that might be embedded in the distributed data. Many existing distributed data mining algorithms do not allow users to express the patterns to be mined according to their intention via the use of constraints. Consequently, these unconstrained mining algorithms can yield numerous patterns that are not interesting to users. Moreover, due to inherited measurement inaccuracies and/or network latencies, the data are often riddled with uncertainty. These call for *constrained mining* and *uncertain data mining*. In this paper, we propose a tree-based system for mining frequent sets that satisfy user-defined constraints from a distributed environment such as a wireless sensor network of uncertain data. Experimental results show effectiveness of our proposed system.

1 Introduction

Data mining searches for implicit, previously unknown, and potentially useful information that is embedded in data. As a common data mining task, *frequent set mining* looks for sets of items (also known as *itemsets*) that are frequently co-occurring together. The mined frequent sets can be used in the discovery of correlation or casual relations, analysis of sequences, and formation of association rules. Since its introduction [4], the research problem of finding frequent sets has been the subject of numerous studies.

Most algorithms in the early days were Apriori-based [3], which depends on a generate-and-test paradigm to find all frequent sets by first generating candidates and then checking their support (i.e., their occurrences) against the traditional databases containing precise data. To avoid the generate-and-test paradigm, the FP-growth algorithm [12] was proposed. Such a tree-based algorithm constructs an extended prefix-tree structure, called Frequent Pattern tree (FP-tree), to capture the contents of the transaction database. Rather than employing the generate-and-test strategy of Apriori-based algorithms, FP-growth focuses on

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 40–53, 2011.

frequent pattern growth—which is a restricted test-only approach (i.e., does not generate candidates, and only tests for support).

In many real-life applications, data are riddled with uncertainty. It is partially due to inherent measurement inaccuracies, sampling and duration errors, network latencies, and intentional blurring of data to preserve anonymity. As such, the presence or absence of items in a dataset is uncertain. Hence, mining uncertain data [1,5,23] is in demand. For example, a physician may highly suspect (but not guarantee) that a patient suffers from asthma. The uncertainty of such suspicion can be expressed in terms of *existential probability* of an item in a probabilistic dataset. To mine frequent sets from these uncertain data, the U-Apriori [7] and UF-growth [17] algorithms were proposed.

Many frequent set mining algorithms, regardless whether they are Apriori-based or tree-based, provide little or no support for user focus when mining precise or uncertain data. However, in many real-life applications, the user may have some particular phenomena in mind on which to focus the mining (e.g., medical analysts may want to find only those lab test records belonging to patients suspected to suffer from asthma instead of all the patients). Without user focus, the user often needs to wait for a long period of time for numerous frequent sets, out of which only a tiny fraction may be interesting to the user. Hence, *constrained frequent set mining* [11,14,15,18], which aims to find those frequent sets that satisfy the user-specified constraints, is needed. CAP [19], DCF [13], and $\mathcal{FIC}$ [20] are examples of algorithms that mine constrained frequent set mining from traditional precise data.

With advances in technology, one can easily collect large amounts of data from not only a single source but multiple sources. For example, in recent years, sensor networks have been widely used in many application areas such as agricultural, architectural, environmental, and structural surveillance. Sensors distributed in these networks serve as good sources of data. However, sensors usually have limited communication bandwidth, transmission energy, and computational power. Thus, data are not usually transmitted to a single distant centralized processor to perform the data mining task. Instead, data are transmitted to their local (e.g., closest) processors within a distributed environment. This calls for *distributed mining* [2,6,8,21,22]—which searches for implicit, previously unknown, and potentially useful frequent sets that might be embedded in the distributed data.

Count Distribution, Data Distribution and Candidate Distribution [2], as well as FDM [6] are some examples of Apriori-based distributed algorithms that find frequent sets in a distributed environment. Parallel-HFP-Leap [10] is a tree-based algorithm for mining in a distributed environment. However, they all do not handle constraints nor do they mine uncertain data. On the other hand, CAP, DCF and $\mathcal{FIC}$ all find *constrained* frequent sets, but they mine a centralized database of precise data. Similarly, the U-Apriori and UF-growth algorithms both mine a centralized database of *uncertain* data for all (unconstrained) frequent sets instead of only those constrained ones. In other words, these existing frequent set mining algorithms fall short in different aspects. Hence, a natural question

to ask is: Is it possible to mine *uncertain* data for only those frequent sets that satisfy user *constraints* in a *distributed* environment? In response to this question, we propose in this paper a tree-based system for mining uncertain data in a distributed environment for frequent sets that satisfy user-specified constraints. Here, our *key contribution* is the non-trivial integration of (i) constrained mining, (ii) distributed mining, (iii) uncertain data mining, with (iv) tree-based frequent set mining. The resulting tree-based system efficiently mines from distributed uncertain data for only those constrained frequent sets. It avoids the candidate generate-and-test paradigm, handles with uncertain data, pushes user constraints inside the mining process, avoids unnecessary computation, and finds only those sets satisfying the constraints in a distributed environment.

This paper is organized as follows. The next section gives some background information on constrained frequent set mining from uncertain data. In Section 3, we propose our non-trivial integration of tree-based frequent set mining, constrained mining, distributed mining, together with uncertain mining. Experimental results are shown in Section 4. Finally, Section 5 presents the conclusions.

2 Background

2.1 Mining Frequent Sets from Uncertain Data

When the problem of frequent set mining was first introduced [4], the corresponding algorithm—namely, Apriori—mined all frequent itemsets from a transaction database (TDB) consisting of *precise data*, in which the contents of each transaction are precisely known. Specifically, if a transaction t_i contains an item x (i.e., $x \in t_i$), then x is precisely known to be present in t_i. On the other hand, if a transaction t_i does not contain an item y (i.e., $y \notin t_i$), then y is precisely known to be absent from t_i. However, this is not the case for probabilistic databases consisting of uncertain data. A key difference between precise and uncertain data is that each transaction of the latter contains items and their *existential probabilities*. The existential probability $P(x, t_i)$ of an item x in a transaction t_i indicates the likelihood of x being present in t_i. For a real-life example, each transaction t_i represents a patient's visit to a physician's office. Each item x within t_i represents a potential disease, and is associated with $P(x, t_i)$ expressing the likelihood of a patient having that disease x in t_i (say, in t_1, the patient has a 60% likelihood of having asthma, and a 90% likelihood of catching a cold regardless of having asthma or not). With this notion, each item in a transaction t_i in datasets of precise data can be viewed as an item with a 100% likelihood of being present in t_i.

Given an item x and a transaction t_i, there are two possible worlds when using the *"possible world" interpretation* of uncertain data [7,9,16,17]: (i) the possible world W_1 where $x \in t_i$ and (ii) the possible world W_2 where $x \notin t_i$. Although it is uncertain which of these two worlds is the *true world*, the probability of W_1 being the true world is $P(x, t_i)$ and that of W_2 is $1 - P(x, t_i)$.

In real-life applications, there are generally many independent items in each of the n transactions in a TDB (where $|\text{TDB}| = n$). Hence, the *expected support*

of an itemset X in the TDB can be computed by summing the support of X in possible world W_j (while taking in account the probability of W_j to be the true world) over all possible worlds. Such a sum can be simplified to become the sum of product of existence probabilities of $x \in X$, as follows [9]:

$$expSup(X) = \sum_{i=1}^{n} \left(\prod_{x \in X} P(x, t_i) \right). \quad (1)$$

An itemset X is *frequent* if its expected support meets or exceeds the user-specified threshold *minsup*.

2.2 Mining Frequent Sets That Satisfy User Constraints

An existing constrained frequent set mining framework [13,19] allows the user to use a rich set of SQL-style constraints to specify his interest for guiding the mining process so that only those frequently occurring sets of market basket items that satisfy the user constraints are found. This avoids unnecessary computation for mining those uninteresting frequent sets. Besides market basket items, the set of constraints can also be imposed on items, events or objects in other domains. The following are some examples of user constraints. Constraint $C_1 \equiv max(X.Price) \leq \$25$ expresses the user interest in finding every frequent set X such that the maximum price of all market basket items in each X is at most \$25. Similarly, $C_2 \equiv min(X.Price) \leq \$30$ says that the minimum price of all items in an itemset X is at most \$30. For domains other than the market basket, constraint $C_3 \equiv X.Location = Canada$ expresses the user interest in finding every frequent set X such that all events in X are held in Canada; $C_4 \equiv X.Symptom \supseteq \{dry\ throat,\ sneezing\}$ says that each individual in X suffers from at least dry throat and sneezing; $C_5 \equiv X.Weight \geq 32$kg says that the weight of each object in X is at least 32kg; and $C_6 \equiv avg(X.Rainfall) \leq 10$mm says that the average rainfall of all meteorological records in X is at most 10mm.

The above constraints can be categorized into several overlapping classes according to the properties that they possess. One of these properties is *succinctness*. A constraint is *succinct* if one can directly generate precisely all and only those itemsets satisfying the constraints without generating and excluding itemsets not satisfying the constraints. In the above examples, all constraints except C_6 are *succinct* because one can use member generating functions [13,19] to precisely generate constrained itemsets. For example, itemsets satisfying $C_1 \equiv max(X.Price) \leq \$25$ can be precisely generated by combining any market basket items having price $\leq \$25$, thereby avoiding the substantial overhead of the generation and exclusion of invalid itemsets. Similarly, itemsets satisfying $C_2 \equiv min(X.Price) \leq \$30$ can be precisely generated by combining at least one market basket item having price $\leq \$30$ with some optional items (of any price values). It is important to note that (i) *a majority of constraints are succinct* and (ii) many non-succinct constraints can be induced into weaker constraints that are succinct (e.g., non-succinct constraint $C_6 \equiv avg(X.Rainfall) \leq 10$mm can

be induced into a succinct constraint $C'_6 \equiv min(X.Rainfall) \leq 10$mm as all frequent sets satisfying C_6 must satisfy C'_6).

Besides succinctness, there are some other properties. One of them is *anti-monotonicity*. A constraint is *anti-monotone* if all subsets of an itemset satisfying the constraint also satisfy the constraint. With this additional property, succinct constraints can be further divided into two subclasses:

- *succinct anti-monotone (SAM) constraints*, and
- *succinct non-anti-monotone (SUC) constraints.*

Among the five succinct constraints C_1–C_5 above, C_1, C_3 and C_5 are SAM constraints. Note that supersets of any itemset violating the SAM constraints also violate the constraints (e.g., if an itemset X contains an item having price > \$25, then X violates C_1 and so does every superset of X). In contrast, C_2 and C_4 are *SUC constraints* because they do not possess such an anti-monotonicity property. For instance, if the minimum price of all items contained within X is higher than \$30, then X violates C_2 but there is no guarantee that all supersets of X would violate C_2. As an example, let *y.Price* be \$50 and *z.Price* be \$10. Then, $X \cup \{y\}$ and $X \cup \{z\}$ are both supersets of X. Among them, the former still violates C_2 but the latter satisfies C_2.

3 Our Proposed Distributed Mining System

Without loss of generality, we assume to have p sites/processors and $m = m_1 + m_2 + ... + m_p$ sensors in a distibuted network such that m_1 wireless sensors transmit data to their closest or designated site/processor P_1, m_2 sensors transmit data to the site/processor P_2, and so on. With this setting, we show in this section how our proposed system finds (i) constrained sets that are locally frequent w.r.t. site/processor P_i and (ii) those that are globally frequent w.r.t. all sites/processors in the entire wireless sensor network.

3.1 Finding Constrained Locally Frequent Sets

Given m_i sensors transmitting data to the processor P_i, a local database TDB_i of uncertain data can be created for P_i. We aim to find sets that are both (i) frequent to P_i and (ii) satisfying a succinct (SAM or SUC) constraint C. For uncertain data, we use the "possible world" interpretation of uncertain data. We find constrained locally frequent itemsets from uncertain data in the following steps.

Identification of Items Satisfying the Constraints. Let $\texttt{Item}^{\texttt{M}}$ be the collection of mandatory items—i.e., the collection of domain items that individually satisfy the SAM or SUC constraint C; let $\texttt{Item}^{\texttt{O}}$ be the collection of optional items—i.e., the collection of domain items that individually violate C.

Then, for any SAM constraint C_{SAM}, an itemset X satisfying C_{SAM} cannot contain any item from $\texttt{Item}^{\texttt{O}}$ due to the anti-monotonicity property. So, any

itemset X satisfying C_{SAM} must consist of *only* items that individually satisfy C_{SAM}. In other words, any itemset X satisfying C_{SAM} must be generated by combining items from $\texttt{Item}^{\texttt{M}}$ (i.e., $X \subseteq \texttt{Item}^{\texttt{M}}$). Due to the succinctness property, items in $\texttt{Item}^{\texttt{M}}$ can be efficiently enumerated (from the list of domain items) by selecting only those items that individually satisfy C_{SAM}. See the following example.

Example 1. *Consider the following transaction database TDB consisting of uncertain data:*

Transactions	*Contents*
t_1	{*a:0.7, b:0.8, d:1.0, e:0.1, f:0.4*}
t_2	{*a:0.7, b:0.8, c:0.8, d:1.0, e:0.2*}
t_3	{*a:0.8, c:0.5, e:0.3, f:0.4*}
t_4	{*b:0.8, c:0.8, d:1.0*}
t_5	{*c:0.8, d:1.0*}

with the following auxiliary information:

Items	*a*	*b*	*c*	*d*	*e*	*f*
Price	*$10*	*$20*	*$100*	*$50*	*$75*	*$25*

In the above TDB of uncertain data, each transaction contains items and their corresponding existential probabilities. For example, there are five domain items a, b, d, e and f in the first transaction t_1, where the existential probabilities of these items are 0.7, 0.8, 1.0, 0.1 and 0.4 respectively. Note that (i) different items may have the same existential probabilities (e.g., the existential probabilities of items b and c in t_2 are the same—with a value of 0.8) but (ii) the existential probabilities of the same item may vary from one transaction to another (e.g., the existential probability of item a in transaction t_2 is 0.7 whereas that in t_3 is 0.8). Let constraint C_{SAM} be the SAM constraint $C_1 \equiv max(X.Price) \leq \$25$. Our proposed system checks each of the six domain items against the constraint C_{SAM}. It first enumerates the valid items a, b and f (i.e., items with individual price $\leq \$25$). So, $\texttt{Item}^{\texttt{M}} = \{a, b, f\}$. Once we identified the domain items that satisfy the succinct anti-monotone constraint C_{SAM}, these items would serve as building blocks of all constrained frequent itemsets satisfying C_{SAM} because all constrained frequent itemsets must comprise only those $\texttt{Item}^{\texttt{M}}$ items.

Next, for any SUC constraint C_{SUC}, any itemset X satisfying C_{SUC} is composed of mandatory items (i.e., items that individually satisfy C_{SUC}) and possibly some optional items (regardless whether or not they satisfy C_{SUC}). Note that, although C_{SUC} possesses the succinctness property (i.e., one can easily enumerate all and only those itemsets that are guaranteed to satisfy C_{SUC}), it does not possess the anti-monotonicity property. So, if an itemset violates C_{SUC}, there is no guarantee that all or any of its supersets would violate C_{SUC}. Hence, not all itemsets satisfying C_{SUC} are composed of only domain items that individually satisfy the constraints (as for SAM constraints). Instead, any itemset X satisfying C_{SUC} must be generated by combining at least one $\texttt{Item}^{\texttt{M}}$ item and

possibly some $\texttt{Item}^{\texttt{O}}$ items. Due to succinctness, items in $\texttt{Item}^{\texttt{M}}$ and in $\texttt{Item}^{\texttt{O}}$ can be efficiently enumerated. See the following example.

Example 2. *Consider the same TDB and auxiliary information in Example 1. Let constraint C_{SUC} be the SUC constraint $C_2 \equiv min(X.Price) \leq \$30$. Our proposed system checks each of the six domain items against C_{SUC}. It first enumerates the valid items a, b and f (i.e., items with individual price $\leq \$30$), giving $\texttt{Item}^{\texttt{M}} = \{a, b, f\}$. The remaining domain items then belong to $\texttt{Item}^{\texttt{O}}$ (i.e., items with individual price $> \$30$). Once we classified the domain items into (i) the $\texttt{Item}^{\texttt{M}}$ items (which satisfy C_{SUC}) and (ii) the $\texttt{Item}^{\texttt{O}}$ items (which violate C_{SUC}), all these items serve as building blocks of all constrained frequent itemsets satisfying C_{SUC} because all constrained frequent itemsets must comprise at least one $\texttt{Item}^{\texttt{M}}$ item and may contain some additional $\texttt{Item}^{\texttt{M}}$ or $\texttt{Item}^{\texttt{O}}$ items.*

Construction of an UF-tree. Once the domain items are classified into $\texttt{Item}^{\texttt{M}}$ and $\texttt{Item}^{\texttt{O}}$ items (no $\texttt{Item}^{\texttt{O}}$ items for C_{SAM}), our system then constructs an UF-tree, which is built in preparation for mining constrained frequent sets from uncertain data. It does so by first scanning the TDB of uncertain data once. It accumulates the expected support of each of the items in order to find all *frequent* domain items. Among these items, the system discards those infrequent ones and only captures those frequent ones in the UF-tree. Note that any infrequent $\texttt{Item}^{\texttt{M}}$ or $\texttt{Item}^{\texttt{O}}$ items can be safely discarded because any itemset containing an infrequent item is also infrequent.

Once the frequent $\texttt{Item}^{\texttt{M}}$ and $\texttt{Item}^{\texttt{O}}$ items are found, our system arranges these two kinds of items in such a way that $\texttt{Item}^{\texttt{M}}$ items appear *below* $\texttt{Item}^{\texttt{O}}$ items (i.e., $\texttt{Item}^{\texttt{M}}$ items are closer to the leaves, and $\texttt{Item}^{\texttt{O}}$ items are closer to the root). Among all the items in $\texttt{Item}^{\texttt{M}}$, they are sorted in non-ascending order of accumulated expected support. Similarly, among all the items in $\texttt{Item}^{\texttt{O}}$, they are also sorted in non-ascending order of accumulated expected support. The system then scans the TDB the second time and inserts each transaction of the TDB into the UF-tree. Here, the new transaction is merged with a child (or descendant) node of the root of the UF-tree (at the highest support level) only if the same item *and the same expected support* exist in both the transaction and the child (or descendant) nodes.

For SAM constraints, the corresponding UF-tree captures only those frequent $\texttt{Item}^{\texttt{M}}$ items; for SUC constraints, the corresponding UF-tree captures both the frequent $\texttt{Item}^{\texttt{M}}$ items and the frequent $\texttt{Item}^{\texttt{O}}$ items. With such a tree construction process, the UF-tree possesses the property that *the occurrence count of a node is at least the sum of occurrence counts of all its children nodes.* See Example 3.

Example 3. *Let us revisit Examples 1 and 2, and let the user-specified support threshold minsup be set to 1.0. Our system builds the UF-tree that captures the frequent items satisfying the SUC constraint C_2 as follows. First, the system scans the TDB once and accumulates the expected support of each $\texttt{Item}^{\texttt{M}}$ item as well as each $\texttt{Item}^{\texttt{O}}$ item. Hence, it finds all frequent $\texttt{Item}^{\texttt{M}}$ items and sorts them in descending order of (accumulated) expected support. It also finds all frequent*

$\texttt{Item}^{\texttt{O}}$ *items and sorts them in descending order of (accumulated) expected support. Among the two kinds of items,* $\texttt{Item}^{\texttt{O}}$ *are arranged on top (near the root) of* $\texttt{Item}^{\texttt{M}}$ *items (which are near the leaves). Specifically, our system obtains* $\texttt{Item}^{\texttt{O}}$ *items* d, c *and* e *(with their corresponding accumulated expected support values of 4.0, 2.9 and 0.6), which are sorted in descending order of their expected support values. Among these* $\texttt{Item}^{\texttt{O}}$ *items,* e *(having accumulated expected support of 0.6* $<$ *minsup) is removed. Then, the system represents the frequent* $\texttt{Item}^{\texttt{O}}$ *items and their expected support as d:4.0 and c:2.9. The expected support of each of these frequent* $\texttt{Item}^{\texttt{O}}$ *items* $\geq$ *minsup. Similarly, the system also obtains* $\texttt{Item}^{\texttt{M}}$ *items* b, a *and* f *(with their corresponding accumulated expected support values of 2.4, 2.2 and 0.8), which are also sorted in descending order of their expected support values. Among these* $\texttt{Item}^{\texttt{M}}$ *items,* f *(having accumulated expected support of 0.8* $<$ *minsup) is removed. Then, the system represents the frequent* $\texttt{Item}^{\texttt{M}}$ *items and their expected support as b:2.4 and a:2.2. The expected support of each of these frequent* $\texttt{Item}^{\texttt{M}}$ *items* $\geq$ *minsup.*

Then, our system scans the TDB the second time and inserts each transaction into the UF-tree. The system first inserts frequent items from the first transaction t_1 *into the tree. It then inserts the frequent items from the second transaction* t_2 *into the UF-tree. Since the expected support of* a *and* b *in* t_2 *is the same as those in an existing branch (i.e., the branch for* t_1*), this node can be shared. So, the system increments the occurrence count for the tree nodes (a:0.7) and (b:0.8) to 2, and adds the remainder of* t_2*—namely,* $\langle$*(c:0.8):1, (d:1.0), (e:0.2):1*$\rangle$*—as a child of the node (b:0.8):2. Afterwards, our system inserts the frequent items from the remaining transactions. At the end of the tree construction process, we get the UF-tree shown in Figure 1(a) capturing the contents of the TDB of uncertain data.*

Mining of Constrained Frequent Itemsets from the UF-tree. Once the UF-tree is constructed with the item-ordering scheme where $\texttt{Item}^{\texttt{O}}$ items are above $\texttt{Item}^{\texttt{M}}$ items, our proposed system extracts appropriate paths to form a projected database for each $x \in \texttt{Item}^{\texttt{M}}$. The system does not need to form projected databases for any $y \in \texttt{Item}^{\texttt{O}}$ because all itemsets satisfying C_{SUC} must be "extensions" of an item from $\texttt{Item}^{\texttt{M}}$ (i.e., all valid itemsets must be grown from $\texttt{Item}^{\texttt{M}}$ items) and no $\texttt{Item}^{\texttt{O}}$ items are kept in the UF-tree for C_{SAM}.

When forming each $\{x\}$-projected database and constructing its UF-tree, our system does not need to distinguish those $\texttt{Item}^{\texttt{M}}$ items from $\texttt{Item}^{\texttt{O}}$ items in the UF-tree for $\{x\}$-projected database. Such a distinction between two kinds of items is only needed for the UF-tree for the TDB (for SUC constraints only) but not projected UF-trees once we found at least one valid item $x \in \texttt{Item}^{\texttt{M}}$ because for any v satisfying C_{SUC},

$$v = \{x\} \cup Others$$
$$\text{where (i) } x \in \texttt{Item}^{\texttt{M}}$$
$$\text{(ii) } Others \subseteq (\texttt{Item}^{\texttt{M}} \cup \texttt{Item}^{\texttt{O}} - \{x\}).$$

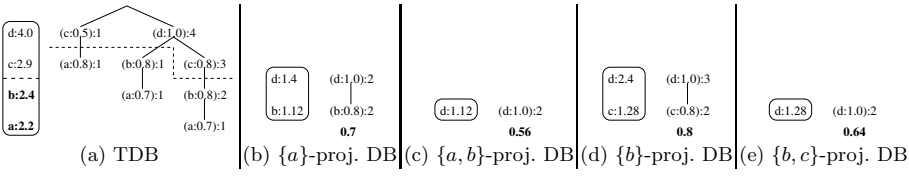


Fig. 1. The UF-trees used in our proposed system

After constructing these projected UF-tree for each $x \in \texttt{Item}^{\texttt{M}}$, our proposed system mines all frequent sets that satisfy C_{SUC} in the same manner as it mines those satisfying C_{SAM}. See Example 4.

Example 4. *Once again, we revisit Example 3. Once the UF-tree is constructed, our proposed system recursively mines constrained locally frequent itemsets from this tree with minsup = 1.0 as follows. From the header table (from top to bottom) containing two* $\texttt{Item}^{\texttt{O}}$ *items d:4.0 = (4 ×1.0) and c:2.9 = (1 ×0.5) + (3 ×0.8) as well as two* $\texttt{Item}^{\texttt{M}}$ *items b:2.4 = (1 ×0.8) + (2 ×0.8) and a:2.2 = (1 ×0.8) + (1 ×0.7) + (1 ×0.7), the system first finds two constrained frequent sets* $\{b\}$ *and* $\{a\}$ *with expected support values of 2.4 and 2.2 respectively.*

Then, our system recursively mines constrained frequent sets from this UF-tree with minsup = 1.0 as follows. From the UF-tree shown in Figure 1(a), our system starts with $a \in \texttt{Item}^{\texttt{M}}$ *and constructs an UF-tree for the* $\{a\}$*-projected database. The resulting tree, as shown in Figure 1(b), consists of a single path—namely,* $\langle$*(d:1.0):2, (b:0.8):2*$\rangle$ *with the expected support of* $\{a\}$ *equal to* **0.7** *(implying that d or b occurs together with a twice in the original database). The expected support values of* $\{a, b\} = 2 \times \mathbf{0.7} \times 0.8 = 1.12$ *and of* $\{a, d\} = 2 \times \mathbf{0.7} \times 1.0 = 1.4$*. Thus, both* $\{a, b\}$ *and* $\{a, d\}$ *are frequent.*

The system then extracts from this single-path tree to form an UF-tree for the $\{a, b\}$*-projected database. The resulting tree, as shown in Figure 1(c), consists of a single node (d:1.0):2 with the expected support of* $\{a, b\}$ *equal to* $\mathbf{0.56} = 0.7 \times 0.8$ *(implying that* $\{d\}$ *occurs together with* $\{a, b\}$ *twice in the original database). Itemset* $\{a, b, d\}$*, with its expected support equals* $2 \times \mathbf{0.56} \times 1.0 = 1.12$*, is frequent. This marks the end of the extensions of* $\{a\}$*.*

Then, the system considers the next item in $\texttt{Item}^{\texttt{M}}$ *(i.e., b) and constructs an UF-tree for the* $\{b\}$*-projected database. The resulting tree, as shown in Figure 1(d), consists of a single path—namely,* $\langle$*(d:1.0):3, (c:0.8):2*$\rangle$ *with the expected support of b equal to* **0.8** *(implying that* $\{b, d\}$ *occurs three times and* $\{b, c\}$ *occurs twice in the original database). The expected support values of* $\{b, c\} = 2 \times \mathbf{0.8} \times 0.8 = 1.28$ *and of* $\{b, d\} = 3 \times \mathbf{0.8} \times 1.0 = 2.4$*. So, they are both frequent.*

The system then extracts from this single-path tree to form an UF-tree for the $\{b, c\}$*-projected database. The resulting tree, as shown in Figure 1(e), consists*

of a single node (d:1.0):2 with expected support of $\{b, c\}$ *equal to* $\mathbf{0.64} = 0.8 \times 0.8$ *(implying that d occurs together with* $\{b, c\}$ *twice in the original database). Itemset* $\{b, c, d\}$*, with its expected support equals* $2 \times \mathbf{0.64} \times 1.0 = 1.28$*, is frequent.*

Since no more item belongs to $\texttt{Item}^{\texttt{M}}$*, this marks the end of the mining process. Our proposed system recursively finds the following eight locally frequent sets that satisfy the SUC constraint* C_2 *from uncertain data:* $\{a\}$*:2.2,* $\{a, b\}$*:1.12,* $\{a, b, d\}$*:1.12,* $\{a, d\}$*:1.4,* $\{b\}$*:2.4,* $\{b, c\}$*:1.28,* $\{b, c, d\}$*:1.28 and* $\{b, d\}$*:2.4.*

3.2 Finding Constrained Globally Frequent Sets

Once the constrained locally frequent sets are found from distributed uncertain data, the next step is to find the constrained globally frequent sets among those contrained locally frequent sets. Note that it is not a good idea to transmit all data TDB_i from each site/processor P_i to a centralized site/processor Q, where all data are merged to form a global database $TDB = \bigcup_i TDB_i$ from which constrained globally frequent sets are found. The problem with such an approach is that it requires lots of communication for transmitting data from each site. This problem is worsen when TDB_i's are huge; wireless sensors can generate huge amount of data. Moreover, such an approach does not make use of constrained locally frequent sets in finding constrained globally frequent sets.

Similarly, it is also not a good idea to ask each site to transmit all its constrained locally frequent sets to a centralized site, where the itemsets are merged. The merge result is a collection of global candidate sets. The problem is that if a constrained set X is locally frequent at a site P_1 but not at another site P_2, then we do not have the frequency of X at P_2. Lacking this frequency information, one may not be able to determine whether X is globally frequent or not.

Instead, our proposed system does the following. Each site/processor P_i (for $1 \leq i \leq p$) applies constraint checking and frequency checking to find locally frequent $\texttt{Item}_i^{\texttt{M}}$ items (and $\texttt{Item}_i^{\texttt{O}}$ items for C_{SUC}), which are then transmitted to a centralized site/processor Q. It takes the union of these items, and broadcasts the union to all P_i's. Each P_i then extracts these items (potentially globally frequent items) from transactions in TDB_i and puts into an UF-tree. Note that all globally frequent sets must be composed of only the items from this union because: (i) if an item A is globally frequent, A must be locally frequent in at least one of P_i's; (ii) if an item B is locally infrequent in *all* the P_i's, B is guaranteed to be globally infrequent.) At each site P_i, the UF-tree contains (i) items that are locally frequent w.r.t. P_i and (ii) items that are potentially globally frequent but locally infrequent items w.r.t P_i. Then, our system recursively applies the usual tree-based mining process (e.g., UF-growth) to each α-projected database (where locally frequent $\alpha \subseteq \texttt{Item}_i^{\texttt{M}}$) of the UF-tree at P_i to find *constrained locally frequent sets* (with local frequency information). These sets are then sent to Q, where the local frequencies are summed. As a result, *constrained globally frequent*

sets can be found. If the sum of available local frequencies of a constrained set X meets the minimum support threshold, then X is globally frequent. For the case where a constrained set is locally frequent at a site P_1 but not at another site P_2, then Q sends a request to P_2 for finding its local frequency. It is guaranteed that such frequency information can be found by traversing appropriate paths in the UF-tree at P_2 (because the UP-tree keeps all potential globally frequent items).

To summarize, given p sites/processors in a distributed environment (e.g., a wireless sensor network), our system makes use of (i) the constrained locally frequent sets and (ii) the UF-trees that keep all potentially global frequent items to efficiently find constrained globally frequent sets (w.r.t. the entire distributed environment). Again, constraints are pushed inside the mining process; the computation is proportional to the selectivity of constraints. Moreover, our proposed system does not require lots of communication among processors (e.g., it does not need to transmit TDB_i).

4 Experimental Results

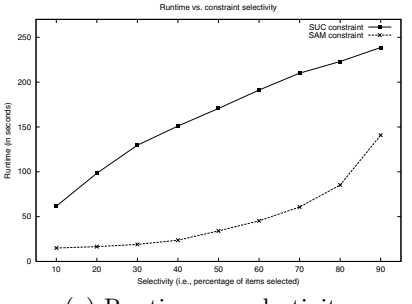
For experimental evaluation, we used many different datasets including IBM synthetic data, real-life databases from the UC Irvine Machine Learning Depository (e.g., mushroom data) as well as those from the Frequent Itemset Mining Implementation (FIMI) Dataset Repository. We cite below those experimental results based on a dataset generated by the program developed at IBM Almaden Research Center [3]. The dataset contains 10M records with an average transaction length of 10 items, and a domain of 1,000 items. Unless otherwise specified, we used $minsup = 0.01\%$. We assigned to each item an existential probability in the range of (0,1]. All experiments were run in a time-sharing environment in a 2.4 GHz machine. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os for constraint checking, UF-tree construction, and frequent set mining steps.

In the first experiment, we evaluated the functionality of our proposed system, which was implemented in C. For instance, we used (i) a dataset of uncertain data and (ii) a *constraint with 100% selectivity* (so that every item is selected). With this setting, we compared our system (which mines *constrained* frequent sets from uncertain data) with U-Apriori [7] and UF-growth [17] (which mine *unconstrained* frequent sets from uncertain data). Experimental results showed that (i) our system returned the *same* collection of frequent sets as those returned by U-Apriori and UF-growth, and (ii) both U-Apriori and UF-growth are confined to finding frequent sets from a centralized dataset of uncertain data when the user-specified constraints are of 100% selectivity, whereas our proposed system is capable of finding frequent sets from *distributed* uncertain data with constraints of *any selectivity*.

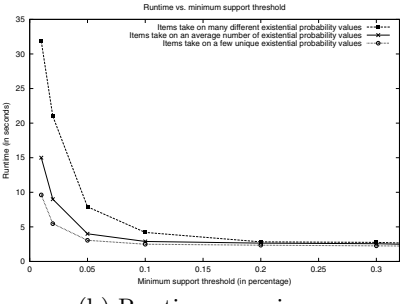
Next, we used (i) a constraint and (ii) a dataset of uncertain data consisting of items *all with existential probability of 1* (indicating that all items are definitely present in the database). With this setting, we compared our system (which mines constrained frequent sets from *uncertain* data) with some existing algorithms that mine constrained frequent sets from *precise* data (e.g., CAP [19]). From the experimental results, we observed that (i) our system returned the *same* collection of frequent sets as those returned by CAP. Note that CAP is confined to finding frequent sets from a centralized dataset of uncertain data when existential probability of all items is of 1. In contrast, our proposed system is capable of finding frequent sets from distributed uncertain data containing items with *various existential probability values* ranging from 0 to 1.

In the third experiment, we measured the amount of communication/data transmitted between the distributed sites P_i's and their centralized site Q. Figure 2(d) shows that the amount of transmitted data decreased when the selectivity of constraints decreased. Figure 2(a) shows the corresponding runtimes. Specifically, runtimes decreased when the selectivity of constraints decreased. Both graphs demonstrate the *effectiveness of constrained mining in a distributed environment*.

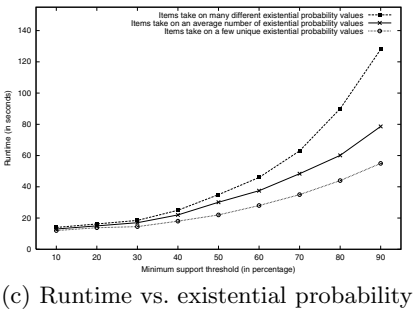
In the fourth experiment, we evaluated the effects of varying the number of distributed sites. When more sites were in the distributed network, our system transmitted more data because an addition of a site implies transmission of an



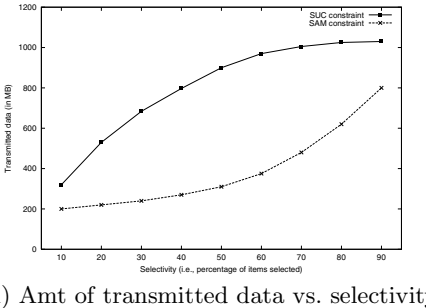
(a) Runtime vs. selectivity



(b) Runtime vs. minsup



(c) Runtime vs. existential probability



(d) Amt of transmitted data vs. selectivity

Fig. 2. Experimental results of our proposed system

additional set of locally frequent items and locally frequent itemsets. In terms of runtime, when more sites were in the network, the runtime of our system increased slightly. This is because the extra communication time (due to extra sites) was offset by the savings in building and mining from a smaller UF-tree at each site. For example, when we doubled the number of sites (from 4 to 8 sites), the amount of communication/data transmitted was almost double (because each site produced a similar set of locally frequent items and itemsets—especially when TDB_i's were similar); but, the runtime just increased slightly (1.07 times; i.e., not doubled) because we built and mined from smaller FP-trees at 8 sites (rather than from bigger trees at 4 sites).

In addition, we conducted a few more sets of experiments. For example, we tested the effect of distribution of existential probabilities of items. When items took on a few unique existential probability values, UF-trees became smaller and thus took shorter runtimes. See Figure 2(c). Regarding the evaluation on the scalability of our proposed system, experimental results showed that mining with our system had linear scalability with respect to the number of transactions. We also tested the effect of *minsup*. When *minsup* increased, fewer itemsets had expected support $\geq$ *minsup*, and thus shorter runtimes were required for the experiments. See Figure 2(b).

All these experimental results showed the importance and the benefits of using our proposed system in mining probabilistic datasets of uncertain data for frequent sets.

5 Conclusions

In this paper, we proposed a tree-based system for mining frequent sets that satisfy user-defined constraints from a distributed environment such as a wireless sensor network of uncertain data. Experimental results show effectiveness of our proposed system.

Acknowledgments. This project is partially supported by NSERC (Canada) in the form of research grants.

References

1. Aggarwal, C.C., et al.: Frequent pattern mining with uncertain data. In: Proc. KDD 2009, pp. 29–37 (2009)
2. Agrawal, R., Shafer, J.: Parallel mining of association rules. IEEE TKDE 8(6), 962–969 (1996)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. VLDB 1994, pp. 487–499 (1994)
4. Agrawal, R., et al.: Mining association rules between sets of items in large databases. In: Proc. ACM SIGMOD 1993, pp. 207–216 (1993)
5. Bernecker, T., et al.: Probabilistic frequent itemset mining in uncertain databases. In: Proc. KDD 2009, pp. 119–127 (2009)

6. Cheung, D.W., et al.: A fast distributed algorithm for mining association rules. In: Proc. PDIS 1996, pp. 31–42 (1996)
7. Chui, C.-K., et al.: Mining frequent itemsets from uncertain data. In: Zhou, Z.-H., Li, H., Yang, Q. (eds.) PAKDD 2007. LNCS (LNAI), vol. 4426, pp. 47–58. Springer, Heidelberg (2007)
8. Coenen, F., et al.: T-trees, vertical partitioning and distributed association rule mining. In: Proc. IEEE ICDM 2003, pp. 513–516 (2003)
9. Dai, X., et al.: Probabilistic spatial queries on existentially uncertain data. In: Medeiros, C.B., Egenhofer, M.J., Bertino, E. (eds.) SSTD 2005. LNCS, vol. 3633, pp. 400–417. Springer, Heidelberg (2005)
10. El-Hajj, M., Zaïane, O.R.: Parallel leap: large-scale maximal pattern mining in a distributed environment. In: Proc. ICPADS 2006, pp. 135–142 (2006)
11. Grahne, G., et al.: Efficient mining of constrained correlated sets. In: Proc. IEEE ICDE 2000, pp. 512–521 (2000)
12. Han, J., et al.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Mining and Knowledge Discovery 8(1), 53–87 (2004)
13. Lakshmanan, L.V.S., Leung, C.K.-S., Ng, R.: Efficient dynamic mining of constrained frequent sets. ACM TODS 28(4), 337–389 (2003)
14. Leung, C.K.-S.: Frequent itemset mining with constraints. In: Encyclopedia of Database Systems, pp. 1179–1183 (2009)
15. Leung, C.K.-S., Brajczuk, D.A.: Efficient algorithms for the mining of constrained frequent patterns from uncertain data. ACM SIGKDD Explorations 11(2), 123–130 (2009)
16. Leung, C.K.-S., Hao, B.: Mining of frequent itemsets from streams of uncertain data. In: Proc. IEEE ICDE 2009, pp. 1663–1670 (2009)
17. Leung, C.K.-S., et al.: A tree-based approach for frequent pattern mining from uncertain data. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 653–661. Springer, Heidelberg (2008)
18. Leung, C.K.-S., et al.: Mining uncertain data for frequent itemsets that satisfy aggregate constraints. In: Proc. ACM SAC 2010, pp. 1034–1038 (2010)
19. Ng, R.T., et al.: Exploratory mining and pruning optimizations of constrained associations rules. In: Proc. ACM SIGMOD 1998, pp. 13–24 (1998)
20. Pei, J., et al.: Pushing convertible constraints in frequent itemset mining. Data Mining and Knowledge Discovery 8(3), 227–252 (2004)
21. Schuster, A., et al.: A high-performance distributed algorithm for mining association rules. KAIS 7(4), 458–475 (2005)
22. Zaki, M.J.: Parallel and distributed association mining: a survey. IEEE Concurrency 7(4), 14–25 (1999)
23. Zhang, Q., et al.: Finding frequent items in probabilistic data. In: Proc. ACM SIGMOD 2008, pp. 819–832 (2008)

Set-to-Set Disjoint-Paths Routing in Recursive Dual-Net

Yamin Li[1], Shietung Peng[1], and Wanming Chu[2]

[1] Department of Computer Science Hosei University
Tokyo 184-8584 Japan
{yamin,speng}@hosei.ac.jp
[2] Department of Computer Hardware University of Aizu
Aizu-Wakamatsu 965-8580 Japan
w-chu@u-aizu.ac.jp

Abstract. Recursive dual-net (RDN) is a newly proposed interconnection network for massive parallel computers. The RDN is based on recursive dual-construction of a symmetric base-network. A k-level dual-construction for $k > 0$ creates a network containing $(2n_0)^{2^k}/2$ nodes with node-degree $d_0 + k$, where n_0 and d_0 are the number of nodes and the node-degree of the base network, respectively. The RDN is a symmetric graph and can contain huge number of nodes with small node-degree and short diameter. Node-to-set disjoint-paths routing is fundamental and has many applications for fault-tolerant and secure communications in a network. In this paper, we propose an efficient algorithm for set-to-set disjoint-paths routing in RDN. We show that, given two sets of $d_0 + k$ nodes, S and T in $RDN^k(B)$, $d_0 + k$ disjoint paths, each connecting a node in S to a node in T, can be found in $O(lglgN * lgN)$ time, where N is the number of nodes in $RDN^k(B)$. The length of the paths is at most $3(D_0/2 + 1)(lgN + 1)/(lgn_0 + 1)$, where D_0 and n_0 are the diameter and the number of nodes of base-network B, respectively.

Keywords: Interconnection network; disjoint paths; set-to-set routing.

1 Introduction

Nowadays, parallel computer systems with large number of processors achieved petaflops computing performance and are opening the door to exaflops. In the last decade, because of the advance in computer technology, computer makers such as IBM and Cray have risen up competition to build supercomputers with hundreds of thousands of processors. It has been predicted that, in the near future, the number of processors will reach millions [3].

Interconnection networks play a critical role for those supercomputers to gain high-performance. It is possible to combine cheap and efficient products to provide almost all components of a parallel computer except for the interconnection network [7]. Therefore, many topologies have been proposed for the interconnection networks and studied eagerly [2,5,14,15].

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 54–65, 2011.

The recursive dual-net (RDN), a newly proposed interconnection network, is based on recursive dual-construction of a symmetric base-network [17]. The dual-construction extends a network with n nodes and node-degree d to a network with $2n^2$ nodes and node-degree $d+1$. The k-level RDN is obtained by recursively applying dual-construction k times starting from the symmetric base network B. The RDN has many merits as long as topological properties are concerned. For example, an RDN can connect a huge number of nodes with just a small number of links per node. It is not difficult to construct an RDN connecting 3-millions nodes with 6 links per node and its diameter equals 22. Therefore, it is an interesting candidate as an interconnection network for massively parallel computers of next generations.

In the research on interconnection networks, it is very important to design and develop efficient routing algorithms. The algorithms include those for solving disjoint-paths problems in node-to-node, node-to-set, and set-to-set routings. These problems are fundamental and essential for fault tolerance and security in parallel computation and communication.

The problem of disjoint-paths routing has been investigated in many topologies. There are several algorithms in star graphs for the node-to-node disjoint paths problem in $O(n^2)$ time [6,11,13]. Gu and Peng gave algorithms for the node-to-set and the set-to-set disjoint-paths problems in hypercubes in $O(n^2)$ and $O(n^2 \log n)$ time, respectively [10]. Gu and Peng gave algorithms for the node-to-set and the set-to-set disjoint-paths problems in star graphs in $O(n^2)$ time [8,9]. In pancake graphs, there are algorithms for the node-to-node and the node-to-set disjoint-paths problems in $O(n^2)$ time [12,19]. Bossard, Kaneko and Peng gave an algorithm for node-to-set disjoint-paths problem in metacube [4]. Li, Peng and Chu gave algorithms for node-to-set disjoint-paths and fault-tolerant routing problem in RDN [18,16].

In parallel programming, collective communication is generally implemented in a way that routing messages from a set of nodes to another set of nodes so they don't interfere with each other. Therefore, it is important to design efficient algorithms for set-to-set, disjoint-paths, routing problem. In this paper, we propose an efficient algorithm that finds disjoint paths for set-to-set routing in recursive dual-nets. For a k-level recursive dual-net with base-network B, $RDN^k(B)$, given two sets of nodes, S and T with $|T| = |S| = m \leq d_0 + k$, the algorithm can find m disjoint paths that connect nodes in S to nodes in T in $O(m * \lg N)$ time, where N is the number of nodes in $RDN^k(B)$. The length of the paths is at most $3(D_0/2 + 1)(lgN + 1)/(lgn_0 + 1)$, where d_0 and D_0 are the node-degree and the diameter of the base-network, respectively.

The rest of the paper is organized as follows. Section 2 introduces some terminologies, definitions, and basic properties of recursive dual-nets. Section 3 gives the proposed algorithms for the set-to-set disjoint-paths problem in RDN. Finally, Section 4 concludes this paper with some possible future works.

2 Recursive Dual-Net

Let G be an undirected graph. The size of G, denoted as $|G|$, is the number of vertices. A path from node s to node t in G is denoted by $s \to t$. The length of the path is the number of edges in the path. For any two nodes s and t in G, we denote $D(s,t)$ as the length of a shortest path connecting s and t. The diameter of G is defined as $D(G) = \max\{D(s,t)|s,t \in G\}$. For any two nodes s and t in G, if there is a path connecting s and t, we say G is connected. If every node in G looks alike, we say G is symmetric.

Given a symmetric connected graph B with n_0 nodes and the node degree d_0, a Recursive Dual-Net of level k, denoted as $RDN^k(B)$ or $RDN^k(B(n_0))$, can be recursively defined as follows:

1. $RDN^0(B) = B$ is a symmetric connected graph with n_0 nodes, called *base-network*;
2. For $k > 0$, an $RDN^k(B)$ is constructed from $RDN^{k-1}(B)$ by a dual-construction as explained below (also see Figure 1).

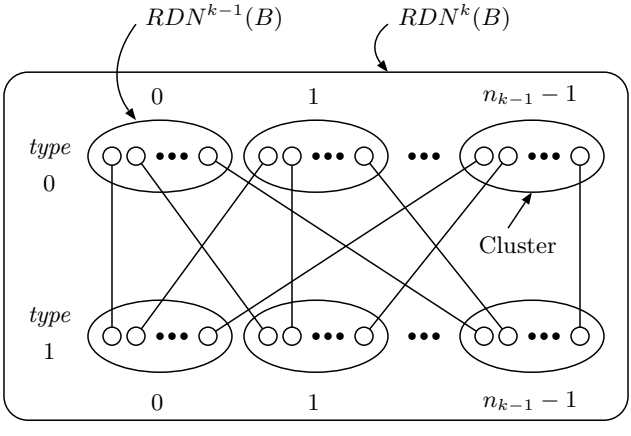


Fig. 1. Build an $RDN^k(B)$ from $RDN^{k-1}(B)$

Dual-Construction: Let $RDN^{k-1}(B)$ be referred to as a *cluster* of level k and the number of nodes $n_{k-1} = |RDN^{k-1}(B)|$. An $RDN^k(B)$ is a graph that contains $2n_{k-1}$ clusters of level k as subgraphs. These clusters are divided into two sets with each set containing n_{k-1} clusters. Each cluster in one set is said to be of *type* 0, denoted as C_i^0, where $0 \le i \le n_{k-1} - 1$ is the cluster ID. Each cluster in the other set is of *type* 1, denoted as C_j^1, where $0 \le j \le n_{k-1} - 1$ is the cluster ID. In the dual-construction at level k, each node has a new link to a node in a distinct cluster of the other type. We call this link *cross-edge* of level k. That is, for each pair of clusters C_i^0 and C_j^1, there is a unique edge connecting

a node in C_i^0 and a node in C_j^1, $0 \leq i, j \leq n_{k-1} - 1$. In Figure 1, there are n_{k-1} nodes within each cluster $RDN^{k-1}(B)$.

Figure 2 shows the $RDN^2(B(3))$ constructed from a ring with 3 nodea by appying dual-construction twice. The number of nodes in $RDN^2(B(3))$ is 2×18^2, or 648.

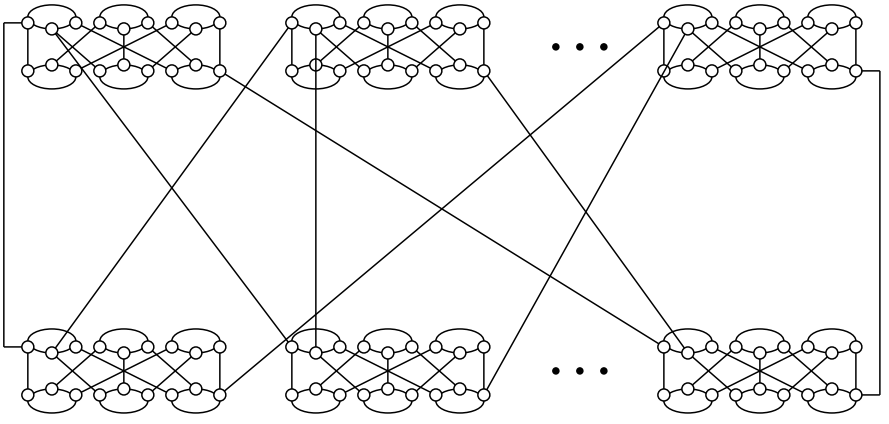


Fig. 2. A Recursive Dual-Net $RDN^2(B(3))$

Similarly, we can construct an $RDN^3(B(3))$ containing 2×648^2, or 839,808 nodes with node-degree 5 and the diameter equals to 22. In contrast, the 839808-node 3D torus machine (adopt by IBM Blue Gene/L [1]) can be configured as $108 \times 108 \times 72$ nodes with node-degree 6 and the diameter of $54 + 54 + 36 = 144$.

A node presentation for $RDN^k(B)$ that provides a unique ID to each node in $RDN^k(B)$ is described as follows. Let the set of IDs of nodes in B, denoted as ID_0, be i, $0 \leq i \leq n_0 - 1$. The ID_k of node u in $RDN^k(B)$ for $k > 0$ is a triple (u_0, u_1, u_2), where u_0 is a 0 or 1, u_1 and u_2 belong to ID_{k-1}. We call u_0, u_1, and u_2 typeID, clusterID, and nodeID of u, respectively.

More specifically, ID_i, $1 \leq i \leq k$, can be defined recursively as follows: $ID_i = (b, ID_{i-1}, ID_{i-1})$, where $b = 0$ or 1, and ID_0 is the set of IDs of nodes in B. With this ID presentation, (u, v) is a cross-edge of level k in $RDN^k(B)$ iff $u_0 \neq v_0$, $u_1 = v_2$, and $u_2 = v_1$. The ID of a node u in $RDN^k(B)$ can also be presented by an unique integer i, $0 \leq i \leq (2n_0)^{2^k}/2 - 1$, where i is the lexicographical order of the triple (u_0, u_1, u_2). For example, the ID of node $(1, 1, 2)$ in $RDN^1(B)$ is $1 * 3^2 + 1 * 3 + 2 = 14$.

The basic topological properties has been explored in [17]. The following lemma is from [17].

Lemma 1. *Assume that the base-network B is a symmetric graph with size n_0, node-degree d_0, and diameter D_0. Then, the size, the node-degree, the diameter, and the bisection bandwidth of $RDN^k(B)$ are $(2n_0)^{2^k}/2$, $d_0 + k$, $2^k D_0 + 2^{k+1} - 2$, and $\lceil (2n_0)^{2^k}/8 \rceil$, respectively.*

3 Set-to-Set Disjoint-Path Routing in RDN

In a graph G, given two disjoint sets of n nodes, S and T, the set-to-set disjoint-paths problem is to find n disjoint paths, each connecting a node in S to a node in T. In this section, we will propose an efficient algorithm for the set-to-set disjoint-paths problem in RDN.

Given two nodes u and v in $RDN^k(B)$, there exists a simple routing algorithm that finds a shortest path from u to v [17]. The routing algorithm is described formally as Algorithm 1.

Algorithm 1. RDN_routing($RDN^k(B), u, v$)
begin
  if $k = 0$ **then** RDN_routing(B, u, v)
  else
    Case 1: $u_0 = v_0$ and $u_1 = v_1$
      RDN_routing($RDN_u^{k-1}(B), u_2, v_2$);
      /* $RDN_u^{k-1}(B)$ is the cluster where u belongs to. */
    Case 2: $u_0 \neq v_0$
      RDN_routing($RDN_u^{k-1}(B), u_2, v_1$);
      RDN_routing($RDN_v^{k-1}(B), v_2, u_1$);
      $u' \leftarrow (u_0, u_1, v_1)$;
      $v' \leftarrow (v_0, v_1, u_1)$;
      connect u' and v' via a cross-edge of level k;
    Case 3: $u_0 = v_0$ and $u_1 \neq v_1$
      route u to w via the cross-edge of level k;
      route node w to node v as in Case 2;
  endif
end

Lemma 2. *In $RDN^k(B)$ with $k > 0$, a path from source s to destination t can be found in $O(\lg N)$ time and the length of the path is at most $(D_0 + 2)(\lg N + 1)/(\lg n_0 + 1) - 2$, where N is the number of nodes in $RDN^k(B)$, D_0 and n_0 are the diameter and the number of nodes in B, respectively.*

Let the $d_0 + k$ neighbors of node u be $u^{(i)}, 1 \leq i \leq d_0 + k$, where $u^{(i)}, 1 \leq i \leq d_0$, are the neighbors of u in B, and edge $(u, u^{(i)}), d_0 + 1 \leq i \leq d_0 + k$, is the cross-edge of level $i - d_0$. Let $u^{(i,j)} = (u^{(i)})^{(j)}$ for $1 \leq i, j \leq d_0 + k$, and so on. For simplicity, we denote $N(u) = \{u^{(i)}, 1 \leq i \leq d_0 + k\}$. C denotes a cluster (of level k). C_u denotes the cluster C with node $u \in C$. $type(C)$ denotes the type of cluster C.

The following lemma is basic and will be used frequently in our algorithms.

Lemma 3. *In $RDN^k(B)$, for any node u, there exist $d_0 + k$ disjoint paths $u \rightarrow u_i, 1 \leq i \leq d_0 + k$, of length at most 2, $u \rightarrow u^{(i)} \rightarrow u^{(i,d_0+k)}, 1 \leq i \leq d_0 + k$, and $u \rightarrow u^{(d_0+k)}$, such that $u_i \notin C_u$ and $C_{u_i} \neq C_{u_j}$ if $i \neq j$.*

The idea of the proposed algorithm is to try to connect each pair of nodes in S and T in different cluster. If a cluster contains multiple nodes in S or T, all nodes except a pair of nodes in S and T, if any, should be distributed to other clusters by paths of length at most 2 for further connection.

While distributing nodes to other clusters, there are certain conditions that should be satisfied. First, these paths of length at most 2 should be disjoint with the path connecting the single pair of nodes inside the cluster. Second, the paths for distributing the nodes in S or T should be disjoint with other distributing paths for the nodes in the same set, and the destination clusters they are routed to should all distinct and does not include any other nodes in the same set. However, it does not matter the paths for distributing nodes in different sets are disjoint or not. If they meet then the connection for this pair of nodes is done.

If the cluster C contains only nodes in one of the two sets S and T and the number of nodes in S (or T) is at least 2, we simply route all the nodes in $C \cap S$ (or $C \cap T$) to distinct clusters that do not contain any node or distributed node in S (or T) by disjoint paths of length at most 2. However, we need to handle the case when a node in S (or T) is distributed to a cluster that contains multiple nodes in T (or S) which are distributed to other clusters already.

After the routing described above, all clusters that are not processed yet have at most one node or one distributed node in the two sets S and T. If there exists two clusters of distinct types such that each contains a node in distinct sets then they can be connected by the basic routing subroutine (Algorithm 1). Otherwise (a pair of nodes are in two clusters of the same type), we should connect them via a cluster (third party) that is of distinct type. The third party should not contain any other nodes or distributed nodes of the two sets.

To describe the algorithm that we propose for the set-to-set, disjoint-paths, routing in RDN, we need the following subroutines, namely, RDN_modified_route, RDN_one_path, and RDN_modified_one_path.

The subroutine RDN_modified_routing has three input parameters: nodes u and v, and a cluster C with $C_u \neq C_v$, $type(C_u) = type(C_v)$, and $type(C) \neq type(C_u)$. It will be called to connect nodes u and v such that the path passes through clusters C_u, C, and C_v only.

The subroutine RDN_one_path has a cluster C and four sets of nodes, U, V, W_1, and W_2 as input. Sets U and V are the sets of the unconnected nodes or distributed nodes of sets S and T, respectively. Sets W_1 and W_2 are the sets of all nodes and distributed nodes of S and T, respectively. It will be called when $C \cap U$ and $C \cap V$ are nonempty. It will connect a pair of nodes in $C \cap U$ and $C \cap V$ and distribute all other nodes in $C \cap U$ or $C \cap V$, if any, to other distinct clusters not containing any node in W_1 or W_2, respectively.

The subroutine RDN_modified_one_path has a cluster C, a node $v \in C$, a set U such that $C \cap U = \{u_1, \ldots, u_r\}$ with $r > 1$, and r disjoint paths of length at most 2, $u_i \rightarrow u'_i, 1 \leq i \leq r$ with $C_{u'_i} \neq C$. It will connect v with a node $u_a \in C \cap U$ in C such that path $v \rightarrow u_a$ is disjoint with $r-1$ paths $u_i \rightarrow u'_i, 1 \leq i \neq a \leq r$, and remove path $u_a \rightarrow u'_a$.

The three subroutines are described formally as Algorithms 2, 3, and 4. The correctness of these algorithms, the upper bounds of their time complexities, and the maximum lengths of the paths found will be shown in the next subsection after the proposed algorithm being introduced.

Algorithm 2. RDN_modified_routing($RDN^k(B), u, v, C$)
Input: a cluster C and two nodes u and v in $RDN^k(B)$ such that $C_u \neq C_v$ and $type(C_u) = type(C_v) \neq type(C)$
Output: a path $u \to v$ that passes through C
begin
RDN_routing(C_u, u, u'), where $(u')^{(d_0+k)} \in C$;
RDN_routing(C_v, v, v'), where $(u')^{(d_0+k)} \in C$;
RDN_routing($C, (u')^{(d_0+k)}, (v')^{(d_0+k)}$);
return path $u \to u' \to (u')^{(d_0+k)} \to (v')^{(d_0+k)} \to v' \to v$;
end

The proposed algorithm is based on a dynamic routing strategy which connects pairs of nodes in S and T via different clusters. If a cluster contains multiple nodes of S or T or both we should connect at most one pair of nodes inside the cluster and all other nodes of S and T should be routed to other clusters for further routing. It uses four sets of nodes (U, V, W_1, and W_2) for bookkeeping and condition identification. Initially, we set $U = W_1 = S$ and $V = W_2 = T$. Once a pair of nodes is connected, they are removed from U and V. In the case, a node in $U \cap C$ (or $V \cap C$) is distributed to other cluster C', it is removed from U (or V) and the distributed node in C' is added to U and W_1 (or V and W_2). In the end, when the algorithm terminates, $U = V = \emptyset$ and W_1 and W_2 contain all nodes in S and T and their distributed nodes, respectively.

The algorithm is divided into three stages to handle the routing in different situations. In the first stage, we handle the clusters C with $C \cap U \neq \emptyset$ and $C \cap V \neq \emptyset$. We call RDN_one_path to generate path $u_a \to v_b$ for a pair of nodes u_a and v_b, and route all other nodes in $C \cap (U \cup V)$ to other distinct clusters that do not contain the nodes in W_1 or W_2 (depending on whether the node is in U or V) by disjoint paths of length at most 2. This process is done by calling to subroutine RDN_one_path. If a destination cluster C' of a node in U (or V) contains some nodes in V (or U) then we call RDN_one_path again on C' until the destination cluster do not contain any other node in $U \cup V$.

In the second stage, we handle the clusters C with either ($|C \cap U| > 1$ and $|C \cap V| = 0$) or ($|C \cap V| > 1$ and $|C \cap U| = 0$). We handle the case $|C \cap U| > 1$ and $|C \cap V| = 0$ first. We route all nodes in $C \cap U$ to other distinct clusters that do not contain nodes in W_1 by disjoint paths of length at most 2. If a node in $C \cap U$ is routed to a cluster C' with $|C' \cap V| \neq 0$, we call RDN_one_path again on C'. After the first case is done, we handle the second case $|C \cap V| > 1$

Algorithm 3. RDN_one_path($RDN^k(B), C, U, V, W_1, W_2$)
Input: A cluster C and four sets of nodes, U, V, W_1 and W_2, in $RDN^k(B)$ such that $C \cap U = \{u_1, \ldots, u_p\}$ and $C \cap V = \{v_1, \ldots, v_q\}$, where $p, q \geq 1$.
Output: Path $u_a \in U \rightarrow v_b \in V$, $p-1$ disjoint paths $u_i \rightarrow u'_i \notin C, 1 \leq i \neq a \leq p$, of length at most 2 such that $C_{u'_i} \cap W_1 = \emptyset$ for all $i, 1 \leq i \neq a \leq p$, and $C_{v'_i} \neq C_{v'_j}$ if $i \neq j$, and $q-1$ disjoint paths $v_i \rightarrow v'_i \notin C, 1 \leq i \neq b \leq q$, of length at most 2 such that $C_{v'_i} \cap W_2 = \emptyset$ for all $j, 1 \leq j \neq b \leq q$, and $C_{v'_i} \neq C_{v'_j}$ if $i \neq j$
begin
pick up $u_a \in U \cap C$ and $v_b \in V \cap C$ randomly;
find $p-1$ disjoint paths of length at most 2, $u_i \rightarrow u'_i, 1 \leq i \neq a \leq p$ such that $C_{u'_i} \cap W_1 = \emptyset$ for all $i, 1 \leq i \neq a \leq p$, and $C_{u'_i} \neq C_{u'_j}$ if $i \neq j$;
find $q-1$ disjoint paths of length at most 2, $v_i \rightarrow v'_i, 1 \leq i \neq b \leq q$ such that $C_{v'_i} \cap W_2 = \emptyset$ for all $i, 1 \leq i \neq b \leq q$, and $C_{v'_i} \neq C_{v'_j}$ if $i \neq j$;
RDN_basic_route(C, u_a, v_b);
if $\exists i \neq a$ such that $\{u_i \rightarrow u'_i\} \cap \{u_a \rightarrow v_b\} \neq \emptyset$ or $\exists j \neq b$ such that $\{v_j \rightarrow v'_j\} \cap \{u_a \rightarrow v_b\} \neq \emptyset$
then find $u_{a'}$ and $v_{b'}$ such that $u_{a'} \rightarrow v_{b'}$ is disjoint with the other paths of length at most 2;
remove paths $u_{a'} \rightarrow u'_{a'}$ and $v_{b'} \rightarrow v'_{b'}$;
replace path $u_a \rightarrow v_b$ by path $u_{a'} \rightarrow v_{b'}$;
find a path of length at most 2, $u_a \rightarrow u'_a$ such that it is disjoint with paths $u_i \rightarrow u'_i, i \neq a$, $C_{u'_a} \cap W_1 = \emptyset$ and $C_{u'_a} \neq C_{u'_j}$ for all $j \neq a$;
find a path of length at most 2, $v_b \rightarrow v'_b$ such that it is disjoint with paths $v_j \rightarrow v'_j, j \neq b$, $C_{v'_b} \cap W_2 = \emptyset$ and $C_{v'_b} \neq C_{v'_j}$ for all $j \neq b$;
endif
end

and $|C \cap U| = 0$. We route all nodes in $C \cap V$ to other distinct clusters that do not contain nodes in W_2 by disjoint paths of length at most 2. However, if

Algorithm 4. RDN_modified_one_path($RDN^k(B), C, v, U$)
Input: A cluster C, a node $v \in C$ and a set of nodes U in $RDN^k(B)$ such that $C \cap U = \{u_1, \ldots, u_r\}$, where $r > 1$, and r disjoint paths of length at most 2, $u_i \rightarrow u'_i, 1 \leq i \leq r$.
Output: Path $u_a \rightarrow v$ that is disjoint with paths $u_i \rightarrow u'_i, 1 \leq i \neq a \leq r$, and path $u_a \rightarrow u'_a$ is removed.
begin
pick up $u_a \in C \cap U$ randomly;
RDN_routing(C, u_a, v);
if $\exists j \neq a$ such that $\{u_j \rightarrow u'_j\} \cap \{u_a \rightarrow v\} \neq \emptyset$
then find $v_{a'}$ such that $u_{a'} \rightarrow v$ is disjoint with any of the other $r-1$ paths of length at most 2;
replace path $u_a \rightarrow v$ by path $u_{a'} \rightarrow v$;
remove path $u_{a'} \rightarrow u'_{a'}$
else remove path $u_a \rightarrow u'_a$;
endif
end

a node in $C \cap V$ is routed to a cluster C' with $|C' \cap W_1| > 1$, we cannot call RDN_one_path on C' since the nodes in $C' \cap W_1$ had been routed to other clusters. We call RDN_modified_one_path on C' instead.

In the third stage, since every cluster contains at most one node in $U \cup V$, we can connect them pairwise by RDN_routing if the pair of nodes are in the clusters of different types by RDN_modified_routing otherwise.

The results of the paper are stated in Lemmas 4 - 6 and Theorem 1. The proofs for the corectness and the time complexities of Algorithms 2 - 5 are omitted due to the page limit. The interested reader can find the proofs in the full paper.

Lemma 4. *Algorithm 2 is correct. The time complexity of Algorithm 2 is* $O(\lg N)$. *The length of the paths is at most* $3(D_0/2+1)(\lg N+1)/(\lg n_0+1)-4$, *where* N *is the number of nodes in* $RDN^k(B)$ *and* D_0 *and* n_0 *are the diameter and the number of nodes in* B, *respectively.*

Lemma 5. *Algorithm 3 is correct for the sets* U,V, W_1 *and* W_2 *defined in Algorithm 5. The time complexity of Algorithm 3 is* $O(\lg N)$. *The length of the path connects* u *to a node in* V *is at most* $(D_0/2+1)*(\lg N+1)/(\lg n_0+1)-2$.

Lemma 6. *Algorithm 4 is correct. The time complexity of Algorithm 4 is* $O(\lg N)$. *The length of the paths is at most* $(D_0/2+1)*(\lg N+1)/(\lg n_0+1)-2$.

Theorem 1. *Assume that* d_0 *and* D_0 *are the node-degree and the diameter of the base network* B, *respectively. Assume that* d_0 *disjoint paths exist in* B *between two sets of* d_0 *nodes in* B. *Let* S *and* T *be two sets of* d_0+k *nodes in* $RDN^k(B), k>0$. *Then* d_0+k *disjoint paths, each connecting a node in* S *to a node in* T *can be found in* $O(\lg\lg N*\lg N)$ *time, where* N *is the number of nodes in* $RDN^k(B)$. *The length of the paths is at most* $3(D_0/2+1)(\lg N+1)/(\lg n_0+1)$, *where* D_0 *and* n_0 *are the diameter and the number of nodes in* B.

We give an example of set-to-set disjoint-paths routing in $RDN^2(B)$ that follows the proposed algorithm step-by-step.

Example 1: In $RDN^2(B)$, where B is a ring with 3 nodes, let $s_1 = (0,(0,1,1),(1,0,2))$; $s_2=(1,(1,0,0),(0,2,0))$; $s_3=(1,(1,1,1),(0,0,0))$; and $s_4=(1,(1,1,1),(0,1,1))$, and let $t_1 = (0,(0,1,0),(1,0,1))$; $t_2 = (0,(0,1,0),(0,1,2))$; $t_3 = (1,(1,0,1),(1,2,0))$; and $t_4 = (1,(1,1,1),(1,0,2))$.

For simplicity, we do not include the updating of set W_1 and W_2. Basically, any nodes generated from a distributing path of length at most 2 from a node in S (or T) should be added to W_1 (or W_2). The clusters contain any node in W_1 (or W_2) cannot be selected as destination clusters while distributing the nodes in a cluster that contains more than one node in S (or T). Initially U and V are set to S and T, respectively

• Stage 1-1: Since cluster $C = (1,(1,1,1),(*,*,*))$ contains both nodes in S and T, we call RDN_one_path to connect s_3 to t_4 and route s_4 to cluster $(0,(0,1,0),(*,*,*)$ by a path of length 2 as follows. Notice that the path of length 1 cannot be used since the destination cluster $(0,(0,1,1),(*,*,*))$ contains s_1.

Algorithm 5. RDN_S2S_disjoint_paths($RDN^k(B), S, T$)
Input: Two disjoint sets S and T of $m \leq d_0 + k$ nodes in $RDN^k(B), k > 0$
Output: m disjoint paths connecting nodes in S and nodes in T
begin
$U = W_1 \leftarrow S$; $V = W_2 \leftarrow T$;
while $\exists$ cluster C with $|C \cap U| = p > 0 \wedge |C_s \cap V| = q > 0$ **do** /* Stage 1 */
RDN_one_path($RDN^k(B), C, U, V, W_1, W_2$);
$U \leftarrow U \cup \{u'_i, 1 \leq i \neq a \leq p\} \setminus (C \cap U)$;
$V \leftarrow V \cup \{v'_i, 1 \leq i \neq b \leq q\} \setminus (C \cap V)$;;
$W_1 \leftarrow W_1 \cup \{u'_i, 1 \leq i \neq a \leq p\}$;
$W_2 \leftarrow W_2 \cup \{v'_i, 1 \leq i \neq b \leq q\}$;
while $\exists$ cluster C with $|C \cap U| = p > 1$ and $|C \cap V| = 0$ **do** /* Stage 2 */
find p disjoint paths $u_i \rightarrow u'_i, 1 \leq i \leq p$ of length at most 2 s.t. clusters $C_{u'_i}$ are all distinct and do not contain any node in W_1;
$U \leftarrow U \cup \{u'_i, 1 \leq i \leq p\} \setminus (C \cap U)$;
$W_1 \leftarrow W_1 \cup \{u'_i, 1 \leq i \leq p\}$
while $\exists$ cluster C with $C \cap U = \{u\}$ and $|C \cap V| > 0$ **do**
RDN_one_path($RDN^k(B), \{u\}, V, W_1, W_2$);
$U \leftarrow U \setminus \{u\}$;
$V \leftarrow V \cup \{v'_i, 1 \leq i \neq b \leq q\} \setminus (C \cap V)$;
$W_2 \leftarrow W_2 \cup \{v'_i, 1 \leq i \neq b \leq q\}$;
while $\exists$ cluster C with $|C \cap V| = q > 1$ and $|C \cap U| = 0$ **do**
find q disjoint paths $v_i \rightarrow v'_i, 1 \leq i \leq q$ of length at most 2 s.t. clusters $C_{v'_i}$ are all distinct and do not contain any node in W_2;
$V \leftarrow V \cup \{v'_i, 1 \leq i \leq q\} \setminus (C \cap V)$;;
$W_2 \leftarrow W_2 \cup \{v'_i, 1 \leq i \leq q\}$;
while $\exists$ cluster C with $|C \cap V| = \{v\}$ and $|C \cap W_1| > 0$ **do**
if $|C \cap W_1| = \{u\}$ /* $C \cap W_1 = C \cap U$ in this case. */
then
RDN_routing($RDN^k(B), u, v$)
$U \leftarrow U \setminus \{u\}$;
$V \leftarrow V \setminus \{v\}$;
else
RDN_modified_one_path($RDN^k(B), C, v, W_1$);
$U \leftarrow U \setminus \{u'_a\}$;
$V \leftarrow V \setminus \{v\}$;
while $\exists$ C and C' with $type(C) \neq type(C') \wedge C \cap U = \{u\} \wedge C' \cap V = \{v\}$ **do**
/* Stage 3 */
RDN_routing($RDN^k(B), u, v$);
$U \leftarrow U \setminus \{u\}$;
$V \leftarrow V \setminus \{v\}$;
while $\exists$ C and C' with $type(C) = type(C') \wedge C \cap U = \{u\} \wedge C' \cap V = \{v\}$ **do**
find a cluster C'' s.t. $type(C'') \neq type(C)$ and $C \cap (W_1 \cup W_2) = \emptyset$;
RDN_modified_routing($RDN^k(B), u, v, C''$);
$U \leftarrow U \setminus \{u\}$;
$V \leftarrow V \setminus \{v\}$;
end

Path #1: $s_3 = (1,(1,1,1),(0,0,0)) \rightarrow (1,(1,1,1),(1,0,0)) \rightarrow t_4 = (1,(1,1,1),(1,0,2))$ $s_4 \rightarrow (1,(1,1,1),(0,1,0)) \rightarrow (0,(0,1,0),(1,1,1)) = s'_4$
The nodes in U and V are updated: $U = \{s_1, s_2, s'_4\}, V = \{t_1, t_2, t_3\}$.

• Stage 1-2: Since cluster $(0,(0,1,0),(*,*,*))$ contain t_1, t_2, and s'_4, we call RDN_one_path again for cluster $(0,(0,1,0),(*,*,*))$.
Path #2: $s'_1 \rightarrow t_1(0,(0,1,0),(0,1,1) \rightarrow (0,(0,1,0),(0,1,0) \rightarrow (0,(0,1,0),(1,0,1) = t_1$
$t_1 \rightarrow (0,(0,1,0),(1,0,0)) \rightarrow (1,(1,0,0),(0,1,0) = t'_1$
The nodes in U and V are updated: $U = \{s_1, s_2\}, V = \{t'_2, t_3\}$.

• Stage 3: Since all the nodes in $U \cup V$ are in distinct clusters, we connect the pair of nodes (s_1 and t_3) that are in the clusters of distinct types by RDN_routing. Finally, we pick up cluster $C = (0,(0,0,0),(*,*,*))$ of type 0 that does not contain any node in $W_1 \cup W_2$ and call RDN_Modified_Routing to connect the last pair of nodes (s_2 and t'_2) via cluster C.
Path #3: $s_1 \Rightarrow (0,(0,1,1),(1,0,1)) \rightarrow (1,(1,0,1),(0,1,1)) \Rightarrow t_3$
Path #4: $s_2 \Rightarrow (1,(1,0,0),(0,0,0)) \rightarrow (0,(0,0,0),(1,0,0)) \Rightarrow (0,(0,0,0),(0,1,2)) \rightarrow (1,(0,1,2),(0,0,0)) \Rightarrow t'_2$

4 Concluding Remarks

Recursive dual-net is a potential interconnection network for supercomputers of next generations because of its low node degree and short diameter for the extremely large parallel computer systems. Its symmetric and recursive structure, and simple routing algorithms are also attractive. In this paper, we proposed an efficient algorithm for the set-to-set disjoint-paths routing on recursive dual-net. There are many other interesting collective communication and computational problems on recursive dual-net that are worth further research.

References

1. Adiga, N.R., Blumrich, M.A., Chen, D., Coteus, P., Gara, A., Giampapa, M.E., Heidelberger, P., Singh, S., Steinmacher-Burow, B.D., Takken, T., Tsao, M., Vranas, P.: Blue gene/l torus interconnection network. IBM Journal of Research and Development 49(2/3), 265–276 (2005)
2. Akers, S.B., Krishnamurthy, B.: A group-theoretic model for symmetric interconnection networks. IEEE Transactions on Computers 38(4), 555–566 (1989)
3. Beckman, P.: Looking toward exascale computing. In: Proceedings of the 2008 International Conference on Parallel and Distributed Computing, Applications and Technologies (December 2008) keynote speaker
4. Bossard, A., Kaneko, K., Peng, S.: Node-to-set disjoint-paths routing in metacube. In: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 8–11. IEEE Computer Society Press, Hiroshima (2009)
5. Corbett, P.F.: Rotator graphs: An efficient topology for point-to-point multiprocessor networks. IEEE Transactions on Parallel and Distributed Systems 3(5), 622–626 (1992)

6. Day, K., Tripathi, A.: A comparative study of topological properties of hypercubes and star graphs. IEEE Transactions on Parallel and Distributed Systems 5(1), 31–38 (1994)
7. Duato, J., Yalamanchili, S., Ni, L.: Interconnection Networks: An Engineering Approach. IEEE Computer Society Press, Los Alamitos (1997)
8. Gu, Q.P., Peng, S.: Set-to-set fault tolerant routing in star graphs. IEICE Trans. on Information and Systems E79-D(4), 282–289 (1996)
9. Gu, Q.P., Peng, S.: Node-to-set disjoint paths problem in star graphs. Information Processing Letters 62(4), 201–207 (1997)
10. Gu, Q.P., Peng, S.: Node-to-set and set-to-set cluster fault tolerant routing in hypercubes. Parallel Computing 24(9), 1245–1261 (1998)
11. Jovanovic, Z., Misic, J.V.: Fault tolerance of the star graph interconnection network. Information Processing Letters 49(3), 145–150 (1994)
12. Kaneko, K., Suzuki, Y.: Node-to-set disjoint paths problem in pancake graphs. IEICE Transactions on Information and Systems E86-D(9), 1628–1633 (2003)
13. Latifi, S.: On the fault-diameter of the star graph. Information Processing Letters 46(3), 143–150 (1993)
14. Li, Y., Peng, S.: Dual-cubes: a new interconnection network for high-performance computer clusters. In: Proceedings of the 2000 International Computer Symposium, Workshop on Computer Architecture, ChiaYi, Taiwan, pp. 51–57 (December 2000)
15. Li, Y., Peng, S., Chu, W.: Metacube - a new interconnection network for large scale parallel systems. Australian Computer Science Communications 24(3), 29–36 (2002)
16. Li, Y., Peng, S., Chu, W.: Disjoint-paths and fault-tolerant routing on recursive dual-net. In: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 48–56. IEEE Computer Society Press, Hiroshima (2009)
17. Li, Y., Peng, S., Chu, W.: Recursive dual-net: A new versatile network for supercomputers of the next generation. Journal of Chinese Institute of Engineer 32, 931–938 (2009)
18. Li, Y., Peng, S., Chu, W.: Node-to-set disjoint-paths routing in recursive dual-net. In: Proceedings of the First International Conference on Networking and Computing, Higashi Hiroshima, Japan, pp. 9–14 (November 2010)
19. Suzuki, Y., Kaneko, K.: An algorithm for node-disjoint paths in pancake graphs. IEICE Transactions on Information and Systems E86-D(3), 610–615 (2003)

Redflag: A Framework for Analysis of Kernel-Level Concurrency

Justin Seyster, Prabakar Radhakrishnan, Samriti Katoch, Abhinav Duggal, Scott D. Stoller, and Erez Zadok

Department of Computer Science, Stony Brook University

Abstract. Although sophisticated runtime bug detection tools exist to root out several kinds of concurrency errors, they cannot easily be used at the kernel level. Our *Redflag* framework and system seeks to bring these essential techniques to the Linux kernel by addressing issues faced by other tools. First, other tools typically examine every potentially concurrent memory access, which is infeasible in the kernel because of the overhead it would introduce. Redflag minimizes overhead by using offline analysis together with an efficient in-line logging system and by supporting targeted configurable logging of specific kernel components and data structures. Targeted analysis reduces overhead and avoids presenting developers with error reports for components they are not responsible for. Second, other tools do not take into account some of the synchronization patterns found in the kernel, resulting in false positives. We explore two algorithms for detecting concurrency errors: one for race conditions and another for atomicity violations; we enhanced them to take into account some specifics of synchronization in the kernel. In particular, we introduce Lexical Object Availability (LOA) analysis to deal with multi-stage escape and other complex order-enforcing synchronization. We evaluate the effectiveness and performance of Redflag on two file systems and a video driver.

1 Introduction

As the kernel underlies all of a system's concurrency, it is the most important front for eliminating concurrency errors. In order to design a highly reliable operating system, developers need tools to find concurrency errors before they cause real problems in production systems. Understanding concurrency in the kernel is difficult. Unlike many user-level applications, almost the entire kernel runs in a multi-threaded context, and much of it is written by experts who rely on intricate synchronization techniques.

Runtime analysis is a powerful and flexible approach to detection of concurrency errors. We designed the *Redflag* framework and system with the goal of airlifting this approach to the kernel front lines. Redflag takes its name from stock car and formula racing, where officials signal with a red flag to end a race. It has two main parts:

1. *Fast Kernel Logging* uses compiler plug-ins to provide *modular* instrumentation that targets specific data structures in specific kernel subsystems for logging. It reserves an in-memory buffer to log operations on the targeted data structures with the best possible performance.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 66–79, 2011.

2. The *offline Redflag analysis* tool performs post-mortem analyses on the resulting logs. Offline analysis reduces runtime overhead and allows any number of analysis algorithms to be applied to the logs.

Currently, Redflag implements two kinds of concurrency analyses: *Lockset* [15] analysis for data races and *block-based* [19] analysis for atomicity violations. We developed several enhancements to improve the accuracy of these algorithms, including *Lexical Object Availability* (LOA) analysis, which eliminates false positives caused by complicated initialization code. We also augmented Lockset to support Read-Copy-Update (RCU) [12] synchronization, a synchronization tool new to the Linux kernel.

The paper is organized as follows. Section 2 describes our system. Section 3 presents experimental results. Section 4 discusses related work. Section 5 concludes and discusses future work.

2 Design

2.1 Instrumentation and Logging

Redflag inserts targeted instrumentation using a suite of GCC compiler plug-ins that we developed specifically for Redflag. Plug-ins are a recent GCC feature that we contributed to the development of. Compiler plug-ins execute during compilation and have direct access to GCC's intermediate representation of the code [2]. Redflag's GCC plug-ins search for relevant operations and instrument them with function calls that serve as hooks into Redflag's logging system.

Redflag currently logs four types of operations: (1) Field access: read from or write to a field in a `struct`; (2) Synchronization: acquire/release operation on a lock or wait/signal operation on a condition variable; (3) Memory allocation: creation of a kernel object, necessary for tracking memory reuse (Redflag can also track deallocations, if desired); (4) System call (syscall) boundary: syscall entrance/exit (used for atomicity checking).

When compiling the kernel with the Redflag plug-ins, the developer provides a list of `structs` to target for instrumentation. Field accesses and lock acquire/release operations are instrumented only if they operate on a targeted `struct`. A lock acquire/release operation is considered to operate on a `struct` if the lock it accesses is a field within that `struct`. Some locks in the kernel are not members of any `struct`: these global locks can be directly targeted by name.

To minimize runtime overhead, and to allow logging in contexts where potentially blocking I/O operations are not permitted (e.g., in interrupt handlers or while holding a spinlock), Redflag stores logged information in a lock-free in-memory buffer. I/O is deferred until logging is complete.

When logging is finished, a backend thread empties the buffer and writes the records to disk. With 1GB of memory allocated for the buffer, it is possible to log 7 million events, which was enough to provide useful results for all our analyses.

2.2 Lockset Algorithm

Lockset is a well known algorithm for detecting *data races* that result from variable accesses that are not correctly protected by locks. Our Lockset implementation is based

on Eraser [15]. A *data race* occurs when two accesses to the same variable, at least one of them a write, can execute together without intervening synchronization. Not all data races are bugs. A data race is *benign* when it does not affect the program's correctness.

Lockset maintains a *candidate set* of locks for each monitored variable. The candidate lockset represents the locks that have consistently protected the variable. A variable with an empty candidate lockset is potentially involved in a race. Before the first access to a variable, its candidate lockset is the set of all possible locks. The algorithm tracks the current lockset for each thread. Each lock-acquire event adds a lock to its thread's lockset. The corresponding release removes the lock.

When an access to a variable is processed, the variable's candidate lockset is refined by intersecting it with the thread's current lockset. In other words, the algorithm sets the variable's candidate lockset to be the set of locks that were held for *every* access to the variable. When a candidate lockset becomes empty, the algorithm revisits every previous access to the same variable, and if no common locks protected both the current access and that previous one, we report the pair as a potential data race.

Redflag produces at most one report for each pair of lines in the source code, so the developer does not need to examine multiple reports for the same race. Each report contains every stack trace that led to the race for both lines of code and the list of locks that were held at each access.

Beyond the basic algorithm described above, there are several common refinements that eliminate false positives (false alarms) due to pairs of accesses that do not share locks but cannot occur concurrently for other reasons.

Variable initialization. When a thread allocates a new object, no other thread has access to that object. until the thread stores the new object's address in globally accessible memory. Most initialization routines in the kernel exploit this to avoid the cost of locking during initialization. As a result, most accesses during initialization appear to be data races to the basic Lockset algorithm.

The Eraser algorithm solves this problem by tracking which threads access variables to determine when each variable become shared by multiple threads [15]. We implement a variant of this idea: when a variable is accessed by more than one thread or accessed while holding a lock, it is considered shared. Accesses to a variable before its first shared access are marked as thread local, and Lockset ignores them.

Memory reuse. When a region of memory is freed, allocating new data structures in the same memory can cause false positives in Lockset, because variables are identified by their location in memory. Eraser solves this problem by reinitializing the candidate lockset for every memory location in a newly allocated region [15]. Redflag also logs calls to allocation functions, so that it can similarly account for reuse.

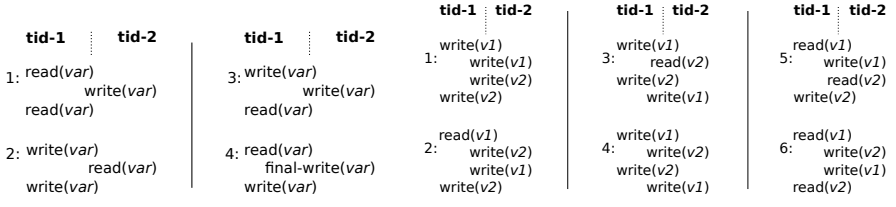
2.3 Block-Based Algorithms

Redflag includes two variants of Wang and Stoller's block-based algorithm [18,19]. These algorithms check for *atomicity*, which is similar to serializability of database transactions and provides a stronger guarantee than freedom from data races. Two atomic functions executing in parallel always produce the same result as if they executed in sequence, one after the other.

When checking atomicity for the kernel, system calls provide a natural unit of atomicity. By default, we check atomicity for each syscall execution. Not all syscalls need to be atomic, so Redflag provides a simple mechanism to specify smaller atomic regions (see Section 2.5).

We implemented two variants of the block-based algorithm: a single-variable variant that detects violations involving just one variable and a two-variable variant that detects violations involving more than one variable.

The single-variable block-based algorithm decomposes each syscall execution into a set of *blocks*, which represent sequential accesses to a variable. Each block includes two accesses to the same variable in the same thread, as well as the list of locks that were held for the duration of the block (i.e., all locks acquired before the first access and not released until after the second access). The algorithm then checks each block, searching all other threads for any access to the block's variable that might interleave with the block in an unserializable way. An access can interleave a block if it is made without holding any of the block's locks, and the interleaving is unserializable if it matches any of the patterns in Figure 1(a).



(a) Single variable (b) Double variable

Fig. 1. The illegal interleavings in the single- and double-variable block-based algorithms [19]. Note that a final write is the last write to a variable during the execution of an atomic region.

The two-variable block-based algorithm also begins by decomposing each syscall execution into blocks. A two-variable block comprises two accesses to *different variables* in the same thread and syscall execution. The algorithm searches for pairs of blocks in different threads that can interleave illegally. Each block includes enough information about which locks were held, acquired, or released during its execution to determine which interleavings are possible. Figure 1(b) shows the six illegal interleavings for the two-variable block-based algorithm; Wang and Stoller give details of the locking information saved for each block [19].

Together, these two variants are sufficient to determine whether any two syscalls in a trace can violate each other's atomicity [19]. In other words, these algorithms can detect atomicity violations involving any number of variables.

Analogues of the Lockset refinements in Section 2.2 are used in the block-based algorithm to eliminate false positives due to variable initialization and memory re-use.

2.4 Algorithm Enhancements

The kernel is a highly concurrent environment and uses several different styles of synchronization. Among these, we found some that were not addressed by previous work

on detecting concurrency violations. This section discusses two new synchronization methods that Redflag handles: multi-stage escape and RCU.

Multi-stage escape. As explained in Section 2.2, objects within their initialization phases are effectively protected against concurrent access, because other threads do not have access to them. However, an object's accessibility to other threads is not necessarily binary. An object may be available to a limited set of functions during a secondary initialization phase and then become available to a wider set of functions when that phase completes. During the secondary initialization, some concurrent accesses are possible, but the initialization code is still protected against interleaving with many functions. We call this phenomenon *multi-stage escape*. As an example, inode objects go through two stages of escape. First, after a short first-stage initialization, the inode gets placed on a master inode list in the file system's superblock. File-system–specific code performs a second initialization and then assigns the inode to a dentry.

The block-based algorithm reported illegal interleavings between accesses in the second-stage initialization and syscalls that operate on files, like `read()` and `write()`. These interleavings are not possible, however, because file syscalls *always* access inodes through a dentry. Before an object is assigned to a dentry—its second escape—the second-stage initialization code is protected against concurrent accesses from any file syscalls. Interleavings are possible with functions that traverse the superblock's inode list, such as the writeback thread, but they do not result in atomicity violations, because they were designed to interleave correctly with second-stage initialization.

To avoid reporting these kinds of false interleavings, we introduce *Lexical Object Availability* (LOA) analysis, which produces a relation on field accesses for each targeted `struct`. Intuitively, the LOA relation encodes observed ordering among lines of code. We use these orderings to infer when an object becomes unavailable to a region of code, marking the end of an initialization phase.

In the inode example, any access from a file syscall serves as evidence that first- and second-stage initialization are finished, meaning that accesses from those initialization routines are no longer possible. Accesses from the writeback thread are weaker evidence, showing that first-stage initialization is finished.

The LOA algorithm first divides the log file into sub-traces. Each sub-trace contains all accesses to one particular instance o of a targeted `struct` S. For each sub-trace, which is for some instance of some `struct` S, the algorithm adds an entry for a pair of statements in the LOA relation for S when it observes that one of the statements occurred after the other in a different thread in that sub-trace. Specifically, for a `struct` S and read/write statements a and b, (a, b) is included in LOA_S iff there exists a sub-trace for an instance of `struct` S containing events e_a and e_b such that:

1. e_a is performed by statement a, and e_b is performed by statement b, and
2. e_a occurs before e_b in the sub-trace, and
3. e_a and e_b occur in different threads.

We modified the block-based algorithm to report an atomicity violation only if the interleaving statements that caused the violation are allowed to interleave by their LOA relation. For an event produced by statement b to interleave a block produced by statements a and c, the LOA relation must contain the pairs (a, b) and (b, c). Otherwise, the algorithm considers the interleaving impossible.

Returning to the inode example, consider a and c to be statements from the secondary initialization stage and b to be a statement in a function called by the `read` syscall. Because statement b cannot access the inode until after secondary initialization is done, (b, c) cannot be in LOA_{inode}, the LOA relation for inodes.

We also added LOA analysis to the Lockset algorithm: it reports that two statements a and b can race only if both (a, b) and (b, a) are in the LOA relation for the `struct` that a and b access.

Although we designed LOA analysis specifically for multi-stage escape, it can also infer other kinds of order-enforcing synchronization. For example, we found that the kernel sometimes uses condition variables to protect against certain operations to inodes that are in a startup state, which lasts longer than its initialization. We constructed the happened-before relation [8] to determine which potential interleavings were precluded by condition variables, but all such interleavings were already filtered by LOA. LOA analysis can also infer *destruction* phases, when objects typically return to being exclusive to one thread.

Because LOA filters interleavings based on the observed order of events, it can cause false negatives (i.e., it can eliminate warnings corresponding to actual errors). The common technique of filtering based on when variables become shared (see Section 2.2) has the same problem: if a variable becomes globally accessible but is not promptly accessed by another thread, neither technique recognizes that such an access is possible. Dynamic escape analysis addresses this problem by determining precisely when an object becomes globally accessible [19], but it accounts for only one level of escape.

Syscall interleavings. Engler and Ashcraft observed that dependencies on data prevent some kinds of syscalls from interleaving [4]. For example, a `write` operation on a file never executes in parallel with an `open` operation on the same file, because userspace programs have no way to call `write` before `open` finishes.

These dependencies are actually a kind of multi-stage escape. The return from `open` is an escape for the file object, which then becomes available to other syscalls, such as `write`. For functions that are called only from one syscall, our LOA analysis already rules out impossible interleavings between syscalls with this kind of dependency.

However, when a function is reused in several syscalls, the LOA relation, as described above, cannot distinguish executions of the same statement that were executed in different syscalls. As a result, if LOA analysis sees that an interleaving in a shared function is possible between one pair of syscalls, it will believe that the interleaving is possible between any pair of syscalls.

To overcome this problem, we augment the LOA relation to contain entries of the form $((syscall, statement), (syscall, statement))$. As a result, LOA analysis treats a function called from different syscalls as separate functions. Statements that do not execute in a syscall are instead paired with the name of the kernel thread they execute in. The augmented LOA relations can express dependencies caused by both multi-stage escape during initialization and dependencies among syscalls.

RCU. Read-Copy Update (RCU) synchronization is a recent addition to the Linux kernel that allows very efficient read access to shared variables [12]. A typical RCU-write first copies the protected data structure, modifies the local copy, and then replaces the

pointer to the original copy with a pointer to the updated copy. RCU synchronization does not protect against lost updates, so writers must use their own locking. A reader needs only to surround read-side critical sections with `rcu_read_lock()` and `rcu_read_unlock()`, which ensure that the shared data structure does not get freed during the critical section.

We extended Lockset to test for correctness of RCU use. When a thread enters a read-side critical section by calling `rcu_read_lock()`, our implementation adds a virtual RCU lock to the thread's lockset. We do not report a data race between a read and a write if the read access has the virtual RCU lock in its lockset. However, conflicting writes to an RCU-protected variable will still produce a data race report.

2.5 Filtering False Positives and Benign Warnings

Bit-level granularity. We found that many false positives in the block-based algorithms were caused by *flag variables*, like the `i_state` field in Figure 2, which group several boolean values into one integer variable. Because several flags are stored in the same variable, an access to any individual flag appears to access all flags in the variable. Erickson et al. observed this same pattern in the Windows 7 kernel and account for it in their DataCollider race detector [5].

Figure 2 shows an example of an interleaving that the single-variable block-based algorithm would report as a violation. The two bitwise assignments in thread 1 both write to the `i_state` field. These two writes form a block between which the conditional in thread 2 can interleave; this is one of the illegal patterns shown in Figure 1(a). However, there is no atomicity problem, because thread 1 writes only the `I_SYNC` bit, and thread 2 reads only the `I_CLEAR` bit.

We eliminate such false positives by modifying the block-based algorithms to treat any variable that is sometimes accessed using bitwise operators as 64 individual variables (on 64-bit systems). Our analysis still detects interleavings between bitwise accesses to individual flags and accesses that involve the whole variable.

```
/* [Thread 1] */
spin_lock(inode->lock);
inode->i_state |= I_SYNC;
spin_unlock(inode->lock);

                                  /* [Thread 2] */

                                  spin_lock(inode->lock);
                                  if (inode->i_state & I_CLEAR) {
                                     /* ... */
                                  }
                                  spin_unlock(inode->lock);

spin_lock(inode->lock);
inode->i_state &= ~I_SYNC;
spin_unlock(inode->lock);
```

Fig. 2. An interleaving that appears to violate the atomicity of the `i_state` field. However, this is a false alarm, because the two threads access different bits of the field.

Idempotent operations. An operation is *idempotent* if, when it is executed multiple times on the same variable, only the first execution changes the variable's value. For example, setting a bit in a flag variable is an idempotent operation. When two threads execute an idempotent operation, the order of these operations does not matter, so atomicity violations involving them are false positives. The user can annotate lines that perform idempotent operations. Our algorithms filter out warnings that involve only these lines.

Choosing atomic regions. We found that many atomicity violations initially reported by the block-based algorithms are benign: the syscalls involved are not atomic, but are not required to be atomic. For example, the `btrfs_file_write()` function in the Btrfs file system loops through each page that it needs to write. The body of the loop, which writes one page, should be atomic, but the entire function does not need to be.

Redflag lets the user break up atomic regions by marking lines of code as *fenceposts*. A fencepost ends the current atomic region and starts a new one. For example, placing a fencepost at the beginning of the page-write loop in `btrfs_file_write()` prevents Redflag from reporting atomicity violations spanning two iterations of the loop. Fenceposts provide a simple way for developers to express expectations about atomicity.

To facilitate fencepost placement, Redflag determines which lines of code, if marked as fenceposts, would filter the most atomicity violations. Any line of code that executes in the same thread as a block between the first and last operations of the block (see Section 2.3 for a description of blocks) can serve as a fencepost that filters all violations involving that block. After the block-based analysis produces a list of atomicity violations with corresponding blocks, fencepost inference proceeds by greedily choosing the fencepost that will filter the most violations, removing these violations from its list, and repeating until no violations remain. The result is a list of potential fenceposts sorted by the number of violations they filter. The user can examine these candidate fenceposts to see whether they lie on the boundaries of logical atomic regions in the code.

3 Evaluation

To evaluate Redflag's accuracy and performance, we exercised it on three kernel components: Btrfs, Wrapfs, and Noveau. Btrfs is a complex in-development on-disk file system. Wrapfs is a pass-through stackable file system that serves as a stackable file system template. Because of the interdependencies between stackable file systems and the underlying virtual file system (VFS), we instrumented all VFS data structures along with Wrapfs's data structures. We exercised Btrfs and Wrapfs with Racer [16], a workload designed to test a variety of file-system system calls concurrently. Nouveau is a video driver for Nvidia video cards. We exercised Nouveau by playing a video and running several instances of `glxgears`, a simple 3D OpenGL example.

Lockset results. Lockset revealed two confirmed locking bugs in Wrapfs. The first bug results from an unprotected access to a field in the `file struct`, which is a VFS data structure instrumented in our Wrapfs tests. A Lockset report shows that parallel calls to the `write` syscall can access the `pos` field simultaneously. Investigating this race, we found an article describing a bug resulting from it: parallel writes to a file may

Table 1. Summary of results of block-based algorithms. From left to right, the columns show: reports caused by `wrapfs_setattr`, reports caused by `touch_atime`, reports caused by reads with no effect, reports involving counting variables, reports caused by coarse-grained reporting of `struct` accesses, and reports that do not fall into the preceding categories. Each column has two sub-columns, with results for the single-variable and two-variable algorithms, respectively. Empty cells represent zero.

	setattr		stat		atime		useless read		counting		struct granularity		untraced lock		other	
Btrfs					5				61	6	2				40	
Wrapfs	34	6	14		43										2	
Nouveau							1				21	2		1		

write their data to the same location in a file, in violation of POSIX requirements [3]. Proposed fixes carry an undesirable performance cost, so this bug remains.

The second bug is in Wrapfs itself. The `wrapfs_setattr` function copies a data structure from the wrapped file system (the *lower inode*) to a Wrapfs data structure (the *upper inode*) but does not lock either inode, resulting in several Lockset reports. We discovered that file truncate operations call the `wrapfs_setattr` function after modifying the lower inode. If a truncate operation's call to `wrapfs_setattr` races with another call to `wrapfs_setattr`, the updates to the lower inode from the truncate can sometimes be lost in the upper inode. We confirmed this bug with Wrapfs developers.

Lockset detected numerous benign races: 8 in Btrfs, and 45 in Wrapfs. In addition, it detected benign races involving the `stat` syscall in Wrapfs, which copies file metadata from an inode to a user process without locking the inode. The unprotected copy can race with operations that update the inode, causing `stat` to return inconsistent (partially updated) results. This behavior is well known to Linux developers, who consider it preferable to the cost of locking [1], so we filter out the 29 reports involving `stat`.

Lockset produced some false positives due to untraced locks: 2 for Wrapfs, and 11 for Noveau. These false positives are due to variable accesses protected by locks external to the traced `structs`. These reports can be eliminated by telling Redflag to trace those locks.

Block-based algorithms results. Table 1 summarizes the results of the block-based algorithms. We omitted four `structs` in Btrfs from the analysis, because they are modified frequently and are not expected to update atomically for an entire syscall. The two-variable block-based algorithm is compute- and memory-intensive, so we applied it to only part of the Btrfs and Wrapfs logs.

For Wrapfs, the `wrapfs_setattr` bug described above causes atomicity violations as well as races; these are counted in the "setattr" column. The results for Wrapfs do not count 86 reports for the file system that Wrapfs was stacked on top of (Btrfs in our test). These reports were produced because we told Redflag to instrument all accesses to targeted VFS structures, but they are not relevant to Wrapfs development.

For Wrapfs, the unprotected reads by `stat` described above cause two-variable atomicity violations, which are counted in the "stat" column. These reads do not cause

Table 2. Number of false positives filtered out by various techniques

	Fenceposts	Bit-level granularity	LOA	Unfiltered
Btrfs	44	0	159	108
Wrapfs	81	6	215	79
Nouveau	-	2	70	22

single-variable atomicity violations, because inconsistent results from `stat` involve multiple inode fields, some read before an update by a concurrent operation on the file, and some read afterwards.

For Noveau, the report in the "Untraced lock" column involves variables protected by the Big Kernel Lock (BKL), which we track.

The "counting" column counts reports whose write accesses are increments or decrements (e.g., accesses to reference count variables). Typically, these reports can be ignored, because the order in which increments and decrements execute does not matter—the result is the same. Our plug-ins mark counting operations in the log, so Redflag can automatically classify reports of this type.

The "struct granularity" column counts reports involving `structs` whose fields are grouped together by Redflag's logging. Accesses to a `struct` that is *not* targeted get logged when the non-targeted `struct` is a field of some `struct` that is targeted and the access is made through the targeted `struct`. However, all the fields in the non-targeted `struct` are labeled as accesses to the field in the targeted `struct`, so they are treated as accesses to a single variable. This can cause false positives, in the same way that bit-level operations can (*cf.* Section 2.5). These false positives can be eliminated by adding the non-targeted struct to the list of targeted structs.

Filtering. Table 2 shows how many reports were filtered from the results of the single-variable block-based algorithm (which produced the most reports) by manually chosen fenceposts, bit-level granularity, and LOA analysis. The "unfiltered" column shows the number of reports not filtered by any of these techniques. We used fewer than ten manually chosen fenceposts each for Btrfs and Wrapfs. Choosing these fenceposts took only a few hours of work. We did not use fenceposts for our analysis of Nouveau because we found that entire Nouveau syscalls are atomic.

LOA analysis is the most effective among these filters. Only a few `structs` in each of the modules we tested go through a multi-stage escape, but those `structs` are widely accessed. It is clear from the number of false positives removed that a technique like LOA analysis is necessary to cope with the complicated initialization procedures in systems code.

Some reports filtered by LOA analysis may be actual atomicity violations, as discussed in Section 2.4. This happened with a bug in Btrfs' inode initialization that we discovered during our experiments. The Btrfs file creation function initializes the new inode's file operations vector just after the inode is linked to a dentry. This linking is the inode's second stage of escape, as discussed Section 2.4. When the dentry link makes the new inode globally available, there is a very narrow window during which another thread can open the inode while the inode's file operations vector is still empty. This

bug is detected by the single-variable block-based algorithm, but the report is filtered out by LOA analysis. LOA analysis will determine that the empty operations vector is available to the `open` syscall only if an open occurs during this window in the logged execution, which is unlikely. Dynamic escape analysis correctly recognizes the possible interleaving in any execution, but has other drawbacks, because it accounts for only one level of escape. In particular, the bug can be fixed by moving the file operations vector initialization earlier in the function: before the inode is linked to a dentry, but still after the inode's first escape. Dynamic escape analysis would still consider the interleaving possible, resulting in a false positive.

We tested the fencepost inference algorithm in Section 2.5 on Btrfs. We limited it to placing fenceposts in Btrfs functions (not, e.g., library functions called from Btrfs functions). The algorithm produced a useful list of candidate fenceposts. For example, the first fencepost on the list is just before the function that serializes an inode, which is reasonable because operations that flush multiple inodes to disk are not generally designed to provide an atomicity guarantee across all their inode operations.

Performance. To evaluate the performance of our instrumentation and logging, we measured overhead with a micro-benchmark that stresses the logging system by constantly writing to a targeted file system. For this experiment, we stored the file system on a RAM disk to ensure that I/O costs did not hide overhead. This experiment was run on a computer with two 2.8GHz single-core Intel Xeon processors. The instrumentation targeted Btrfs running as part of the 2.6.36-rc3 Linux kernel. We measured an overhead of 2.44× for an instrumented kernel without logging, and 2.65× with logging turned on. The additional overhead from logging includes storing event data, copying the call stack, and reserving buffer space using atomic memory operations.

Schedule sensitivity of LOA. Although LOA is very effective at removing false positives, it is sensitive to the observed ordering of events, potentially resulting in false negatives, as discussed in Section 2.4. We evaluated LOA's sensitivity to event orderings by repeating a workload under different configurations: single-core, dual-core, quad-core, and single-core with kernel preemption disabled. We then analyzed the logs with the single-variable block-based algorithm. The analysis results were quite stable across these different configurations, even though they generate different schedules. The biggest difference is that the non-preemptible log misses 13 of the 201 violations found in the quad-core log. There were only three violations unique to just one log.

4 Related Work

A number of techniques, both runtime and static, exist for tracking down difficult concurrency errors. This section discusses tools from several categories: runtime race detectors, static analyzers, model checkers, and runtime atomicity checkers.

Runtime race detection. Our Lockset algorithm is based on the Eraser algorithm [15]. Several other variants of Lockset exist, implemented for a variety of languages. LOA analysis is the main distinguishing feature of our version. Some features of other race

detectors could be integrated into Redflag, for example, the use of sampling to reduce overhead, at the cost of possibly missing some errors, as in LiteRace [11].

Microsoft Research's DataCollider [5] is the only other runtime data race detector that has been applied to an OS kernel, to the best of our knowledge. Specifically, it has been applied to several modules in the Windows kernel and detected numerous races. It detects actual data races when they occur, in contrast to Lockset-based algorithms that analyze synchronization to detect possible races. At runtime, DataCollider pauses a thread about to perform a memory access and then uses hardware watchpoints to intercept conflicting accesses that occur within the pause interval. This approach produces no false positives but may take longer to find races and may miss races that happen only rarely. DataCollider uses sampling to reduce overhead.

Static analysis. Static analysis tools, typically based on the Lockset approach of finding variables that lack a consistent locking discipline, have uncovered races even in some large systems. For example, RacerX [4] and RELAY [17] found data races in the Linux kernel. Static race detection tools generally produce many false positives, due to the well-known difficulties of analyzing aliasing, function pointers, calling context, etc.

Static analysis of atomicity has been studied (e.g., [7,14]) but not applied to large systems software. Generally, these analyses check whether the code follows certain safe synchronization patterns.

Runtime atomicity checking. To the best of our knowledge, we are the first to apply a runtime atomicity checker to components of an OS kernel. Although we used the block-based algorithms, other runtime techniques for checking atomicity and similar properties could be adapted to work on Redflag's logs. Atomicity checkers based on Lipton's reduction theorem [9,6,19] are computationally much cheaper than the block-based algorithms, because they check a simpler condition that is sufficient but not necessary for ensuring atomicity. As a result, however, they usually produce more false positives.

AVIO [10] and CTrigger [13] use heuristics to infer programmers' expectations about atomicity, and then check for violations thereof (i.e., atomicity violations). An important difference from our work is that the block-based algorithm reports potential and actual atomicity violations, while AVIO and CTrigger report only actual atomicity violations (i.e., atomicity violations that manifest in the monitored run). They actively perturb the schedule to increase the likelihood that atomicity bugs will manifest during testing. Also, they do not detect atomicity violations involving multiple variables. As a result, they are computationally cheaper and produce fewer false positives, but they are more schedule-sensitive and may miss bugs that the block-based algorithms would report. Their implementations use binary instrumentation and are not integrated with the compiler, so it would be difficult to target their analysis to specific data structures.

5 Conclusions

We have described the design of Redflag and shown that it can successfully detect data races and atomicity violations in components of the Linux kernel. To the best of our knowledge, Redflag is the first runtime race detector applied to the Linux kernel, and the first runtime atomicity detector for any OS kernel.

Redflag's runtime analyses are designed to detect potential concurrency problems even if actual errors occur only in rare schedules not seen during testing. The analyses are based on well-known algorithms but contain a number of extensions that significantly improve accuracy, such as LOA analysis. Although the cost of thorough logging can be high, we have shown that Redflag's performance is sufficient to capture traces that exercise many system calls and execution paths.

Future work. We plan to extend Redflag with dynamic escape analysis and active analysis (i.e., schedule perturbation) and experiment with the interaction between these techniques and LOA analysis. We also plan to extend Redflag with an analysis that identifies where memory barriers are needed. Memory barriers, which are usually necessary only in low-level systems code, prevent memory operation reorderings that would otherwise be allowed by the weak (not sequentially consistent) memory models used in modern compilers and processors. Another direction for future work is to apply Redflag for performance improvement of concurrent code. By examining locking and access patterns in execution logs, Redflag could identify critical sections that can employ double-checked locking and data structures that would benefit from RCU use. We plan to release the entire Redflag framework and tools publicly under an open source license.

Acknowledgements. Research supported in part by NFS grants CNS-0509230 and CNS-0831298, AFOSR grant FA0550-09-1-0481, and ONR grant N00014-07-1-0928.

References

1. Bacik, J.: Possible race in btrfs (2010), http://article.gmane.org/gmane.comp.file-systems.btrfs/5243/
2. Callanan, S., Dean, D.J., Zadok, E.: Extending GCC with modular GIMPLE optimizations. In: Proceedings of the 2007 GCC Developers' Summit, Ottawa, Canada (July 2007)
3. Corbet, J. write(), thread safety, and POSIX, http://lwn.net/Articles/180387/
4. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 237–252. ACM Press, New York (2003)
5. Erickson, J., Musuvathi, M., Burckhardt, S., Olynyk, K.: Effective data-race detection for the kernel. In: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, Berkeley (2010)
6. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: POPL 2004: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 256–267. ACM, New York (2004)
7. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proc. ACM SIGPLAN Conference on Programming Language Design and IMPLEMENTATION (PLDI), pp. 338–349. ACM Press, New York (2003)
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
9. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
10. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting atomicity violations via access interleaving invariants. In: ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 37–48. ACM, New York (2006)

11. Marino, D., Musuvathi, M., Narayanasamy, S.: LiteRace: Effective sampling for lightweight data-race detection. In: PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 134–143. ACM, New York (2009)
12. McKenney, P.E.: What is RCU? (2005), http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.33.y.git;a=blob;f=Documentation/RCU/whatisRCU.txt
13. Park, S., Lu, S., Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. In: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 25–36. ACM, New York (2009)
14. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analaysis of data races and atomicity. In: Proceedings of the Tenth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming (June 2005)
15. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: ERASER: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems 15(4), 391–411 (1997)
16. Modak, S.: Linux Test Project, LTP (2009), http://ltp.sourceforge.net/
17. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: FSE 2007: Proceedings of the 6th ESEC/SIGSOFT International Symposium on Foundations of Software Engineering, pp. 205–214. ACM, New York (2007)
18. Wang, L., Stoller, S.D.: Run-time analysis for atomicity. In: Proceedings of the Third Workshop on Runtime Verification (RV). Electronic Notes in Theoretical Computer Science, vol. 89(2), Elsevier, Amsterdam (2003)
19. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Softw. Eng. 32(2), 93–110 (2006)

Exploiting Parallelism in the H.264 Deblocking Filter by Operation Reordering

Tsung-Hsi Weng, Yi-Ting Wang, and Chung-Ping Chung

Department of Computer Science, National Chiao-Tung University,
1001 University Road, Hsinchu, Taiwan, ROC
{chwong,yitwang,cpchung}@cs.nctu.edu.tw

Abstract. In the H.264 video compression standard, the deblocking filter contributes about one-third of all computation in the decoder. With multi-processor architectures becoming the future trend of system design, computation time reduction can be achieved if the deblocking filter well apportions its operations to multiple processing elements. In this paper, we apply a 16 pixel long boundary, the basic unit for deblocking in the H.264 standard, as the basis for analyzing and exploiting possible parallelism in deblocking filtering. Compared with existing approaches using a macroblock as a basic unit for analysis, a 16 pixel long boundary by having a finer granularity can improve the chances of increasing the degree of parallelism. Moreover, a possible compromise to fully utilize limited hardware resources and hardware architectural requirements for deblocking are also proposed in this paper. Compared with the 2D wave-front method order for deblocking both 1920*1080 and 1080*1920 pixel sized frames, the proposed design gains speedups of 1.57 and 2.15 times given an un-limited number of processing elements respectively. Using this approach, the execution time of the deblocking filter is proportional to the square root of the growth of the frame size (keeping the same width/height ratio), pushing the boundary of practical real-time deblocking of increasingly larger video sizes.

Keywords: deblocking, parallelization, multi-core.

1 Introduction

The H.264 standard provides acceptable image quality combined with a reduction in bit-rate compared with existing video compression standards. Besides this, it can also provide higher adaptability and better error resilience for a wider range of applications. With regards to the compression rate, the bit rate of H.264 is almost 50% lower than that of the MPEG-2, H.263v2 and MPEG-4 Advanced Simple Profile video compression standards for the same picture quality [7].

Deblocking is intended to smooth block-edge artifacts caused by the decoding process and enhance picture quality. In the encoding process, the H.264 encoder uses the macroblock (MB, 16x16 pixel square) as the basic coding unit. Quantization of the macroblocks causes visual discontinuities between the edges of decoded macroblocks. Pixels located on macroblock boundaries with a similar value may for

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 80–92, 2011.

the above reason be decoded with a larger difference in values, resulting in a decline in picture quality. Therefore, the purpose of deblocking is to smooth block artifacts caused by the decoding process to enhance picture quality. Another advantage of deblocking is to increase coding efficiency. Decoded and deblocked images will be referenced later, and because the picture is of higher quality, there will be a reduction in the encoded bit rate.

Deblocking filtering accounts for one-third of all computation in the decoder [1]. With multi-core becoming the trend, if deblocking can be processed using a multi-core parallel processing architecture, the processing can be distributed to different computing processing elements (PEs) to address and reduce execution time. Currently parallel processing of deblocking focuses on parallelization at the MB-level. We found that parallelizing deblocking at a finer granularity can be developed according to our presented design.

We analyze the deblocking order to obtain the dependency between the various boundaries, and then propose an execution order, with execution of deblocking in this order giving higher parallelism.

The rest of this paper is organized as follow. Section 2 introduces the background of the deblocking filter and related work for deblocking filter parallelization. Section 3 shows our parallelized design. Section 4 analyzes the proposed method and compares it with related works. Section 5 shows our proposed hardware architectural requirements. Finally, the conclusion is given along with further work.

2 Background

2.1 H.264 Deblocking Filter

The deblocking filter is used in order to smooth block-edge artifacts. Figure 1 shows the idea.

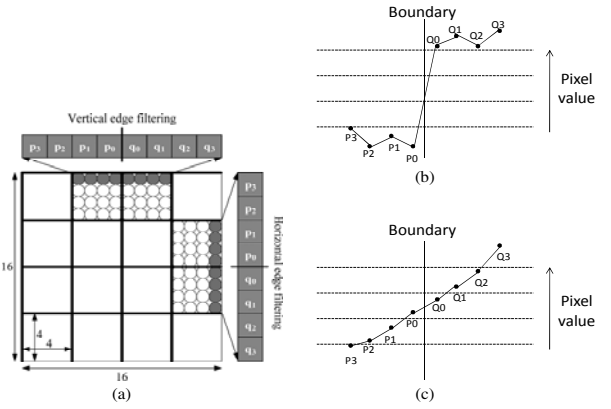


Fig. 1. (a) Affected pixels in deblocking (b) The pixel values before deblocking filtering; the P0~P3 and Q0~Q3 pixel value gap causes a visual discontinuity. (c) After deblocking filtering; the pixel values are now smooth.

Figure 1(b) shows a block-edge artifact caused by a large difference in luma values. The pixels P0~P3 and Q0 ~Q3 in Fig. 1(b) can be located either vertically or horizon-tally as shown in Fig. 1(a). A deblocking filter is applied on the P0~P3 and Q0~Q3 pixel luma values to make these eight values visually smooth. The pixel luma value distribution after applying the deblocking filter is shown in Fig. 1(c).

Deblocking is needed for both MB boundaries and 4*4 block boundaries. As the MB is the basic coding unit in H.264, block-edge artifacts occur easily at MB boundaries. In addition, there are some coding modes using 4*4 blocks for inter prediction and intra prediction. For these cases deblocking is needed to smooth the block-edge artifacts.

The MB deblocking internal (intra MB) execution order as defined by the H.264 standard is shown in Fig. 2(a). Execution starts by deblocking a column of pixels moving horizontally left to right, and then a row of pixels moving vertically top to bottom. The inter MB execution order is shown in Fig. 2(b), and moves from left to right, top to bottom.

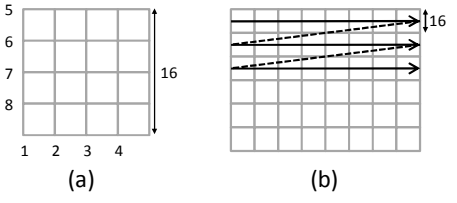


Fig. 2. (a) Intra MB order. (b) Inter MB order.

Although the H.264 standard defines the deblocking order as shown above, as long as the final decoding results in the correct output, the above order can be changed. Changing the order in which the calculation is performed is an opportunity for parallelizing deblocking filtering. We propose a conceptual design to improve the parallelizability of the deblocking filter.

2.2 Related Work

The 2D wave-front method is based on using the MB as a unit for parallelization [2].

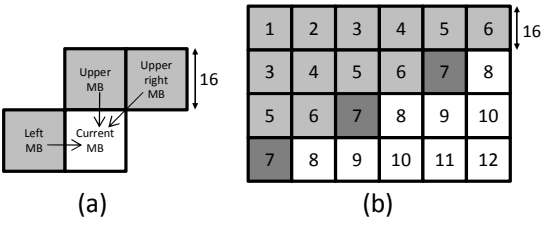


Fig. 3. (a) Data dependencies in inter MB deblocking. (b) MBs that can be processed simultaneously.

In Fig. 3(a), according to the deblocking order, we find the current MB has a data dependency on the Upper, Upper-Right and Left MBs. So using an MB as the parallelization unit, the Upper, Upper-Right and Left MB must be deblocked before the Current MB. In Fig. 3(b), MBs that can be processed simultaneously are numbered together.

According to observation, this method does not have a fixed degree of parallelism. The degree of parallelism initially steadily increases. Some startup time is needed before reaching maximum parallelism. After maintaining maximum parallelism for some time, the degree of parallelism will begin to steadily decrease. In Fig. 4, the units of time are in terms of the time to deblock one MB, and the frame size is 1920*1080.

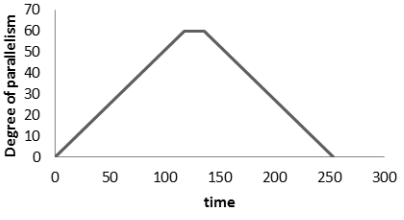


Fig. 4. Time and parallelism relationship. The vertical axis is the number of MBs processed in parallel, the horizontal axis is time. The time unit here is time required for deblocking a MB.

The 2D wave-front method's maximum parallelism and required startup time and ending time can be expressed by the equation:

$$\text{Maximum parallelism (P)} = \text{Min}\left(\left\lceil\frac{1}{2}\text{M}_\text{W}\right\rceil, \text{M}_\text{H}\right)$$

$$\text{Startup time and Ending time} = 2 * (\text{P} - 1)$$

Where the M_W is the number of columns of MBs in frame and M_H is the number of rows of MBs in frame.

The 3D wave-front method [3] is based on the 2D wave-front method, but also uses inter frame parallelism, meaning more MBs can be processed in parallel. This method can significantly enhance the parallelism. In Fig. 5 [4], the dark gray MBs can be processed in parallel.

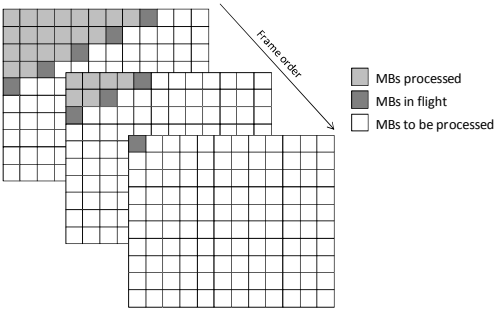


Fig. 5. The dark gray MBs can be processed in parallel [4]

3 Algorithm

Analyzing applications at a finer granularity usually opens extra opportunities for parallelization. In H.264, the standard defines the order for deblocking using a 16 pixel long boundary as its basic unit. As a result, in this section we analyze the data dependencies within the deblocking filter, and then propose our deblocking order and design.

3.1 Analysis of Data Dependencies

We separate the data dependencies when using a 16 pixel long boundary for deblocking into 3 cases:

Case 1: Intra MB 16 Pixel Long Boundary Data Dependencies. In Fig. 6(a), the result after deblocking MB_{b1} (boundary b1) is input into the deblocking filter for MB_{b2}, with that result then becoming the input into the deblocking filter for MB_{b3} and so on. Through this analysis the data dependency chain is MB_{b1}⇨MB_{b2}⇨MB_{b3} ⇨MB_{b4} and MB_{b5}⇨MB_{b6}⇨MB_{b7}⇨MB_{b8}. Moreover, the deblocking result of MB_{b4} is input to MB_{b5}, so the data dependency chain for intra MB deblocking is MB_{b1}⇨MB_{b2}⇨MB_{b3}⇨MB_{b4}⇨MB_{b5}⇨MB_{b6}⇨MB_{b7}⇨MB_{b8} as shown in Fig. 6(b).

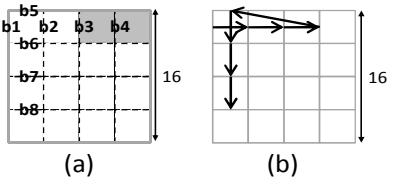


Fig. 6. (a) Intra MB deblocking execution order, the gray blocks are data dependencies from boundary 4 to boundary 5. (b) The data dependency chain for intra MB deblocking.

Case 2: Same Row Inter-MB 16 Pixel Long Boundary Data Dependencies. In Fig. 7, part of the deblocking result of Current MB_{b8} (the gray blocks) is the deblocking input to Right MB_{b1}, so Right MB_{b1} depends on Current MB_{b8}. In other words, Right MB_{b1} can begin execution after the Current MB_{b8} has completed execution. This shows that using 16 pixel long boundaries, MBs within the same row cannot be deblocked at the same time.

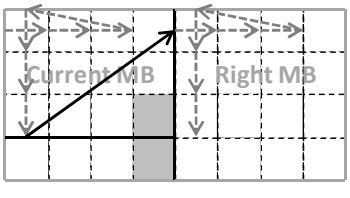


Fig. 7. The deblocking data dependency chain for MBs in the same row

Case 3: Adjacent Row Inter-MB 16 Pixel Long Boundary Data Dependencies. In Fig. 8, the deblocking input of Current MB_{b5} needs 4 4x4 blocks from Upper MB (gray blocks). According to Case 2, we find that the dark gray block is the last to be modified. The dark gray block is modified by deblocking Upper-right MB_{b1} after which it is able to become the deblocking input to Current MB_{b5}. Therefore, Current MB_{b5} depends on Upper-right MB_{b1}, with the data dependency chain shown as a black arrow in Fig. 8.

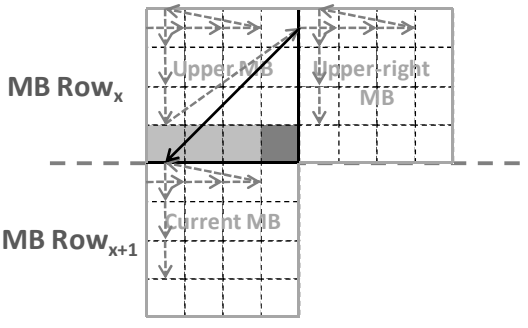


Fig. 8. The data dependency chain between adjacent rows of MBs

3.2 Proposed Deblocking Order

According to the above 3 cases, we propose a new execution order. This order fulfills the required data dependencies whilst providing an extra degree of deblocking parallelism. The time that deblocking is performed on each 16 pixel long boundary is shown in Fig. 9(a). If the time of execution for deblocking Current MB_{b1} is t, by the above Case 2 the execution time of Right MB_{b2} is t+9, by Case 3 the execution time of Lower MB_{b5} is also t+9, and by Case 1 the execution time of Lower MB_{b1} is t+5.

If the time of execution of Current MB_{b1} is t, it shows in Fig. 9(b) the execution time of Lower MB_{b1} is t+5 in our proposed order, and the execution time of Lower MB_{b1} is t+16 in the 2D wave-front method.

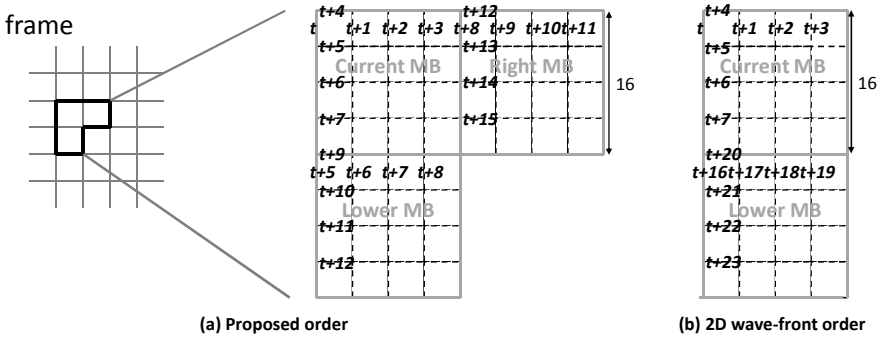


Fig. 9. Comparison of (a) Proposed execution order, (b) 2D wave-front order

According to Case 2 mentioned above, when deblocking on 16 pixel long boundaries within the same MB row, MBs cannot be deblocked at the same time. As a result, we can assign one PE to each row of MBs. Due to the relationship between the number of PEs and the aspect of the frame to be deblocked, there are two cases that can occur:

Case I: Degree of Parallelism Depends on Frame Aspect. Assuming there are more PEs than needed, the degree of parallelism will be limited only by the frame aspect. While processing 16 pixels horizontally (the width of one MB) takes 8 stages, processing 16 pixels vertically takes only 5 stages in the proposed order. As a result, deblocking of the first row of MBs will finish before starting the last row of MBs, if the number of rows of MBs is less than (8/5) × the number of columns of MBs in a frame. We categorize the effects of frame aspect ratio into the following two situations:

*Situation 1. # rows of MBs in frame ≤ 8/5 * # columns of MBs in frame (Degree of parallelism limited by # rows of MBs in frame).* In this situation, the maximum parallelism is equal to the number of rows of MBs in the frame. Our method has a startup and ending time similar to the 2D wave-front method. A diagram is shown in Fig. 10(a) to help explain. The upper-left gray region is the starting up of the deblocking filter, and the lower-right gray region is the finishing of the deblocking filter. In these regions, the deblocking filter is not able to reach maximum parallelism. The white region is where the deblocking filter is able to reach maximum parallelism. The degree of parallelism and timing relationship diagram is shown in Fig. 10(b).

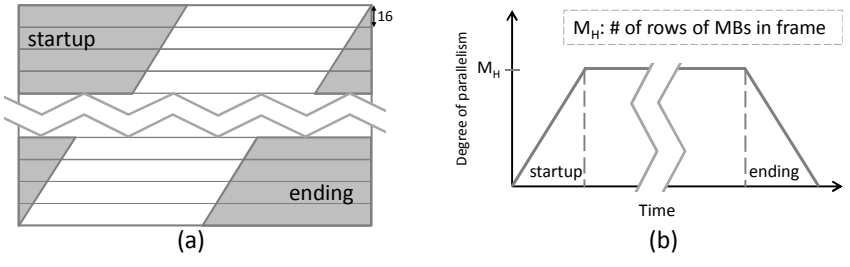


Fig. 10. (a) Zones of startup and ending of deblocking order and (b) the corresponding degree of parallelism and time relationship diagram. The time unit is the time required for deblocking a 16 pixel long boundary.

*Situation 2. # rows of MBs in frame > 8/5 * # columns of MBs in frame (Degree of parallelism limited by # rows of MBs in frame).* In this situation shown in Fig. 11(a), the degree of parallelism is equal to the number of rows of MBs that can start their deblocking before the deblocking has completed for the first row of MBs. As explained in the beginning of case I, if the ratio of the height to width is larger than 8/5, the degree of parallelism will be limited by the frame width. The degree of parallelism and timing relationship diagram is shown in Fig. 11(b).

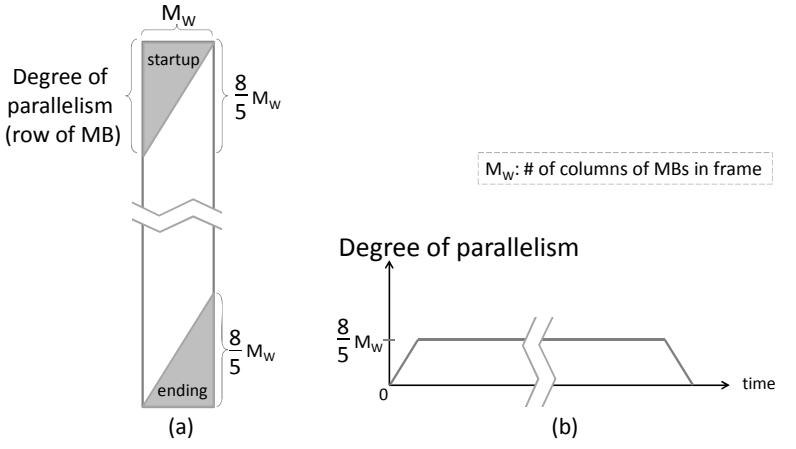


Fig. 11. (a) The degree of parallelism is limited when the frame height is larger than 8/5 times of the frame width. (b) The degree of parallelism and timing relationship diagram.

Case II: # of PEs Not Enough for Maximum Parallelism. In this case, the frame has to be split into multiple pieces for deblocking. Here we first show a naive approach, and then propose an improved one.

Naive Approach. In Fig. 12(a), assume the number of PEs is K, and then divide the frame into pieces where each piece contains K rows of MBs. The execution order of the pieces is from top to bottom. We find that each piece has a startup and ending time, meaning PEs remaining idle often occurs. The degree of parallelism and timing diagram is shown in Fig. 12(b).

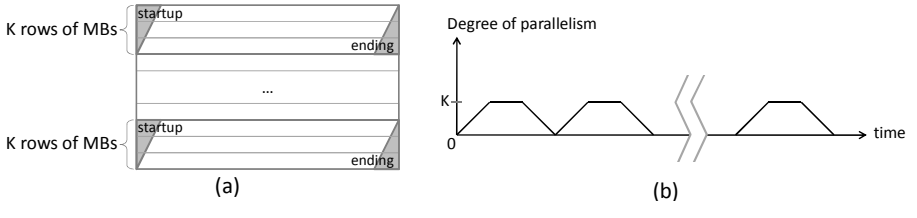


Fig. 12. When # of PEs is not enough for maximum parallelism, (a) frame is split into multiple pieces for deblocking, and (b) PEs idle often occurs

Improved Approach. In the naive approach, PEs are frequently idle between the deblocking of pieces. But after analyzing the details, we find that the execution of the ending of one piece and the startup of the next piece can be overlapped to fully utilize the PEs. They are able to be overlapped because there are no direct data dependencies between the ending of this piece and the startup of the next piece. Fig. 13 shows a reduction in the idle time of PEs after this overlapping.

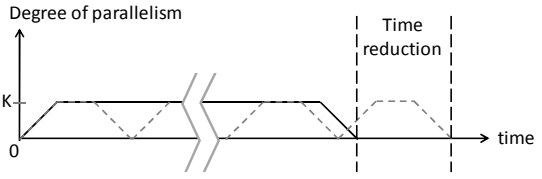


Fig. 13. The degree of parallelism and timing relationship after overlapping

4 Analysis

The proposed order has been shown in the previous section, so the focus of this section is on determining the degree to which parallelism and execution time can be improved from this design. In this section, we first model the parallelism and time of deblocking a frame for both the proposed order and 2D wave-front method order. Then the time required will be compared using both 1920*1080 and 1080*1920 pixel sized frames as examples. After that, we construct figures to show the effects of the number of PEs and the benefits from overlapping the deblocking of adjacent rows of MBs. In the end, we explain that our design is also complementary to the 3D wave-front method.

The proposed execution order's maximum parallelism and required start-up time, ending time and execution time can be expressed by the equations:

Maximum parallelism of a frame (P_F)
= Maximum # of rows of MBs that can be deblocked in parallel

$$P_F = Min\left(M_H, \left\lceil \frac{8}{5} M_W \right\rceil\right)$$

Maximum parallelism (P)
= Min (Maximum parallelism of a frame, # of available PEs)

$$P = Min\left(M_H, \left\lceil \frac{8}{5} M_W \right\rceil, \#\ of\ PEs\right)$$

Startup time
= Time to reach the row of maximum parallelism
= (delay between processing rows) × (maximum parallelism - 1)

$$\text{Startup time} = 5 \times (P - 1)$$

Ending time

$$= \begin{cases} \text{Time after finishing the last Pth row of MBs.} & \text{, if enough PEs} \\ \text{Time after finishing 1st row of MBs in last piece.} & \text{, if limited PEs} \end{cases}$$

$$= \begin{cases} (\text{delay between processing rows}) \times (\text{Max. parallelism} - 1) & \text{, if \#of PEs} \geq P_F \\ (\text{delay between processing rows}) \times (\text{\# rows in last piece} - 1) & \text{, if \#of PEs} < P_F \end{cases}$$

$$\text{Ending time} = \begin{cases} 5 \times (P-1) & \text{, if \#of PEs} \geq P_F \text{ or \# rows in last piece} = P \\ 5 \times ((M_H \bmod P) - 1) & \text{, otherwise} \end{cases}$$

Total Execution time

$$= \begin{cases} 5 \times (M_H - 1) + 8 \times M_W & \text{, if \#of PEs} \geq P_F \\ 8 \times M_W \times \left\lceil \frac{M_H}{P} \right\rceil + 5 \times (P-1) & \text{, if \#of PEs} < P_F \text{ \# rows in last piece} = P \\ 8 \times M_W \times \left\lceil \frac{M_H}{P} \right\rceil + 5 \times ((M_H \bmod P) - 1) & \text{, if \#of PEs} < P_F \text{ \# rows in last piece} \neq P \end{cases}$$

Above time unit is the time required for deblocking a 16 pixel long boundary. In order to compare the proposed method with the 2D wave-front method, we have to modify the original equations using following rules. First, taking the number of PEs into consideration; and second, adjusting the time unit. Deblocking one macroblock is equivalent to deblocking eight 16 pixel long boundaries, so assuming the computation power of all PEs are the same, we multiply the time by 8.

For the case of degree of parallelism depends on frame aspect, the proposed method's execution time, startup time, ending time and degree of parallelism is shown in comparison with the 2D wave-front method for both (a) 1920*1080 and (b) 1080*1920 pixel sized frames in Fig. 14.

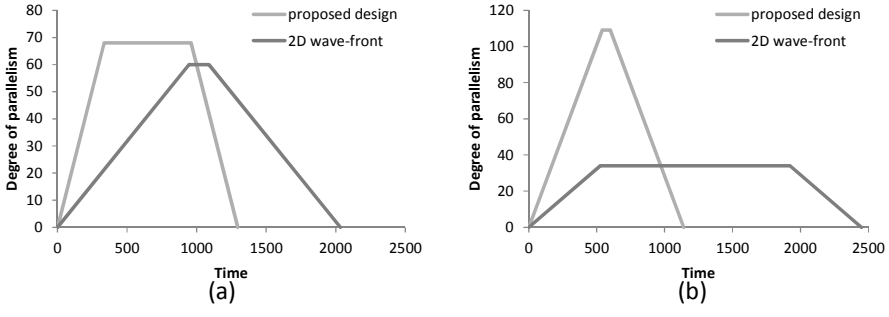


Fig. 14. Proposed method compared with the 2D wave-front method in degree of parallelism and time for deblocking when frame size is (a) 1920*1080 and (b) 1080*1920. The time unit is the time required for deblocking a 16 pixel long boundary.

Observing Fig. 14, we find the proposed method has a faster startup time and ending time than the 2D wave-front method, which means a faster execution time. The reason for the higher degree of parallelism than the 2D wave-front is the frame aspect ratio. In the proposed method, while the first row of MBs is being processed, the last row of MBs begins to be processed, so the theoretical maximum degree of parallelism is achieved. For both 1920*1080 and 1080*1920 pixel sized frames, the proposed design is 1.57 and 2.15 times faster than the 2D wave-front method given an un-limited number of processing elements respectively.

To show the effects of the number of PEs and the benefits from overlapping the deblocking of adjacent rows of MBs, two figures have been constructed. Figure 15(a)

shows the total execution time for the following 4 configurations for a 1920*1080 frame size, and Fig. 15(b) is similar but for a 1080*1920 frame size:

- 2D wave-front method without deblocking overlapping,
- 2D wave-front method with deblocking overlapping,
- Proposed method without deblocking overlapping,
- Proposed method with deblocking overlapping.

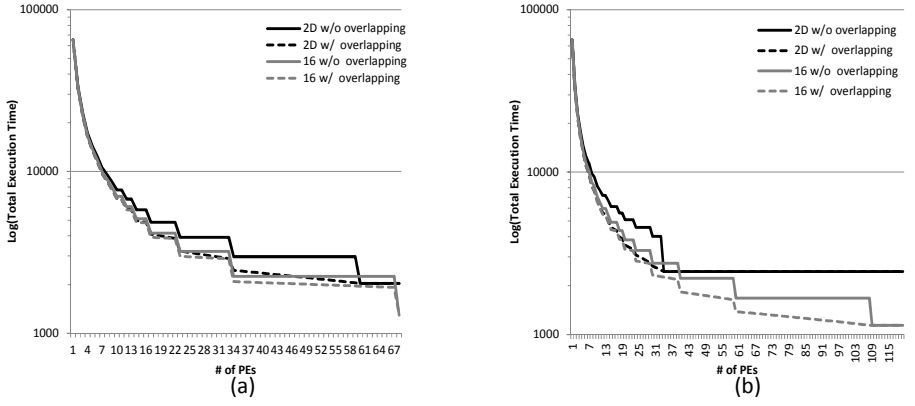


Fig. 15. Proposed method compared with the 2D wave-front method in time for deblocking and number of PEs when frame sizes are (a) 1920*1080 and (b) 1080*1920

We find that for both the 2D wave-front method and proposed method, the greater the number of PEs, the greater the benefit to the time to deblock a frame. However, our proposed method gains more benefit than the 2D wave-front method which comes from the shorter startup and ending time requirement, especially for the vertically shaped frame. Figure 15(b) shows the evidence that while the speedup of the 2D wave-front method stops at 34 PEs, the speedup of our proposed method keeps improving until 109 PEs.

In Fig. 15, we find the total execution time curves have a step-like pattern. This characteristic comes from the splitting of frames. When the number of PEs passes a threshold in which the number of PEs can divide evenly into the total number of MB rows, the total execution time is greatly reduced, thus forming the curves.

Moreover, we find that whilst overlapping benefits both designs, it is actually more useful for the 2D wave-front method because it has a longer startup and ending time.

Last but not least, when considering if our design is complementary with the 3D wave-front method as the 2D wave-front is, the answer is yes. Due to the deblocking filter having no inter-frame data dependencies, our approach is definitely complementary with the 3D wave-front method.

5 Architectural Requirements

In order to deblock a video frame in the proposed order, some hardware support may be necessary. In this section, we list some major hardware requirements such as dedicated

buses between PEs, data loop-backs, and internal buffers. Any hardware that fulfills these requirements should be capable of gaining the benefits from proposed order.

The following are the requirements for the hardware design:

1. As mentioned in case 1 and 2 of section III, 16 pixel long boundaries a row of MBs are required to be deblocked in sequential order, so we can assign one PE for each row of MBs. While deblocking a row of MBs, some intermediate pixel values should be looped back to the PE itself or be kept in internal buffers for further use later.
2. As mentioned in case 3 of section III, we know that the deblocking of every MB_{b5} requires the pixel values that come from its Upper and Upper-right MB. Since we assign a PE to the deblocking of a row of MBs, we need a dedicated bus for data bypassing between PEs dealing with adjacent rows of MBs. This bus can be unidirectional from the upper PE to its adjacent lower PE.

A schematic of the hardware architectural requirements is shown in Fig. 16. The Input Buffer stores pixels that are not yet deblocked, the Output Buffer stores pixels that have been deblocked, and the Internal Buffer stores pixels whose value is needed later. Moreover, PEs will use a shared bus to access memory.

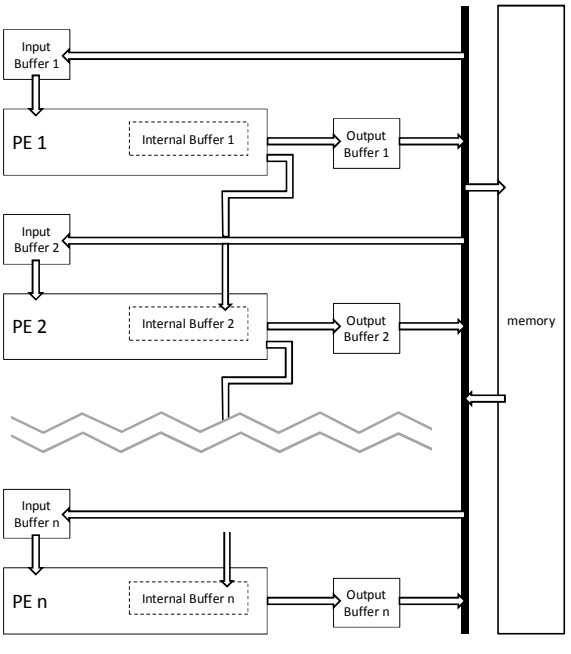


Fig. 16. Schematic of hardware architectural requirements

6 Conclusion and Future Work

As shown in our proposed order, examining the deblocking algorithm at a finer granularity did bring additional opportunities for exploiting parallelism, and thus

speed up the execution time of the deblocking filter. Compared with the 2D wave-front method order in deblocking both 1920*1080 and 1080*1920 pixel sized frames, we gain a speedup of 1.57 and 2.15 times given an un-limited number of PEs respectively. Besides this, we also provided hardware architectural requirements to arrange our PEs in an efficient way to fulfill the requirements of the proposed order. For an environment with limited hardware resources, we also provide an algorithm able to fully utilize available resources for the deblocking filter.

Considering the trend of digital video codecs, larger frame sizes and reduced coded video size are both essential. In order to achieve this goal, the deblocking filter plays an important role because dealing with larger frames takes time proportional to the frame size. The proposed design can limit the growth in time spent deblocking by the maximum of the frame width and height, which are often proportional to the square root of the frame size. Thus it brings the opportunity for practical real-time deblocking of larger sized videos in the future.

Can finer granularity always bring more parallelism for the deblocking filter? This may be true. We can still further analyze the deblocking algorithm using a 4 pixel long boundary or one pixel long boundary to see if there are any opportunities using a similar approach to that in this paper.

References

1. List, P., Joch, A., Lainema, J., Bjontegaard, G., Karczewicz, M.: Adaptive deblocking filter. IEEE Transactions on Circuits and Systems for Video Technology 13(7), 614–619 (2003)
2. Van der Tol, E., Jasper, E., Gelderblom, R.H.: Mapping of H.264 Decoding on a Multiprocessor Architecture. In: Proceeding of SPIE Conference on Image and Video Communications 2003, pp. 707–709 (2003)
3. Meenderinck, C., Azevedo, A., Alvarez, M., Juurlink, B., Ramirez, A.: Parallel Scalability of H.264. In: Proc. First Workshop on Programmability Issues for Multi-Core Computers (January 2008)
4. Zhao, Z., Liang, P.: Data partition for wavefront parallelization of H.264 video encoder. In: IEEE International Symposium on Circuits and Systems (2006)
5. Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264/ISO/IEC 14496-10 AVC) (March 2003)
6. Xu, K., Choy, C.-S.: A Five-Stage Pipeline, 204 Cycles/MB, Single-Port SRAM-Based Deblocking Filter for H.264/AVC. IEEE Transactions on Circuits and Systems for Video Technology 18(3), 363–374 (2008)
7. Chang, Y.-S.: Improvements of H.264 De-blocking filter and DST Implementation of H.264 Decoder. A Thesis Submitted to Institute of Electrical Engineering National Yunlin University of Science & Technology in Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical Engineering (July 2007)

Compiler Support for Concurrency Synchronization

Tzong-Yen Lin, Cheng-Yu Lee, Chia-Jung Chen, and Rong-Guey Chang

Department of Computer Science, National Chung Cheng University
{lty93,lcyu95m,ccj98p,rgchang}@cs.ccu.edu.tw

Abstract. How to write a parallel program is a critical issue for Chip multi-processors (CMPs). To overcome the communication and synchronization obstacles of CMPs, transactional memory (TM) has been proposed as an alternative for controlling concurrency mechanism. Unfortunately, TM has led to seven performance pathologies: DuelingUpgrades, FutileStall, StarvingWriter, StarvingElder, SerializedCommit, RestartConvoy, and FriendlyFire. Such pathologies degrade performance during the interaction between workload and system. Although this performance issue can be solved by hardware, the software solution remains elusive. This paper proposes a priority scheduling algorithm to remedy these performance pathologies. By contrast, the proposed approach can not only solve this issue, but also achieve higher performance than hardware transactional memory (HTM) systems on some benchmarks.

1 Introduction

Using lock synchronization to write a parallel program on chip multi-processors (CMPs) is difficult and error-prone. Thus TM has been proposed as an alternative way to control concurrency mechanism [11]. For lock-based synchronization, programmers must lock and unlock the shared data carefully in a parallel program to avoid significant performance degradation, deadlock, and livelock. Conversely, TM prevents deadlock and livelock entirely by employing priority mechanism. High priority transactions can abort the conflicting transactions with lower priority, and the mechanism can also prevent priority inversion.

There are three major manageable mechanisms in TM: version management (VM), conflict detection (CD), and conflict resolution (CR). In HTM, there are read-sets and write-sets to control the concurrency mechanism. The selection of storing the new data in memory or a log is managed using a VM mechanism. Determining whether the data version in the read-sets and write-sets is conflict or not is called CD. When conflict occurs, we must achieve a CR to abort one of the transactions and allow the program proceed to execute.

Based on VM, CD, and CR, we can specify these design points as LL (Lazy CD/Lazy VM/Committer Wins) systems, EL (Eager CD/Lazy VM/Requester Wins) systems, and EE (Eager CD / Eager VM / Requester Stalls) systems. Specifically, there is currently no Lazy CD/Eager VM system on hardware transactional memory.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 93–105, 2011.

Jayaram Bobba et al. [4] reported that TM spends more time on useless execution behaviors including abort, commit, stall, and backoff. They also indicated the performance pathologies that harm performance of TM programs and presented a hardware solution to solve these pathologies. Most previous research has accelerated programs using hardware to improve TM program performance, and only few software transactional memory (STM) approached was presented. Scherer and Scott [12] indicated that conflict resolution is the critical goal in STMs to avoid many pathologies to degrade performance. Although HTM achieves high performance, modifying hardware architecture is costly. In this paper, we endeavor to propose another way to accelerate TM programs via parsing transactions in advance using software, and apply a priority scheduling scheme to resolve conflict for TM. Using software costs less than using hardware and our approach can achieve the same, even higher performance by comparing our work with the work presented in [4].

Section2 is the related research. We describe the performance pathologies of HTM and a brief introduction of hardware solution [4] in Section3. Section 4 presents the proposed priority scheduling approach. Finally, we describe the implementation details and show the result.

2 Related Work

HTM systems may have to modify processors, cache and bus protocol, to control transactions such as LogTM variants [15,18], LTM [2], TCC [8], and Bulk [6]. The Log-based TM is based on EE systems and uses MOESI directory protocol. It is fast for commit and can detect conflicts quickly. However, when transactions abort, they must roll back the old values. Furthermore, the LL system is similar to both the work presented in Bulk[6] and TM coherence and consistency (TCC)[8]. TM coherence and consistency (TCC) stores the new values in the processor's L1 cache. If transactions commit, TCC overwrites the L2 cache and memory. When transactions commit, TCC detects any conflicts. The EL system's sample is LTM [2], which allows the old value to remain in the main memory and stores the new value in the cache. LTM detects conflict for each memory reference. STM provides transactional memory semantics in a runtime library or the programming language [5], and requires some hardware support such as an atomic compare and swap operation.

Some research has examined TM with priority. First, Justin and Daniel [9] extended a TM contention manager for user-defined priority-based transactions. Their system focused on real systems or some strict systems that require more restrictions. Our method does not restrict systems based on type. In addition, Karma [12,13] has tracked the cumulative number of blocks opened by a transaction as its priority. When a transaction commits, it resets the priority values to zero. When the transaction opens the block, it increases their priority values. The contention manager compares the transactions' priority and aborts the smaller one when transactions conflict. When transaction retries, it will increase the transaction's priority value. The most noteworthy difference with this

manager and our approach is that their manager does not analyze the transactions in advance. We analyze the transactions to obtain more information to modify the priority values.

3 Background

The seven performance pathologies simplify the interaction of TM system design and program transactions, leading to interesting patterns of execution that can affect performance. Pathologies degrade performance by preventing a transaction from making progress or by performing useless work that is discarded when a transaction aborts. They are described as follows.

DuelingUpgrades: When two concurrent transactions read and later attempt to modify the same cache block, this pathology arises. Because both transactions add the block to their read-sets, the conflict is detected when they write the same data, causing one of the transactions to abort. This behavior is pathologic only for EE systems because of their slower aborts.

FutileStall: Eager conflict detection may cause a transaction that ultimately aborted to stall for another transaction. In this case, the stall is not needed, because it did not resolve a conflict with a transaction that performed useful work. Eager version management exacerbates this pathology because the HTM system must restores the old values to maintain isolation on its write-set. Thus, a transaction could stall on another transaction that ultimately aborts and continues to stall while the system restores the old values from the log. StarvingWriter: This pathology occurs when a transactional writer conflicts with a set of concurrent transactional readers. The writer stalls waiting for the readers to finish their transactions and release isolation. The writer may starve if new readers arrive before existing readers commit [7]. The writer is blocked by a series of committing readers. In a more favorable case, the readers make progress and only the writer starves. In the least favorable case, none of the transactions make progress, because the readers encounter a cyclic dependence with the writer after reading the block, abort (releasing isolation), but then retry before the writer acquires access. StarvingElder: A lazy conflict detection system and a "committer-wins" policy may cause the pathology. That is because the system allows small transactions to starve longer transactions [10]. Small transactions naturally reach their commit phase faster and the committer-wins policy allows repeated small transactions to always abort the longer transaction. The resulting load imbalance may have broad performance repercussions. SerializedCommit: Lazy conflict detection HTM systems serialize transactions during commit to ensure a global serial order. Thus, committing transactions may stall while waiting for other transactions to commit. The case is generated in a program with many small transactions. RestartConvoy: This pathology happens in HTM systems with a lazy conflict detection. When one committing transaction conflicts with (and aborts) multiple instances of the same static transaction, the aborted transactions restart simultaneously, compete for system resources, and, due to their similarity, finish together. The crowd of transactions competes to commit,

and the winner aborts the others. Convoys can persist indefinitely if a thread that commits a transaction rejoins the competition before all other transactions have had a chance to commit [3]. FriendlyFire: This pathology arises when one transaction conflicts with and aborts another, which then subsequently aborts before committing any useful work. In the least favorable case, this pathology repeats indefinitely with concurrent transactions continually aborting each other, resulting in livelock.

4 The Proposed Priority Scheduling Algorithm

In this section, we present our priority scheduling algorithm to resolve the seven performance pathologies and show how to set the priority.

4.1 Algorithm

The symbols used in the algorithm are defined as follows. For a transaction T,

TR_i is the ith transaction,
$P(T)$ is the priority of thread T,
$P_S(T)$ is the static priority of thread T,
$P_D(T)$ is the dynamic priority of thread T,
$C(T)$ is the total clock cycles to execute thread T,
TC is the total clock cycles of the whole program,
$S(T)$ is different store address counts of thread T,
$LS(T)$ is the number of all loads and stores address counts of thread T,
$CE(T)$ is the current execution time (cycles) of thread T,
$N_R(T)$ is the number of RETRY times of thread T,
$BO(T)$[1] : backoff time (cycles) of thread T, and
$BO_{base}(T)$ is the original backoff time (cycles) of thread T.

Figure 1 presents our priority scheduling algorithm. First, we must obtain the static priority of each thread, which is owned by each transaction. The static priority consists of the execution time of each transaction and the different store addresses of each transaction. If we know the execution steps of transactions, including its memory reference and execution time in advance, we can use the information to set the priority. The information is defined before execution and does not change upon run time; therefore, we call them "static priority". Next, we set the "dynamic priority" which is changed by the current execution time and the number of retries at run time. Dynamic priority records the cycles of each transaction that has been executed. If a transaction has been executed for a while, it should not be aborted by the other one. Thus, the larger transaction does not waste the retry time. However, if a small transaction is always aborted by larger transactions, we should consider this case. Therefore, the number of retry times is of considerable concern to this study. Furthermore, the priority values influence the backoff time. If a transaction has higher priority, it has a smaller backoff time (that is, the transaction restarts faster). In our algorithm,

the additional step involves the transaction needing to compete with priorities. The transaction having the highest priority can abort others and is not interrupted by other transactions. When a transaction is aborted, we need to update the 'dynamic priority" and backoff time.

```
   Let i = 1
   For TR_i in all transactions
      Assign P_S(T_i) to T_i
Begin
   Let j = 1
   For T_j in all transactions
      Assign P_D(T_j) to T_j based on according CE(T_j) and N_R(T_j)
   if("Data Conflict" appears between T_i and T_j
      if (P(T_i) > P(T_j))
          Aborting T_j
         Update P_D(T_j) and BO(T_j)
      else
         Aborting T_i
         Update P_D(T_i) and BO(T_i)
End
```

Fig. 1. The proposed priority scheduling algorithm

4.2 Priority Assignment

This section present the priority setting in details. Initially, each transaction is assigned a priority value. The priority values vary despite different processors executing the same transaction, due to the priority value being composed of static priority and dynamic priority. Even transactions executed by different processors have the same static priority; the dynamic priority is determined by the current execution time and number of retry times. Thus, this approach can prevent the transaction executed by different processors that cannot compete with each other. Our purpose of priority is to change traditional TM conflict resolution. The traditional TM conflict resolutions are inflexible and do not consider all aspects. Therefore, this paper contributes a method that can consider all aspects and improve the original TM conflict resolution. Most importantly, PS mitigates the performance pathologies as well as improves TM program performance. The following is the priority scheme.

$$P(T) = P_S(T) + P_D(T)$$
$$P_S(T) = C(T)/TC \times \alpha + S(T)/LS(T) \times \beta$$
$$P_D(T) = CE(T)/TC \times \gamma + N_R(T) \times \delta$$
$$BO(T) = (1/P(T)) \times BO_{base}$$

Notice that $\alpha, \beta, \gamma, \delta$ are the initial weights. The static priority is defined according to the previous execution tracks of threads. We monitor the execution time of each transaction and store different address counts to determine the static

priority. To determine whether the transaction is large or not, we record the execution time of each transaction. If the transaction often modifies the values of memory, we assume it spends more time. In this case, we assign it a higher priority. Notably, static priority must be defined before invoking our priority scheduler. We exemplify how to calculate the static priority as follows.

Example:

Cycles	Instruction
100	begin transaction(id)
101	LOAD A
102	STORE B
103	STORE A
104	STORE B
105	STORE C
106	LOAD B
107	commit transaction(id)

TC is 107-100 = 7
The transaction store A,B,C, so I_{stw} is 3
Based on the priority scheme,
$P_S(T) = C(T)/TC \times \alpha + S(T)/LS(T) \times \beta$
$= 7/TC \times \alpha + 3/LS(T) \times \beta$

4.3 PS on TM Pathologies

In this section, we exemplify the seven pathologies with our priority scheduling algorithm and we identify the priority relationships among transactions.

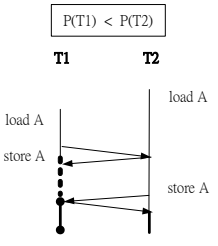


Fig. 2. PS : DUELINGUPGRADES

Figure 2 illustrates DuelingUpgrades as solved by our approach. We assume $P(T2) > P(T1)$, because T2's execution time is longer than that of T1's, while the original conflict resolution stalls the requester and aborts the requester on possible deadlock. Here, we stall the requester but abort the transaction having the smaller priority. T1 makes a request to store A the first time; hence, T1 is stalled by T2. After that, T2 requests to store A. If we also stall the requester, it causes a deadlock. Therefore, we must abort one of the transactions. Based on priority values, we abort the smaller one of the two. A hardware solution EE_P can decrease the pathology ratio. The EE_P system uses a small write-set predictor to predict the deadlock [15]. We hope our algorithm can combine with EE_P to achieve the most favorable result having the smallest pathology ratio.

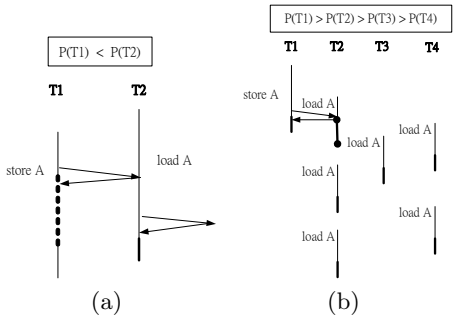


Fig. 3. The proposed priority scheduling for (a) FUTILESTALL and (b) STARVINGWRITER

FutileStall: EE system stalls the requester and aborts the requester on possible deadlock. Figure 3(a) illustrates FutileStall as solved by our algorithm. FutileStall is similar to DuelingUpgrades, but the situation in T2 is aborted by transactions other than T1. We also assume $P(T2) > P(T1)$. The original T2 is aborted by the other transaction; but in our assumption, T2 has a higher priority and can continue executing without wasting the execution time. Hardware does not target this pathology, but we can resolve this pathology in this example. We have an advantageous position regarding this pathology.

StarvingWriter: This is also on the EE system; therefore, it has the same conflict resolution as DuelingUpgrades and FutileStall. This case occurs when readers stall writer successively and writer may be starvation. The hardware solution EE_{HP} extends EE_P in an attempt to also reduce STARVINGWRITER, by allowing an older writer to simultaneously abort a number of younger readers. Our approach is similar to EE_{HP} by giving the writer higher priority than readers; hence, the writer does not starve and does not waste the stall time. Figure 3(b) illustrates StarvingWriter as solved by PS. This is the reason that we consider the store address times to set a priority value.

StarvingElder: This problem is on the LL system. If the small transactions appear many times as well as the conflict resolution is committer wins, it prevents the larger transaction from having a chance to commit. The larger transaction may be a starvation. Figure 4(a) illustrates StarvingElder as solved by the proposed priority scheduling. Our approach gives $P(T2) = P(T3) > P(T1)$ at the initial time. To avoid low-priority transaction starves, we also consider the number of retry times. When T1 restarts the transaction, our approach raises the dynamic priority immediately. In our dynamic priority, T1 retries more times, and obtains a higher priority. In Figure 4(a), T1 may retry three times and owns the highest priority, causing $P(T1) > P(T2) = P(T3)$. When T1 can abort other transactions, it does not starve because we account for retry times in dynamic priority. The hardware solution LL_B addresses RestartConvoy and also can mitigate StarvingElder and SerializedCommit. Like LL system, LL_B is based on the committer-wins policy. However, restarting transactions use randomized linear backoff to delay the restart of an aborted transaction. By staggering the restart

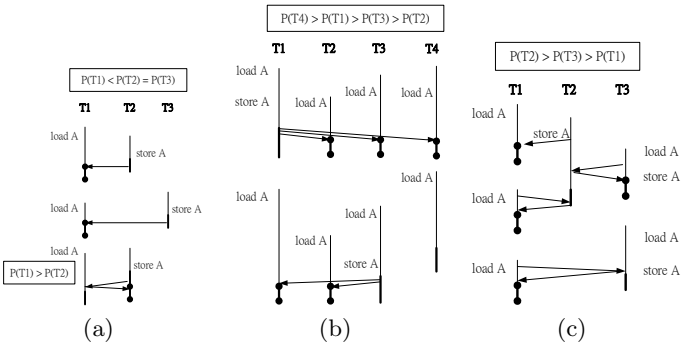


Fig. 4. The proposed priority scheduling for (a) STARVINGELDER, (b) RESTART-CONVOY and (c) FRIENDLYFIRE

of each transaction in the group of transactions aborted by a given commit, LL_B mitigates convoy formation. But in our speculation, we think that randomized linear backoff can not ensure the same case will not happen again. We assign the backoff time based on priority value. If the priority value is large, it means that the transaction should have a higher priority to execute. Thus, we give the high-priority transaction smaller backoff time. In Figure 4(a) is different in the backoff time every time the transaction restarts.

SerializedCommit: This problem arises from the hardware restriction with only one write buffer. Hhardware solution uses backoff time to stagger the transactions. We also use backoff to stagger the transactions' commit time based on priority. The high-priority transaction should execute first, we seek to improve the hardware solution to achieve higher performance. However, we do not illustrate with this pathology because there is no fixed solution of execution sequence in the previous example. We can only ensure that using backoff time does not allow transactions to commit simultaneously.

RestartConvoy: Figure 4(b) illustrates RestartConvoy as solved by PS. This problem affects resource contention and serialized execution. If transactions restart simultaneously, it causes many transactions to restart simultaneously. This result lets these transactions cause the same problem again. The solution involves setting the backoff time to stagger the restart time. The hardware solution LL_B also involves setting the backoff time. The difference between hardware and our approach is that we use priority value to decide the backoff time. If the transaction has a higher priority, we assign it a shorter backoff time and let it restart immediately. Higher priority transactions mean they must finish earlier than others; therefore, we use priority value to decide backoff time. The higher priority transactions can commit earlier. Conversely the hardware solution cannot ensure that the same case will not happen again.

FriendlyFire: The EL system detects conflicts on each memory reference and writes the new value to the writer buffer. FriendlyFire causes livelock on TM programs. The hardware solution EL_T behaves the same as EL, but instead of always aborting in favor of the requester, transaction conflicts are resolved

according to the logical age of the transaction, as has been done before for implicit transactions [16] and eager alternative [6]. Our approach is similar to EL_T system, as shown in Figure 4(c). Our approach can solve the problem in advance because the higher-priority transaction such as T2 can abort others no matter if T2 commits or not. In this figure, it is not important that setting which one of transactions be the highest transaction, and it will avoid livelock.

5 Experimental Results

5.1 Setup

We simulate a full-system infrastructure using Simics [1] and a customized memory model built with the Wisconsin GEMS toolset [14]. The HTM interface is implemented using "magic" instructions: special no-ops that are caught by Simics and passed onto the memory model. The software components of the TM systems are implemented using hand-coded assembly routines and C functions.

We model a 32-core CMP system, which is an in-order, single-issue cores. Each core has 32KB private writeback L1 I and D caches. All cores share a multi-banked 8 MB L2 cache consisting of 32 banks interleaved by a block address. On-chip cache coherence is maintained via an on-chip directory, which maintains a bit vector of shares and implements the MESI protocol.

Figure 5 is the flow, which is divided into three parts. The first is the original TM execution. According to the traditional program execution process, we input the codes and compile them. After compiling, we attain the executable file and execute it. Because we run TM programs, we can obtain the information of the transactions. Based on the information, we profile the transactions' execution states.

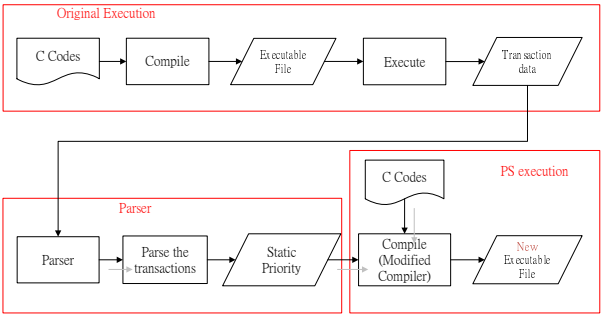


Fig. 5. Experimental Flow

Table 1. Workloads Parameters

BENCHMARK	Input	Work	Units
Cholesky	tk14.O	Factorization	1
Btree	Uniform random	BTree operation	100000
Deque	Uniform random	Deque operation	100000

Table 2. Pathologies on EE system: Base VS HW VS PS(% Execution Time)

	Base			HW			PS		
	DU	FS	SW	DU	FS	SW	DU	FS	SW
deque	5.1	0.2	3.0	< 0.1	< 0.1	< 0.1	0.3	0.3	0.5
cholesky	0.9	< 0.1	0.4	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	0.2
btree	1.1	< 0.1	0.8	0.9	12.1	0.3	0.2	0.2	0.2
barnes	0.2	< 0.1	0.6	< 0.1	< 0.1	< 0.1	< 0.1	0.2	0.4
radiosity	1.1	< 0.1	0.5	< 0.1	< 0.1	< 0.1	0.1	< 0.1	0.2

Table 3. Pathologies on EL system: Base VS HW VS PS(% Execution Time)

	Base	HW	PS
	FF	FF	FF
deque	3.2	< 0.1	< 0.1
cholesky	< 0.1	< 0.1	NA
btree	0.8	< 0.1	1.1
barnes	< 0.1	< 0.1	< 0.1
radiosity	< 0.1	< 0.1	< 0.1

Here, we know each transaction's execution cycles as they store different address times. Further using these information and weights do multiplication can get the "static priority". Finally, we use the modified compiler to combine the static priority into the source codes. Therefore, the new executable file contains the static priority. We run the new executable file and integrate the priority scheduling in the simulator. Following the above steps, we can compare the performance pathologies ratio in Section 5.2. In the next section, we describe the workload implementation, which is analyzed and regarded as our standard in this paper.

We select the SPLASH [17] suite benchmark and two microbenchmarks. Stanford Parallel Applications for Shared Memory (SPLASH) is a suite of multiprocessor applications. For Barnes, Cholesky, and Radiosity, these scientific programs were taken from the SPLASH benchmark suite and were selected because they demonstrate significant critical section based synchronization. We replace the critical sections with transactions while retaining barriers and other synchronization mechanisms. To reduce simulation times, we do not measure the entire parallel segment of the program. Instead, we take representative sections of the program and measure performance in terms of well-defined units of work. For Btree and Deque, these microbenchmarks present different data structures. Btree performs a lookup (with 80% probability) or an insert (20%). Deque is a benchmark such that each transaction first enqueues (dequeues) a value on the left (or right) of a global deque. It then performs a local job and finally increments the global counter.

5.2 Results

In this section, we present the results of pathologies ratio and the performance for the workloads introduced previously. Tables 2, 3,4, and 5 show the comparison

Table 4. Pathologies on LL system: Base VS HW VS PS(% Execution Time)

	Base			HW			PS		
	SE	SC	RC	SE	SC	RC	SE	SC	RC
deque	< 0.1	< 0.1	< 0.1	0.1	0.6	< 0.1	0.2	< 0.1	< 0.1
cholesky	< 0.1	< 0.1	0.6	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
btree	0.2	2.3	< 0.1	0.2	2.1	< 0.1	< 0.1	< 0.1	< 0.1
barnes	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
radiosity	< 0.1	< 0.1	< 0.1	NA	NA	NA	< 0.1	< 0.1	< 0.1

Table 5. Pathologies : PS and write-set predictor (% Execution Time)

	PS + Write-set Predictor		
	DU	FS	SW
deque	< 0.1	< 0.1	< 0.1
btree	0.7	9.1	< 0.1
cholesky	< 0.1	< 0.1	< 0.1

among the base result and three types of methods. "BASE" represents the original TM process, HW is the hardware solution, and PS is our proposed priority scheduling. Here, we also use PS with a write-set predictor present in Mix. We list seven pathologies: DuelingUpgrades(DU), FutileStall(FS), StarvingWriter(SW), StarvingElder(SE), SerializedCommit(SC), RestartConvoy(RC), and Friendly-Fire(FF), as well as the percent of total cycles for each workload in these tables.

We discovered that the hardware solution can eliminate these pathologies. While we compared the base results and PS results, PS mitigates most of these pathologies. FutileStall arises in Btree because there are numerous reads in this benchmark. Although the pathology ratio arises, the speedup for Btree with PS is 1.5. Therefore, we can improve the entire performance. Occasionally, PS cannot achieve the highest performance, but if we use PS with write-set predictor in Table 5, it achieves the highest performance. The write-set predictor can decrease the percentage of wrong cases.

6 Conclusion

To improve transactional memory performance, we cite the seven performance pathologies of Jayaram Bobba et al.'s [4], as well as their hardware solution. Scherer and Scott [12] indicated that, in STMs, conflict resolution is critical in avoiding numerous pathologies. In this paper, we add a new conflict resolution to resolve conflict. We use software to assist TM to achieve a high performance base on HTM systems. Our approach uses a profiling mechanism with the previous hardware solutions' idea, to propose a priority scheduling algorithm. We use static priority via profiling and dynamic priority to adjust the mechanism to remedy these pathologies. Using software to achieve some effects that HTM can not accomplish is our major goal. Our contribution is to propose another way to

accelerate TM program via parsing transactions in advance, using software to add a priority scheduling scheme to HTM. We also explain how research [4] has used HTM to mitigate these pathologies, while comparing our method with HTM solutions. Although hardware solutions achieve high performance, modifying the hardware architecture is costly. Using software costs less than using hardware and can achieve higher performance. In the experimental results, we found that our proposed solution, priority scheduling (PS), can mitigate these pathologies and improve the entire performance.

References

1. Magnusson, P.S., et al.: Simics: A full system simulation platform. IEEE Computer, 50–58 (February 2002)
2. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 316–327 (January 2005)
3. Blasgen, M., Gray, J., Mitoma, M., Price, T.: The convoy phenomenon. SIGOPS Oper. Syst. Rev. 20–25 (1979)
4. Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swiftand, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, pp. 387–394 (June 2007)
5. Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., Olukotun, K.: The atomoσ transactional programming language. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (June 2006)
6. Ceze, L., Tuck, J., Cascaval, C., Torrellas, J.: Bulk disambiguation of speculative threads in multiprocessors. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture (June 2006)
7. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with readers and writers. Communications of the ACM, 667–668 (1971)
8. Damron, P., Fedorova, A., Lev, Y., Luchango, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (October 2006)
9. Gottschlich, J., Connors, D.A.: Extending contention managers for user-defined priority-based transactions. In: Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (April 2008)
10. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistencys. In: Proceedings of the 31st Annual International Symposium on Computer Architecture (June 2004)
11. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300 (May 1993)
12. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: 24th ACM Symposium on Principles of Distributed Computing (July 2005)

13. Scherer III, W.N., Scott, M.L.: Randomization in stm contention management. In: Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (July 2005)
14. Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Min Xu, A.R.A., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator toolset. In: Computer Architecture News, pp. 92–99 (September 2005)
15. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: Log-based transactional memory. In: Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture, pp. 258–269 (February 2006)
16. Rajwar, R., Goodman, J.R.: Transactional lock-free execution of lock-based programs. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (October 2002)
17. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24–37 (June 1995)
18. Yen, L., Bobba, J., Marty, M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: Logtm-se: Decoupling hardware transactional memory from caches. In: Proceedings of the 13th IEEE Symposium on High-Performance Computer Architecture, pp. 261–272 (February 2007)

Fault-Tolerant Routing Based on Approximate Directed Routable Probabilities for Hypercubes

Thuy Dinh Duong and Keiichi Kaneko

Department of Computer and Information Sciences
Graduate School of Engineering
Tokyo University of Agriculture and Technology
Koganei-shi, Tokyo, Japan
{50010646150@st,k1kaneko@cc}.tuat.ac.jp

Abstract. Recently, parallel processing systems have been studied very actively, and many topologies have been proposed. A hypercube is one of the most popular topologies for interconnection networks. In this paper, we propose two new fault-tolerant routing algorithms for hypercubes based on approximate directed routable probabilities. The probability represents ability of routing to any node at a specific distance and also taking account of from which direction the message was received. Each node chooses one of its neighbor nodes to send a message by comparing the approximate directed routable probabilities. We also conducted a computer experiment to verify the effectiveness of our algorithms.

Keywords: multicomputer, interconnection network, parallel processing, fault-tolerant routing, hypercube, performance evaluation.

1 Introduction

The hypercube topology has been one of the most popular network topologies used for interconnection multicomputer networks especially in the parallel processing field. The properties of regular and recursive structure and low diameter [7] have attracted a considerable attention. As a results, there are several commercial and experimental systems that have been adopted this topology. Figure 1 shows an example of a 4-dimensional hypercube Q_4.

In a parallel processing system, many nodes process tasks together. Hence, efficiency of message passing has a high influence on the completion of task processing since communications between the nodes are accomplished by passing messages. The role of a routing algorithm is to specify a path to send a message from a source node to a destination node. Therefore, efficient message routing is one of the most vital problems in parallel processing systems.

Since the sizes of tasks to be processed are increasing, larger networks are required. However, the more the number of nodes in a network becomes, the higher the possibility of occurrences of faulty elements becomes. Hence, even in the presence of faulty nodes in a parallel processing system, finding a path which is fault-free and as short as possible between a source node and a destination node

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 106–116, 2011.

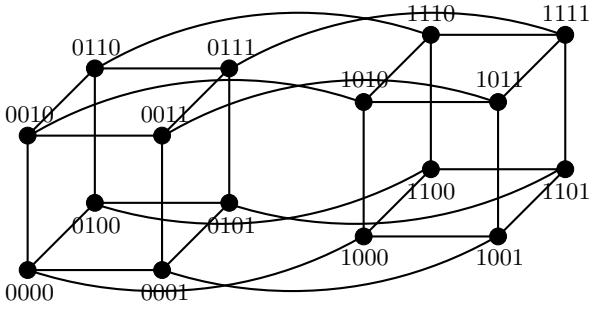


Fig. 1. An example of 4-dimensional hypercube Q_4

has become an emerged problem. Though there are several approaches to address the problem. But an efficient fault-tolerant routing algorithm must satisfy the following conditions. First, given a pair of a source node and destination node, the algorithm should find out a possible fault-free path between them. Second, if a node has the knowledge of the status of all other nodes in a network, it can be possible to calculate an optimized path. Because of the limitation in resources, that is, time and space complexities, information stored in each node should be so small that each node cannot identify all of faulty nodes. Therefore, in this paper we propose two fault-tolerant routing algorithms that find a fault-free path between non-faulty nodes. Also, in our proposed algorithms, we assume that in an n-dimensional hypercube Q_n with faulty nodes, information stored in each node must be of polynomial time and space complexities of n.

The rest of this paper is structured as follows. We survey related works in Section 2. In Section 3, we define terminology and notations that are necessary for discussion. Then, we introduce the directed routable probabilities, their approximate values, and a simplified calculation method for them in Section 4. Furthermore, we proposed two fault-tolerant routing algorithms in Section 5, and evaluate their performance by a computer experiment in Section 6. Finally, we give a conclusion and future works in Section 7.

2 Related Works

Recently, the research on fault-tolerant routing in hypercubes has gained attention and there are many attempts in research for this problem. By recursively classifying non-faulty nodes into safe, ordinary unsafe, and strongly unsafe nodes depending on the classification of neighbor nodes, Chiu and Wu have proposed an efficient fault-tolerant routing algorithm [4]. To improve the algorithm, Chiu and Chen introduced the routing capabilities that are obtained by classifying the safety nodes with respect to the Hamming distance to the destination nodes [3]. Wu has also proposed a similar fault-tolerant routing algorithm independently by introducing the safety vectors [8]. In addition, Kaneko and Ito have proposed a fault-tolerant routing algorithm based on classification of ordinary

and strongly unsafe nodes with respect to the Hamming distance as well as an efficient method to obtain classification of them [6].

All of the above attempts are based on information if a message is surely routed to the destination node or not. On the other hand, Al-Sadi et al. and Duong and Kaneko have proposed fault-tolerant routing algorithms based on probabilities that a message is sent from the source node to the destination node with a path of length of Hamming distance between them [1,2,5]. Though these algorithms take different approaches, the results are very similar. In this study, we extend the approach by Duong and Kaneko [5] by introducing a new concept of approximate directed routable probabilities and a simplified calculation method for them. Then, we propose two fault-tolerant routing algorithms based on them. Moreover, we carry out a computer experiment to evaluate performance of the algorithms.

3 Preliminaries

In this section, we define a hypercube network and introduce requisite notations. Hamming distance $H(\boldsymbol{a}, \boldsymbol{b})$ between 2 nodes $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$ and $\boldsymbol{b} = (b_1, b_2, \ldots, b_n)$ is defined by the number of different bits between a_i and b_i $(1 \leq i \leq n)$.

Definition 1. An n-dimensional hypercube Q_n is an undirected graph, which consists of 2^n nodes. Each node $\boldsymbol{a}$ is an n-bit sequence $(a_1, a_2, \ldots, a_n)$ where $a_i \in \{0, 1\}$ $(1 \leq i \leq n)$, and a_i is called the bit of i-th dimension. For two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$ in Q_n, there is an edge $(\boldsymbol{a}, \boldsymbol{b})$ between them if and only if the Hamming distance between them $H(\boldsymbol{a}, \boldsymbol{b})$ is equal to 1. The neighbor node of $\boldsymbol{a}$ along the dimension j is $\boldsymbol{a} \oplus 2^j$. □

In general, a path in a graph is represented by an alternate sequence of nodes and edges $\boldsymbol{a}_1$, $(\boldsymbol{a}_1, \boldsymbol{a}_2)$, $\boldsymbol{a}_2$, …, $\boldsymbol{a}_{k-1}$, $(\boldsymbol{a}_{k-1}, \boldsymbol{a}_k)$, $\boldsymbol{a}_k$. The length of the path P is the number of edges included in the path, and it is denoted by $L(P)$. If Q_n is fault-free, the length of the shortest path between $\boldsymbol{a}$ and $\boldsymbol{b}$ is equal to $H(\boldsymbol{a}, \boldsymbol{b})$.

Definition 2. For a node $\boldsymbol{a}$ in Q_n, a set of nodes $N(\boldsymbol{a})$ defined by

$$N(\boldsymbol{a}) = \{\boldsymbol{n} \mid H(\boldsymbol{a}, \boldsymbol{n}) = 1\}.$$

is called a set of neighbor nodes of $\boldsymbol{a}$. □

In a hypercube Q_n with a set of faulty nodes F, for a source node $\boldsymbol{s}$ and a destination node $\boldsymbol{d}$ that are both non-faulty, a fault-tolerant routing algorithm finds a fault-free path between $\boldsymbol{s}$ and $\boldsymbol{d}$.

Definition 3. In Q_n, for two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$, the set of preferred neighbor nodes of $\boldsymbol{a}$ for $\boldsymbol{b}$ is denoted by $N_0(\boldsymbol{a}, \boldsymbol{b})$, and is defined by $N_0(\boldsymbol{a}, \boldsymbol{b}) = \{\boldsymbol{n} \mid \boldsymbol{n} \in N(\boldsymbol{a}), H(\boldsymbol{n}, \boldsymbol{b}) = H(\boldsymbol{a}, \boldsymbol{b}) - 1\}$. In addition, the set of spare neighbor nodes of $\boldsymbol{a}$ for $\boldsymbol{b}$ is denoted by $N_1(\boldsymbol{a}, \boldsymbol{b})$, and is defined by $N_1(\boldsymbol{a}, \boldsymbol{b}) = \{\boldsymbol{n} \mid \boldsymbol{n} \in N(\boldsymbol{a}), H(\boldsymbol{n}, \boldsymbol{b}) = H(\boldsymbol{a}, \boldsymbol{b}) + 1\}$. It is easy to see that $N(\boldsymbol{a}) = N_0(\boldsymbol{a}, \boldsymbol{b}) \uplus N_1(\boldsymbol{a}, \boldsymbol{b})$. □

Note that, in Q_n, the number of nodes that are apart from a node $\boldsymbol{a}$ by Hamming distance h is equal to ${}_nC_h$. Also note that, for two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$ in Q_n, if $H(\boldsymbol{a},\boldsymbol{b}) = h$, then $|N_0(\boldsymbol{a},\boldsymbol{b})| = h$ holds.

4 Directed Routable Probabilities

In this section, we give the idea of directed routable probabilities. In an n-dimensional hypercube Q_n with a set of faulty nodes F, let us consider a situation where a non-faulty node $\boldsymbol{a}$ received a message from its non-faulty neighbor node $\boldsymbol{a} \oplus 2^j$ along the dimension j. Then, the directed routable probability $\vec{P}_h^{*j}(\boldsymbol{a})$ of $\boldsymbol{a}$ with respect to a Hamming distance h and a dimension j represents the probability that, for an arbitrary non-faulty node $\boldsymbol{b}$ with $h = H(\boldsymbol{a},\boldsymbol{b})$ and $h+1 = H(\boldsymbol{a} \oplus 2^j, \boldsymbol{b})$, there is a fault-free path of length h between $\boldsymbol{a}$ and $\boldsymbol{b}$.

Since it is difficult to calculate the directed routable probabilities precisely, we use the following approximate values.

Definition 4. For a node $\boldsymbol{a}$ in an n-dimensional hypercube Q_n with a set of faulty nodes F, approximate directed routable probabilities $\vec{P}_h^j(\boldsymbol{a})$ of $\boldsymbol{a}$ with respect to Hamming distance h and the dimension j is defined as follows:

$$\vec{P}_h^j(\boldsymbol{a}) = \begin{cases} 1 & (h = 0) \\ 0 & (1 \le h \le n, \boldsymbol{a} \in F) \\ \displaystyle\sum_{\substack{I \subset N(\boldsymbol{a}) \setminus \{\boldsymbol{a} \oplus 2^j\} \\ |I| = h}} \max_{\boldsymbol{n} \in I} \{\vec{P}_{h-1}^{\log_2(\boldsymbol{a} \oplus \boldsymbol{n})}(\boldsymbol{n})\} \Big/ {}_{n-1}C_h & (1 \le h \le n, \boldsymbol{a} \notin F) \end{cases}$$

□

Definition 4 for $\vec{P}_h^j(\boldsymbol{a})$ has the following meaning. First, an arbitrary node including a faulty node can send a message to itself with probability 1. Next, if a node $\boldsymbol{a}$ is faulty, it cannot send a message to any node other than itself. Hence, for any positive Hamming distance h, $\vec{P}_h^j(\boldsymbol{a}) = 0$ holds. Otherwise, by taking all the combinations of h nodes from the set of neighbor nodes $N(\boldsymbol{a}) \setminus \{\boldsymbol{a} \oplus 2^j\}$ of $\boldsymbol{a}$, the expectation value of the maximum routable probabilities of the combinations of h nodes with respect to the Hamming distance $h-1$ and the dimension which is used to send a message from $\boldsymbol{a}$ is calculated.

The approximate directed routable probabilities with respect to the Hamming distance 0 and an arbitrary dimension j is defined to be 1 for all the nodes including faulty nodes. Therefore, for any non-faulty node $\boldsymbol{a}$, $\vec{P}_1^j(\boldsymbol{a}) = 1$ holds.

To calculate the approximate directed routable probabilities easily, we introduce the following theorems.

Theorem 1. In an n-dimensional hypercube Q_n with a set of faulty nodes F, for a non-faulty node $\boldsymbol{a}$ and two natural numbers h $(1 \le h \le n)$ and j $(0 \le j \le n-1)$,

$$\vec{P}_h^j(\boldsymbol{a}) = \left(\sum_{k=1}^{\phi(j)-1} {}_{k-1}C_{h-1} p_k + \sum_{k=\phi(j)+1}^{n} {}_{k-2}C_{h-1} p_k \right) \Big/ {}_{n-1}C_h$$

where $p_1 \leq p_2 \leq \ldots \leq p_n$ are obtained by sorting $\vec{P}^0_{h-1}(\boldsymbol{a} \oplus 2^0)$, $\vec{P}^1_{h-1}(\boldsymbol{a} \oplus 2^1)$, $\ldots$, $\vec{P}^{n-1}_{h-1}(\boldsymbol{a} \oplus 2^{n-1})$.

(Proof) Let $\phi(\cdot)$ is a bijection that satisfies the condition $\vec{P}^j_{h-1}(\boldsymbol{a}^{\oplus 2^j}) = p_{\phi(j)}$. Then, from Definition 4, $p_k = \max_{\boldsymbol{n} \in I}\{\vec{P}^{\log_2(\boldsymbol{a} \oplus \boldsymbol{n})}_{h-1}(\boldsymbol{n})\}$ holds if and only if $p_k \in \cup_{\boldsymbol{n} \in I}\{\vec{P}^{\log_2(\boldsymbol{a} \oplus \boldsymbol{n})}_{h-1}(\boldsymbol{n})\}$ and $\cup_{\boldsymbol{n} \in I}\{\vec{P}^{\log_2(\boldsymbol{a} \oplus \boldsymbol{n})}_{h-1}(\boldsymbol{n})\} \subset \{p_1, p_2, \ldots, p_k\} \backslash \{p_{\phi(j)}\}$ hold. Therefore, the number of occurrences such that p_k becomes the maximum value is equal to ${}_{k-1}C_{h-1}$ if $k < \phi(j)$ and to ${}_{k-2}C_{h-1}$ if $k > \phi(j)$. Hence, the theorem holds. □

Theorem 2. In Q_n with a set of faulty nodes F, for a node $\boldsymbol{a}(\notin F)$, h $(1 \leq h \leq n)$ and l $(1 \leq l \leq n-1)$, $\vec{P}^{\phi^{-1}(l+1)}_h(\boldsymbol{a}) = \vec{P}^{\phi^{-1}(l)}_h(\boldsymbol{a}) + {}_{l-1}C_{h-1}(p_l - p_{l+1})/{}_{n-1}C_h$ holds.

(Proof) It is trivial from Theorem 1. □

Theorem 3. In Q_n with a set of faulty nodes F, for a node $\boldsymbol{a}(\notin F)$ and h $(1 \leq h \leq n)$, $\vec{P}^{\phi^{-1}(1)}_h(\boldsymbol{a}) = \vec{P}^{\phi^{-1}(2)}_h(\boldsymbol{a}) = \cdots = \vec{P}^{\phi^{-1}(h)}_h(\boldsymbol{a})$

(Proof) From Theorem 2, if $l < h$, ${}_{l-1}C_{h-1} = 0$ and $\vec{P}^{\phi^{-1}(l+1)}_h(\boldsymbol{a}) = \vec{P}^{\phi^{-1}(l)}_h(\boldsymbol{a})$ holds. □

From Theorems 1 to 3, we can obtain a function ADRP to calculate the approximate directed routable probabilities. It is shown in Figure 2.

```
function ADRP(a, h, j, F)
begin
    if h = 0 then P^j_h(a) := 1
    else if a ∈ F then P^j_h(a) := 0
    else begin
        collect {P^{log2(a⊕n)}_{h-1}(n) | n ∈ N(a)};
        sort {P^{log2(a⊕n)}_{h-1}(n) | n ∈ N(a)} to obtain p1 ≤ p2 ≤ ... ≤ pn;
        define φ(·) s.t. P^j_{h-1}(a ⊕ 2^j) = p_{φ(j)};
        P^{φ^{-1}(1)}_h(a) := Σ_{k=h+1}^{n} {}_{k-2}C_{h-1} p_k / {}_{n-1}C_h;
        for l := 1 to n - 1 do
            P^{φ^{-1}(l+1)}_h(a) := P^{φ^{-1}(l)}_h(a) + {}_{l-1}C_{h-1} × (p_l - p_{l+1})/{}_{n-1}C_h
    end;
    return P^j_h(a)
end
```

Fig. 2. Function to calculate approximate values of directed routable probabilities

In fault-tolerant routing based on approximate directed routable probabilities, the following theorem is very helpful.

Theorem 4. In an n-dimensional hypercube Q_n with a set of faulty nodes F, for any two nodes $\boldsymbol{s}$ and $\boldsymbol{d}$ with $h = H(\boldsymbol{s}, \boldsymbol{d})$, if there is a fault-free path between

$\boldsymbol{s}$ and $\boldsymbol{d}$, then there is a node $\boldsymbol{a}$ in the set of preferred neighbor nodes of $\boldsymbol{s}$ for $\boldsymbol{d}$, $N_0(\boldsymbol{s},\boldsymbol{d})$, that satisfies $\vec{P}_{h-1}^{\log_2(\boldsymbol{a}\oplus\boldsymbol{s})}(\boldsymbol{a}) > 0$ or there is a node $\boldsymbol{b}$ in the set of spare neighbor nodes of $\boldsymbol{s}$ for $\boldsymbol{d}$, $N_1(\boldsymbol{s},\boldsymbol{d})$, that satisfies $\vec{P}_{h+1}^{\log_2(\boldsymbol{b}\oplus\boldsymbol{s})}(\boldsymbol{b}) > 0$.

(Proof) Let $\boldsymbol{c}_0 \rightarrow \boldsymbol{c}_1 \rightarrow \cdots \rightarrow \boldsymbol{c}_l$ be the fault-free path between $\boldsymbol{s}$ and $\boldsymbol{d}$ where $\boldsymbol{c}_0 = \boldsymbol{s}$, $\boldsymbol{c}_l = \boldsymbol{d}$, $\boldsymbol{c}_i \neq \boldsymbol{c}_{i+1}$ $(0 \leq i \leq l-1)$, $\boldsymbol{c}_{i-1} \neq \boldsymbol{c}_{i+1}$ $(1 \leq i \leq l-1)$, and $l \geq h$. If $\boldsymbol{c}_1 \in N_0(\boldsymbol{s},\boldsymbol{d})$, let us consider a sub path $\boldsymbol{c}_0 \rightarrow \boldsymbol{c}_1 \rightarrow \cdots \rightarrow \boldsymbol{c}_h$. Then, from Definition 4, $\vec{P}_0^{\log_2(\boldsymbol{c}_h\oplus\boldsymbol{c}_{h-1})}(\boldsymbol{c}_h) > 0$. In addition, for any r $(0 \leq r \leq h-1)$, if $\vec{P}_{h-r-1}^{\log_2(\boldsymbol{c}_{r+1}\oplus\boldsymbol{c}_r)}(\boldsymbol{c}_{r+1}) > 0$, then

$$\vec{P}_{h-r}^{\log_2(\boldsymbol{c}_r\oplus\boldsymbol{c}_{r-1})}(\boldsymbol{c}_r) = \sum_{\substack{I\subset N(\boldsymbol{c}_r)\setminus\{\boldsymbol{c}_{r-1}\} \\ |I|=h}} \max_{\boldsymbol{n}\in I}\{\vec{P}_{h-r-1}^{\log_2(\boldsymbol{c}_r\oplus\boldsymbol{n})}(\boldsymbol{n})\}/{}_{n-1}C_h$$
$$> \vec{P}_{h-r-1}^{\log_2(\boldsymbol{c}_{r+1}\oplus\boldsymbol{c}_r)}(\boldsymbol{c}_{r+1})/{}_{n-1}C_h > 0.$$

Hence, $\vec{P}_{h-1}^{\log_2(\boldsymbol{c}_1\oplus\boldsymbol{c}_0)}(\boldsymbol{c}_1) > 0$ holds. If $\boldsymbol{c}_1 \in N_1(\boldsymbol{s},\boldsymbol{d})$, let us consider a sub path a sub path $\boldsymbol{c}_0 \rightarrow \boldsymbol{c}_1 \rightarrow \cdots \rightarrow \boldsymbol{c}_{h+2}$. Then, from a similar discussion to the case that $\boldsymbol{c}_1 \in N_0(\boldsymbol{s},\boldsymbol{d})$, $\vec{P}_{h+1}^{\log_2(\boldsymbol{c}_1\oplus\boldsymbol{c}_0)}(\boldsymbol{c}_1) > 0$ holds. □

From Theorem 4, for any two non-faulty nodes $\boldsymbol{s}$ and $\boldsymbol{d}$ with $h = H(\boldsymbol{s},\boldsymbol{d})$, if there is not a node $\boldsymbol{a}$ $(\in N_0(\boldsymbol{s},\boldsymbol{d}))$ such that $\vec{P}_{h-1}^{\log_2(\boldsymbol{a}\oplus\boldsymbol{s})}(\boldsymbol{a}) > 0$ nor a node $\boldsymbol{b}$ $(\in N_1(\boldsymbol{s},\boldsymbol{d}))$ such that $\vec{P}_{h+1}^{\log_2(\boldsymbol{b}\oplus\boldsymbol{s})}(\boldsymbol{b}) > 0$, then there is not any fault-free path between $\boldsymbol{s}$ and $\boldsymbol{d}$.

5 Fault-Tolerant Routing Algorithms

In this section, we show how to find a path in a hypercube with faulty nodes by using approximate directed routable probabilities. The routing strategy is based on the approximate directed routable probabilities of neighbor nodes stored in each node.

In an n-dimensional hypercube with a set of faulty nodes F, we assume that each node $\boldsymbol{a}$ stores the approximate directed routable probabilities for all of the neighbor nodes $\boldsymbol{n}$ $(\in N(\boldsymbol{a}))$ with respect to all Hamming distances h $(0 \leq h \leq n)$. Then, for a non-faulty source node $\boldsymbol{s}$ and a non-faulty destination node $\boldsymbol{d}$, we propose two fault-tolerant routing algorithms which establish fault-free paths between them.

5.1 Naive Algorithm `ADRP0`

First, we propose a simple fault-tolerant routing algorithm `ADRP0`. It takes the current node $\boldsymbol{c}$ and the destination node $\boldsymbol{d}$ as its arguments. Initially it is called with the source node and the destination node.

In the algorithm, the Hamming distance $h = H(\boldsymbol{c},\boldsymbol{d})$ is calculated first. If it is equal to 0, the algorithm delivers the message to $\boldsymbol{c}$ and terminates. Otherwise, that is, if $h > 0$, Algorithm `ADRP0` selects the node $\boldsymbol{n}_0^*$ among the preferred

neighbor nodes of the current node for the destination node that has the largest positive approximate directed routable probability with respect to $H(\boldsymbol{c}, \boldsymbol{d}) - 1$, and sends the message to the selected neighbor node. If the approximate directed routable probabilities of the preferred nodes are all zero, then the node $\boldsymbol{n}_1^*$ with the largest approximate probability with respect to $H(\boldsymbol{c}, \boldsymbol{d}) + 1$ is selected among the spare neighbor nodes, and the message is sent to it.

Figure 3 shows the pseudo code for the algorithm where exception handling for the case $h = n$ is omitted. From Theorem 4, $\vec{P}_{h-1}^{\log_2(\boldsymbol{c} \oplus \boldsymbol{n}_0^*)}(\boldsymbol{n}_0^*) > 0$ or $\vec{P}_{h+1}^{\log_2(\boldsymbol{c} \oplus \boldsymbol{n}_1^*)}(\boldsymbol{n}_1^*) > 0$ holds. Hence, the routing always fails by an infinite loop.

```
procedure ADRP0(c, d)
begin
    h := H(c, d);
    n0* := arg max_{n∈N0(c,d)} {P⃗_{h-1}^{log2(c⊕n)}(n)};
    n1* := arg max_{n∈N1(c,d)} {P⃗_{h+1}^{log2(c⊕n)}(n)};
    if h = 0 then deliver the message to c
    else if P⃗_{h-1}^{log2(c⊕n0*)}(n0*) > 0 then ARDP0(n0*, d)
    else ARDP0(n1*, d)
end
```

Fig. 3. Routing algorithm `ADRP0`

5.2 Improved Algorithm `ARDP1`

Next, we propose an alternate fault-tolerant routing algorithm `ARDP1`. It takes the previous node $\boldsymbol{p}$, the current node $\boldsymbol{c}$, and the destination node $\boldsymbol{d}$ as its input. From the previous node, the message is sent to the current node. Initially, it is called with the source and destination nodes.

In the algorithm, the Hamming distance $h = H(\boldsymbol{c}, \boldsymbol{d})$ is calculated firt. If it is equal to 0, the algorithm delivers the message to $\boldsymbol{c}$ and terminates. Otherwise, that is, if $h > 0$, among the preferred neighbor nodes except for the previous node $N_0(\boldsymbol{c}, \boldsymbol{d}) \setminus \{\boldsymbol{p}\}$, Algorithm `ARDP1` first tries to select the node $\boldsymbol{n}_0^*$ that has the largest positive approximate directed routable probability with respect to $H(\boldsymbol{c}, \boldsymbol{d}) - 1$, and send the message to the selected neighbor node if it exists. If the approximate directed routable probabilities of the preferred nodes are all zero, then the node $\boldsymbol{n}_1^*$ with the largest approximate probability with respect to $H(\boldsymbol{c}, \boldsymbol{d}) + 1$ is selected among the spare neighbor nodes except for the previous node $N_1(\boldsymbol{c}, \boldsymbol{d}) \setminus \{\boldsymbol{p}\}$, and the message is sent to it. Note that Algorithm `ARDP1` excludes the previous node $\boldsymbol{p}$ from the candidate nodes for message sending. This exclusion avoids a simple loop of message sending between two adjacent nodes.

Figure 4 shows the pseudo code for the algorithm where exception handling for the case $h = n$ is omitted. Note that Algorithm `ADRP1` may explicitly fail to send a message.

```
procedure ADRP1(p, c, d)
begin
    h := H(c, d);
    n*_0 := arg max_{n∈N_0(c,d)\{p}} {P⃗^{log2(c⊕n)}_{h-1}(n)};
    n*_1 := arg max_{n∈N_1(c,d)\{p}} {P⃗^{log2(c⊕n)}_{h+1}(n)};
    if h = 0 then deliver the message to c
    else if P⃗^{log2(c⊕n*_0)}_{h-1}(n*_0) > 0 then ADRP1(c, n*_0, d)
    else if P⃗^{log2(c⊕n*_1)}_{h+1}(n*_1) > 0 then ADRP1(c, n*_1, d)
    else error('unable to deliver')
end
```

Fig. 4. Routing algorithm ADRP1

6 Performance Evaluation

In this section, we first analyze the time complexity of calculation of approximate directed routable probabilities, which is the first step of our algorithms. Next, we compare our algorithms with the algorithm by Al-Sadi et al. [1,2] and the algorithm by Duong and Kaneko [5] by a computer experiment.

6.1 Time Complexity

Time complexity for calculation of approximate directed routable probabilities in each node depends on the expression $\vec{P}_h^j(\boldsymbol{a}) = (\sum_{k=1}^{\phi(j)-1} {}_{k-1}C_{h-1}p_k + \sum_{k=\phi(j)+1}^{n} {}_{k-2}C_{h-1}p_k)/{}_{n-1}C_h$. Combinations of ${}_{k-1}C_{h-1}$, ${}_{k-2}C_{h-1}$ and ${}_nC_h$ are calculated at first, and stored in an array. It takes $O(n^2)$ time complexity by making use of the relation ${}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k$ with the boundary condition ${}_nC_0 = {}_nC_n = 1$. For a Hamming distance h, sorting of p_k $(1 \leq k \leq n)$ takes $O(n \log n)$ time complexity, and calculation of $\vec{P}_h^j(\boldsymbol{a})$ takes $O(n)$ time complexity. Therefore, for all h $(1 \leq h \leq n)$, sorting p_k's and calculation of $\vec{P}_h^j(\boldsymbol{a})$ require $O(n^2 \log n)$ time complexity. From the above discussion, calculation of $\vec{P}_h^j(\boldsymbol{a})$ is dominant, and it takes $O(n^2 \log n)$ time complexity in total. Each node has to exchange information n times with each of its neighbor nodes.

6.2 Computer Experiment

In this section, we give the detail of the results of a computer experiment conducted to compare Algorithms ADRP0 and ADRP1 we proposed with Algorithms ADO0 and ADO1 by Al-Sadi et al. and Algorithms DK0 and DK1 by Duong and Kaneko.

Algorithms ADO0 and DK0 do not make use of the information from which the message is sent to the current node while Algorithms ADO1 and DK1 are obtained by restricting the candidate nodes to suppress infinite loops between two adjacent nodes as similar to Algorithm ADRP0.

A computer experiment was carried out for n-dimensional hypercubes where $5 \leq n \leq 10$ changing the ratio of faulty nodes ρ from 10% to 80%. Concretely,

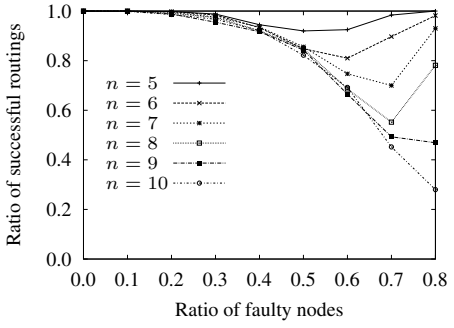


Fig. 5. Ratio of successful routings by Algorithm ADO0

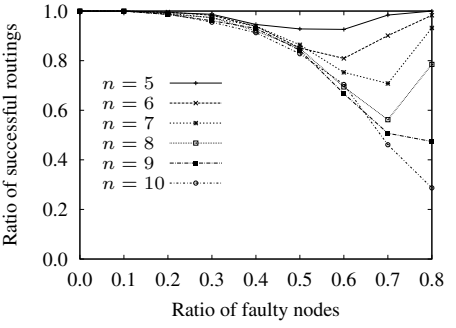


Fig. 6. Ratio of successful routings by Algorithm DK0

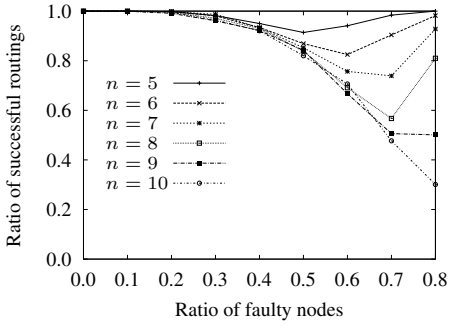


Fig. 7. Ratio of successful routings by Algorithm ADRP0

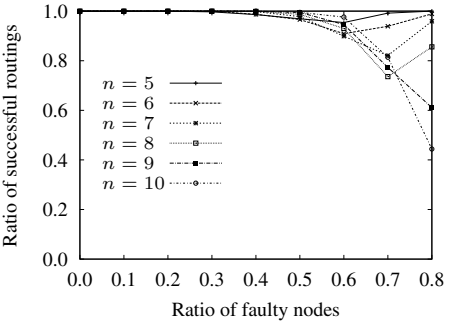


Fig. 8. Ratio of successful routings by Algorithm ADO1

first, in Q_n $(5 \leq n \leq 10)$, we selected faulty nodes randomly with the ratio ρ. Next, we selected the source node $\boldsymbol{s}$ and the destination node $\boldsymbol{d}$ from non-faulty nodes. Finally, after checking the connectivity of $\boldsymbol{s}$ and $\boldsymbol{d}$, we applied the six fault-tolerant routing algorithms, ADO0, DK0, ADRP0, ADO1, DK1, and ADRP1, to measure the ratio of successful routings. If $\boldsymbol{s}$ and $\boldsymbol{d}$ are not connected, that is, there is no fault-free path between them, we start over from the selection of faulty nodes. For each pair of a dimension and a ratio of faulty nodes, we executed at least 1,000 trials.

Figures 5, 6, 7, 8, 9, 10 show the results by Algorithms ADO0, DK0, ADRP0, ADO1, DK1 and ADRP1, respectively.

As a result, we can see that performance of ADO0 and ADO1 is almost equivalent to that of DK0 and DK1, respectively. In addition, performance of ADRP0 is a bit better than that of ADO0 and DK0. For example, the ratio of successful routings of ADRP0 is 0.302 with $n = 10$ and $\rho = 0.8$ while those of ADO0 and DK0 are 0.280 and 0.287, respectively. Moreover, performance of ADRP1 is significantly better than that of ADO0 and DK0. For example, the ratio of successful routings of ADRP1 is 0.571 with $n = 10$ and $\rho = 0.8$ while those of ADO0 and DK0 are 0.444 and 0.453, respectively.

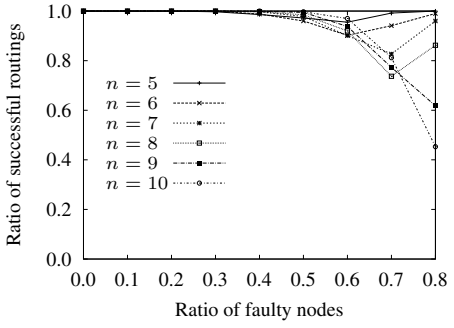


Fig. 9. Ratio of successful routings by Algorithm `DK1`

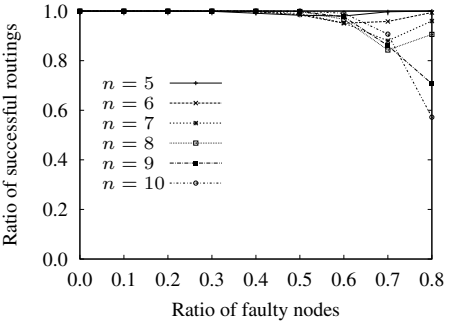


Fig. 10. Ratio of successful routings by Algorithm `ADRP1`

6.3 Discussion

The computer experiment shows that with high ratios of faulty nodes ($\rho = 0.7$ or $\rho = 0.8$), the successful routing ratios are increased by 0.1 to 0.2 by algorithms `ADRP0` and `ADRP1`. Hence, we give a small discussion about the effects of approximate directed routable probabilities.

Let us consider a situation where the current node is $\boldsymbol{c}$, and there is a non-faulty neighbor node $\boldsymbol{a}$ ($\in N(\boldsymbol{c})$) whose neighbor nodes are all faulty except for $\boldsymbol{c}$. Then, it is easy to see that if $\boldsymbol{a}$ is selected as the next node, it would lead to a routing failure since there is no available node in $N(\boldsymbol{a}) \setminus \{\boldsymbol{c}\}$. See Figure 11.

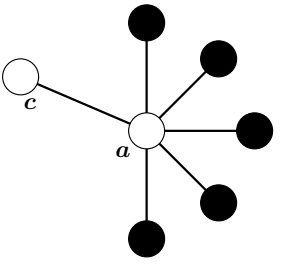


Fig. 11. A typical example of routing failure

In the proposed algorithms, the approximate directed routable probabilities $\vec{P}_h^j(\boldsymbol{c})$ with $h \geq 2$ are calculated to be 0 by taking the direction $j = \log_2(\boldsymbol{c} \oplus \boldsymbol{a})$ into consideration. Therefore, $\boldsymbol{a}$ cannot be selected by the algorithms. On the other hand, in the other algorithms `ADO0`, `ADO1`, `DK0`, and `DK1`, the corresponding probabilities are not 0, and it may happen that $\boldsymbol{a}$ is selected for message sending.

This is the main reason that the proposed algorithms based on approximate directed routable probabilites show the better performance than others. When the faulty ratio is getting higher, such situations occur more frequently.

7 Conclusion

In this paper, we have proposed two new fault-tolerant routing algorithms for hypercubes based on approximate directed routable probabilities, which represent ability of routing to any node at a specific distance considering from which direction the message is received. Each node selects one of its neighbor nodes to send a message by taking the approximate directed routable probabilities into consideration.

We also conducted a computer experiment to verify the effectiveness of our algorithms. As a result, we proved that our algorithms have better performance than the algorithms proposed by Al-Sadi et al. and the algorithms proposed by Duong and Kaneko.

Our next step is to extend the concept of the approximate directed routable probabilities to apply other topologies for parallel processing systems. The path lengths are also an important problem. Therefore, we also intend to improve the routing algorithm so it can find the shorter fault-free paths between any fault-free nodes in a hypercube.

Acknowledgement. This study was partly supported by a Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science under Grant No. 22500041.

References

1. Al-Sadi, J., Day, K., Ould-Khaoua, M.: Fault-tolerant routing in hypercubes using probablity vectors. Parallel Computing 27(10), 1381–1399 (2001)
2. Al-Sadi, J., Day, K., Ould-Khaoua, M.: Probability-based fault-tolerant routing in hypercubes. The Computer Journal 44(5), 368–373 (2001)
3. Chiu, G.M., Chen, K.S.: Use of routing capability for fault-tolerant routing in hypercube multicomputers. IEEE Transactions on Computers 46(8), 953–958 (1997)
4. Chiu, G.M., Wu, S.P.: A fault-tolerant routing strategy in hypercube multicomputers. IEEE Transactions on Computers 45(2), 143–155 (1996)
5. Duong, D.T., Kaneko, K.: Fault-tolerant routing algorithms based on approximate routable probabilities for hypercube networks. In: Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (July 2011)
6. Kaneko, K., Ito, H.: Fault-tolerant routing algorithms for hypercube interconnection networks. IEICE Transactions on Information and Systems E84-D(1), 121–128 (2001)
7. Seitz, C.L.: The cosmic cube. Communications of the ACM 28(1), 22–33 (1985)
8. Wu, J.: Adaptive fault-tolerant routing in cube-based multicomputers using safety vectors. IEEE Transactions on Parallel and Distributed Systems 9(4), 322–334 (1998)

Finding a Hamiltonian Cycle in a Hierarchical Dual-Net with Base Network of p-Ary q-Cube

Yamin Li[1], Shietung Peng[1], and Wanming Chu[2]

[1] Department of Computer Science
Hosei University
Tokyo 184-8584 Japan
{yamin,speng}@hosei.ac.jp
[2] Department of Computer Hardware
University of Aizu
Aizu-Wakamatsu 965-8580 Japan
w-chu@u-aizu.ac.jp

Abstract. We first introduce a flexible interconnection network, called the hierarchical dual-net (HDN), with low node degree and short diameter for constructing a large-scale supercomputer. The HDN is constructed based on a symmetric product graph (base network). A k-level hierarchical dual-net, $\mathrm{HDN}(B, k, S)$, contains $(2N_0)^{2^k}/(2\prod_{i=1}^{k} s_i)$ nodes, where $S = \{s_i | 1 \leq i \leq k\}$ is the set of integers with each s_i representing the number of nodes in a super-node at the level i for $1 \leq i \leq k$, and N_0 is the number of nodes in the base network B. The node degree of $\mathrm{HDN}(B, k, S)$ is $d_0 + k$, where d_0 is the node degree of the base network. The benefit of the HDN is that we can select suitable s_i to control the growing speed of the number of nodes for constructing a supercomputer of the desired scale. Then we show that an HDN with the base network of p-ary q-cube is Hamiltonian and give an efficient algorithm for finding a Hamiltonian cycle in such hierarchical dual-nets.

Keywords: Interconnection networks, Hamiltonian cycle embedding.

1 Introduction

Recently, because of the advances in computer technology and competition among computer makers, supercomputers containing hundreds of thousands of nodes have been built [13]. It was predicted that the parallel systems of the next decade will contain 10 to 100 millions of nodes [2]. The interconnection network plays an important role for achieving high-performance in such parallel systems. The performance of a large-scale parallel computers depends largely on the time complexities of communication schemes, and in turn depends on the diameter of the interconnection network.

An interconnection network consists of switches with multiple communication ports and cables that connect switch ports by following certain topologies. Most of the supercomputers listed in Top500 [13] use Infiniband or Gigabit Ethernet as their switches. For an Ultra-scale parallel computer, the traditional

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 117–128, 2011.

interconnection networks may no longer satisfy the requirements for high-performance computing or efficient communication. For such an Ultra-scale parallel computer, the node degree and the diameter will be the critical measures for the effectiveness of the interconnection networks. The node degree is limited by the hardware technologies and the diameter affects all kinds of communication schemes directly. Other important measures include symmetricity, scalability, and efficient routing algorithms.

The following two categories of interconnection networks have attracted a great research attention. One is the networks of hypercube-like family that has the advantage of short diameters for high-performance computing and efficient communication. Some typical are hierarchical cubic networks [3], dual-cube [6,8], and cube-connected cycles [12]. The other is the family of 2D/3D meshes or tori that has the advantage of small and fixed node degrees and easy implementations. Most supercomputers including those built by CRAY [4], IBM [1], SGI, and Intel use 3D tori or hypercubes.

In this paper, we first introduce a flexible interconnection network, called the *hierarchical dual-net* (HDN). The HDN can connect a large number of nodes with a small node degree, meanwhile keeping the diameter short. The HDN was motivated by recursive dual-net (RDN) [11]. The RDN was proposed recently as a candidate of high-performance interconnection networks for supercomputers of next generations. The problem of the RDN is that it grows too fast in size, and there is no mechanism to control the rate of its growth. Different from the RDN, the scale of the HDN can be controlled by setting a set of suitable parameters while generating an expanded network through dual-construction. We investigate the topological properties of the HDN and show some examples of HDNs with simple base networks of small size.

Linear array and ring are two fundamental networks and many algorithms were designed based on linear array and ring [5]. Thus embedding linear array or ring in networks is important for emulating those algorithms. A *Hamiltonian cycle* of an undirected graph G is a simple cycle that contains every vertex of G exactly once. A *Hamiltonian path* in a graph is a simple path that visits every node exactly once. A graph that contains a Hamiltonian cycle is said to be *Hamiltonian*. A graph is said to be *Hamiltonian connected* if there is a Hamiltonian path between *any* two distinct nodes in the graph. Two algorithms for embedding a Hamiltonian cycle in Dual-Cube and Metacube were given in [7] and [9], respectively. The RDN was proved to be Hamiltonian connected in [10]. In this paper, we show that the hierarchical dual-net is Hamiltonian if the base network is p-ary q-cube and give an efficient algorithm for the cycle construction.

2 The Hierarchical Dual-Net

Since the hierarchical dual-net was motivated by the recursive dual-net, we begin with a brief introduction to RDN [11]. The RDN is constructed recursively by a dual-construction. The dual-construction is a way to expand a given symmetric graph G of size n to a new symmetric graph G' of size $2n^2$. It generates $2n$ copies

of G as subgraphs (denoted as clusters) of G'. Half of them are of class 0 and the others are of class 1. In G', each node in a cluster has a new link that connects the node to a node in a distinct cluster of the other class.

If G is symmetric then the expanded graph G' is unique and symmetric. Therefore, the dual-construction can be applied recursively from a symmetric network (the base network). $RDN(m,k)$ denotes an RDN generated from a base network of size m by applying dual-construction k times.

The problem of an RDN is that its growth rate in size is super-exponential ($O((2m)^{2^k})$). There is very little space for selection of the size of an RDN. For example, let B be a 3-cube, then the sizes of $RDN(8,k)$ will be 2^7, 2^{15}, and 2^{31} for $k = 1, 2$, and 3, respectively. In HDN, we provide a mechanism to control the growth rate through its expansion from a base network. This new interconnection network has a very flexible way for adjusting its size.

The hierarchical dual-net, HDN(B,k,S), has three sets of parameters: B is a *symmetric product graph*, we call it base network; k is an integer that indicates the *level* of the HDN (the times of dual-construction applied); and $S = \{s_1, s_2, \ldots, s_k\}$, where s_i is the number of nodes in a *super-node* at the level i for $1 \leq i \leq k$. All of these terminologies will be defined in the following paragraphs.

Given r graphs $G_i = (V_i, E_i)$, $1 \leq i \leq r$, their product graph $G = G_1 \times G_2 \times \ldots \times G_r$ is defined as the graph $G = (V, E)$, where $V = \{(v_1, v_2, \ldots, v_r) | v_i \in V_i, 1 \leq i \leq r\}$ and $E = \{[(u_1, u_2, \ldots, u_r), (v_1, v_2, \ldots, v_r)]|$ for some j, $(u_j, v_j) \in E_j$ and for $i \neq j, u_i = v_i\}$. In other words, the nodes of the product graph G are labeled with r-tuples, where the ith element of the r-tuples is chosen from the node set of the ith component graph. The edges of the product graph connect pairs of nodes whose labels are identical in all but the jth element, and the two nodes corresponding to the jth elements in the jth component graph are connected by an edge.

Meshes, tori, and hypercubes are typical examples of product graphs. The 2D $p \times q$ torus is $C_p \times C_q$, where C_p and C_q are rings with p and q nodes, respectively. Any node in the torus can be presented by an ordered pair (u, v), where $u \in C_p$ and $v \in C_q$. Note that the product graph $G = G_1 \times G_2$ can be viewed as being constructed from $|V_1|$ copies of G_2 or $|V_2|$ copies of G_1. Similarly, an r-cube is a product of r numbers of K_2 (complete graph of two nodes represented by 0 and 1). So nodes in r-cube can be represented by an r-bit binary number which is an r-tuple of 0 and 1, and two nodes are connected iff they differ in exactly 1 bit.

Given a product graph $G = G_1 \times G_2 \times \ldots \times G_r$, $1 \leq i \leq r$, we define a *quotient graph* Q as $Q = G/G'$ where G' is a sub-product graph of G such that $G = G' \times Q$. We can consider a quotient graph Q as a reduced graph of G with G' mapped into a single node. A graph G is symmetric (node-symmetric) if all its nodes looks alike. A product graph is symmetric if all its component graphs are symmetric.

We use the symmetric product graph as the base network for generating a hierarchical dual-net through dual-constructions. We denote the base network as $B = B_1 \times B_2 \times \ldots \times B_r$ where all of the B_i, $1 \leq i \leq r$, are symmetric. We define a super-node SN of B as a sub-product graph of B. That is, $SN = B_{i_1} \times B_{i_2} \times \ldots \times B_{i_q}$, where $i_j, 1 \leq j \leq q$, are distinct and $q \leq r$.

Let $|B_i| = b_i$ be the number of nodes in B_i for $1 \leq i \leq r$. The HDN$(B, 0, S) = B$ $(S = \emptyset)$ is the base network. For $i > 0$, the HDN(B, i, S), $S = \{s_j | 1 \leq j \leq i\}$, is generated from HDN$(B, i-1, S)$ by a construction to be explained below. First, we define a super-node of level i, denoted as SN^i, to be a sub-product graph of size s_i in B. Then, we define graph Q^i as the quotient graph HDN$(B, i-1, S)/SN^i$. Suppose that there are N_{i-1} nodes in the HDN$(B, i-1, S)$, then the number of nodes n_i in Q^i is N_{i-1}/s_i. The s_i can be 1 or $\prod_{j=1}^{q} |B_{i_j}|$, where $1 \leq i_j \leq r$ and $q \leq r$. That is, s_i can be a product of any number of integers in $\{b_1, b_2 \ldots, b_r\}$. For example, if $r = 3$, $b_1 = 2$, $b_2 = 3$, and $b_3 = 5$, the possible s_i can be 1, 2, 3, 5, 2×3, 2×5, 3×5, or $2 \times 3 \times 5$.

The construction of HDN(B, i, S), $1 \leq i \leq k$, can be defined by a two-steps process: First, we perform a dual-construction on the quotient graph $Q^{i-1} =$ HDN$(B, i-1, S)/SN^i$. Let the graph generated by the dual-construction be Q^i, and the subgraph of two nodes that is connected by a cross-edge of level i be K_2. Then HDN(B, i, S) is the graph which replaces every K_2 in Q^i by a product graph $K_2 \times SN$. We call HDN$(B, i-1, S)$ *cluster* of HDN(B, i, S).

Referring to Fig. 1, suppose that there are n_i super-nodes in an HDN$(B, i-1, S)$, an HDN(B, i, S) consists of $2n_i$ clusters which are divided into two *classes*: class 0 and class 1 with each class containing n_i clusters. That is, the number of clusters in each class is equal to the number of super-nodes in a cluster. At level i, each super-node in a cluster has s_i new links to a super-node in a distinct cluster of the other class. Because there are s_i nodes in a super-node, one node contributes a new link.

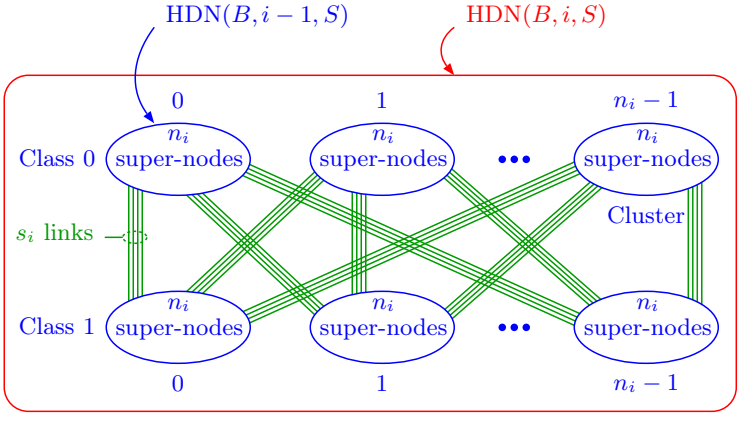


Fig. 1. Building an HDN(B, i, S) from HDN$(B, i-1, S)$

The indexes of the nodes in HDN(B, k, S) can be defined as follows. Let SN^k_{id} be a *super-node_ id* in a cluster of HDN(B, k, S) and N^k_{id} be a *node_ id* in a super-node, then a node in the HDN(B, k, S) can be represented by $(C^k, U^k_{id}, SN^k_{id}, N^k_{id})$ where C^k is the *class_ id* (0 or 1) and U^k_{id} is the *cluster_ id*. A cross-edge at level k connects node $(C^k, U^k_{id}, SN^k_{id}, N^k_{id})$ and node $(\overline{C^k}, SN^k_{id}, U^k_{id}, N^k_{id})$.

We show two small HDN examples in Fig. 2 and Fig. 3, where the base network is a 2-cube. Fig. 2 shows an HDN$(B, 1, S)$ with $s_1 = 2$. There are 2 super-nodes (SN 0 and SN 1) in a cluster and each contains 2 nodes: node 0 and node 1. Each class has 2 clusters (the number of clusters in a class is equal to the number of super-nodes in a cluster). Fig. 3 shows an HDN$(B, 2, S)$ with $s_2 = 4$ based on HDN$(B, 1, S)$ with $s_1 = 2$.

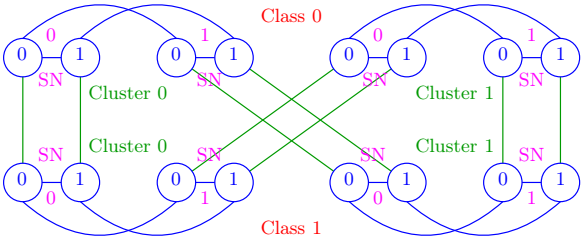


Fig. 2. An HDN$(B, 1, S)$ with $s_1 = 2$

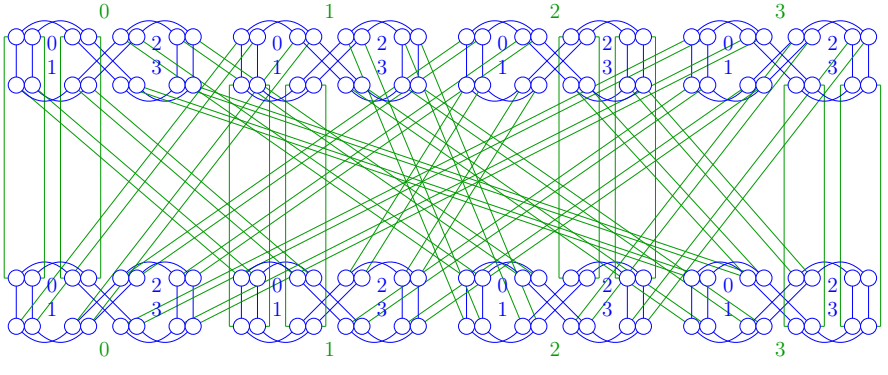


Fig. 3. An HDN$(B, 2, S)$ with $s_1 = 2$ and $s_2 = 4$

We can use a conventional 3D torus as the base network and implement it on silicon or even on printed circuit boards (PCB). The HDN can be implemented by wiring the node boards or the computer racks with cables. It will be interesting to investigate the possibility and scaling behavior of the opposite solution, having HDN at the PCB or at least rack level, and connecting the racks via conventional 3D torus topology.

3 Topological Properties of HDN

Suppose that the node degree of the base network B is d_0, the node degree of the HDN(B, k, S) is d_0+k. Let N_{i-1} be the number of nodes in the HDN$(B, i-1, S)$. there are $N_i = 2 \times N_{i-1}/s_i \times N_{i-1} = 2N_{i-1}^2/s_i$ nodes in the HDN(B, i, S) for $1 \leq i \leq k$, where N_{i-1}/s_i is the number of clusters in each class. That is, the

number of nodes in the HDN(B, k, S) is $(2N_0)^{2^k}/(2\prod_{i=1}^{k} s_i)$, where N_0 is the number of nodes in the base network.

The diameter of an interconnection network is the maximum of the shortest distances between any two nodes. Let the diameter of the HDN$(B, i-1, S)$ be D_{i-1} and the diameter of the super-node be $D(SN^i)$. Then, if we map a super-node into a single node, the diameter of the quotient graph Q^{i-1} is $D(Q^{i-1}) = D_{i-1} - D(SN^i)$. To route a node u in a cluster of class 0 (or 1) to a node v in a different cluster of the same class, we can route u along with a direct link of level i to a node u' in a cluster of class 1 (or 0). This takes one step. Then, we route u' inside the cluster to a node w' which can reach a node w in the same cluster of node v along with direct link of level i. The longest distance between nodes u' and w' is $D(Q^{i-1})$. Similarly, we can route node w' to a node w by one step and then to a node v' which is in the same super-node of v by $D(Q^{i-1})$ steps. Finally, we route v' to node v, this takes $D(SN^i)$ steps. Therefore, we have the following recurrence:

$$D_i = 2 \times (1 + D(Q^{i-1})) + D(SN^i) = 2D_{i-1} - D(SN^i) + 2 \tag{1}$$

Solving the above recurrence, we get the diameter D_k of HDN(B, k, S):

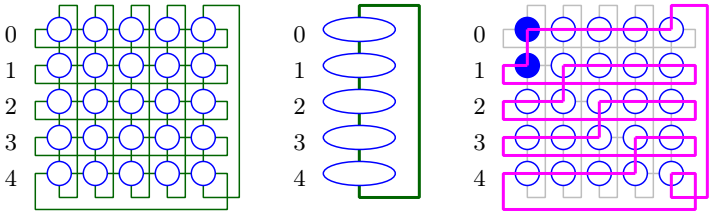
$$D_k = 2^k D(B) - \sum_{j=0}^{k-1} 2^j D(SN^{k-j}) + 2^{k+1} - 2 \tag{2}$$

where $D(B)$ and $D(SN^i), 1 \le i \le k$, are the diameters of the base network and the super-nodes, respectively. The results of the analysis in this section are summarized in the following theorem.

Theorem 1. *Assume that the base network B is a symmetric product graph and $SN^i, 1 \le i \le k$, are sub-product graphs of B with $|SN^i| = s_i$. Let the number of nodes, the node-degree, and the diameter of B be N_0, d_0, and D_0, respectively. Let the diameters of $SN^i, 1 \le i \le k$, be $D(SN^i)$. Then, the number of nodes, the node-degree, and the diameter of HDN(B, k, S), where $S = \{s_1, \ldots, s_k\}$, are $(2N_0)^{2^k}/(2\prod_{i=1}^{k} s_i)$, $d_0 + k$, and $D_k = 2^k D(B) - \sum_{j=0}^{k-1} 2^j D\ (SN^{k-j}) + 2^{k+1} - 2$, respectively.*

4 Hamiltonian Cycle Embedding

This section shows how to construct a Hamiltonian cycle in HDN(B, k, S) with the base network of p-ary q-cube. A p-ary q-cube connects p^q nodes. The term p refers to the number of nodes per dimension and the term q represents the network dimension. Each node can be identified by a q-digit radix-p address $(a_0, a_1, \ldots, a_i, \ldots, a_{q-1})$, where the ith digit a_i represents the node position in the ith dimension. There is a link connecting node A with address $(a_0, a_1, \ldots, a_{q-1})$ and node B with address $(b_0, b_1, \ldots, b_{q-1})$ if and only if there exists i $(0 \le i \le q-1)$ such that $a_i = (b_i + 1)\%\, p$ and $a_j = b_j$ for $0 \le j \le q-1$ and $i \ne j$, where % is a modular operation. For examples, a 4-ary 2-cube is a



(a) 5-ary 2-cube (b) Super-nodes (c) A cycle

Fig. 4. A Hamiltonian cycle in a 5-ary 2-cube

4×4 torus and a 2-ary 4-cube is a 4-cube. A p-ary q-cube is a product graph $G_0 \times G_2 \times \ldots \times G_{q-1}$ with every G_i $(0 \leq i \leq q-1)$ is a p-node ring.

Fig. 4(a) shows a 5-ary 2-cube. Suppose that a super-node contains 5 nodes in a same row in the 5-ary 2-cube, then there are 5 super-nodes that form a ring as shown in Fig. 4(b), the *id* of each super-node is given in the left side. When we construct an HDN(B, k, S), we connect node $(C^k, U^k_{id}, SN^k_{id}, N^k_{id})$ and node $(\overline{C^k}, SN^k_{id}, U^k_{id}, N^k_{id})$ with a link of level k. The super-node_id SN^k_{id} is assigned as below: Supposing that there are n_k super-nodes, super-nodes i and $(i+1)\%\, n_k$ are neighbors. Fig. 4(c) shows a cycle that contains all nodes in the 5-ary 2-cube, starting from a node in super-node 0 to a node in super-node 1.

Lemma 1. *There is a Hamiltonian cycle in a p-ary q-cube.*

Proof. Consider a p-ary 2-cube (2D torus). Suppose that (x, y), for $0 \leq x, y \leq p-1$, is a node address where x is the row number and y is the column number in the 2D torus. We can build a cycle $(0,0) - (1,0) - (1,1) - (2,1) - (2,2) - (3,2) - (3,3) - \ldots - (p-1, p-2) - (p-1, p-1) - (0, p-1) - (0,0)$, then replace every horizontal edge with a path that contains all nodes in a row, see the example in Fig. 4(c). For $q > 2$, we can treat the p-ary $(q-1)$-cube as a ring and do the similar work above. □

We first consider building an HDN$(B, 1, S)$ from base network. There are $2n_1$ clusters and each cluster contains n_1 super-nodes. A *virtual Hamiltonian cycle* that connects all $2n_1$ clusters can be build easily. From the definition of the HDN, we know that there is a cross-edge at level k between node $(C^k, U^k_{id}, SN^k_{id}, N^k_{id})$ and node $(\overline{C^k}, SN^k_{id}, U^k_{id}, N^k_{id})$. By ignoring the node_id N^k_{id}, we can select some super-nodes (two super-nodes per cluster) and links to connect all of the clusters as below.

(0,0,0) – (0,0,1) – (1,1,0) – (1,1,1) – (0,1,1) – (0,1,2) – (1,2,1) – (1,2,2) –
(0,2,2) – (0,2,3) – (1,3,2) – (1,3,3) – ⋯ –
(0,n_1−1,n_1−1) – (0,n_1−1,0) – (1,0,n_1−1) – (1,0,0).

All super-nodes can be inserted to the cycle by replacing the two super-nodes with the Hamiltonian path in each cluster. For example, in the cluster 0 of type 0, two super-nodes 0 and 1 will be replaced with $0 - (n_1-1) - \cdots - 2 - 1$, i.e.,

$$\underline{\mathbf{(0,0,0)}} - \underline{\mathbf{(0,0,1)}} \implies$$
$$\underline{\mathbf{(0,0,0)}} - (0,0,n_1{-}1) - \cdots - (0,0,2) - \underline{\mathbf{(0,0,1)}}.$$

After all super-nodes in clusters were included, we perform *super-node renaming*, by which the renamed super-nodes 0, 1, ..., $n_k - 1$ form a cycle, where super-nodes i and $(i+1)\%n_k$ are neighbors. We assign an id to each super-node increasingly, starting from 0 for super-node (0,0,0).

A Hamiltonian cycle that contains all nodes can be obtained by expanding from super-node level to node level. Fig. 5 shows the Hamiltonian cycle in HDN$(B, 1, S)$ with base network of 4-ary 2-cube and $s_1 = 4$. All super-nodes in the figure were renamed.

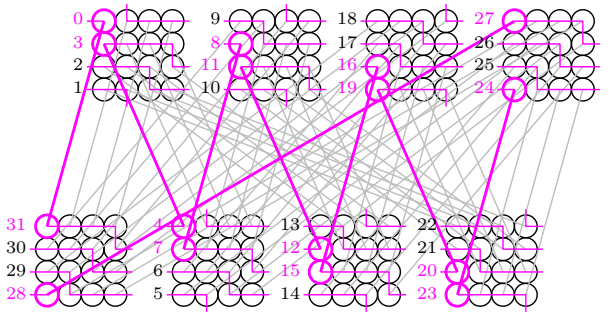


Fig. 5. A Hamiltonian cycle in HDN$(B, 1, S)$ with $s_1 = 4$

To find a Hamiltonian cycle in HDN$(B, 2, S)$, we can use the cycle built in level 1 to

1. build a virtual Hamiltonian cycle connecting all clusters at level 2,
2. insert all super-nodes to the cycle, and
3. expand super-nodes to nodes.

But in the last step, there is a problem: some nodes in a super-node can not be included in the cycle. Super-nodes i and $(i+1)\%n$ are in the virtual Hamiltonian cycle. When we use the cycle built in level 1, only one node in super-node i can be included.

To solve this problem, we re-construct the Hamiltonian path of level 1. The idea is described below. For any node u in super-node i, there is a path $(u \to w)$ that contains all nodes in the super-node. Then, starting from node w, we can build a path connecting all nodes in super-nodes $(i-1)\%n$, $(i-2)\%n$, ..., 1, 0, $n-1$, ..., $(i+2)\%n$, and reach a node in super-node $(i+1)\%n$. In super-node $(i+1)\%n$, we can have a path containing all nodes in the super-node. Suppose that this path ended at node v, then we can use the level-2 cross-edge of node v to reach a node in the next cluster of level 2.

Lemma 2. *For an HDN with the base network of p-ary q-cube, given any node $u \in$ renamed super-node i in a cluster, for $i = 0, 1, \ldots, n-1$, there is a path $u \to v$ containing all of the nodes in the cluster with $v \in$ renamed super-node $(i+1)\%n$.*

Proof. Only considering the super-nodes, the renamed super-nodes form a Hamiltonian cycle in the cluster. Within the super-node i, there is a path $u \rightarrow w$ connecting all of the nodes in the super-node. Because every node in the super-node i has a link to the corresponding node in the super-node $(i-1)\%n$, from node w, we can go to the super-node $(i-1)\%n$. Then we can add all of the nodes in the super-node $(i-1)\%n$. By repeat this procedure, finaly, we can reach the last super-node $(i+1)\%n$ and add all of the nodes in the super-node, ended with node v. □

By using this idea, we can find Hamiltonian cycle in HDN$(B, 2, S)$ for $s_1 = s_2 = 4$. There are 32 clusters (0, 1, ..., 31) in each class, 32 super-nodes (0, 1, ..., 31) in each cluster, and 4 nodes in each super-nodes.

For the case of $s_2 < s_1$, the Hamiltonian cycle can be constructed easily in the same algorithm described above. Compared to the case of $s_1 = s_2$, the number of nodes is reduced.

If $s_2 > s_1$, a super-node of level 2 will contain s_2/s_1 super-nodes of level 1. For example, if $s_1 = 4$ and $s_2 = 16$, the super-node 0 of level 2 contains super-nodes 0, 1, 2, and 3 of level 1, the super-node 1 of level 2 contains super-nodes 4, 5, 6, and 7 of level 1, and so on.

To build a Hamiltonian cycle in an HDN$(B, 2, S)$ with $s_2 > s_1$, we can use the same algorithm described above, and note that when super-node i and super-node $(i+1)$ are used for building a virtual Hamiltonian cycle at level 2, all nodes in super-node i (containing s_2/s_1 super-nodes of level 1) must be included in the path $u \rightarrow w$ before going to super-node $i-1$ (see Fig. 6). This can be done also by applying Lemma 2. In order to go to super-node $(i-1)$, nodes u and w must be in the same super-node of level 1. Other super-nodes, including the super-node $(i+1)$, can be normally included by following the Hamiltonian cycle of level 1.

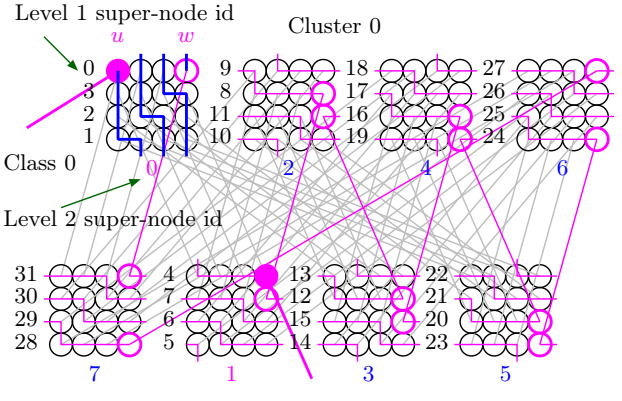


Fig. 6. Part of Hamiltonian cycle at level 2 (cluster 0 of class 0)

In Fig. 6, there are 8 super-nodes in cluster 0 of class 0 ($s_2 = 16$). Super-nodes 0 and 1 are used for constructing a virtual Hamiltonian cycle at level 2. To include all of the super-nodes within the cluster, we construct a super-node path (0−7−6−5−4−3−2−1). The super-node 7 of level 2 contains super-nodes

28, 29, 30, and 31 of level 1. We construct a Hamiltonian path that contains all of the nodes in the cluster based on the Hamiltonian cycle of level 1, but all of the nodes in the super-node 0 of level 2 (containing super-nodes 0, 1, 2, and 3 of level 1) must be routed before going to super-node 7 of level 2 (actually to a node in super-node 31 of level 1). Then the rest part of the path can be constructed as $31-30-\cdots-6-5-4$ (super-node id of level 1), terminated at a node in super-node 1 of level 2.

By using this algorithm, a Hamiltonian cycle in an HDN$(B, 2, S)$ with $s_1 = 4$ and $s_2 = 16$ can be built. In each cluster, the Hamiltonian path is terminated at two solid nodes. The number in the left side of each super-node is the super-node id of level 1; the number in the bottom/up side of each super-node is the super-node id of level 2. We can see from the figure that all of the nodes in a super-node are included before going to the next super-node.

As a summary, no matter of the s_i, a Hamiltonian cycle in HDN(B, i, S) can be found by following the Hamiltonian cycle in HDN$(B, i-1, S)$ and routing all nodes in a super-nodes before going to the next super-node. We formally give the algorithm for finding a Hamiltonian cycle in HDN(B, k, S) in Algorithm 1.

Algorithm 1: HDN_HC(HDN(B, k, S))
begin
  pc_0 = Hamiltonian cycle of base network;
  for $j \leftarrow 1$ **to** k **do** /* k levels */
    group nodes to super-nodes based on s_j;
    n_j = the number of super-nodes in HDN$(B, k-1, S)$;
    based on the cycle build at level $j-1$,
    rename the super-node_id in HDN$(B, j-1, S)$ such
    that super-nodes i and $(i+1)\%n_j$ are neighbors;
    $u = 0$; /* starting node_id in a super-node */
    for $i \leftarrow 0$ **to** $n_j - 1$ **do** /* n_j clusters of each class */
      /* build a Hamiltonian path in cluster i of class 0 */
      /* based on $j-1$ level Hamiltonian cycle pc_{j-1} */
      $hp_i^0 = (0, i, i, u) \rightarrow (0, i, (i+1)\%n_j, v)$;
      /* build a Hamiltonian path in cluster $(i+1)\%n_j$ of class 1 */
      /* based on $j-1$ level Hamiltonian cycle pc_{j-1} */
      $hp_i^1 = (1, (i+1)\%n_j, i, v) \rightarrow (0, i, (i+1)\%n_j, w)$;
      $u = w$; /* end node_id → starting node_id of next cluster */
    endfor
    /* we get the j level Hamiltonian cycle pc_j */
    $pc_j = \emptyset$;
    for $i \leftarrow 0$ **to** $n_j - 1$ **do** /* n_j clusters of each class */
      $pc_j = pc_j \cup hp_i^0 \cup hp_i^1$;
    endfor
  endfor
end

Theorem 2. *If the base network B is a p-ary q-cube, then HDN(B, k, S) is Hamiltonian for any $k > 0$.*

Proof. By following the Algorithm 1, a Hamiltonian *path* can be constructed. To prove the correctness of the theorem, we need to show that the two terminate nodes of the path are neighbors. From Algorithm 1, we know that the two *super-nodes* that contain the terminate nodes of the path are neighbors, so we just prove that these two terminators' node_ids are the same. This means that there is a link between the two nodes.

For the case of $s_i \leq s_{i-1}$, without loss of generality, we select node (0,0,0,0) as the starting node to construct the Hamiltonian path. First, by including all of the nodes inside the super-node (0,0,0), we reach a node with the node_id $s_i - 1$ (a super-node has s_i nodes). Then, from node $(0,0,0,s_i - 1)$, we can add all of the nodes in the rest super-nodes of the cluster (0,0) to a path $p_{0,0}$, terminated at node (0,0,1,0). We show this path in detail as below (the symbol "→" denotes a path, "–" is a link, and there are n_i nodes in the cluster): $p_{0,0} = (0,0,0,0) \rightarrow (0,0,0,s_i-1) - (0,0,n_i-1,s_i-1) - (0,0,n_i-1,0) - (0,0,n_i-1,1) \rightarrow (0,0,n_i-1,s_i-2) - (0,0,n_i-2,s_i-2) - (0,0,n_i-2,s_i-1) - (0,0,n_i-2,0) - (0,0,n_i-2,1) - \cdots - (0,0,s_i-1,s_i-1) - (0,0,s_i-1,0) - (0,0,s_i-1,1) \rightarrow (0,0,s_i-1,s_i-2) - (0,0,s_i-2,s_i-2) - \cdots - (0,0,1,s_i-1) - (0,0,1,0)$.

From node (0,0,1,0), we can go to node (1,1,0,0) via the cross-edge. Similarly, we can construct a path $p_{1,1}$ containing all of the nodes in the cluster (1,1) and terminated at node (1,1,1,0). To include all of the nodes in all clusters, we got a cycle $(0,0,0,0) \rightarrow (0,0,1,0) \rightarrow (1,1,0,0) \rightarrow (1,1,1,0) \rightarrow (0,1,1,0) \rightarrow (0,1,2,0) \rightarrow (1,2,1,0) \rightarrow (1,2,2,0) \rightarrow \cdots \rightarrow (0,n_i-1,n_i-1,0) \rightarrow (0,n_i-1,0,0) \rightarrow (1,0,n_i-1,0) \rightarrow (1,0,0,0) - (0,0,0,0)$.

For the case of $s_i > s_{i-1}$, we route all nodes inside the super-nodes of level i. Suppose that we also select node (0,0,0,0) as the starting node to construct the Hamiltonian path. First, by including all of the nodes inside the super-node (0,0,0), we reach a node with the node_id $s_{i-1} - 1$ which has the same super-node_id 0 of the level $i - 1$. Then, from node $(0,0,0,s_{i-1} - 1)$, we can have a path $p_{0,0}$ containing all of the nodes in the cluster (0,0), terminated at node $(0,0,1,s_{i-1} - 1)$. Next, we go to the node $(1,1,0,s_{i-1} - 1)$ along with a cross-edge of level i. To include all of the nodes in all clusters, we got a cycle $(0,0,0,0) \rightarrow (0,0,1,s_{i-1}-1) - (1,1,0,s_{i-1}-1) \rightarrow (1,1,1,s_{i-1}-2) - (0,1,1,s_{i-1}-2) \rightarrow (0,1,2,s_{i-1}-3) - \cdots \rightarrow (0,n_i-1,n_i-1,2) \rightarrow (0,n_i-1,0,1) - (1,0,n_i-1,1) \rightarrow (1,0,0,0) - (0,0,0,0)$. □

5 Concluding Remarks

The hierarchical dual-net can connect a large number of nodes with a small node-degree and a short diameter. It is a potential candidate for the interconnection network of the supercomputers of the next generation that may have more than one million of nodes. We can select a popular network of small size that is a product graph as the base network and then connect multiple base modules with cross links (cables) to construct a very large-scale hierarchical dual-net. We

can also select a suitable set of integers based on the base network to control the number of nodes in the supercomputer. The base networks can be implemented in a NoC VLSI and high-speed line cables may be used as the cross links to connect PCB modules in cabinets. We presented an algorithm for finding a Hamiltonian cycle in HDN with the base network of p-ary q-cube. There still remain a lot of open issues in the study of the HDN. The future work might include the design of algorithms for collective communication in HDN, the design of fault-tolerant routing algorithms in HDN, and the fault-tolerant cycle embedding in the hierarchical dual-net.

References

1. Adiga, N.R., Blumrich, M.A., Chen, D., Coteus, P., Gara, A., Giampapa, M.E., Heidelberger, P., Singh, S., Steinmacher-Burow, B.D., Takken, T., Tsao, M., Vranas, P.: Blue gene/l torus interconnection network. IBM Journal of Research and Development 49(2/3), 265–276 (2005)
2. Beckman, P.: Looking toward exascale computing, keynote speaker. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2008), University of Otago, Dunedin, New Zealand (December 2008)
3. Ghose, K., Desai, K.R.: Hierarchical cubic networks. IEEE Transactions on Parallel and Distributed Systems 6(4), 427–435 (1995)
4. Cray xt3 supercomputer (2004), http://www.cray.com/products/xt3/index.html
5. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes. Morgan Kaufmann Pub., San Francisco (1992)
6. Li, Y., Peng, S.: Dual-cubes: a new interconnection network for high-performance computer clusters. In: Proceedings of the 2000 International Computer Symposium, Workshop on Computer Architecture, ChiaYi, Taiwan, pp. 51–57 (December 2000)
7. Li, Y., Peng, S., Chu, W.: Hamiltonian cycle embedding for fault tolerance in dual-cube. In: Proceedings of the IASTED International Conference on Networks, Parallel and Distributed Processing, and Applications (NPDPA 2002), Tsukuba, Japan, pp. 1–6 (October 2002)
8. Li, Y., Peng, S., Chu, W.: Efficient collective communications in dual-cube. The Journal of Supercomputing 28(1), 71–90 (2004)
9. Li, Y., Peng, S., Chu, W.: An algorithm for constructing hamiltonian cycle in metacube networks. In: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2007), pp. 285–292. IEEE Press, Adelaide (2007)
10. Li, Y., Peng, S., Chu, W.: Hamiltonian connectedness of recursive dual-net. In: Proceedings of the 9th International Conference on Computer and Information Technology, Xiamen, China, pp. 203–208 (October 2009)
11. Li, Y., Peng, S., Chu, W.: Recursive dual-net: A new universal network for supercomputers of the next generation. In: Hua, A., Chang, S.-L. (eds.) ICA3PP 2009. LNCS, vol. 5574, pp. 809–820. Springer, Heidelberg (2009)
12. Preparata, F.P., Vuillemin, J.: The cube-connected cycles: a versatile network for parallel computation. Commun. ACM 24, 300–309 (1981)
13. TOP500: Supercomputer Sites (June 2011), http://top500.org/

Adaptive Resource Remapping through Live Migration of Virtual Machines

Muhammad Atif[1] and Peter Strazdins[2]

[1] ANU Supercomputer Facility
[2] Research School of Computer Science
The Australian National University, Canberra, ACT, 0200, Australia
{Muhammad.Atif,Peter.Strazdins}@anu.edu.au

Abstract. In this paper we present ARRIVE-F, a novel open source framework which addresses the issue of heterogeneity in compute farms. Unlike the previous attempts, our framework is not based on linear frequency models and does not require source code modifications or off-line profiling. The heterogeneous compute farm is first divided into a number of virtualized homogeneous sub-clusters. The framework then carries out a lightweight 'online' profiling of the CPU, communication and memory subsystems of all the active jobs in the compute farm. From this, it constructs a performance model to predict the execution times of each job on all the distinct sub-clusters in the compute farm. Based upon the predicted execution times, the framework is able to relocate the compute jobs to the best suited hardware platforms such that the overall throughput of the compute farm is increased. We utilize the live migration feature of virtual machine monitors to migrate the job from one sub-cluster to another.

The prediction accuracy of our performance estimation model is over 80%. The implementation of ARRIVE-F is lightweight, with an overhead of 3%. Experiments on a synthetic workload of scientific benchmarks show that we are able to improve the throughput of a moderately heterogeneous compute farm by up to 25%, with a time saving of up to 33%.

1 Introduction

Compute farms, whether for research department clusters, data centers or supercomputing facilities tend to become heterogeneous over time. This is due to incremental extension over a period of time and/or particular nodes being purchased for users with particular needs and the relatively small price differences between these various options. After couple of upgrade cycles, the compute farm becomes a heterogeneous compute farm (HC) constituted of a federation of homogeneous sub-clusters.

Parallel applications have varied computation and communication requirements depending on the domain and the nature of their algorithms. For instance, some applications are floating point intensive while others can be memory or communication intensive. This diverse nature of applications results in varied execution time in the compute farm due to heterogeneity of the nodes.

The issue of effective mapping (or scheduling) of parallel applications onto such heterogeneous system is therefore of great interest to researchers. Two

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 129–143, 2011.

distinct methodologies have evolved overtime for effective resource allocation in an HC. One is to develop a heterogeneity-aware scheduler [8,6] and the other is to write heterogeneity-aware applications [7,11]. In both cases, the allocation of nodes to a parallel job in an HC requires some sort of performance modeling techniques. In performance modeling, an application is profiled to gain an understanding of its performance characteristics. The performance models are evaluated on the different compute nodes and sub-networks to determine the expected speedups (or slowdowns) on various hardware architectures/nodes.

Fine grained performance modeling is capable of reasonably accurate prediction but the associated cost of profiling can be very high in terms of the wall-clock time of the job [11,15]. Due to these costs, these techniques must be applied to applications in an 'offline' mode. The application or some of its iterations are profiled on a set of hardware and scheduling decisions are based on the resultant profile metrics. In the case of a workload change, the application behavior changes and the application needs to be profiled again.

In this paper, we present the design and results of our resource remapping framework, which is able to exploit the heterogeneity in a compute farm to improve throughput. We deal with this issue of heterogeneity by breaking the heterogeneous compute farm into a number of homogeneous sub-clusters. The runtime characteristics of applications are determined with the combination of hardware performance counters/units (PMUs) and the profiling interface to MPI (PMPI). This enables us to predict the performance of the running MPI applications on all the other sub-clusters present in our heterogeneous compute farm. All this is done without the need of changing the application binary or requiring off-line profiling and analysis. We then propose the best-suited sub-cluster for the compute job on the compute farm and migrate the job to this sub-cluster to improve the overall throughput and the average waiting time. For the job migration (or remapping), we make use of the live migration facility provided by most virtual machine monitors. As concluded in [3,17], the benefits of virtualization in HPC environments outweigh the potential overheads, namely potentially slower communication and CPU overhead due to virtualization.

The rest of the paper is organized as follows: We briefly discuss the related work in Section 2. We then present our theoretical framework in Section 3. Implementation details are provided in Section 4. Sections 5 and 6 detail our experiments and results. Conclusions and future work are given in Section 7.

2 Related Work

There is a significant amount of work related to this paper which can be broadly divided into two categories: heterogeneity-aware schedulers and heterogeneity-aware applications.

2.1 Heterogeneity-Aware Schedulers

A number of 'static' heterogeneous cluster scheduling solutions have been proposed which try to map the application processes on the available compute nodes

to improve the application's runtime. The general problem of optimally mapping parallel programs or tasks to machines in a heterogeneous compute farm has been shown to be NP-complete [8], and hence requires heuristics. Braun et al. [6] have presented a detailed comparison of a number of heuristics proposed earlier[1]. However these heuristics have a hard requirement of the estimated execution times of the application on all the possible distinct compute nodes (processors). This either requires the complete execution of the application on all the possible compute nodes before scheduling or 'offline' performance estimation to predict the runtime of the application. All of the subsequent work in the area is static in nature and dynamic load balancing of the HC is missing in these frameworks.

2.2 Heterogeneity-Aware Applications

One approach proposed by several researchers [7,10] to address the computation and communication imbalance of the nodes in an HC is to adapt the parallel application according to the heterogeneity of the compute farm. Here, the parallel application is required to distribute computations unevenly to account for the varied speed and architecture of processors [7]. These techniques require source-code modifications. Charm++ [10] , Prophet [16] and EasyGrid [5] are examples of similar attempts that require source code modifications to build scheduling and load balancing capabilities inside an application.

2.3 Comparison with ARRIVE-F

The approaches for dealing with heterogeneity mentioned above are either based on linear CPU frequency models, require off-line profiling, source code modification or combinations thereof. Our framework is different in that it has all of the following desirable properties: (i) It does not require source code changes or even recompilation of the parallel application. (ii) The applications are profiled and analyzed only during runtime. (iii) The execution time estimates are based on real hardware metrics like floating point operations and L2 misses. Our framework uses the live migration facility of Xen hypervisor to dynamically relocate applications, a feature missing in any job scheduling/estimation framework.

3 Framework

The objective of our framework is to increase the overall throughput of the computer cluster by reducing the average wait time of the submitted jobs. The live migration facility provided by Xen is utilized to move the compute job to the best suitable hardware.

3.1 Assumptions

Our scheduling and profiling framework is targeted towards iterative scientific applications with runtimes in the order of minutes or larger. We need applications

[1] Details and original citations of these heuristics can be found in the paper.

to run for such periods of time in order to recover the time lost during migration. As most scientific applications are iterative in nature [15] and typically run for hours, these assumptions are realistic in most real-world scenarios. We do not cater for the disk I/O intensive jobs, as this introduces complexities beyond the scope of our current work

The heterogeneous compute farm is divided into a number of homogeneous sub-clusters on the basis of similar hardware specifications. No job may be dispatched across two or more of these sub-clusters. Each virtual machine (VM) is provided with exactly one CPU, The one VM per CPU requirement enables us to keep the minimum grain of migration to one process.

We also assume that the approximate wallclock time is provided at the time of submission. It should be noted that the approximate wallclock time is a basic requirement of the widely used backfill algorithm. In Section 3.3, we discuss how this assumption can be relaxed.

3.2 Performance Modeling

We divide our performance model into three sub-models: the computational model, the communication model and the memory utilization model, as will be described below.

The model is built on an execution profile gathered a specific time period of 'τ' seconds. The value of τ can be any positive value large enough to cover at least a single iteration of the parallel application.

Computational Model. The computation model is responsible for generating the CPU profile of the application. We profile all processes on the compute cluster for the characteristics which are exposed through the hardware performance counters. We take an average of the hardware performance counter events for all processes, as there is no significant variation in these events between processes for the scientific applications that we have studied. The events include (but are not limited to) the floating point operation and the L1/L2 cache miss counts.

Once the hardware performance counter (Pctr) event counts over the profiling period is obtained, we use a standard approach to model the computational time by weighing them with their associated penalties. The time for a process belonging to a parallel application 'j', executing on a distinct cluster 'A', is given by:

$$t^{\mathrm{P}}_{A,j} = \sum_{i} Pctr_{A,j,i} \times \frac{Cycles_{A,i}}{f_A} \tag{1}$$

where $Pctr_{A,j,i}$ is the count of a specific performance counter event 'i' (e.g. L2 cache misses) performed by job j. $Cycles_{A,i}$ is the total number of the CPU cycles required to perform the task identified by the $Pctr_{A,i,j}$. E.g. in the case of floating point operations $Cycles_{A,i}$ represents the average number of cycles required to perform a single floating point operation. f_A denotes the CPU frequency of sub-cluster 'A'. The fraction $\frac{Cycles_{A,i}}{f_A}$ is a hardware dependent constant for the sub-cluster 'A'.

In order to predict the computation time ($\tilde{t}^{\mathrm{P}}_{B,j}$) for the job j on a different cluster 'B' , we simply substitute the hardware dependent fraction in Equation 1.

we assume that the event counts will remain approximately the same on cluster 'B' (i.e. $Pctr_{B,j,i} \approx Pctr_{A,j,i}$). This may not necessarily hold for events such as cache misses, and may introduce some inaccuracy into the prediction.

Communication Model. To determine the communication characteristics of a process belonging to a parallel job, we use the MPI profile wrappers known as PMPI, present in most of the MPI-2 compliant implementations [2].

In order to determine the time spent by the process in blocking communication, we log the frequency of distinct messages according to the message size. Let '$n_j^B(s)$' be the total number of distinct messages of size 's' in a time period 'τ'. The total time spent by the process j executing on sub-cluster 'A' in blocking communications, $t_{A,j}^{\mathrm{B}}$, is given by:

$$t_{A,j}^{\mathrm{B}} = \sum_s n_j^{\mathrm{B}}(s) \times l_A(s) \tag{2}$$

where $l_A(s)$ is the communication network's latency for a message size of 's' on sub-cluster 'A'. The latency is determined through micro-benchmarks. To predict the time spent in blocking communication for the target cluster 'B', we simply use Equation 2 to evaluate the expression $t_{B,j}^{\mathrm{B}}$.

Predicting accurate communication times for non-blocking communication is more difficult because of the potential overlap of communication and computation [15]. A non-blocking communication is usually followed by an `MPI_Wait()`. Here, the time that an application has to wait for the communication to complete is more relevant than the network latency as it can capture the degree of overlap. We have devised a lightweight approximation for the non-blocking communication. This is done by logging the MPI_Request object in each non-blocking message and comparing it against the corresponding `MPI_Wait()` call.

The time a process waits for all the non-blocking communication to finish during the time period τ can be given as:

$$t_{A,j}^{\mathrm{N}} = \sum_s n_j^{\mathrm{N}}(s) \times w_A(s) \tag{3}$$

where $n_j^{\mathrm{N}}(s)$ represents the number of non-blocking messages of distinct size 's'. $w_A(s)$ is the average waiting time for all messages of size 's'. $w_A(s)$ is calculated by logging each non-blocking send or receive and calculating the corresponding time a process had to wait for that particular message to complete.

In order to predict the waiting time for cluster 'B' with a different interconnect, we assume that the degree of overlap will be approximately preserved. Thus, we simply multiply the term $n_j^{\mathrm{N}} \times w_A(s)$ in Equation 3 with the ratio of latencies of cluster 'B' to cluster 'A'. The approximate time on cluster 'B' that will be spent in non-blocking communication can then be given by:

$$\tilde{t}_{B,j}^{\mathrm{N}} = \sum_s n_j^{\mathrm{N}}(s) \times w_A(s) \times \frac{l_B(s)}{l_A(s)} \tag{4}$$

Memory Utilization. The swap partition utilization by any process belonging to an HPC application has a time penalty in the order of minutes even for the

application with a wall clock time in seconds. Our framework does not predict the performance of such a case; however, it is able to detect take appropriate actions so that the application can avoid thrashing. The implementation details of memory utilization are given in Section 4.

Predicted Execution Time on Another Cluster. The time gained or lost by the job if it was executed on cluster 'B' can be obtained by subtracting the predicted computation and communication times for sub-cluster 'B' from the profiled times of sub-cluster 'A':

$$t_{A\rightarrow B,j} = (t^{\mathrm{P}}_{A,j} - \tilde{t}^{\mathrm{P}}_{B,j}) + (t^{\mathrm{B}}_{A,j} - t^{\mathrm{B}}_{B,j}) + (t^{\mathrm{N}}_{A,j} - \tilde{t}^{\mathrm{N}}_{B,j}) \tag{5}$$

where $t_{A\rightarrow B,j}$ is the predicted time saved or lost by the job j on sub-cluster 'B' for every τ seconds. Here a negative value means that the application will run slower on sub-cluster 'B'.

Equation 5 forms the basis of the migration prediction model as discussed in Section 3.3. In order to determine the execution time $T_{A\rightarrow B,j}$ of job j on sub-cluster 'B' based on the profile data on sub-cluster 'A', $t_{A\rightarrow B,j}$ is multiplied with the total number of τ blocks in the actual runtime of the application on sub-cluster 'A', $T^{act}_{A,j}$:

$$T_{A\rightarrow B,j} = t_{A\rightarrow B,j} \times \frac{T^{act}_{A,j}}{\tau} \tag{6}$$

The accuracy of the model is discussed in Section 5.3.

3.3 Migration Prediction

As discussed, the main objective of the framework is to increase the throughput of the compute farm by migrating jobs to the best suited sub-cluster. This requires the framework to compute the predicted execution time of each active job on all sub-clusters. As the sub-clusters in the compute farm might be busy entertaining a number of other parallel jobs (e.g. k, l, m), the framework inspects all jobs and treats all sub-clusters as 'potential targets'. If the potential target cluster is free, then the job is migrated from the source cluster to the target cluster. However, if the potential target cluster is busy servicing another job, the impact of job migration on both sub-clusters is calculated, i.e. the time lost of by both jobs during the migration.

The total time saved by the compute farm $T^{A\leftrightarrow B}_{j,k}$ if two jobs j and k swap their respective sub-clusters can be given by:

$$T^{A\leftrightarrow B}_{j,k} = \eta_j t_{A\rightarrow B,j} \times \frac{T^{\mathrm{rem}}_j}{\tau} + \eta_k t_{B\rightarrow A,k} \times \frac{T^{\mathrm{rem}}_k}{\tau} - T^{M}_{j,k} \tag{7}$$

where η_j and η_k are the number of processes of the respective jobs. The terms $\frac{T^{\mathrm{rem}}_j}{\tau}$ and $\frac{T^{\mathrm{rem}}_k}{\tau}$ give the remaining time blocks for the applications j and k respectively. The remaining time T^{rem} is calculated by subtracting the elapsed time from the users' runtime estimate. The term $t_{B\rightarrow A,k}$ is the predicted time for the job k on sub-cluster 'A' and is derived in the similar way as $t_{A\rightarrow B,j}$.

The term $T^M_{j,k}$ is the average time stretch introduced in the wall clock time of jobs j and k by the migration, weighted by the number of processes involved. The value of $T^M_{j,k}$ depends on the network bandwidth and CPU frequency of the involved sub-clusters, as well as the network utilization and memory dirtying rate of the involved jobs [3]. $T^M_{j,k}$ can be determined through the profile data but, for simplicity, we use an approximation $T^M_{j,k} = T^m(\eta_j + \eta_k)$, where T^m is the average wall clock time stretch introduced by a single process migration. This approximation assumes that the migration of multiple processes is effectively overlapped. A negative value of $T^{A\leftrightarrow B}_{j,k}$ means that the proposed migration will result in reduced throughput of the compute farm.

The job j should at least run for $\frac{T^m}{t_{A\rightarrow B,j}}$ profile time blocks (τ) to recover the cost of migration, i.e.

$$\frac{T^{rem}_j}{\tau} > \frac{T^m}{t_{A\rightarrow B,j}} \tag{8}$$

Only if the above condition correspondingly holds on job k as well is the pair further considered for migration.

To remove any further dependence on the users' runtime estimates (in particular the tendency to over-estimate the application run-time to avoid getting their jobs killed [14]), we introduce a parameter β. We assume that the candidate processes will run for at least another $\beta\tau$ time, and only proceed with the migration if there is sufficient benefit over this restricted interval. β is a configurable parameter and its value can be adjusted according the migration overheads and the average job length in the compute cluster such that $\beta\tau > T^m$.

The total time saved by the compute farm if the migration proceeds over this period can then be calculated as:

$$\bar{T}^{A\leftrightarrow B}_{j,k} = (\eta_j t_{A\rightarrow B,j} + \eta_k t_{B\rightarrow A,k}) \times \beta - T^M_{j,k} \tag{9}$$

Equation 9 suggests that, for $\bar{T}^{A\leftrightarrow B}_{j,k} > 0$, the compute farm will post an improved throughput.

To take into account possible inaccuracies in our performance model, we introduce a further threshold value T^{Thresh}, which is the minimum expected benefit required for the migration sequence to proceed. The value of T^{Thresh} is a configurable parameter, and can be a percentage of $\beta\tau(\eta_j + \eta_k)$. For example a value of $T^{Thresh} = 0.1\beta\tau(\eta_j + \eta_k)$ suggests that the jobs j and k should only be migrated if the expected gain in time from the proposed migration is at least 10% in the next time interval $\tau\beta$.

$$\bar{T}^{A\leftrightarrow B}_{j,k} > T^{Thresh} \tag{10}$$

Equations 8 and 10 form the basis of the migration decisions.

3.4 Migration Decisions

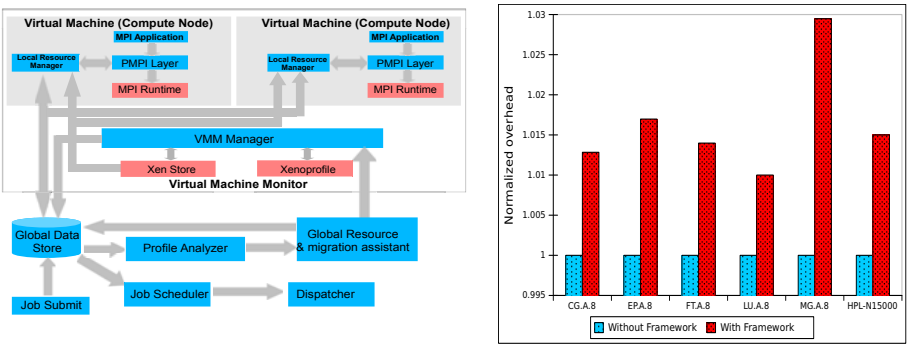
At any given time, a compute farm may have a number of sub-clusters with a number of active jobs. In order to make migration decisions that improve the throughput of the compute farm, we compare the active jobs against each other;

except for the ones which are on the same sub-cluster. The left hand side of Equation 10 is used to develop a comprehensive list in descending order. We then traverse through the list in order and migrate the jobs if they qualify for final migration check, i.e. enough resources are available on both the sub-clusters to entertain the swapped jobs.

We have identified a number of possible migration scenarios. An extended description of these scenarios may be found in our technical report [4]. To ensure that the framework benefits from the migration decisions, a migrated job has to wait for $\beta\tau$ seconds to be considered for another migration.

4 Implementation

The framework requires a runtime engine to generate the computation and communication profiles of the active jobs. A high level diagram of the scheduling and profiling framework is presented in the Figure 1(a).



(a) Block view of the framework. (b) Overheads of the framework.

Fig. 1.

The user submits the job through the 'job submit' routine. The job queue is continuously analyzed and jobs are dispatched to based on the easy backfill algorithm [14]. The scheduler ensures that no job is dispatched among two different sub-clusters. The scheduler also tries to 'co-locate' the application processes if possible.

Each compute node has a *local resource manager daemon* (LRM) responsible for updating the node statistics like memory utilization and the communication network utilization. The memory profile of each process is obtained through the `/proc` filesystem, whereas the communication network profile is obtained through the PMPI layer. In order to achieve a lightweight MPI message profile, the LRM shares memory with the PMPI layer. The PMPI layer logs the frequency of every distinct message in terms of message size, communication destination, and message type. The LRM periodically updates the summary of these statistics to the global data store, which is a MySQL database.

The CPU profiles are generated through passive domain profiling provided by the Xenoprofile [13]. The CPU profiles are read by the *VMM daemon* (VMMD) which resides in domain-0. Like the LRM, the VMMD periodically sends the data to the global data store. To minimize profile overhead, the event counter threshold is kept at 500K and the sample directory is kept in a RAMFS.

The *profile analyzer* is responsible for predicting the job completion times and is based on the CPU and communication models explained in Sections 3.2 and 3.2. The profile analyzer reads the profile information from the global data store and does not contribute to the profiling overhead. The *migration assistant* performs the live migration of compute nodes if required.

We have named our framework as ARRIVE-F (Adaptive Resource Relocation in Virtualized Environments). ARRIVE-F is licensed under GPL version 3, and its source code can be downloaded from `http://cs.anu.edu.au/~Muhammad.Atif/opensource/arrivef`.

5 Experimental Results

In this section, we present initial results of the profiling and resource relocation framework. We use simple scenarios to show that our framework is capable of improving the throughput of the compute cluster and reduce the average wait time of the submitted jobs.

5.1 Experimental Platform

For the experiments we use Xen 3.3.0 compiled from source using GCC 4.2.4. Xen 3.3.0 is patched with our live migration optimization, which is able to reduce the migration overhead by 50% [3]. The domain-0 kernel is Xeno-linux 2.6.31.12.

Table 1. Heterogeneous Compute Farm

Cluster	CPU Type	Memory	**Total Machines**
A	4 × Opteron 2.2 Ghz	4 GB	2
B	4 × Phenom II 3.0 Ghz	4 GB	2
C	4 × Phenom II 3.0 Ghz	4 GB	2
D	2 × Athlon 2.0 Ghz	1.2 GB	4

The experimental cluster consists of a number of sub-clusters as given in Table 1. Each VM is provided with one CPU and two Gigabit Ethernet network interfaces, with the exception of sub-cluster 'C' which utilizes Fast Ethernet for MPI communication for added heterogeneity. Both the network interfaces use the Xen bridge architecture to allow VM migration. The RAM of each VM is set to 512 MB initially. The VMs are mounted through NFS to enable migration.

5.2 Applications

We utilize a selection of the NAS Parallel Benchmarks (NPB) [1] and High Performance Linpack (HPL) [9] to validate the framework. For these applications,

we have found it sufficient to limit the performance counter set used for profiling to floating point operations and L1/L2 cache misses.

All the benchmarks were compiled with GCC 4.3.2 with amdfam10 optimization. In Section 6, we have modified the NPB by increasing the number of iterations. This is done to make the benchmark run for a longer period of time as required by the framework. Where applicable, the total number of iterations is shown along with the benchmark name, class and number of processes.

For the experiments in the Section 6, the value of τ is 50 seconds and β is 20 time blocks. Jobs with an approximate remaining runtime of $\beta\tau$ or more seconds were considered for migration. $T^{Thresh} = 0.1\beta\tau\eta$ is utilized. where η is the total number of processes of all the jobs involved in the migration sequence.

5.3 Comparison of Predicted and Actual Execution Times

In this section, we present the comparison of the actual execution times with the predicted times as generated by the framework. For the ease of comparison, we present two tables: one for the computational model accuracy and the other for the communication model accuracy. We have used NPB class 'B' for these experiments as their wall clock times are higher than those for class 'A'.

For the CPU model, we utilize sub-clusters 'A' and 'B', which have similar network but different CPU architectures. We execute the benchmark on cluster 'A' and the framework predicts the wall clock time for the cluster 'B'.

Table 2. CPU Model Accuracy

Benchmark	T_A^{act}	T_B^{act}	$T_{A\to B}^{act}$	% Acc **CPU**	% Acc **Prof**
CG.B.8	104.5	57.9	71.2	75.5	81.3
FT.B.8	98.2	81.6	77.2	88.6	94.6
LU.B.8	240.7	81.3	103.4	46.0	78.6
HPL.N15K	150.7	62.2	68.5	56.2	90.8

Table 3. Comm. Model Accuracy

Benchmark	T_C^{act}	T_B^{act}	$T_{C\to B}^{act}$	% Acc **Prof**
CG.B.8	141.0	57.9	66.2	88.8
FT.B.8	375.1	81.6	79.3	97.2
LU.B.8	106.8	81.3	61.8	76.0
HPL.N15K	150.7	62.2	80.2	71.1

The columns T_A^{act} and T_B^{act} give the time taken by the benchmark on the respective sub-clusters. The column $T_{A\to B}^{act}$ gives the wall clock time predicted by the framework for the cluster 'B' based on the data collected from the cluster 'A' and is calculated from Equation 6 . The column '% Acc CPU' shows the percentage accuracy of the execution time projection based on linear CPU frequency scaling, which forms the basis of the research works presented in Section 2. The CPU frequency speedup ratio is calculated by f_B/f_A. The column '% Acc Prof' gives the execution time percentage accuracy of our framework. It is clear that our framework consistently outperforms the CPU frequency method.

For the communication profile, we utilized clusters 'B' and 'C'. The only difference between these two clusters is the communication link.

It can be seen from Table 3 that the accuracy of the communication model decreases for the application which sends or receives a high frequency of messages. This is due to the fact that Xen uses a split driver interface which is highly CPU intensive For applications like LU, which sends a high number of messages

of small size, the CPU load due to processing on domain-0 is high, and this skews the projections. We plan to introduce domain-0 profiles in the framework to overcome this issue.

5.4 Framework Overheads

Figure 1(b) gives the comparison of the relative wall clock times for a set of NPB class 'B' and HPL benchmarks with and without the prediction framework. Approximately 1.2% of the profile overhead is due to Xenoprofile. The communication profiles have an overhead of 0.3% to 0.7%, depending upon the communication rate of the benchmark. An exception is the MG benchmark, which uses non-blocking messages and the overhead is due to the retrieval of the corresponding non-blocking requests, as discussed in Section 3.2.

6 Compute Farm Throughput

To determine the throughput of the compute farm with the resource relocation framework, the scheduler is supplied with workloads which are representative of real-world data. We then compare it against the 'base-run' using the same scheduler but without the profiling framework active. The scheduler is based on FCFS easy backfill algorithm, as discussed in Section 4. In both the experimental runs, the scheduler is supplied with the same stream of jobs with the approximate expected runtime given for each job. The Lublin-Feitelson model [12] was used to generate the job queue.

The list of jobs generated by the model was randomly allocated to one of the NPB kernels. The number of iterations of these benchmarks were changed to match the approximate expected runtime provided by the Lublin-Feitelson model. This was done by calculating an iteration size of each benchmark on cluster 'A'. We have conducted several experiments with different job streams. Due to space limitation we are presenting the results of only one scenario. The complete list, with the expected and actual runtimes and detailed analysis of the migration decisions, can be found in our technical report [4].

6.1 Experiment 1

In this experiment a stream of 330 jobs was given to the scheduler. The number of jobs was arbitrarily chosen to keep the experiment runtime to approximately 3 hours. The inter-arrival-time of the jobs was selected to represent a normal rush hour.

The scenario was tested three times and results of each run are presented in Table 4. The columns 'Base-run' and 'Migration Run' give the statistics of the respective runs. The columns 'W.Time' and 'TA.Time' represent the average wait time and the average turnaround time of jobs respectively. The column 'Total time' gives the total time taken by the respective base-run or the migration run to complete the execution of all the 330 jobs supplied to the scheduler. The

column 'jobs @ mig.run' gives the total number of jobs completed by the base-run in the time that took migration run to complete the execution of all the 330 jobs.

The average throughput improvement achieved for this particular case is 27%. The average time saved by the framework is 3104 seconds, which reflects an impressive time saving of 32%. Compared to the base-run, the migration run reduced the average wait time for jobs by 55% and the average turnaround time of the jobs was improved by 54%.

As a single second difference in the execution of any job can result in different sub-cluster allocations to subsequent jobs, the total times are different for each experimental run.

Table 4. Base-run vs migration run

Sr. No	Base-run				Migration Run		
	W.Time	TA.Time	Total Time	Jobs@mig.run	W.Time	TA.Time	Total Time
1	4493	4654	12434	255	2747	2913	9740
2	4469	4684	12782	267	2961	3129	9380
3	4531	4749	12837	258	2982	3119	9619
Avg.	*4498*	*4696*	*12684*	*260*	*2896*	*3053*	*9580*

For the experiment run with the migration framework, three distinct migration decisions were made and are detailed in Table 5.

Table 5. Job Migration Details

Migration Number	Job Name	Sub-cluster	T^{est}	T^{act} Base	$T^{act}_{A \leftrightarrow B}$ Mig.
Migration 1	FT.B.4.20	D	92	1148 (D)	415
	FT.A.4.156	C	230	95 (C)	108
Migration 2	MG.B.8.5132	A	2697	2332 (A)	1769
	FT.B.8.506	B	2174	2226 (B)	2222
Migration 3	CG.B.4.2286	A	3268	3408 (D)	2043
	LU.A.1.7385	A	5869	5870 (A)	4161
	LU.A.8.12334	C	1850	1058 (B)	1838
	LU.B.1.455	A	1500	1850 (A)	1447

1- Like the base-run, in the migration run, FT.B.4.20 was allocated to Cluster 'D' which has 1.2 GB of physical memory per physical machine. FT.B.4 benchmark requires a minimum of 800 MB of physical memory to avoid thrashing and takes approximately 90 seconds to complete. As each VM on cluster 'D' can have a maximum of 512 MB of memory, the job used the swap partition. This resulted in the job taking 1148 seconds to complete. The migration framework was able to detect the case and the job was swapped with the job (FT.A.4.156) running on cluster 'C'. The column 'T^{est}' shows the estimated time generated by the Lublin-Feitelson model. 'T^{act} Base' gives the actual time taken by the job to complete in the base run. The brackets contain the sub-cluster on which the job executed in the base-run. The column '$T^{act}_{C \leftrightarrow D}$ Mig.' gives the time taken by the jobs in the migration run.

2- The second migration performed by the framework was the sub-cluster swap between MG.B.8.5132 and FT.B.8.506. FT.B.8 is bandwidth bound and does

not benefit from the faster CPU clock. The framework was able to determine that the compute farm will benefit from the migration swap of these two jobs and proceeded accordingly.
3- The last migration was in fact a sequence of migrations and represents the complex migration scenario presented in Section 3.4. Here, the framework migrated LU.A.8.12334 from sub-cluster 'C' to sub-cluster 'A'. As seen from the previous experiments, LU.A.8 has less penalty on cluster 'C' compared to the other NPB kernels. However, the framework was able to co-locate all the processes of CG.B.4.2286, eliminating the inter-node communication through slower network interface. This resulted in CG.B.4 benefiting from the faster CPUs. Similarly LU.A.1 and LU.B.1 benefited from the higher flop rate of sub-cluster 'C'. LU.A.8.12334 lost CPU time which was made up by the faster communication network of cluster 'A'.

The migration decisions enabled the HC to save a total of 3,984 seconds. However, the actual time difference from the base run by the HC was somewhat less than this (see Table 4), due to the fact that, once migration occurs, different scheduling decisions were made for subsequently introduced jobs.

6.2 Further Experiments

In a second experiment, we removed the FT.B.4.* benchmark from the workload generation to ensure that no benchmark thrashes the swap partition. A list of 212 jobs was generated. The average total time of the base run was 7792s; a single migration of FT.B.8.1093 and LU.B.8.3088 between clusters A and B in each of the three trials resulted in an averaged total time of 7262s, an improvement of 7%. The average waiting and turnaround times improved by less than 1%.

In a third experiment, we removed the sub-cluster 'C' (Fast Ethernet) and the Lublin-Feitelson model generated a workload reflecting a longer arrival time between the jobs (i.e. a lesser load). This saw ARRIVE-F making three migration decisions, all based on computational requirements. The overall throughput was improved by 13% and total time saved was 3360 seconds, representing an improvement of 33%. The average waiting time was reduced by a very impressive 298% and the average turnaround time was improved by 230%.

Further details of experiments may be found in our technical report [4].

7 Conclusions and Future Work

In this paper, we have demonstrated that the issue of heterogeneity in compute farms can be successfully addressed by using the live migration facility provided by virtual machine monitors. We show that the benefits of virtualization go far beyond its traditional use in the data center environments. We have presented a framework which can measure application performance and potential bottlenecks by leveraging hardware performance counters and the PMPI layer. We have shown that the models based on the CPU frequency do not make accurate predictions and incorporating hardware performance counter data leads to an improved prediction.

By implementing a lightweight profile engine with overheads of less than 3%, we are able to predict the execution times for all the jobs on every distinct hardware platform (sub-cluster) with sufficient accuracy to make appropriate migration decisions. In two of our experiments, these improved the throughput of an HC by 25% and the total time saved by over 30%.

We also believe that this paper contributes towards the understanding of how the system level measurements can be used to characterize an application and estimate its execution times in a heterogeneous compute farms.

We envision that such 'online profiling and migration frameworks' will become an essential part of any cloud deployment. For this we are working to extend the framework to the high performance cloud and grid infrastructures. Our main focus is a balance of the throughput of an HC with power savings. In the future we also plan to run real life workloads and test the framework on a larger compute farm.

Acknowledgments. We thank Alexander Technology and Platform Computing for donating us the hardware. We especially thank Richard Alexander for his vision that enabled this research work.

References

1. NAS Parallel Benchmarks (September 2010), `http://www.nas.nasa.gov/Software/NPB`
2. OpenMPI (June 2010), `http://www.open-mpi.org/`
3. Atif, M., Strazdins, P.: Optimizing live migration of virtual machines in smp clusters for hpc applications. In: IFIP International Conference on Network and Parallel Computing Workshops, pp. 51–58 (2009)
4. Atif, M., Strazdins, P.: Adaptive Resource Remappin In Virtualized Environments - Framework. Computer Science Technical Report TR-CS-11-01, Australian National University (May 2011), `http://cs.anu.edu.au/techreports/`
5. Boeres, C., Rebello, V.E.F.: Easygrid: towards a framework for the automatic grid enabling of legacy mpi applications: Research articles. Concurr. Comput.: Pract. Exper. 16, 425–432 (2004)
6. Braun, T.D., Siegel, H.J., Beck, N., Boloni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. Journal of Parallel and Distributed Computing 61, 810–837 (2001)
7. Dongarra, J., Lastovetsky, A.: An overview of heterogeneous high performance and grid computing. In: Di Martino, B., Dongarra, J., Hoisie, A., Yang, L., Zima, H. (eds.) Engineering the Grid: Status and Perspective (2006)
8. Ibarra, O.H., Kim, C.E.: Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. ACM 24(2), 280–289 (1977)
9. University of Tennesse Innovative Computing Laboratory. High performance linpack benchmark (March 2009), `http://www.netlib.org/benchmark/hpl/`
10. Kale, L.V., Kale, L.V.: Charm++ and ampi: Adaptive runtime strategies via migratable objects. In: Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications, no. 9780470558027. John Wiley & Sons, Inc., Chichester (2009)

11. Katramatos, D., Chaplin, S.J.: A cost/benefit estimating service for mapping parallel applications on heterogeneous clusters. In: IEEE International Conference on Cluster Computing, Cluster 2005 (2005)
12. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing, 1105–1122 (November 2003)
13. Menon, A., Santos, J.R., Yoshio, T.: Diagnosing performance overheads in the xen virtual machine environment. In: VEE 2005: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, pp. 13–23 (2005)
14. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel Distrib. Syst. 12, 529–543 (2001)
15. Nakazawa, M., Lowenthal, D.K., Zhou, W.: The mheta execution model for heterogeneous clusters. In: SC 2005: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. IEEE Computer Society, Washington, DC, USA (2005)
16. Weissman, J., Zhao, X.: Scheduling parallel applications in distributed networks. Cluster Computing 1, 109–118 (1998) 10.1023/A:1019073113216
17. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Paravirtualization for HPC systems. In: Min, G., Di Martino, B., Yang, L.T., Guo, M., Rünger, G. (eds.) ISPA Workshops 2006. LNCS, vol. 4331, pp. 474–486. Springer, Heidelberg (2006)

LUTS: A Lightweight User-Level Transaction Scheduler

Daniel Nicácio[1], Alexandro Baldassin[2], and Guido Araújo[1]

[1] IC-UNICAMP
{dnicacio,guido}@ic.unicamp.br
[2] UNESP - Univ Estadual Paulista, Rio Claro, Brazil
alex@rc.unesp.br

Abstract. Software Transactional Memory (STM) systems have poor performance under high contention scenarios. Since many transactions compete for the same data, most of them are aborted, wasting processor runtime. Contention management policies are typically used to avoid that, but they are passive approaches as they wait for an abort to happen so they can take action. More proactive approaches have emerged, trying to predict when a transaction is likely to abort so its execution can be delayed. Such techniques are limited, as they do not replace the doomed transaction by another or, when they do, they rely on the operating system for that, having little or no control on which transaction should run. In this paper we propose LUTS, a Lightweight User-Level Transaction Scheduler, which is based on an execution context record mechanism. Unlike other techniques, LUTS provides the means for selecting another transaction to run in parallel, thus improving system throughput. Moreover, it avoids most of the issues caused by pseudo parallelism, as it only launches as many system-level threads as the number of available processor cores. We discuss LUTS design and present three conflict-avoidance heuristics built around LUTS scheduling capabilities. Experimental results, conducted with STMBench7 and STAMP benchmark suites, show LUTS efficiency when running high contention applications and how conflict-avoidance heuristics can improve STM performance even more. In fact, our transaction scheduling techniques are capable of improving program performance even in overloaded scenarios.

1 Introduction

The recent advent of multicore processors has renewed the interest in concurrent programming and effectively moved it into mainstream. During the last few years, extensive research has been carried out into new programming models and abstractions, of which transactions (or atomic blocks) is certainly a promising one [1]. A transaction can be viewed as an atomic block of code that is executed in isolation from the rest of the system, offering a convenient synchronization mechanism. One of the major advantages of transactions relies on the fact that it moves most of the burden from programmers to the underlying transactional memory (TM) subsystem.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 144–157, 2011.

While transactions perform well on workloads with reasonable amounts of parallelism, they may substantially degrade performance on those which exhibit higher data contention. In such cases, two or more transactions tend to compete for the same data and, since at least one of the accesses is an update, a *conflict* occurs. Deciding what to do when a conflict arises is the responsibility of the so called *contention manager* [2], an important component of any TM system. Usually, a conflicting transaction is aborted and restarted after some time. Although certain contention policies may provide good performance for some workloads, it can also considerably downgrade performance for others.

Contention management is a central issue when the system is characterized by *pseudo parallelism* [3]. Pseudo parallelism occurs when the total number of system-level threads is greater than the available number of processor cores, forcing several threads to share the same core. In such scenario, performance is greatly degraded as: (i) more transactions running simultaneously naturally increase the likelihood of conflicts; (ii) same-core transactions are more likely to conflict to each other, specially when they are longer than the scheduler quantum; and (iii) more threads sharing the same core tend to cause more cache misses and page faults. Dealing with the pseudo parallelism is a central problem in the design of efficient contention managers.

In this letter we propose LUTS, a Lightweight User-level Transaction Scheduler. LUTS implements a fully cooperative scheduler, and does not rely on the system-level scheduler for context switching the transaction threads. LUTS novel user-level cooperative approach presents two main advantages when compared to state-of-the-art TM schedulers. First, it deals with the pseudo parallelism issue in an elegant way, by only spawning as many system-level threads as the number of available processor cores and handling the exceeding threads internally. Second, LUTS allows the TM subsystem to efficiently access the runnable transaction queue and switch the execution to any of them. This allows the design of more precise proactive scheduling schemes, not possible with the current approaches, which are restricted to either serialization or yielding.

We discuss in this letter LUTS implementation along with two proactive conflict-avoidance heuristics. To evaluate LUTS feasibility, we make use of the STMBench7 [4] and STAMP [5] benchmarks. Overall, we notice that LUTS performs better than existing techniques on STAMP and STMBench7 benchmarks.

More precisely, the contributions of this paper are as follows:

- A cooperative scheduler (Section 3), designed with the goal of eliminating pseudo parallelism. It creates at most as many system-level threads as available processor cores, transparently handling the exceeding threads.
- Three novel proactive conflict-avoidance heuristics (Section 4) that uses LUTS scheduling capabilities to avoid starting transactions that are likely to conflict. When a transaction is about to start, we check the probability of conflict among executing transactions. If this probability is high, we choose a transaction that is less likely to abort from LUTS runnable queue.
- An evaluation of LUTS approach (Section 5) to contention management through STMBench7 [4] and STAMP 0.9.10 [5] benchmarks. In general, the

experimental results show LUTS efficiency in dealing with pseudo parallelism issues. When compared to earlier approaches such as [6] and [7], LUTS presented the best overall performance.

2 Related Work

STM contention management has been primarily researched in the context of modular contention managers, introduced by Herlihy et al. [8] for obstruction-free STM implementations. Due to its modular nature, a plethora of contention policies [2, 9, 10, 11] have been devised with the purpose of decreasing the number of conflicts and enabling system progress. Lock-based implementations have usually employed simpler heuristics, such as aborting the conflicting transaction and delaying the restart by using an exponential backoff mechanism. Despite the progress on contention managers, STM systems have not been able to anticipate conflicts and increase system throughput [6, 12]. On the contrary, traditional contention managers use a reactive (damage-control) strategy, instead of focusing on avoiding conflicts in the first place.

Recently, the research focus has shifted to scheduling-based contention management [3, 6, 7, 12, 13, 14]. In the scheduling approach, questions such as when to start a transaction or whether two transactions should run concurrently are taken into account. Ideally, we would like to avoid starting two transactions that will conflict in the future.

Earlier works [3, 6, 13] have proposed serialization as the main contention management mechanism: they keep track of the likelihood of a transaction to abort and, when its conflict probability reaches a given threshold, they serialize such transaction. Hence, in a high contention scenario, transactions with repeated aborts will be serialized one after another, thus reducing the total number of conflicts and wasted work.

Later proposals [7, 14] have suggested a proactive approach, wherein the decision of whether to start a given transaction is taken before the transaction starts executing. Most of the scheduling-based contention management proposals are implemented at the user-level and rely on costly synchronization primitives such as locks and condition variables, which usually involve system calls. A recent work [12] have proposed kernel-level scheduling support in order to reduce the overhead presented in the previous user-level approaches, at the cost of changing the OS scheduler.

3 Overview

The design of LUTS is based on two main assumptions: (1) the number of system-level threads (henceforth referred to as SLTs) should not exceed the number of available cores, and (2) a STM system would benefit from advanced scheduling capabilities. Assumption (1) aims at reducing pseudo parallelism issues, while assumption (2) is guided by the idea that more robust conflict-avoidance mechanisms can be created if scheduling details are exposed to the STM system.

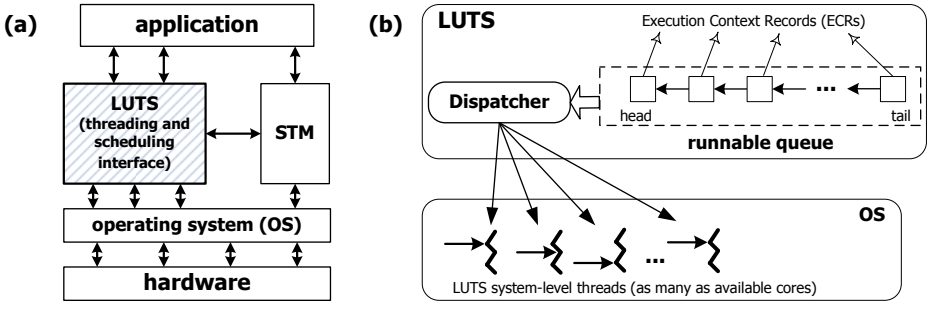


Fig. 1. LUTS overview: (a) application and STM interface; (b) mapping between ECRs and SLTs

Figure 1a shows a general overview of a LUTS-based system. The application and the software transactional memory (STM) library use LUTS interface for managing threading and scheduling operations. A key distinction of LUTS approach is that it does not automatically create as many system-level threads as required by the application. Instead, it creates a number of *execution context records* (ECRs), which encapsulate the state of a thread. Each ECR is inserted into the scheduler runnable queue and is eligible for later execution. LUTS only creates as many system-level threads as the number of available cores. The dispatcher is responsible for taking an ECR from the runnable queue and mapping it to a system-level thread (see Figure 1b).

The main reason to adopt LUTS threading design is to overcome the potential shortcomings caused by pseudo parallelism. In a conventional system, multiple transactions may run on the same processor core: the OS scheduler can preempt a thread that is running a transaction at any time and switch to another thread running a different transaction. LUTS design aims at executing at most one transaction thread per core. This is enforced by: (i) creating at most as many system-level threads as the number of available cores, and (ii) setting the thread affinity to a specific core. This practically avoids having the same core executing more than one transaction thread.

3.1 Scheduling Transactions

After an ECR is mapped and starts executing, there are only two ways for the corresponding SLT to become free again: either the work is finished or the ECR voluntarily relinquishes control. Hence, LUTS employs a cooperative approach to scheduling.

There are two main advantages in adopting a cooperative over a preemptive scheduler in the context of this paper. First, it is very simple to implement and efficient. Second, coordination avoids the risks caused by pseudo parallelism and preemption as discussed previously. Therefore, we avoid blocking a thread that is running a transaction and potentially decrease the probability of inter-transaction conflicts.

The basic interface to our scheduler is `luts_yield`, with a similar semantic to the `sched_yield` routine provided by UNIX-like operating systems. This routine forces the current context to relinquish the SLT. The context is inserted into the tail of the scheduler runnable queue, and another one is taken from the head and executed by the dispatcher. Most importantly, LUTS also allows the STM to switch to another transaction by calling one of two methods:

- `luts_switch` : takes as argument the ID of a transaction and forces, when possible, the execution to switch to a transaction that has a different ID.
- `luts_switch_to_id` : takes as argument the ID of a transaction and forces, when possible, the execution to switch to a transaction with the specified ID.

With `luts_switch` and `luts_switch_to_id` interface, STMs can provide different conflict-avoidance heuristics in the hope of decreasing system contention. In a more general scenario, it is possible for the STM library to provide a callback routine that is invoked for every transaction in the runnable queue. Using this approach, an STM could implement any scheduling strategy as it sees fit, at the cost of the extra overhead.

At the current state LUTS does not implement automatic progress guaratees. However, the programmer can achieve this using synchronization barriers. LUTS provide methods to access such barriers in its interface (i.e. `luts_barrier_wait`), in which case the execution only proceeds after all threads reach that barrier.

It should be noticed that the main contribution of this paper is not in the scheduler itself, but in its application to the domain of memory transactions and the corresponding ability to develop new proactive conflict-avoidance heuristics. The LUTS design can also be seen as a simplication of scheduler activations [15]. LUTS also has similarities with McRT [16], since both implement a run-time thread package. However, LUTS also provides means to schedule transactions and build conflict-avoidance heuristics using its compabilities.

4 LUTS-Based Heuristics

This section presents three heuristics built around LUTS scheduling capabilities: `CILUTS`, `CTLUTS` and `HASHLUTS`. Common to all the heuristics is the fact that we first determine proactively the conflict probability of starting a transaction and invoke LUTS to switch the execution to a different transaction when possible, thus avoiding serializing the execution.

4.1 `CILUTS`

In the first approach, `CILUTS`, each transaction maintains internally a variable describing its conflict probability. To quantify this variable we use the concept of contention intensity (CI), firstly described by Yoo and Lee [6] in their Adaptive Transaction Scheduling (ATS) proposal. Our technique differs from ATS when we find that a transaction is likely to abort. Instead of serializing the transaction, like

Listing 1.1. CTLUTS

```
1 double conflictTable[][];
2 int activeTx[];
3
4 upon stm_init
5   for each transaction pair (t1, t2)
6     resetCT(t1, t2);
7   for each core i
8     activeTx[i] = INVALID;
9
10 upon start
11   for each core i {
12     int other_id = activeTx[i];
13     if (other_id == INVALID) continue;
14     if (ProbCT(tx_id, other_id) > threshold) {
15       luts_switch(tx_id);
16       break; } }
17   activeTx[thisCore] = tx_id;
18
19 upon abort
20   activeTx[thisCore] = INVALID;
21   for each core i {
22     int other_id = activeTx[i];
23     if (other_id == INVALID) continue;
24     increaseProbCT(tx_id, other_id); }
25
26 upon commit
27   activeTx[thisCore] = INVALID;
28   for each core i {
29     int other_id = activeTx[i];
30     if (other_id == INVALID) continue;
31     decreaseProbCT(tx_id, other_id); }
```

ATS does, we use LUTS to find a more appropriate transaction to take its place, by using the interface method `luts_switch` to switch the current transaction to another one with a different ID. By doing so, we can reduce the number of conflicts and do not waste processor time waiting for a transaction to commit.

4.2 CTLUTS

CTLUTS uses a conflict table to keep track of the conflict probabilities among transactions. The pseudo code for the heuristic is presented in Listing 1.1. Besides the conflict table itself (line 1), we also maintain a vector of active transactions (line 2). For each processor core we keep in this vector the ID of the transaction it is currently executing. The conflict table is made as large as the number of different transactions in the application. Transactions are differentiated by the address of their first instruction. When the transactional system is initialized, we reset the conflict table (lines 5 and 6) and set all entries in the active vector to invalid (lines 7 and 8).

Since CTLUTS adopts a proactive approach, the most important part of the heuristic takes place upon transaction start (lines 10-17). For each valid transaction ID in the active vector, we check the conflict probability between the transaction pair and, if above a certain threshold (line 14), we invoke LUTS switch routine (line 15) to choose another transaction. The main idea of the

heuristic is to avoid starting a transaction that is doomed to abort and reduce system contention. Ideally, a transaction with better chances to commit will be chosen from the runnable queue by LUTS. If there is no such transaction then the current transaction is started anyway. Therefore, the worst case scenario for the heuristic happens when no transaction from the runnable queue is eligible to execute (its ID is the same of the current one). When the transaction is successfully initiated, it sets the corresponding entry in the active vector to its ID (line 17).

The conflict table is updated when a transaction fails (lines 19-24) or commits (lines 26-31). Both cases present a similar behavior: the ID of the transaction executing in each concurrent thread is retrieved from the active vector and, if it is different from invalid, the conflict probability of the transaction pair is increased in case of an abort (line 24), or decreased in case of success (line 31). It is worth noticing that the accesses to the active vector and the conflict table are not explicitly synchronized, therefore allowing data races. For instance, when we scan the active vector to update the conflict table, a read access can yield stale values. We consider these data races benign: they add an imprecision to the heuristic prediction but they cannot cause any execution fault. For `CTLUTS`, it is preferable to incur some imprecision than paying the high cost of synchronization. The same approach was adopted by Blake et al. [14] in a similar context.

4.3 `HASHLUTS`

`HASHLUTS` uses a different approach to schedule transactions. While `CILUTS` and `CTLUTS` checks if a starting transaction is likely to abort and then replace it with any other transaction, `HASHLUTS` tries to choose the best transaction to start based on the current set of active transactions. The pseudo code for the heuristic is presented in Listing 1.2. We keep an array (`activeTx`) that contains what are the currently executing transactions on each core, these transactions form a group of active transactions. For example, in a 4-core machine, we may have `activeTx` = {tx1, tx3, tx5, tx1} and we distinguish this group from {tx1, tx5, tx3, tx1}, in other words, we take in account which core is executing each transaction (line 1). We also keep a conflict table to store the probability of a transaction to abort when it executes in parallel with each group of active transactions (line 2), and another array with the best transaction choice (less likely to abort) for each group of active transactions (line 3). When the system initiates we reset those structures (lines 5-9).

Every time a thread is going to start executing a transaction, it first check which group of active transactions is currently executing, a hash function is applied on this information and the result is an index for the bestTx array (line 12). This way the thread knows which transaction is less likely to abort when executing in parallel with the current group of active transactions (line 13). Then, if the thread was about to start a different transaction, it calls the method `luts_switch_to_id` from LUTS interface to switch its context and execute (when possible) the best transaction (line 14). At the end it updates activeTx to reflect the new group of active transactions (line 15).

Listing 1.2. HASHLUTS

```
1 int activeTx[];
2 double conflictTable[][];
3 int bestTx[];
4
5 upon stm_init
6   resetBestTx();
7   resetCT();
8   for each core i
9     activeTx[i] = INVALID;
10
11 upon start
12   int index = hash(activeTx);
13   int tx_id = bestTx[index];
14   luts_switch_to_id(tx_id);
15   updateActiveTx(thisCore, tx_id);
16
17 upon abort
18   updateActiveTx(thisCore, INVALID);
19   int index = hash(activeTx);
20   increaseProbCT(index, tx_id);
21   if(bestTx[index] == tx_id){
22     for each transaction tx {
23       if(conflictTable[index][tx] <
24           conflictTable[index][tx_id])
25       {
26         bestTx[index] = tx;
27       }
28     }
29   }
30
31 upon commit
32   updateActiveTx(thisCore, INVALID);
33   int index = hash(activeTx);
34   decreaseProbCT(index, tx_id);
35   if(conflictTable[index][tx_id] <
36       conflictTable[index][bestTx[index]])
37   {
38       bestTx[index] = tx_id;
39   }
```

Upon abort, activeTx is updated, the hash function is applied to activeTx and the hash function result is used as an index for the conflict table. Then the conflict table entry for the pair (index, abortedTx) has its abort probability increased (lines 18-20). If the aborted transaction was the best transaction for this group of active transaction, we must check if it still is the best choice, if it is not, we update the bestTx for this activeTx (21-29). The commit operation has a similar behavior, but it decreases the abort probability on the conflict table and then only check if the new probability is lower than the current bestTx probability, if that is the case, bestTx is updated (lines 31-39).

HASHLUTS adds more overhead to system, since it calls a context switch more often than the other two heuristics, but it is more accurate since it is able to tell which transaction is less likely to abort at the moment.

5 Experimental Results

In this section we investigate the performance of our prototype implementation of LUTS and the three proposed heuristics using tinySTM 1.0.0 as the base STM system. We conducted the experiments on a single node with four Intel Xeon X7560 processors (32 cores in total), 24MB L3 cache, and 256GB of RAM. The machine runs a typical Linux distribution with kernel version 2.6.18-194. All applications were compiled using gcc version 4.5.1. The evaluation was done using programs from the STMBench7 [4] and STAMP 0.9.10 benchmarks [5].

We report results for 7 different configurations. All configurations use the same base code, providing fair comparisons. More specifically, the following configurations were used:

- **Original:** the baseline tinySTM implementation (version 1.0.0). For the experiments we configured tinySTM with the write-back and encounter-time locking (ETL) strategy. For comparison, we adopted the `CM_SUICIDE` contention policy, which immediately restarts a transaction on abort.
- **ATS:** An implementation of Yoo and Lee adaptive transaction scheduling technique [6]. We use a single global queue for all transactions, as suggested by the authors. Before running the experiments reported here, we conducted a sensibility study on α with values 0.3, 0.5 and 0.75, and on threshold with values 0.3, 0.5 and 0.7. We found out that the combination of 0.75 for α and 0.5 for threshold resulted in best overall performance, and thus was adopted in the experiments.
- **Shrink:** An implementation of the Shrink scheduler proposed by Dragojevic et al. [7] and also integrated with SwissTM [17]. The code was taken from the authors' website[1] for tinySTM version 0.9.5, and adapted to the current version (1.0.0) by us. We did not change the parameters in the code: succ_threshold = 0.5, locality_window = 4, confidence_threshold = 3, `c1` = 3, `c2` = 2, `c3` = 1.
- **LUTS:** The scheduler with a fixed circular order scheduling policy (round-robin). The scheduling interface is not used by the STM code in this configuration, allowing us to measure the gains of our approach in pseudo parallelism scenarios.
- **CILUTS:** The heuristic using contention intensity and LUTS scheduling interface as discussed in Section 4.1. Similarly to ATS, we adopted $\alpha = 0.75$ and threshold = 0.5 in the experiments.
- **CTLUTS:** The heuristic using the conflict table to track conflict probabilities and LUTS switching feature, as explained in Section 4.2. In the experiments we use a threshold value of 0.5.
- **HASHLUTS:** The heuristic using a hash table to track conflict probabilities and LUTS switching feature to select the best transaction for execution, as explained in Section 4.2.

[1] http://lpd.epfl.ch/site/research/tmeval

For every application we report the average over 10 runs in order to reduce the variance in the results. Even so, we noticed a high variance for the Bayes application and omitted it. We also omitted the results for SSCA2 since it presented a livelock scenario in all configurations.

5.1 Speedup

We show speedup results for the 6 configurations discussed with respect to the baseline tinySTM with a single thread. Figure 2 shows the speedup comparison for 8 STAMP programs and 4 STMBench7 configurations when the number of threads is varied from 1 to 128.

If we consider LUTS performance when the number of threads does not exceed 32, no important overhead can be noticed relative to the base STM. From 64 to 128 threads, LUTS achieves the same performance as running 32 threads and, unlike the original implementation, it sustains this performance as the number of threads increases; this behavior is consistently maintained for every STAMP program. With 128 concurrent threads we can notice a speedup of 20x for VacationLow, whereas ATS and Shrink reached about 6x and 5x, respectively.

Besides the gains achieved with LUTS alone, we also notice a performance boost in some applications due to the heuristics `CILUTS`, `CTLUTS` and `HASHLUTS`. Programs Labyrinth, Yada and all STMBench7 configurations had their performance improved even further. While LUTS achieved speedups of 8x in program Small (128 threads), our LUTS-based heuristics managed to speed it up by 10x. In fact, `CTLUTS`, `CILUTS` and `HASHLUTS` are the first transaction scheduling techniques to improve performance in overloaded scenarios even further. With 32 threads, the best speedup of configuration Medium was 8.1x, `CILUTS` achieved 11.3x speedup with 128 threads.

5.2 Overhead

This section presents the overhead added by each configuration discussed in this paper. Figure 3 shows an average of how many processor cycles are spent per transaction on four different programs: Intruder, Labyrinth, Small, and Huge. We noticed that all other programs have a very similar behavior to at least one of those programs, making those four programs enough to represent both benchmarks. Each bar of the graph is composed by four segments: (1) cycles spent on the transaction itself, (2) cycles spent on heuristic code, (3) cycles spent on the LUTS scheduler, and (4) cycles spent on transaction aborts.

Intruder has small transactions, so the scheduler overhead was dominant on techniques `CTLUTS` and `HASHLUTS` for this application. Labyrinth has longer transactions and spends many cycles on aborted transactions; therefore, the overhead introduced by all techniques was proportionally smaller. Heuristic `HASHLUTS` was able to slightly reduce the number of cycles on aborts by adding just a small scheduler overhead. On programs Small and Huge, techniques ATS and Shrink eliminate most of abort cycles, but its overhead does not make it worthwhile.

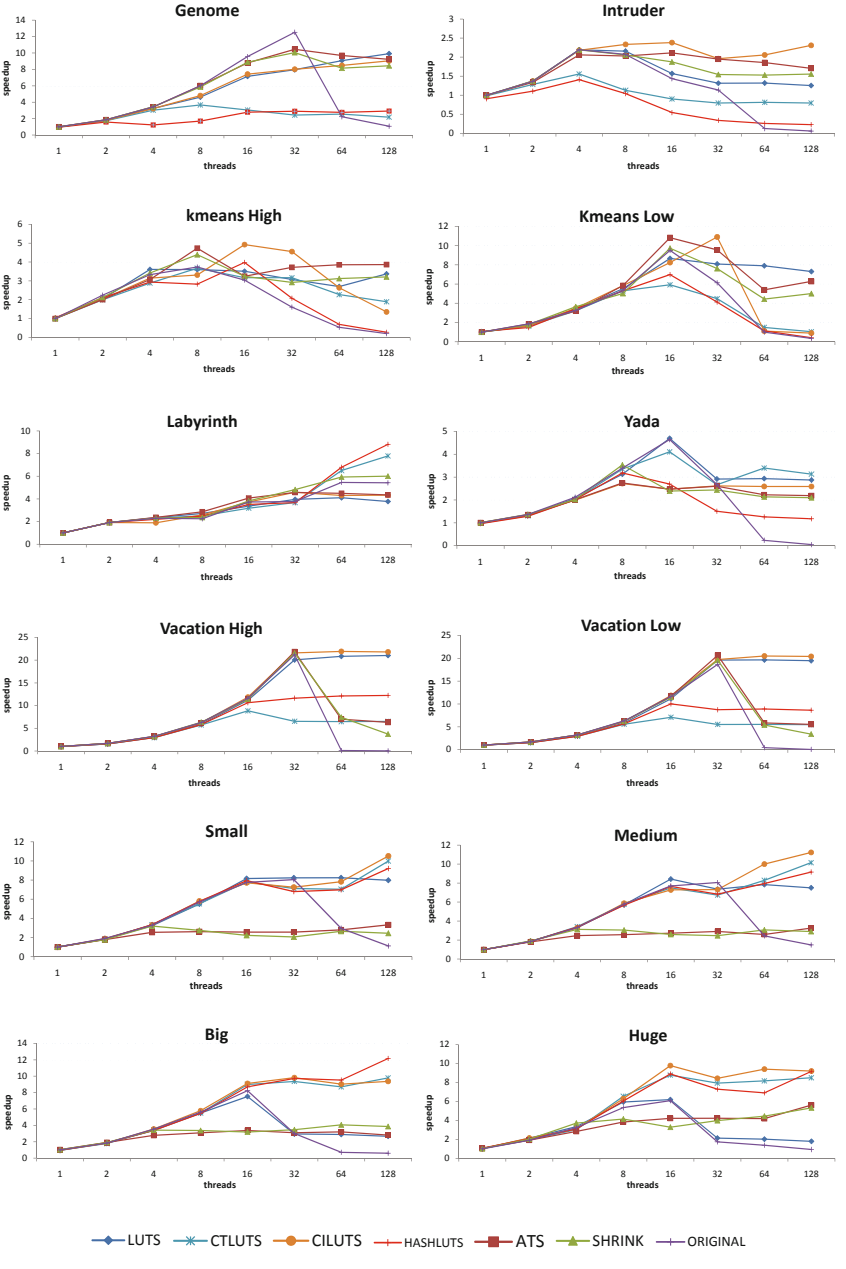


Fig. 2. Speedup achieved on benchmarks STAMP and STMBench7

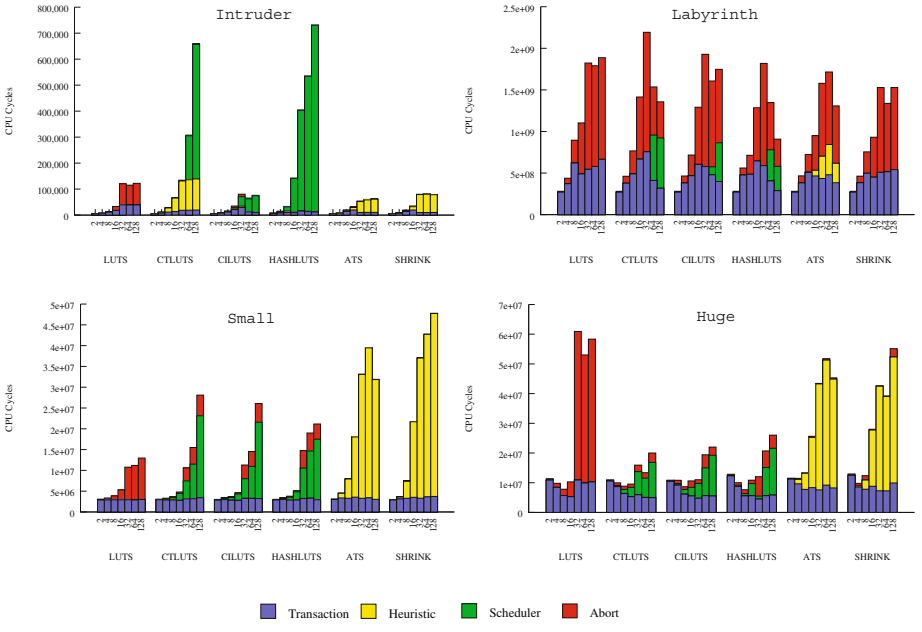


Fig. 3. Overhead of six scheduling techniques

On the other hand, LUTS-based heuristics were capable of avoid such aborts with an acceptable overhead.

5.3 Discussion

Results show that LUTS-based heuristics perform better in programs with longer transactions, like Labyrinth, Yada and the STMBench7 benchmark. This behavior is made clearer with the STMBench7 programs. The only parameter difference between these programs is the data structure size used, increasing the structure size used by STMBench7 programs also increases the length of its transactions. As shown in Figure 2, the performance of LUTS-based heuristics increases as the data structure size increases. Configurations Big and Huge do not scale well with 16+ threads, but LUTS-based heuristics manage to keep those configurations improving until 32 threads and sustain this performance with 64+ threads.

In programs with short transactions and few different transactions, like Intruder, KmeansHigh and KmeansLow, LUTS-based heuristics generated a slight overhead and did not improve LUTS performance. Especially for Kmeans, they were surpassed by ATS and Shrink.

The overhead source for LUTS-based heuristics comes from the extra actions performed during the transaction start, commit, and abort operations. Therefore, the overhead is proportionally larger in programs with short transactions, making programs with large transaction better candidates for those heuristics.

Moreover, avoiding the abort of a large transaction is more effective than avoiding the abort of a short one.

All STMBench7 configurations have 48 different transactions, while STAMP programs have only 3-6 transactions. A high number of different transactions brings more scheduling possibilities for LUTS-based heuristics, improving its performance. This is another reason why STMBench7 programs achieved better results with LUTS-based heuristics.

In general, we can conclude that LUTS delivers good performance on overloaded scenarios no matter how many threads are launched at the application startup, effectively dealing with pseudo parallelism issues. Moreover, the ability to switch to another transaction instead of applying serialization also can pay off, as we observed in the applications with large transactions.

6 Conclusion

We have introduced in this paper LUTS, a Lightweight User-level Transaction Scheduler, and three proactive conflict-avoidance heuristics. LUTS effectively controls the contention level in pseudo parallelism scenarios, and provides the means to increase system performance by avoiding starting transactions that are likely to abort in the near future. In order to accomplish its goal, LUTS relies on two key factors: (i) the number of spawned threads is limited to the number of available processor cores, and (ii) STM libraries can exert influence on the scheduling policy and devise new proactive conflict-avoidance heuristics. LUTS scheduling capabilities is in sharp contrast with prior works, which primarily resorted to serialization.

Experimental results show that LUTS and the proposed conflict-avoidance heuristics provide better performance than existing techniques on some applications taken from the STAMP and STMBench7 benchmarks, reaching speedups up to 23x.

References

1. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan & Claypool Publishers (June 2010)
2. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC 2005, pp. 258–264 (July 2005)
3. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In: PODC 2008, pp. 125–134 (August 2008)
4. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A benchmark for software transactional memory. In: EUROSYS 2007, pp. 315–324 (March 2007)
5. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC, pp. 35–46 (September 2008)
6. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA 2008, pp. 169–178 (June 2008)

7. Dragojevic, A., Guerraoui, R., Singh, A.V., Singh, V.: Preventing versus curing: Avoiding conflicts in transactional memories. In: PODC 2009, pp. 7–16 (August 2009)
8. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003, pp. 92–101 (July 2003)
9. Scherer, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005, pp. 240–248 (July 2005)
10. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 303–323. Springer, Heidelberg (2005)
11. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: PPoPP 2009, pp. 141–150 (February 2009)
12. Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J.L., Muller, G.: Scheduling support for transactional memory contention management. In: PPOPP 2010, pp. 79–90 (January 2010)
13. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-Abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
14. Blake, G., Dreslinski, R.G., Mudge, T.: Proactive transaction scheduling for contention management. In: MICRO 2009, pp. 156–167 (December 2009)
15. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: effective kernel support for the user-level management of parallelism. SIGOPS Oper. Syst. Rev. 25, 95–109 (1991)
16. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In: PPoPP 2006, pp. 187–197. ACM, New York (2006)
17. Dragojevic, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: PLDI 2009, pp. 155–165 (June 2009)

Verification of Partitioning and Allocation Techniques on Teradata DBMS

Ladjel Bellatreche[1], Soumia Benkrid[2], Ahmad Ghazal[3], Alain Crolotte[3], and Alfredo Cuzzocrea[4]

[1] LISI/ENSMA Poitiers University
Futuroscope, France
bellatreche@ensma.fr
[2] National High School for Computer Science (ESI)
Algiers, Algeria
s_benkrid@esi.dz
[3] Teradata Corporation
San Diego, CA, U.S.A.
(Ahmad.Ghazal,Alain.Crolotte)@teradata.com
[4] ICAR-CNR and University of Calabria, Italy
cuzzocrea@si.deis.unical.it

Abstract. Data fragmentation and allocation in distributed and parallel Database Management Systems (DBMS) have been extensively studied in the past. Previous work tackled these two problems separately even though they are dependent on each other. We recently developed a combined algorithm that handles the dependency issue between fragmentation and allocation. A novel genetic solution was developed for this problem. The main issue of this solution and previous solutions is the lack of real life verifications of these models. This paper addresses this gap by verifying the effectiveness of our previous genetic solution on the Teradata DBMS. Teradata is a shared nothing DBMS with proven scalability and robustness in real life user environments as big as 10's of petabytes of relational data. Experiments are conducted for the genetic solution and previous work using the SSB benchmark (TPC-H like) on a Teradata appliance running TD 13.10. Results show that the genetic solution is faster than previous work by a 38%.

1 Introduction

Data warehousing is becoming more complex in terms of applications, data size and queries, including joins and aggregations. Data warehouse projects always stress performance and scalability because of the data volumes and the query complexity. For instance, *eBay's* data warehouse include *2 petabytes* of user data and Millions of queries per day[1]. The parallelism technology is one of the relevant solutions to deal with these mountains and complex queries. More and more organizations are relying on parallel processing technologies to achieve the

[1] http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 158–169, 2011.

performance, scalability, and reliability they need. Most of the major commercial database systems support parallelism (Teradata, Oracle, IBM, Microsoft SQL Server 2008 R2 Parallel Data Warehouse, Sybase, etc.). Rather than relying on a single monolithic processor, parallel systems exploit fast and inexpensive micro processors to achieve high performance.

Designing parallel databases was widely studied in the contexts of OLTP (On-Line Transaction Processing) [1,9,13,7,15,16,18,20] and OLAP (On-Line Analytical Processing) [2,4,12,14,22,23,21]. Most of these studies are usually performed from theoretical point of view. On the other words, none deployment on a real parallel database machine is given.

The main important steps that designers need to perform to construct parallel data warehouses are: **(i)** the *choice of the hardware platform*, **(ii)** the *partitioning of data warehouse schema*, **(iii)** the *allocation of the generated fragments*, **(iv)** the *load balancing over the nodes* of the chosen parallel machine, **(v)** the *query processing* and **(vi)** the deployment of solutions of the previous phases. All these steps got a lot of attention of data warehouse research community, since most of them are inherited from traditional parallel database design, except the last one.

(i) There are three widely used architectures for parallelizing work: (a) *shared memory* (b) *shared disk* and (c) *shared nothing*. In a *shared-memory approach*, all of the CPUs share a single memory and a single collection of disks. This approach is relatively easy to implement, since the lock manager and buffer pool are both stored in the memory system where they can be easily accessed by all the processors. Unfortunately, it has fundamental scalability limitations, as all I/O and memory requests have to be transferred over the same bus that all of the processors share, causing the bandwidth of this bus to rapidly become a bottleneck [10,7]. In *shared-disk platform*, there are a number of independent processor nodes, each with its own memory. These nodes all access a single collection of disks. This architecture has been used to design parallel data warehouse by [25]. It has a number of drawbacks that severely limit scalability. First, the interconnection network that connects each of the CPUs to the shared-disk subsystem can become an I/O bottleneck. Second, since there is no pool of memory that is shared by all the processors, there is no obvious place for the lock table or buffer pool to reside. In a *shared-nothing approach*, each processor has its own set of disks. This architecture is used by *Teradata*. Data is horizontally partitioned across nodes, such that each node has a subset of the rows from each table in the database. Each node is then responsible for processing only the rows on its own disks. Such architectures are especially well suited to the star queries running on data warehouses modelled using a star schema, as only a very limited amount of communication bandwidth is required to join one or more (typically small) dimension tables with the (typically much larger) fact table [10].
(ii) Once the architecture is chosen, data warehouse designer partitions its schema. Fragmentation[2] is a *pre condition* of parallel data warehouse design. It

[2] In this paper, we use fragmentation and partitioning interchangeably.

may be *horizontal*, where table instances are decomposed into disjoint partitions or *vertical*, where tables are split in disjoint sets of attributes. The horizontal partitioning is mainly used for designing parallel data warehouses [25,26,12,14,2]. Two main types of horizontal partitioning exist [5]: *mono table partitioning* and *table-dependent partitioning*. In the mono table partitioning, a table is partitioned using its own attributes. It is quite similar to the *primary horizontal partitioning* proposed in traditional databases [18]. Several modes exist to support mono table partitioning: *Range, List, Hash, Round Robin* (supported by Sybase), *Composite* (Range-Range, Range-List, List-List, etc.), *Virtual Column partitioning* recently proposed by *Oracle11G*. In *table-dependent partitioning*, a table inherits the partitioning characteristics from other table. For instance a fact table may be partition based on the fragmentation schemes of dimension tables. This partitioning is feasible if a parent-child relationship among these tables exists [8,11]. Two main implementations of this partitioning are possible: *native referential partitioning* and *simulated referential partitioning*. The native referential partitioning is recently supported by Oracle11G to equi-partition tables connected by a parent child referential constraint. A native DDL is given to perform this partitioning [11] (*Create Table ... Partition by Reference ...*). This *diversity* of existing modes poses problem of deploying existing research studies on parallel data warehouse design, since a direct deployment is hard to perform. To summarize, we can notice that fragmentation schemes obtained by some partitioning algorithms can be directly implemented in a priori known DBMS (this situation is called a *turnkey solutions*). Others need to be adapted according the target DBMS (*non turnkey solutions*). [2] is an example of turnkey solutions, where algorithms were proposed to *referential partition* a data warehouse. This partitioning was initially supported by Oracle [11]. [12] is an example of non turnkey solutions, where referential partitioning is implemented manually in the context of parallel database machine as follows: (i) a dimension table is first horizontally partitioned using its primary key, then the fact table is decomposed based on its foreign key referencing that dimension table.
(iii) The data allocation is the process that places generated fragments over nodes of parallel machine. This allocation may be either *redundant* (with replication) or *non redundant* (without replication). **(iv)** Once fragments are placed, global queries are then rewritten over fragments and executed on the parallel machine. During their execution phase, **(v)** the load balancing should be verified. Load balancing refers to workload allocation over nodes [20]. **(vi)** The deployment is usually done in simulated environments or using mathematical cost models quantifying the quality of parallel design.

In [2], we propose a parallel data warehouse design approach, where fragmentation and allocation are done in joint way in order to capture the interdependency between these two steps. The decision of allocation fragments in done during the fragmentation process. The quality of this method is measured by the means of a cost model estimating the query processing cost in terms of inputs outputs required for executing a set of queries. The main objective of this paper is to verify our results on a real life parallel DBMS. Based on our collaboration

with the Teradata labs, we managed to run our results on a Teradata appliance running TD 13.10. Teradata is a known MPP DBMS and have been in the Gartner's "Data Warehouse DBMS Magic Quadrant" for many years.

The paper is organized as follows. Section 2 summarizes existing approaches for designing parallel data warehouses. In Section 3, we give background related to the joint design methodology for parallel data warehouse. Section 4 describes the validation of our joint approach on *Teradata* machine using star schema benchmark data set [17]. Finally, Section 5 concludes the paper summarizing the main findings of our research, and proposing directions for future work.

2 Related Work

This section reviews the most important studies on parallel data warehouse design from academic [2,4,6,12,14,22,23] and industrial perspective [21].

Academic studies were essentially focused on proposing solutions for designing data warehouses for a given parallel machine architecture. Furtado [12] discusses partitioning strategies for *node-partitioned data warehouses*. The main suggestion coming from [12] can be synthesized in a "best-practice" recommendation stating to partition the fact table on the basis of the *larger* dimension tables (given a ranking threshold). In more detail, each larger dimension table is first partitioned by means of the *Hash mode* approach via its primary key. Then, the fact table is again partitioned by means of the Hash mode approach via foreign keys referencing the larger dimension tables. Finally, the so-generated fragments are allocated according to two alternative strategies, namely *round robin* and *random*. Smaller dimension tables are instead fully-replicated across the nodes of the target data warehouse. In [14], Lima *et al.* focus the attention on data allocation issues for database clusters. Authors recognize that how to place data/fragments on the different PC of a database cluster in the dependence of a given criterion/goal (e.g., query performance) plays a critical role, hence the following two straightforward approaches can be advocated: (*i*) full replication of the target database on *all* the PC, or (*ii*) meaningful partition of data/fragments across the PC. Starting from this main intuition, authors propose an approach that combines partition and replication for OLAP-style workloads against database clusters. In more detail, the fact table is partitioned and replicated across nodes using the so-called *chained de-clustering*, while dimension tables are fully-replicated across nodes. This comprehensive approach enables the *middleware layer* to perform load balancing tasks among replicas, with the goal of improving query response time. Furthermore, the usage of chained declustering for replicating fact table partitions across nodes allows the designer not to detail the way of selecting the number of replicas to be used during the replication phase. In [22], the allocation of relational data warehouses based on a star schema and utilizing bitmap index structures in a *shared disk architecture* is proposed. The fragments are generated by the means of multi-dimensional hierarchical data fragmentation of the fact table. The proposal is validated by

a *simulation model* [23]. In these studies, fragmentation and allocation are done in sequential way. In [2,4], another trend of parallel data warehouse design was proposed in which partitioning and allocation processes are done simultaneously. These works were done in a shared nothing architecture [2] and heterogeneous database cluster [4].

To summarize, we figure out that the academic studies are validated either by the means of simple cost models estimating the number of inputs outputs required for executing a set of queries or by simulators. None deployment in a real machine is given.

From industrial perspective, DB2 DBMS [21] proposed a solution for data partitioning in shared nothing architecture. Based on this work, data partitioning advisor is developed to recommend user the number of partitions of each fragments. As academic studies, this work considers fragmentation and allocation are done sequentially.

3 Background

In this section, we review the joint methodology for designing parallel data warehouses, where fragmentation and allocation are done simultaneously. To facilitate the understanding of our methodology, we give a formalization of the parallel data warehouse design problem [2]:

- a data warehouse schema composed by d dimension tables $\mathcal{D} = \{D_0, \ldots, D_{d-1}\}$ and one fact table $\mathcal{F}$ – as in [12,14]. Figure 1 shows an example of a star schema of a relational data warehouse used by Star Schema Benchmark [17];
- a shared nothing architecture with M nodes $\mathcal{N} = \{N_0, N_1, \ldots, N_{M-1}\}$;
- a set of star queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_{L-1}\}$ to be executed over the warehouse schema, being each query Q_l, with $0 \leq l \leq L-1$;
- a *maintenance constraint* $\mathcal{W} : W > M$ representing the number of fragments W that the designer considers relevant for his/her target allocation process, called *fragmentation threshold*;

The problem of designing a parallel data warehouse consists in *fragmenting the fact table $\mathcal{F}$ into N_F fragments and allocating them over different* nodes *such that the total cost of executing all the queries in $\mathcal{Q}$ can be minimized while processing constraints are satisfied across nodes, under the maintenance constraint $\mathcal{W}$.*

Based on the formal statement above, it follows that our investigated problem is composed by two sub-problems, namely data partitioning and fragment allocation. Each one of these problems is known to be *NP-complete* [3,24,13]. In order to deal with the parallel design problem, two main classes of methodologies are possible: *sequential design* and *joint design*. Sequential design methodology has been proposed in the context of traditional distributed and parallel database design research. The basic idea underlying this methodology consists in first fragmenting the data warehouse using *any* partitioning algorithm, and then allocating the so-generated fragments by means of *any* allocation algorithm. In

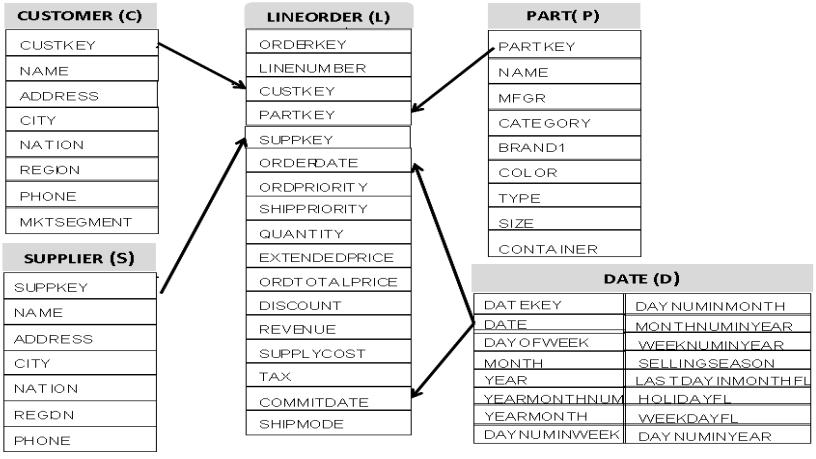


Fig. 1. An Example of a Star Schema

the most general case, each partitioning and allocation algorithm has its own cost model. The main advantage coming from these traditional methodologies is represented by the fact that they are straightforwardly applicable to a large number of even-heterogeneous parallel and distributed environments. Contrary to this, their main limitation is represented by the fact that they neglect the inter-dependency between the data partitioning and the fragment allocation phase, respectively. Another limitation of this approach is the fact that it uses two different cost models: one for fragmentation process and another for allocation process.

To take into account de inter-dependency between and partitioning and fragments allocation, the joint approach is proposed. During the fragmentation phase, each potential solution is tested for allocation process. At the end, the solution with minimum cost is chosen. Only one integrated cost model is used for both processes: fragmentation and allocation. Figure 2 summarizes the steps of joint design methodology [2].

4 Validation on Teradata

Empirical results in previous work [2] were based on custom simulation for a distributed system using a single CPU machine. These simulations lacked the real life aspect to demonstrate the efficacy of the new results. In collaboration with *Teradata* Labs, we verified our results on a *Teradata* system running TD 13.10 software.

In this section we provide a high level description of the Teradata DBMS and the SSB benchmark [17] and how customized it. Finally, we present the results of the SSB benchmark on both the joint and sequential methods.

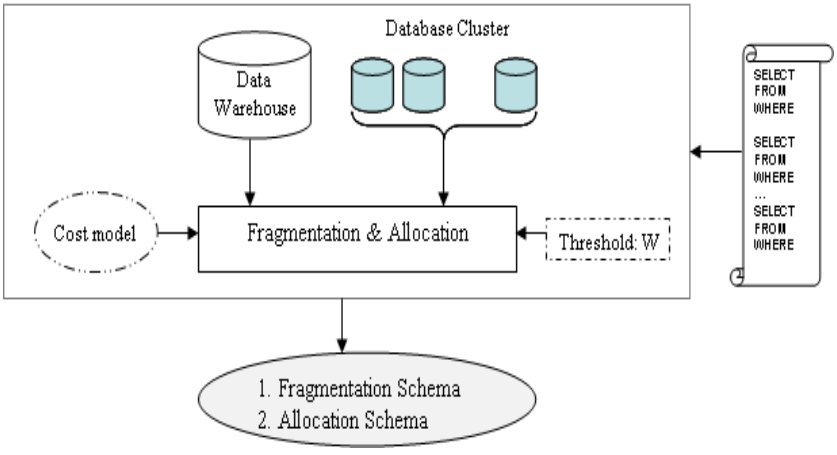


Fig. 2. Joint Design Methodology

4.1 Teradata Description

Teradata is a massively parallel processing system running shared nothing architecture. The *Teradata* DBMS is linearly and predictably scalable in all dimensions of a database system workload (data volume, breadth, number of users, complexity of queries). The basic unit of parallelism in *Teradata* is a virtual processor (called Access Module Processor or AMP) which is assigned a data portion. Each *AMP* executes DBMS functions on its own data. This allows locks and buffers do not have to be shared which ensures scalability. Figure 3 illustrates the *Teradata* architecture by a two node system. A node is a multi-core system with disks and memory. It provides a pool of resources (disk, memory, etc) for the AMPS in that node. BYNET is the network used to link different *AMPs* within a node and across different nodes as well.

Data entering a *Teradata* Database are processed through a sophisticated hashing algorithm and automatically distributed across all *AMPs* in the system. In addition to being a distribution technique, this hash approach serves as an indexing strategy. This significantly reduces the amount of DBA work normally required to set up direct access. To define a *Teradata* Database, the DBA simply chooses a column or set of columns as the primary index for each table. The value contained in these indexed columns is used to determine the *AMP*, which owns the data, as well as a logical storage location within the *AMP*'s associated disk space, all without performing a separate CREATE INDEX operation. To retrieve a row, the primary index value is again passed to the hash algorithm, which generates the two hash values, *AMP* and *Hash-ID*. These values are used to immediately determine which *AMP* owns the row and where the data are stored.

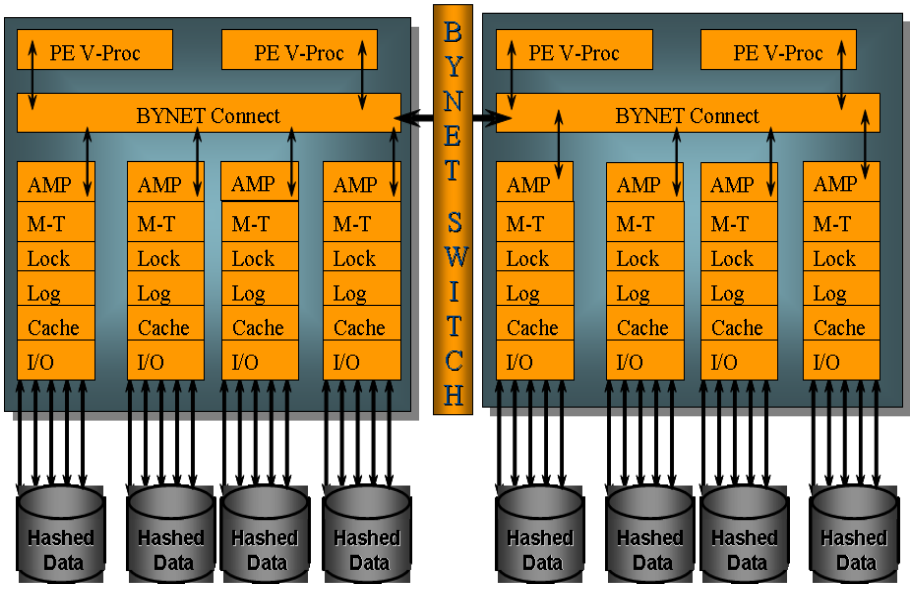


Fig. 3. Teradata Internal Architecture (MPP)

4.2 Experiments

The experiments were designed around the SSB benchmark [17]. This benchmark is based on a star schema derived from the TPC-H schema [19]. Like TPC-H the SSB benchmark comes with a data generation utility (dbgen) which is scalable. For our experiments we used 10 scale factor (10GB). The SSB benchmark is built around a fact table called *lineorder* and 4 dimension tables: *part*, *supplier*, *customer* and *date*. All tables are derived from the TPC-H database except the date dimension which is new. The main table is the fact table *lineorder* with the following DDL:

```
CREATE TABLE dbo10_sq.LINEORDER(
        LO_ORDERKEY int NOT NULL,
        LO_LINENUMBER int NOT NULL,
        LO_CUSTKEY int NOT NULL, -> FK to CUSTOMER
        LO_PARTKEY int NOT NULL, -> FK to PART
        LO_SUPPKEY int NOT NULL, -> FK to SUPPLIER
        LO_ORDERDATE int NOT NULL,
        LO_ORDERPRIORITY char(15) NOT NULL,
        LO_SHIPPRIORITY char(1) NOT NULL,
        LO_QUANTITY int NOT NULL,
        LO_EXTENDEDPRICE int NOT NULL,
        LO_ORDTOTALPRICE int NOT NULL,
        LO_DISCOUNT int NOT NULL,
        LO_REVENUE int NOT NULL,
```

```
LO_SUPPLYCOST int NOT NULL,
LO_TAX int NOT NULL,
LO_COMMITDATE int NOT NULL,
LO_SHIPMODE char(10) NOT NULL);
```

At scale factor 10 the row counts are given in Table 1. More details on the SSB

Table 1. Scale Factor

lineorder	59986052
part	800000
customer	300000
supplier	20000
ddate	2556

benchmark can be found at [17].

4.3 Implementation and Testing Joint and Sequential Approaches

To implement the Joint and Sequential fragmentation/allocation schemes and assess their respective effects in *Teradata* we proceeded as follows:

1. The joint and sequential algorithms is applied to the SSB workload (query descriptions shown later). We used a cost model that is focused on the I/O factor. For more details on these algorithms refer to [2]. Sequential approach starts by partitioning the SSB benchmark schema using a genetic algorithm [2]. The obtained fragmentation schema is then allocated over various nodes.
2. The obtained theoretical results from our algorithms are implemented to *Teradata* as follows:
   - The dimension tables are hash distributed using their primary key field.
   - The fact table is partitioned using the results of each of the *sequential* and *joint* algorithms. Each fragment is represented as a separate table. These fragments were then allocated to particular AMPs based on the hash function that reflect the allocation scheme.
   - Finally, the fact table *lineorder* is defined as a view with UNION of these fragments

For experiments in this section, we have considered a query workload of 22 queries. They are based on the original 13 queries (except Q1.2, Q1.3) but with varied predicates. We have used 50 selection predicates defined on 11 different attributes: (*d.d_year, p.p_category, d.d_yearmonth, s.s_region, p.p_brand , c.c_region, c.c_nation, s.s_nation, c.c_city, s.s_city, p.p_mfgr*). The domains of these attributes are split into: 7, 2, 2, 7, 3, 5, 2, 6, 3, 3 and 3 sub domains, respectively to perform the genetic algorithms for joint and sequential approaches [2]. Note that each selection predicate has a selectivity factor computed using SQL queries executed on the data set of SSB benchmark (these queries are available at: http://www.lisi.ensma.fr/ftp/pub/documents/reports/2011/2011-LISI-.pdf).

Table 2. Results (Time in Seconds)

Queries	Joint	Sequential
Q01.1	0.12	0.59
Q01.2	0.11	0.11
Q01.3	0.12	0.11
Q04.1	0.60	0.58
Q04.2	0.54	0.53
Q04.3	1.08	1.13
Q05.0	0.36	0.15
Q06.0	0.53	0.13
Q07.0	0.46	0.58
Q08.0	0.14	2.15
Q09.0	0.08	0.14
Q10.0	0.07	0.16
Q11.0	0.32	0.62
Q12.0	0.34	0.72
Q13.0	0.18	0.63
Q14.0	0.18	0.24
Q15.0	0.18	0.57
Q16.0	0.18	0.51
Q17.0	0.19	0.24
Q18.0	0.29	0.64
Q19.0	0.56	0.66
Q20.0	0.49	0.33
total time	7.12	11.52

This technique was applied for both *Joint* and *Sequential* approaches. Table 2 shows the run time (in seconds) of the queries for both joint and sequential methods. Each method performed better in certain queries. However, the joint method performed 38% better for the overall workload which validates the results since the workload performance is the objective of both algorithms.

5 Conclusion

This work is the fruit of collaboration between academician and industrial representing by *Teradata Labs* established during the *Conference on Data Warehousing and Knowledge Discovery* (DAWAK) that hold in Bilbao Spain in 2010, where we presented our paper on *joint parallel data warehouse design*. In our previous studies, the verification of the joint method is based on a single CPU system simulating parallel and distributed systems. In this paper, we verified the superiority of our joint method over sequential method using the Teradata DBMS running the SSB benchmark. Overall, the joint method performed 38% better than the sequential method. Based on this collaboration, a deployment methodology of parallel databases design is identified that can be generalized on other parallel DBMS.

Future extensions include incorporating the Teradata cost model in the joint solution. This will insure that the actual cost of CPU, I/O and network cost are reflected. Also, other models beyond star schema like the TPC-H model can be benchmarked.

References

1. Apers, P.M.G.: Data allocation in distributed database systems. ACM Transactions on Database Systems 13(3), 263–304 (1988)
2. Bellatreche, L., Benkrid, S.: A joint design approach of partitioning and allocation in parallel data warehouses. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 99–110. Springer, Heidelberg (2009)
3. Bellatreche, L., Boukhalfa, K., Richard, P.: Data partitioning in data warehouses: Hardness study, heuristics and ORACLE validation. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2008. LNCS, vol. 5182, pp. 87–96. Springer, Heidelberg (2008)
4. Bellatreche, L., Cuzzocrea, A., Benkrid, S.: *F&a*: A methodology for effectively and efficiently designing parallel relational data warehouses on heterogeneous database clusters. In: Bach Pedersen, T., Mohania, M.K., Tjoa, A.M. (eds.) DAWAK 2010. LNCS, vol. 6263, pp. 89–104. Springer, Heidelberg (2010)
5. Bellatreche, L., Woameno, K.Y.: Dimension table driven approach to referential partition relational data warehouses. In: ACM 12th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 9–16 (2009)
6. Bernardino, J., Madeira, H.: Experimental evaluation of a new distributed partitioning technique for data warehouses. In: International Database Engineering & Applications Symposium, IDEAS, pp. 312–321 (2001)
7. Bouganim, L., Florescu, D., Valduriez, P.: Dynamic load balancing in hierarchical parallel database systems. In: Proceedings of the International Conference on Very Large Databases, pp. 436–447 (1996)
8. Ceri, S., Negri, M., Pelagatti, G.: Horizontal data partitioning in database design. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices, pp. 128–136 (1982)
9. DeWitt, D.J., Gray, J.: Parallel database systems: The future of high performance database systems. Communnications ofthe ACM 35(6), 85–98 (1992)
10. DeWitt, D.J., Madden, S., Stonebraker, M.: How to build a high-performance data warehouse, `http://db.lcs.mit.edu/madden/high_perf.pdf`
11. Eadon, G., Chong, E.I., Shankar, S., Raghavan, A., Srinivasan, J., Das, S.: Supporting table partitioning by reference in oracle. In: SIGMOD 2008 (2008)
12. Furtado, P.: Experimental evidence on partitioning in parallel data warehouses. In: DOLAP, pp. 23–30 (2004)
13. Karlapalem, K., Pun, N.M.: Query driven data allocation algorithms for distributed database systems. In: Tjoa, A.M. (ed.) DEXA 1997. LNCS, vol. 1308, pp. 347–356. Springer, Heidelberg (1997)
14. Lima, A.B., Furtado, C., Valduriez, P., Mattoso, M.: Parallel olap query processing in database clusters with data replication. Distributed and Parallel Databases 25(1-2), 97–123 (2009)
15. Mehta, M., DeWitt, D.J.: Data placement in shared-nothing parallel database systems. VLDB Journal 6(1), 53–72 (1997)

16. Menon, S.: Allocating fragments in distributed databases. IEEE Transactions on Parallel and Distributed Systems 16(7), 577–585 (2005)
17. O'Neil, P., O'Neil, E.B., Chen, X.: The star schema benchmark (2007), http://www.cs.umb.edu/~poneil/starschemab.pdf
18. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice Hall, Englewood Cliffs (1999)
19. TPC Home Page. Tpc benchmark^{TM}d (decision support), http://www.tpc.org
20. Rahm, E., Marek, R.: Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In: Proceedings of the International Conference on Very Large Databases, pp. 182–193 (1993)
21. Rao, J., Zhang, C., Megiddo, N., Lohman, G.M.: Automating physical database design in a parallel database. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 558–569 (2002)
22. Röhm, U., Böhm, K., Schek, H.: Olap query routing and physical design in a database cluster. In: Zaniolo, C., Grust, T., Scholl, M.H., Lockemann, P.C. (eds.) EDBT 2000. LNCS, vol. 1777, pp. 254–268. Springer, Heidelberg (2000)
23. Röhm, U., Böhm, K., Schek, H.: Cache-aware query routing in a cluster of databases. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 641–650 (2001)
24. Saccà, D., Wiederhold, G.: Database partitioning in a cluster of processors. ACM Transactions on Database Systems 10(1), 29–56 (1985)
25. Stöhr, T., Märtens, H., Rahm, E.: Multi-dimensional database allocation for parallel data warehouses. In: Proceedings of the International Conference on Very Large Databases, pp. 273–284 (2000)
26. Stöhr, T., Rahm, E.: Warlock: A data allocation tool for parallel warehouses. In: Proceedings of the International Conference on Very Large Databases, pp. 721–722 (2001)

Memory Performance and SPEC OpenMP Scalability on Quad-Socket x86_64 Systems

Daniel Molka, Robert Schöne, Daniel Hackenberg, and Matthias S. Müller

Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, 01062 Dresden, Germany
{daniel.molka,robert.schoene,daniel.hackenberg,
matthias.mueller}@tu-dresden.de

Abstract. Because of the continuous trend towards higher core counts, parallelization is mandatory for many application domains beyond the traditional HPC sector. Current commodity servers comprise up to 48 processor cores in configurations with only four sockets. Those shared memory systems have distinct NUMA characteristics. The exact location of data within the memory system significantly affects both access latency and bandwidth. Therefore, NUMA aware memory allocation and scheduling are highly performance relevant issues. In this paper we use low-level microbenchmarks to compare two state-of-the-art quad-socket systems with x86_64 processors from AMD and Intel. We then investigate the performance of the application based OpenMP benchmark suite SPEC OMPM2001. Our analysis shows how these benchmarks scale on shared memory systems with up to 48 cores and how scalability correlates with the previously determined characteristics of the memory hierarchy. Furthermore, we demonstrate how the processor interconnects influence the benchmark results.

1 Introduction

The performance demands of applications continuously grow and to face this challenge, increasing core counts have become the dominant factor in microprocessor development. For server and HPC workloads with high degrees of parallelism, it is common to use multiple processors to further improve performance. Such multi-socket, multi-core systems are usually implemented as cache coherent shared memory systems. They provide a global view on the available memory, allowing multiple threads to share a single address space and exchange data via the jointly used memory. Based on cache coherent shared memory, the language extension OpenMP enables developers to easily parallelize applications written in C/C++ or Fortran. OpenMP provides an API that allows programmers for example to define parallel regions and to parallelize loops. Within a parallel region, multiple threads are executed concurrently and the workload can be distributed among them. The number of threads is determined at runtime. Therefore, one executable can be flexibly used on systems of different scale and still utilize all available cores. However, this simple programming model

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 170–181, 2011.

does not consider any hardware specifics. The shared memory can be physically distributed among the sockets in the system. The non-uniform memory access (NUMA) results in different memory latencies and bandwidths depending on the location of the data in the system. Therefore, NUMA optimized memory allocation and the placement of threads that access the data are strongly performance relevant issues. Additionally, data can be cached at different locations and resources are shared by multiple cores what increases the complexity even further. In this paper we therefore use sophisticated low-level memory benchmarks to characterize the performance of memory accesses and data transfers between caches on two quad-socket NUMA systems using multi-core x86_64 processors from AMD and Intel. Furthermore, we use SPEC OMPM2001 to investigate how these performance characteristics influence the performance and scalability of typical shared memory high performance computing application.

2 Related Work

The microbenchmarks we use to examine the latencies and bandwidths between the different memory components were first introduced in [6]. They have been further adapted to support additional coherence states and to collect fundamental performance data of two socket x86_64 servers in [4]. SPEC OMP2001 is a well-established shared memory benchmark suite. Saito et al. introduce these benchmarks and present a scalability analysis for up to 128 threads running SPEC OMP benchmarks on medium and large datasets [8]. Aslot et al. use a quad-processor UltraSPARC II system to gather performance related information for specific code sections of the SPEC OMP benchmark applications [1]. They list the most time consuming code regions and discuss scalability issues. Moreover, Müller et al. present scalability and performance results for SPEC OMPL2001 and SPEC HPC2002 benchmarks [7]. Fürlinger et al. analyze the scalability of SPEC OMPM2001/OMPL2001 benchmarks on a 32 processor Itanium system and use the profiling tool `ompP` to break scalability issues down to reasons like thread management, imbalances, and synchronization [3]. All these analyses have been performed on systems with single-core processors and therefore do not provide performance insights for systems consisting of multi-core processors with shared L3 cache and integrated memory controllers.

3 Test Systems

We analyze two quad-socket cache coherent NUMA systems with processors from Intel and AMD. Table 1 summarizes the hardware configuration.

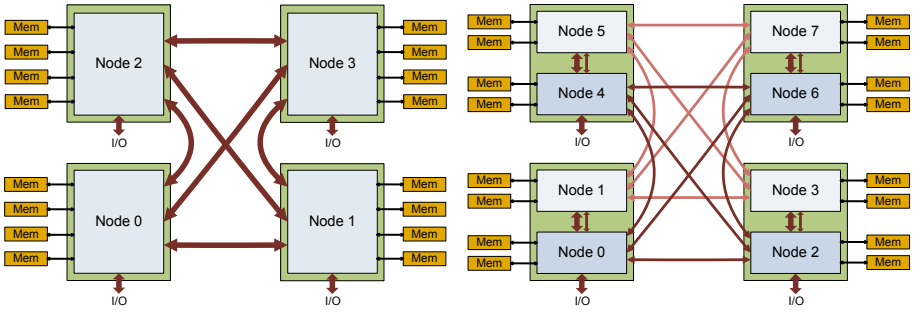
The 8-core Intel Xeon 7500 series processors consist of a monolithic die and feature 24 MiB of shared L3 cache. The two integrated memory controllers each provide two scalable memory interface (SMI) channels. Connected to each SMI channel is a scalable memory buffer (SMB) that controls the DDR3 memory. Each processor features 4 QuickPath Interconnect (QPI) links. One link is used

Table 1. Hardware configuration of test systems

System	Nehalem-EX	Magny Cours
Processors	4x Intel Xeon X7560	4x AMD Opteron 6172
Cores	32 (SMT disabled)	48
Core clock	2.266 GHz (w/o Turbo Boost)	2.100 GHz
Cache	2x 32 KiB L1, 256 KiB L2 per core 24 MiB L3 per processor	2x 64 KiB L1, 512 KiB L2 per core 2x 6 MiB L3 per processor
Interconnect	6.4 GT/s QPI (25.6 GB/s)	6.4 GT/s HT 3.0 (25.6 GB/s)
Memory configuration	256 GiB DDR3-1066 4x SMI per socket	64 GiB DDR3-1333 4x DDR3 per socket
OS	Red Hat EL6 2.6.32-71.el6.x86_64	Ubuntu 10.10 2.6.35-22-server
Compiler	gcc 4.4.4, icc 11.1 (20091130)	gcc 4.4.5, icc 11.1 (20091130)

to connect to the chipset, three links can be used to connect to other processors. The 12-core AMD Opteron 6100 series processors are multi-chip-modules (MCM). They consist of two six-core dies that are internally connected via HyperTransport (HT) links. Each die includes 6 MiB shared L3 cache and a dual-channel DDR3 controller for a total of 12 MiB L3 and four DDR3 channels per processor. Each socket supports four 16-Bit HT 3.0 links, one for communication with the chipset and three to connect to other sockets.

The topologies of the test systems are depicted in Figure 1. The four sockets in the Intel system are fully connected via QPI links. Each link provides 25.6 GB/s of raw bandwidth (12.8 GB/s per direction). Data is transferred in 64 Byte packages each with an 8 Byte header [5,10]. This protocol overhead limits the achievable bandwidth to 11.37 GB/s per direction. The AMD system has four sockets as well, however it consists of eight NUMA nodes. The three 16-Bit links per socket that connect the processors are actually used as six 8-Bit links that provide 12.8 GB/s (6.4 GB/s per direction) each. The dies within a MCM are



(a) four socket Xeon 7500 (b) four socket Opteron 6100

Fig. 1. System topology comparison

connected via one 16-Bit and one 8-Bit link [2]. Every die is directly connected to three dies in other sockets. There are two sets of four fully connected nodes: {0,2,4,6} and {1,3,5,7}. Data transfers between those groups require two HT hops if the nodes are not in the same MCM.

Both systems use snooping based protocols to maintain cache coherence. The additional messages reduce the effective bandwidth. AMD implements a probe filter, called HT Assist, that uses a portion of the L3 cache as directory in order to filter unnecessary messages [2]. However, that reduces the L3 capacity and the L3 bandwidth has to be shared between data accesses and probe filter accesses. Furthermore, only 16 MiB of a node's memory can be cached somewhere in the system as the directory size is limited.

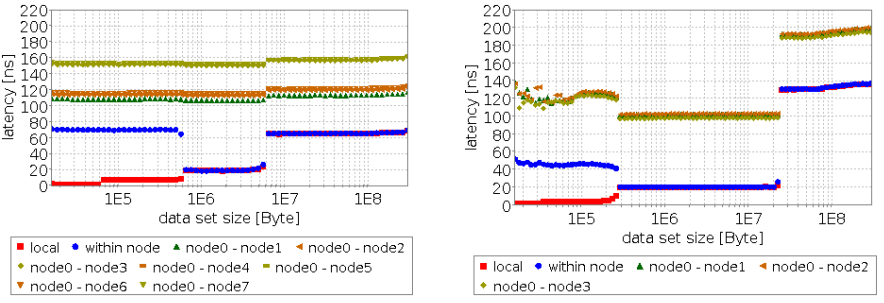
4 Benchmarks

4.1 Microbenchmarks

We use a set of open source microbenchmarks [6,4] to perform a low-level analysis of each system's memory performance. Highly optimized assembler routines and time stamp counter based timers enable precise performance measurements of data accesses in cache coherent NUMA systems with a 64 Bit x86 processors. The benchmarks are parallelized using pthreads and individual threads are pinned to single cores using `sched_setaffinity()`. Coordinated data access sequences are performed in consideration of the coherence protocol to transfer data into a selected cache in a well-defined coherence state. Latencies and bandwidths of accesses to any core's caches and any socket's memory can be measured as well as the aggregated bandwidth of shared caches and memory controllers. The benchmarks support different data allocation schemes: `localalloc` results in all threads using memory local to their NUMA node while `globalalloc` forces all memory to be allocated at the first NUMA node.

4.2 SPEC OMPM2001

To show the impact of the different hardware characteristic of the two platforms on application performance we use the SPEC OMP Benchmark suite V3.2. The 11 codes from SPEC OMPM2001 cover a wide range of applications and use different OpenMP constructs for the parallelization. The philosophy to use real applications results in relatively complex performance properties. More details about the benchmark and its performance properties can be found in [8,1,7,3]. We use the Intel Compiler Suite in version 11.1 on both systems to create the benchmark executables. The same optimization flags are used for all benchmarks, namely `-O3 -ipo -openmp`. To consider the different SIMD extensions we additionally use `-msse3` on the AMD and `-xSSE4.2` on the Intel system. `Likwid-pin` [9] is used to avoid thread migration.



(a) Opteron 6172 (b) Xeon X7560

Fig. 2. Access latencies: Thread on core 0 accesses local or other core's memory

5 Results

5.1 Memory Latency

The microbenchmark results of cache and memory latencies are depicted in Figure 2 and summarized in Table 2.

Figure 2a depicts the latencies in the quad-socket Opteron 6172 system. Accesses to local caches require 1.4 ns (3 cycles) for the L1, 7.1 ns (15 cycles) for the L2, and 19 ns (40 cycles) for the L3 cache. Reading data from the L1 or L2 cache of another core on the same die requires 70 ns. This is significantly higher than measured on older Opteron models without HT Assist [4] as probing other cores is delayed until the HT Assist directory has been checked. With 109 ns the latency between the two dies within a MCM is only marginally lower than the 113 ns for accessing directly connected dies in other sockets. A second HT hop adds another 40 ns to the latency. Memory requests have a latency of 65 ns for local memory, 113-119 ns for memory that can be reached with one HT hop, and 159 ns if two hops are necessary. Without HT Assist, responses from memory would be delayed until the farthest nodes reply to the probe message. This would require approximately 133 ns (152 ns for 2-hop L3 accesses includes HT Assist lookup of about 19 ns). Thus, the HT Assist reduces the local memory latency by more than 50%.

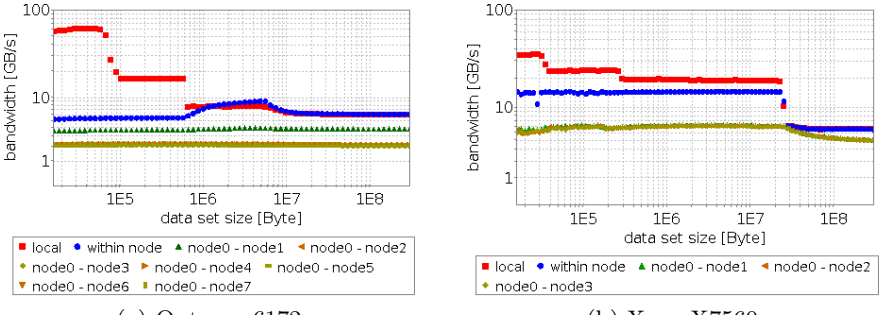
Table 2. Access latencies in ns, cache lines in state modified

Processor	Opteron 6172					Xeon X7560		
Source	local	within socket		other socket		local	within socket	other socket
		on-die	2nd die	1 hop	2 hops			
L1	1.4	70.4	109.5	113.3	153.3	1.8	48.2	126.0
L2	7.1	70.4	109.5	113.3	153.3	4.4	46.4	128.2
L3	19.0	19.0	107.6	111.9	152.4	20.3	20.3	103.0
RAM	65.7	65.7	114.3	119.0	159.0	130.4	130.4	192.8

Figure 2b shows the latencies on the quad-socket Xeon X7560 system. With 1.8 ns (4 cycles), the L1 cache is slightly slower than the AMD implementation. The L2 is faster (4.4 ns; 10 cycles) and the L3 has almost the same latency (around 20 ns). With around 47 ns on-die transfers from other L1 and L2 caches are notably faster. As can be expected from a fully connected system, the latencies for all other sockets are almost identical. However, the single QPI hop adds about 62 ns to the latency compared to 40 ns per HT hop. Memory latencies are 130 ns for the local memory and 192 ns for memory at other sockets which is much more than on the AMD system. It is also significantly more than the 103 ns latency of remote L3 cache accesses that provide an estimate for the snoop responses. The high latency is therefore not a problem of the coherence mechanism but can be attributed to the SMBs that translate from the processor's SMI interface to DDR3.

5.2 Memory Bandwidth

Figure 3 depicts the read bandwidths that we measured for a single thread. A summary that also includes write bandwidths can be found in Table 3. While the Opteron's L1 caches have two 128-Bit read ports (32 Byte per cycle), the Xeon's L1 has only one 128-Bit read port. The L1 write bandwidth is 16 Byte per cycle on both systems. On the Intel system, the L2 and L3 caches provide more bandwidth. Data exchanges between cores on a single die achieve higher bandwidths as well. The single threaded bandwidth from/to main memory is relatively low on both systems. Remote accesses again show the more complex topology of the AMD system. Data can only be read with about 3.8 GB/s from caches or memory of the processor's second die. The bandwidth drops further down to 2.1 GB/s for reading from other sockets. On the Intel system, data can be read with 6.3 GB/s from caches in other processors and 3.9 GB/s from remote memory. These transfer rates are limited by the interconnect as well as by the outstanding requests supported by a single core.



(a) Opteron 6172 (b) Xeon X7560

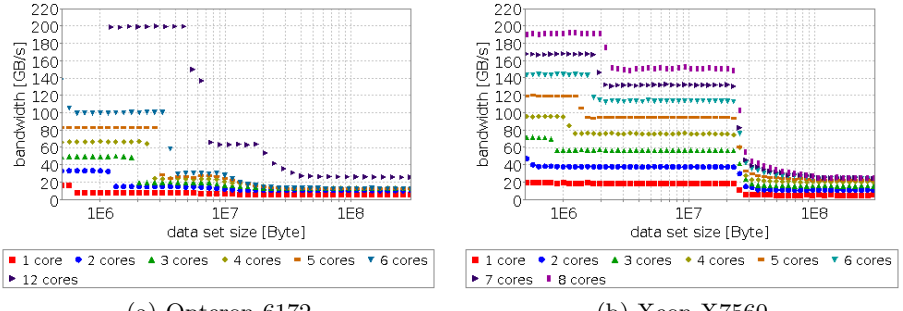
Fig. 3. Memory read bandwidth: Thread on core 0 accesses other core's memory

Table 3. Single thread read (write) bandwidth in GB/s. Write bandwidth listed only for local caches and memory as it cannot be written into other cores' caches.

Processor	Opteron 6172					Xeon X7560		
Source	local	within socket		other socket		local	within socket	other socket
		on-die	2nd die	1 hop	2 hops			
L1	62.9 (31.8)	5.4	3.7	2.1	2.1	35.2 (35.2)	14.3	6.3
L2	16.7 (10.0)	5.4	3.7	2.1	2.1	24.1 (22.3)	14.3	6.3
L3	7.8 (7.0)	9.0	3.9	2.1	2.1	19.2 (12.7)	14.3	6.3
RAM	6.1 (4.4)	6.1 (4.4)	3.8 (3.2)	2.1 (2.0)	2.1 (2.0)	5.5 (4.9)	5.5 (4.9)	3.9 (3.5)

By using eight concurrent threads , 11.0 GB/s can be transferred over one QPI link. This is very close to the expected peak performance of 11.37 GB/s (see Section 3). Six cores on one die of the Opteron 6172 processor can read data with 5.3 GB/s from the second die in the MCM which is much lower than expected. However, no more than 2.1 GB/s can be read from dies in other sockets via the 8-Bit links even with multiple cores reading in parallel. Thus, the bandwidth limit is not caused by too few outstanding requests per core but by the width of the HT links. The low performance indicates that only one link is used to exchange data between two dies in different sockets, even though two links are available between the sockets.

The scaling behavior of the L3 and main memory read bandwidth is depicted in Figure 4 for single NUMA nodes. The results as well as the corresponding write bandwidths are summarized in Table 4. On the AMD system the six cores of one die share an aggregate L3 bandwidth of 30.9 GB/s which is identical for reading and writing. Each dual-channel memory controller delivers a read and write bandwidth of 13.2 GB/s and 7.1 GB/s, respectively. On the Intel system, L3 read and write bandwidths scale linearly with the number of cores and reach a total of 152 and 101 GB/s, respectively. Thus, the Xeon's shared



(a) Opteron 6172 (b) Xeon X7560

Fig. 4. Memory read bandwidth scaling for concurrent accesses of multiple cores

Table 4. Aggregate read (write) bandwidth per socket in GB/s

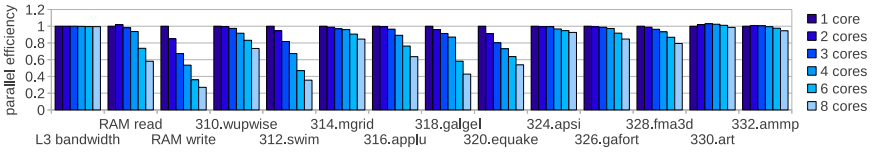
Source		1 core	2 cores	4 cores	6 cores	8 cores	12 cores
Opteron 6172	L3	7.8 (7.1)	15.2 (14.1)	24.1 (24.5)	30.8 (30.9)	n/a	63.7 (61.8)
	RAM	6.1 (5.1)	10.7 (6.6)	13.2 (7.1)	13.2 (7.1)	n/a	26.2 (14.0)
Xeon X7560	L3	19.2 (12.7)	38.3 (25.4)	76.6 (50.8)	114 (76.0)	152 (101)	n/a
	RAM	5.5 (4.9)	11.4 (8.5)	20.6 (10.8)	24.5 (10.9)	25.7 (10.9)	n/a

L3 cache provides significantly more bandwidth than both of the Opteron's L3 partitions combined while being larger and shared by all cores. The memory read bandwidth per socket is almost identical on the two systems (AMD: 26.2 GB/s, Intel: 25,7 GB/s) while the AMD system provides higher write bandwidth (AMD: 14.0 GB/s, Intel: 10,9 GB/s).

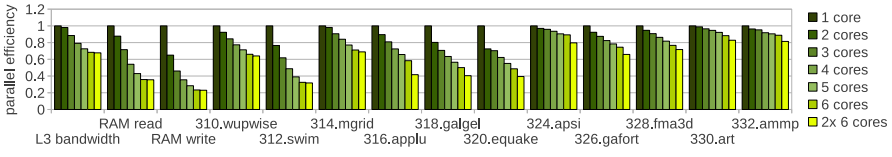
The L3 and memory bandwidths scale linearly with the number of sockets on both systems if memory is allocated by each thread (localalloc). However, if all memory is allocated and initialized by the master thread (globalalloc), memory bandwidth is reduced significantly. In this case performance is limited as only the memory controllers of the first NUMA node are used. The bandwidth limitations of the processor interconnects affect the performance as well. The Intel system provides 17.7 GB/s memory bandwidth if all four sockets use memory from node0. On the AMD system, the single socket bandwidth is already reduced to 7.5 GB/s as two of the processor's four memory channels remain unused and half the cores access memory of the second die via the intra-socket HT connection. If all sockets are used, the 8-Bit links that connect the individual dies become the limiting factor resulting in an even lower bandwidths of 6.6 GB/s for this worst-case scenario. Furthermore, the limited coverage of the HT Assist directory causes invalidations of cache lines if more than 16 MiB of node0's memory are cached somewhere in the system. The 48 L2 caches (24 MiB) already exceed that capacity, thus the effective L3 size is reduced to zero. While both systems require NUMA optimized memory allocation to achieve optimal memory performance, the low-level benchmarks strongly indicate that the AMD memory subsystem is much more sensitive to non-optimal memory allocation.

5.3 SPEC OMPM2001 Scaling with Multiple Cores

For our OpenMP scalability analysis we first examine the parallel efficiency of SPEC OMPM2001 applications running on a single socket of our quad-socket test systems (see Figure 5). Also depicted in Figure 5 are the L3 bandwidth as well as main memory read and write bandwidth. On the Intel system, the L3 cache design is remarkably powerful, allowing the L3 bandwidth to scale linearly. The L3 cache of the AMD processor does not scale linearly, neither does the main memory bandwidth of both systems. Especially the memory write bandwidth scales poorly with a growing number of cores. The limited scaling of shared resources (see Table 4) has a strong influence on the benchmark results, which results in significantly different scaling behavior on multi-core processors than



(a) Xeon X7560



(b) Opteron 6172

Fig. 5. SPEC OMPM2001 scaling with multiple cores of a single socket. A parallel efficiency of 1 indicates linear speedup.

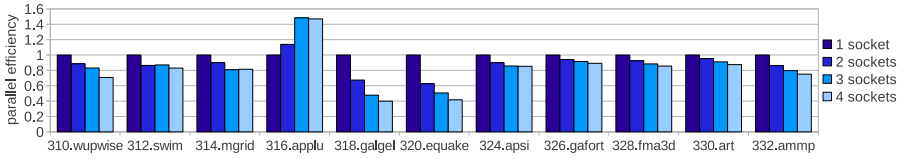
reported in [3] for single-core multi-socket systems. Based on the results shown in Figure 5 the benchmarks can be divided into three groups:

- *Group 1* includes mainly compute bound benchmarks. 324.apsi, 330.art, and 332.ammp show marginal memory influence on the Intel system as well as a minor L3 cache dependence.
- *Group 2* consists of 310.wupwise, 314.mgrid, 326.gafort, and 328.fma3d that are subject to a higher influence of L3 cache and main memory bandwidths.
- *Group 3* contains the strongly memory bound benchmarks 312.swim, 316.applu, 318.galgel, and 320.equake.

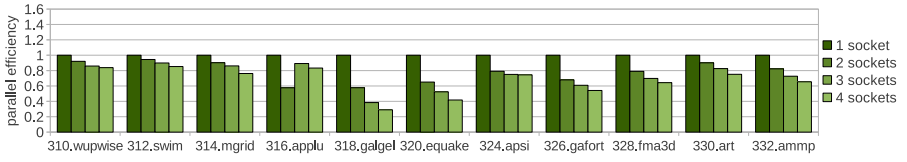
Figure 5b also shows the scalability effects on the Opteron 6172 processor when both dies (2x 6 cores) are being used. 310.wupwise, 312.swim, and 314.mgrid scale almost linearly. They are not affected by the limited interconnect bandwidths and increased access latencies. The remaining benchmarks show significantly reduced parallel efficiency if both NUMA nodes of a single socket are being used.

5.4 SPEC OMPM2001 Scaling with Multiple Sockets or NUMA Nodes

Application scaling on multiple sockets is additionally influenced by the inter-socket connectivity via HT or QPI links. Higher latencies and lower bandwidths compared to the intra-socket case worsen multi-socket scalability. In addition to the coherence traffic, the links are stressed by accesses to distant memory due to NUMA unaware memory allocation. These considerations need to be taken into account for the evaluation of all performance results depicted in Figure 6. Unfortunately, the grouping with respect to memory boundedness does not apply here as the interconnect influence is different for the individual benchmarks. We



(a) Xeon X7560 (8 cores per socket)



(b) Opteron 6172 (12 cores per socket)

Fig. 6. SPEC OMPM2001 scaling with multiple sockets. A parallel efficiency of 1 indicates linear speedup.

observe good multi-socket scaling of 310.wupwise, 312.swim, and 314.mgrid on both systems. 316.applu shows strong super-linear speed-up on the Intel system in accordance to the findings of [8] and [3]. The data set of this benchmark apparently exceeds the available cache size of two AMD sockets and fits well into the caches of three or more sockets. However, the cache effect is alleviated by other effects that hinder multi-socket scalability. This behavior likely results from non-local memory usage, as e.g. the array initialization in `ssor.f:49` ff can provoke non-local memory accesses. The parallelized loop iterates over j from jst to $jend$ to initialize eight arrays that are later used for calculations in disregard of the locality information (e.g. in subroutine `blts`). 318.galgel and 320.equake scale poorly on both systems in accordance to [3]. A non-optimal memory allocation is likely a contributor to that behavior as single-socket scaling appears to be similar to memory bound benchmarks such as 312.swim. The remaining benchmarks scale significantly better on the Intel system, most likely due to the higher intra-socket bandwidths. This is particularly noticeable for 326.gafort, that accesses random indices of a computation matrix in one of the main loops (`shuffle-do#10`). This access pattern as well as the surrounding OMP locks lead to high inter-socket traffic.

5.5 SPEC OMPM2001 Performance Comparison

For a comparison of the overall performance of our test systems, it is important to note that the AMD system has an approx. 40% higher peak computing performance. This potentially benefits applications that are computationally bound. On the other hand, the Intel Xeon processor strongly benefits from its large L3 cache as well as the high L3 bandwidth, potentially improving the performance of memory bound applications. Figure 7 depicts the relative performance (AMD/Intel) of the two test systems. Note that the 'half socket' main

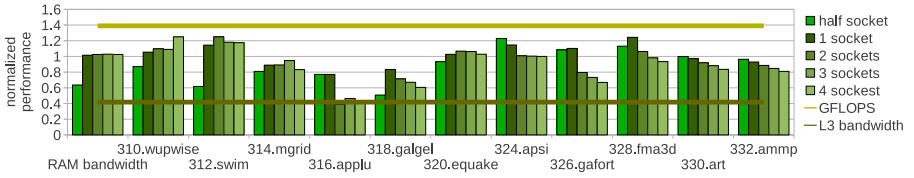


Fig. 7. SPEC OMPM2001 application performance of Opteron 6172 relative to Xeon X7560. Bars > 1 indicate a performance advantage for the Opteron.

memory bandwidth is much lower on AMD compared to Intel, as only two of the Opteron's four memory channels are used in this case. 312.swim and 318.galgel are particularly sensitive to this. The Intel system generally scales better over multiple sockets as illustrated by the decreasing relative performance of 318.galgel, 326.gafort, 328.fma3d, 330.art, and 332.ammp. One outlier is 316.applu, which shows super-linear speedup as stated in Section 5.4 and should therefore not be analyzed in detail at this point. As pointed out in Section 5.3, 310.wupwise, 312.swim, and 314.mgrid are not sensitive to inter-socket communication performance. The scaling behavior of these benchmarks is therefore competitive on the AMD system, with 310.wupwise and 312.swim showing superior overall performance. On average the scaling is worse on the Opteron system, despite AMD's HT Assist feature. This indicates that the unexpectedly low HyperTransport bandwidths (see Table 3) are significantly limiting the performance for benchmarks that perform remote cache or memory accesses. Therefore, the total SPEC OMPM2001 result is significantly higher on the Intel system while single-socket performance is almost identical.

6 Conclusions

Today's shared memory x86 systems are complex cache coherent NUMA architectures. Multiple processors are connected via point-to-point interconnects and each processor features multiple cores. This results in different performance levels for data transfers depending on the exact location of the data source. Sophisticated low-level microbenchmarks that are tailored to investigate the performance of data transfers within shared memory systems are used to present precise results for latencies and bandwidth for on-die and off-die data transfers in two state-of-the-art quad-socket x86_64 systems. Considerable limitations of the inter-socket communication bandwidths are identified. We also provide detailed data for the scaling behavior of shared resources such as last level caches and main memory.

In our scalability analysis of SPEC OMPM2001 applications, both test systems show severe deficiencies in terms of overall parallel efficiency. Scaling is limited by shared resources within each NUMA node that do not scale linearly with the number of cores as well as non-linear scaling with the number of NUMA nodes. This shows that the bottlenecks identified by the microbenchmarks also limit the performance and scalability of real applications. For the majority of

applications, the Intel test system performs better regarding both aspects. While the superior scaling for one processor is expected due to the single-die solution and the extremely well-performing L3 cache, the lower performance of the AMD system in multi-socket configurations is surprising. With respect to the tested workloads, the HT Assist feature does not prove to be sufficient to compensate for the disadvantages that come with the more complex topology and the comparatively low HyperTransport bandwidth.

Acknowledgment. The authors would like to thank Intel Germany for providing us with the Intel Nehalem-EX test system. We also thank NEC Deutschland GmbH for granting us access to the AMD Magny Cours test system.

References

1. Aslot, V., Eigenmann, R.: Quantitative performance analysis of the SPEC OMPM2001 benchmarks. Sci. Program. 11, 105–124 (2003)
2. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE Micro 30, 16–29 (2010)
3. Fürlinger, K., Gerndt, M., Dongarra, J.: Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessors. In: Shi, Y., van Albada, G., Dongarra, J., Sloot, P. (eds.) ICCS 2007. LNCS, vol. 4488, pp. 815–822. Springer, Heidelberg (2007)
4. Hackenberg, D., Molka, D., Nagel, W.E.: Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In: MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 413–422. ACM, New York (2009)
5. Intel. An Introduction to the Intel QuickPath Interconnect (January 2009)
6. Molka, D., Hackenberg, D., Schöne, R., Müller, M.S.: Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In: PACT 2009: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 261–270. IEEE Computer Society, Washington, DC, USA (2009)
7. Muller, M.S., Kalyanasundaram, K., Gaertner, G., Jones, W., Eigenmann, R., Lieberman, R., Van Waveren, M., Whitney, B.: SPEC HPG benchmarks for high performance systems. Int. J. High Perform. Comput. Netw. 1, 162–170 (2004)
8. Saito, H., Gaertner, G., Jones, W., Eigenmann, R., Iwashita, H., Lieberman, R., van Waveren, M., Whitney, B.: Large system performance of SPEC OMP benchmark suites. International Journal of Parallel Programming 31, 197–209 (2003)
9. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: International Conference on Parallel Processing Workshops, pp. 207–216 (2010)
10. Ziakas, D., Baum, A., Maddox, R.A., Safranek, R.J.: Intel® quickpath interconnect architectural features supporting scalable system architectures. In: 2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI), pp. 1–6 (2010)

Anonymous Communication over Invisible Mix Rings

Ming Zheng[1,2], Haixin Duan[1], and Jianping Wu[1]

[1] Department of Computer Science and Technology, Tsinghua University, 100084 Beijing, P.R. China
[2] College of Humanities and Social Sciences, National University of Defense Technology, 410073 Changsha, Hunan, P.R. China
Zhengm09@gmail.com, duanhx@tsinghua.edu.cn, jianping@cernet.edu.cn

Abstract. Protect the identity of participants may be advantageous or essential and even critical for many internet applications. Mix rings architecture give better performance than mix-nets while maintaining anonymity that is stronger than onion routing. This paper presents an enhancement of mix rings, which is a hybrid P2P system and is designed to provide anonymity under a strong adversarial model in terms of identification, anonymity and resilience to collusion, along with low latency of data delivery and the link utilization. We use a double-layer overlay network, composed of nodes that are interested in anonymity and trusted index nodes and introduce several cluster escape and random extend mechanisms into mix rings. We present a description of the protocol, an analysis of common attack defense, and evaluate the degree of anonymity using MATLAB simulations.

Keywords: anonymous communication, mix rings, hybrid P2P, cluster escape.

1 Introduction

The Internet grows rapidly and is public acceptable as a means of communication. And anonymity has become an essential requirement for many Internet applications. There is an awkward situation now lot of anonymous communication systems and protocols are proposed for protecting the user's privacy. However, even if Internet users adopt these anonymous mechanisms, such as Crowds [1] or Onion Router [2], , when the users use static IP addresses for a long time the adversary can still collect the related data to figure out the real addresses and user identities behind the systems. To avoid the static address problem, Jia Zhang et al. suggests using DHCPv6 to get the "seed IP address", and generating changeable addresses during the communications[3]. Because of the limited capability of DHCPv4 protocol and the crisis of IPv4 address space, the proposed method is not feasible in the IPv4 world.

Matthew Burnside and Angelos D. Keromytis presents mix rings[4], which is stronger at terms of anonymity than onion routing while we performance than mix-nets. They show that mix rings provide anonymity that is stronger than Onion Router, and comparable to mix-nets. We present an enhancement of mix rings which provides sender anonymity described above, specifically, supports unlinkability anonymous

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 182–193, 2011.

communication against stronger adversary. Our system is built on top of hybrid peer-to-peer network which is completely distributed, semi-self-organized, and scalable with thousands of active participants who may communicate simultaneously. Our system can provide high level of anonymity while reducing the latency of data delivery and the link utilization. We achieve the anonymity and reductions by making use of mix rings and cluster escape. A number of peers joined to different mix rings dynamically change communication port and they only know little information about the ring. These properties make them anonymous. Also, our system is designed against passive and active attacks from outsider nodes and insider peers.

In our system, the sender anonymity is achieved by making everyone mix and directed relay messages on the ring, for example, when Alice sends a message to the ring and no one knows whether Alice is the originator or the relaying person. There may be a collusion attack, that if Alice relays messages from Bob to Carol as well as originates messages and sends to Carol, then if Bob and Carol collude they can identify messages originated by Alice. We want a higher degree of protection against collusion.

Unlinkability anonymity is also important. When Alice sends an anonymous message, no one would know she is using an anonymous communication services. But using Internet, sometimes the link information may be known by the receiver or adversary. In case of Internet connection-based applications, the sender is anonymous to the receiver but the adversary knows the link. A. Serjantov and P. Sewell's research [16] shows that all of the connection-based anonymity systems are still susceptive to the threat of passive attacks. We try to make our anonymity guaranties stricter – no entity must know anything about the senders or the link, even if collusion should reveal very little information.

The basic approach is that we first initialize the system by building an efficient trusted index nodes overlay in a decentralized manner, and then the messages which sender as a node of ring send flow through the variable-length mix ring . Any peer sets up its own forwarding table to forward messages it received. In that way, sender anonymity is achieved. Also the structure of trusted index nodes has a high degree of ring node distribution,, and making sender anonymity "collusion-free" that neighbors of the sender will not enter its ring. That is at least 2 neighbor nodes could not collude because they belong to different rings.

In Section 2, we overview related work. And then describe our protocol in details in section 3. In section 4, we present various attacks and analyze the anonymity provided by our system. Section 5 shows the evaluation of our system's anonymity degree. In section 6, we analyze the performance of our system. At last we offer concluding remarks in Section 7.

2 Related Work

In 1981 Chaum first proposed the concept of anonymous communication[5], through the store-mix-forward policy, encryption, digital pseudonyms and other technical methods, to hide the identities of the email senders. In the following 30 years, anonymous communication technology has been tremendous developed. As very representatives of the anonymous communication routing protocols, Crowds, Tarzan

[6] and Onion Routing greatly promoted the development of anonymous communication. DC-net[7], Xor-trees[8], P5 [9]are another category of anonymous communication which based on broadcast or multicast. In 2004, R. Dingledine N. Mathewson and P. Syverson presented Tor [10]. Tor has a good user experience, and also does better in the actual deployment of the system, attracting a large number of users and relay nodes. The relay nodes need to spontaneously even voluntarily contribute their bandwidth and system resources, it becomes scale on the Internet's largest users of anonymous communication systems.

Anonymous communication protocol currently proposes most typical feature-based connections, that the communication path by a number of anonymous proxies, usually a serial connection. The communication process is divided into two stages: first, the path establishment phase, that the intermediate nodes on the path connect to its successor, predecessor to build the forwarding path; Second, in the data transmission phase, that the data is transferred to the recipient along the forwarding path which sender has built. Path construction throughout the protocol by ensuring the safety and traceability of information transmission is not for achieving communication anonymity. And it's generally based on a key infrastructure, which needs to trust the key of the central node or the key parameters of the prior distribution. Between such anonymous communications, the nodes usually rely on pre-construction phase of the existing road or shared key encryption, public key transmission path information and consultation successor shared session key. In the data transmission phase, nodes rely on shared session key which has been negotiated on the information received decryption operations, thereby it leads to that the entry and exit information could not be associated.

3 Our Solution

Matthew Burnside and Angelos D. Keromytis propose an anonymous solution by mix rings. We enhance the solution against stronger adversary while keep low latency.

3.1 Structure

The core idea in our approach is a participant group, structured as mix rings. The use of ring is reminiscent of the Metro Link, which only passengers know where to get off. Mix rings provide sender anonymity and unlinkability.

In mix rings, every message received on a peer from predecessor peer is checked. If the peer checks successfully, it will view the content, else forwarded to the successor node. A generated message is repackaged and sent to successor. We follow a simple technique that the nodes only store the successor's address instead of the predecessor's.

The set of participants is formed by each node joining a mix ring and is likely to subway systems. When a message is received by a node, the node knows the message is from a node of group but don't know which one in the group exactly. Hence, the larger the group is, the higher the anonymity could be. But the ring length should be restricted.

3.2 Topology

In our system, we organize the participating peers into one or another directed ring, and each peer may connect to multiple rings. For a specific ring, each peer has a predecessor and a successor. As shown in Figure 1, all the peers are form different mix rings, and a peer could connect to two or more rings.

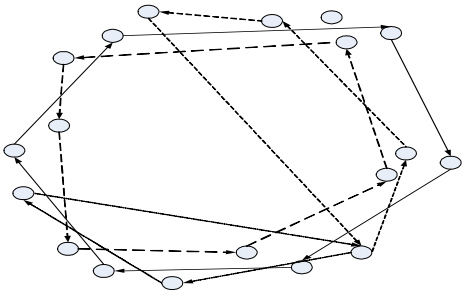


Fig. 1. Client peers construct different mix rings

In order to improve the network efficiency and reduce system communication cost, as shown in Figure 2, we adapt an ideas of building a manage overlay based on the network condition which is including distribution, latency, and bandwidth. And the goal of managing overlay is to control the physical network connections, so that the link usage and the latency could be minimized. The reason, for which we choose peers, is to keep each ring consist of at least n peers to provide a desired level of anonymity and performance balance. The larger number of nodes each ring consisted of, the higher level of anonymity.

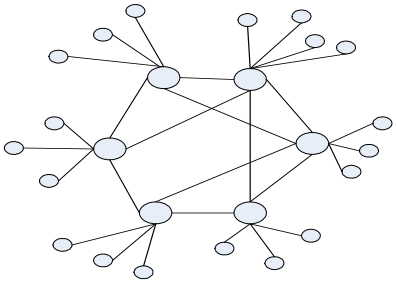


Fig. 2. Hybrid P2P structure/Big node is Local index peer & small node is Client peer

Initialization. In our solution, we assume that each client peer knows a index peer from bootstrap, and the peer is called global index peer. We achieve the overlay building by the following three steps. First, the client peer sends out local-registration-message to the global index peer. Second, global index peer receives the local-registration-message will forward the local-registration-message to a local index peer. Third, local index peer receives local-registration-message will register the

request and response to the client peer. If the client peer has received the response in fixed time, it start register process, otherwise, it will try to connect another global index peer, and keep connecting, until success. Client peers can always join/rejoin the overlay at any time from any global index peer in the overlay.

To register successfully, client peers are required to give a list of information what they know about the local index peer, so that their local index peer could add the list to their list. And then they can randomly pick peer from lists to construct the ring when request returns. A client peer checks its network condition by sending out start-test-request. Local index peers, who receive a start-test-request, could start a network condition test to check network transparency and source address validation rule. The results are stored to database of local index peers.

Maintenance. To maintain the system topology, we use two heartbeat timers at each client peer in the overlay. One timer is used to send out heartbeat message periodically and the other is for replacing communication ports to its successor. If it does not receive its local index peer's heartbeat on time, it will start selecting new local index peer. If local index peer does not receive its client peers' heartbeat, it will delete the client from its on-use ring client nodes list and re-select a client peer for the on-use ring. If a client peer receives a replace-port-message, it will change its service port immediately.

Balance. In the process of balance, the overlay topology mimics the real physical network connections so that the difference between them is minimized. The local index peers execute the balance algorithm periodically to adjust their client peers choice in the ring dynamically for achieving better system performance and scalability. The heartbeat messages contain the client peers list of the index, the successor of the ring client and communication port, etc. The local index peer follows the balance algorithm to find better ring latency upon receiving the heartbeat message from its clients.

The basic idea of the balance algorithm is that: first, if there is a ring for the peer to get the message from its predecessor, there is the physical path from the predecessor to the client peer which goes through the ring, the ring is seen as the client peer's load, and we can reduce the load stress between the client peers, which means we balance clients resources. Second, if one client peer fails, the local index peer sends replay-successor-message to its predecessor. The method that predecessor peer receives replay-successor-message from local index peers will follow balance algorithm.

3.3 Anonymous Communications

Anonymous communications are achieved through the cooperation of all participant peers. Each peer provides four basic functions. Based on these basic functions, the system provides four system functions, and then any peer can use the system for anonymous communications as they desire.

Basic Functions

Send. Any peer can be the originator of a message. Before each message is sent, the message should first be encrypted and packaged as common packet structure. A section random number and a message sequence number are used for the receivers to re-construct the messages.

Receive. In our system, all the messages are sent to ring. Each peer sets a filter to get the interested messages, so no one knows which peers get which kind of messages.

Relay. All peers in the system are responsible for message relay. For any message coming to, the peers should check it. If it is not the receiver of the message, the peer will send the message to its successor until the receiver received the message.

Pass. Pass is the function preformed by peers which could not forge source address only. As shown in Figure 2, any spoof-limited peers, which gets messages from its predecessor, makes a random choice that is either passing the message to another client peer or sending the message to successor. There is a local wide parameter called the forward probability, which indicates the probability according to which a peer will choose to pass. This is cluster escape mechanism 1.

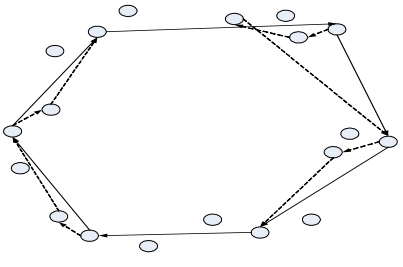


Fig. 3. Limited client peer random choose neighbor for helping pass the message

System Functions

Message repackage. Message repackage is used to repackage a message to another common packet structure. We use a lot of application packet structure such as http, ftp, e-mail packet structure and so on to repackage the message. If the nodes have capability to forge source address, we will use spoofed source address to replace real source address. This is cluster escape mechanism 2.

Key exchange. Key exchange is used for the client peers to send messages without expose their identity. If the client peer send message by sending plaintext to its receiver, it will be exposed. To overcome this shortcoming, the client peer sends the encrypted messages through mix ring channel to receiver and that receiver will send the encrypted response message to ring again. To exchange the key, the initiator client peer sends a random key to its local index peer, and the local index peer randomly chooses a public key to encrypt the random key and sends it to the target local index

peer. After that the target local index peer decrypts the random key with private key and sends it to responder client peer.

Mix traffic. To provide anonymity against a global eavesdropper, we use mix traffic to maintain peers' traffic patterns, i.e. the traffic pattern should be statistically independent of its originating data traffic. In our system, all peers in the same ring send messages to the ring at a system pre-defined rate. When a peer wants to send a message, it exchanges the random dummy message with the signal message, and a fixed bit indicates whether it's a dummy message or not. The messages are encrypted with the random key which exchanges between sender and receiver through local index peers, so it's not possible for the outsiders to know which messages are dummy messages. For all signal messages coming in, peers first try to decrypt and check the integrity. If the messages are decrypt successfully, peers will re-encrypt and re-package the messages, else they put them to outgoing buffer. At last the peers re-order the messages in the outgoing buffer and send them one by one. This is cluster escape mechanism 3.

Ring Adjustment. The ring is build to mimic the physical connections; the ring structure will remain relative stable once the structure is settled down. The attackers could figure out the client peers of a specific ring and compromise all its client peers, which will result in the expose of message from the specific peer. Besides the instability of the peers which goes on and off constantly, we design a protocol to let the ring adjust itself periodically, so that the ring structure is not predictable at the degree and the attacker could not compromise all peers of a ring within that adjustment period. We also change the send port and receive port periodically and randomly pass the message to another client under the same local index peer control. This is cluster escape mechanism 4.

Anonymous Communications

Sender-receiver anonymous communication. Sender-receiver anonymous communication is achieved by ring transfer. If the sender is a ring client, it will send the message to its successor, since the forming peers of the ring are randomly chosen by multiple local index peers; any peer in it except the sender could not know who the sender is. Once the message is sending to the ring, the intermediate node could not know who the send is either. It could be any client in the ring, and the anonymous set for random pass is even larger. Hence, the communication is sender anonymous. Since every client receives all the messages, and they filter out everything they want by themselves, it is receiver anonymous.

Sender anonymous communication. In this case, the sender makes a random key and sends it to local index peer. The local index peers exchange random keys for sender and randomly selected receiver. The sender encrypts the message using the random key, and then sends out the message. The receiver filters all messages on the key field, thus could receive the correct reply. For further anonymity the key can be changed periodically.

4 Attacks and Defense

Anyone outside the peer-to-peer system can monitor all the traffic going through the network. Since all messages are encrypted, the attackers could not know the content of the messages. Since the traffic through the system maintains a stable traffic pattern, each peer maintains in multiple ring. So there is no way to figure out who is the sender or the receiver even the hacker can monitor all the traffic. Because all the messages are mixed at each hop, and the messages are re-ordered before passing on, the hackers could not correlate the incoming and outgoing messages. Alteration at the network layer is prevented by integrity check at each hop, and altered messages will be dropped. A replay attack will re-send an incoming packet and watch for an outgoing packet. A duplicate will correlate the incoming and outgoing packet. In our system, messages are sent to successor in the ring through an unstable topology, replay could not expose the receiver, or the sender. Also, each peer passing messages have a threshold rate, so the DoS attack won't work.

The internal adversary can monitor all the communication between peers and in addition it is also trying to compromise the internal peers of the network. An adversary agent at such a compromised peer can gather information about messages that traverse the peer. The attack that is most likely to compromise the anonymity is the collusion attack. We categorize the major passive attacks [11] which a P2P anonymity protocol should be able to defend against, and also discuss why our system is invulnerable. At last we use Table 1 to show which cluster escape mechanisms are introduced to against passive attacks.

- Message coding attack[12]: An attacker can trace messages that do not change their coding during the session. We use common packet structure to repack the each anonymous packet passed by. Every packet will change its coding when it is rerouted.
- Local collaborating attack[13]: When local index peers create a mix ring, each peer on the ring will belong to different AS. Local collaboration attack could not gain anything.
- Timing attack[14]: We used spoofed source address, dynamic source and target port. No two continued packet can be judge to one data flow. Adversary could not make sure which packets belong to one data flow, so timing attack will be immune.
- Predecessor attack [15]: Because of ring structure, no node is the first responder. A malicious node could not gain the real address of its predecessor while spoofed source addresses are used.
- Traffic analysis attack[16] : As timing attack, adversary could not make sure which packets belong to one data flow. Hence, they could not analyze the traffic.
- Trace back attack[17]: The malicious node could not gain the real address of its predecessor while spoofed source addresses are used. There is no path to trace back.

Table 1. Common Passive Attack and Our Defense Method

Common Passive Attack	**Defense** Method
Message Coding Attack	Message repackage
Local Collaborating Attack	Topology
Timing Attack	Pass, Message repackage, Mix traffic, Ring Adjustment
Predecessor Attack	Topology, Pass, Message repackage, Ring Adjustment
Traffic Analysis Attack	Pass, Message repackage, Mix traffic, Ring Adjustment
Trace Back Attack	Topology, Pass, Message repackage, Ring Adjustment

5 Anonymity Evaluation

Suppose there are N peer nodes in system. The communication behavior of all the N peers looks alike under the ideal circumstance, so each node has a possibility of 1/N to be identified as the sender. The entropy can be calculated as:

$$H(X) = -\sum_{i=1}^{N} p_i \log_2 p_i = \log_2 N \tag{1}$$

However, it is unavoidable in a P2P system that malicious nodes exist. Suppose, M nodes of N are malicious. If we do not consider other special attack strategies, the entropy can be calculated as H(X) = log2(N - M). And the anonymity degree of the system with M malicious nodes is:

$$D(X) = H(X)/H_{Max}(X) = \log_2(N - M)/\log_2 N \tag{2}$$

Because of the specificity of system, suppose one unlimited node can spoof or pass R addresses and Section 4 show we can immune above passive attack. If a malicious peer joined the ring, it can gain the real address of its successor. The anonymity degree of system with M malicious nodes is:

$$D(X) = H(X)/H_{Max}(X) = \frac{\log_2((N - M)(RN - RM + M)/N)}{\log_2(\mathrm{R(N - M) + M})} \tag{3}$$

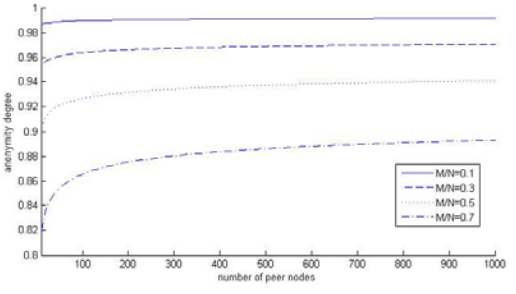


Fig. 4. Anonymity degree of system in different malicious nodes rate

We use MATLAB to evaluate four malicious nodes rate scenes while R is 255. The comparison result between these four malicious nodes rate is shown in Fig. 3. At the beginning, the anonymity degree rises rapidly with the number of nodes increasing, but when the number exceeds 100, the anonymity degree will keep rising, but rise slowly. The malicious node rate will directly affect the anonymity degree of our system, but the anonymity degree is still on high level.

6 Performance Analysis

As we approach, all messages sent to the ring of all participating peers, the performance could be a big problem. However, in order to defend against various attacks, including traffic analysis, message correlation and collaboration, more and more anonymous communication solutions use cover traffic, which means each participating node sends out noisy messages at a fixed rate. Comparing to the systems using cover traffic at the same transition rate, our system does not show any performance degrading, because in both kind systems, each node sends out mixed messages to its successor. Comparing to other P2P solutions, our topology is built to mimic the physical network connections and the local index peer will balance their clients' load, so the link usage and the latency in our system will be better than those systems which use the application layer topology directly. From the security point of view, at the same transfer rate, we send messages to the rings to enhance the anonymity, which could show better anonymity than the systems that only send messages to dedicated receivers, the rate of signal message over noisy message will be higher in our system.

Most current solutions only support sender anonymous communications. There are two solutions supporting sender- receiver anonymous: DC net requires a bus for all the participating nodes, which is not realistic, and Xor-tree requires nodes knowing all the other nodes. Our system adapts their ideas of bus messages to all the participants on the same ring to achieve anonymity. However, we apply the mix ring imitation and re-routing of the physical connection, it provides an inexpensive, scalable and efficient way of anonymity.

Encryption overhead in anonymous communication systems comprises a large proportion of the total cost. Mix in the anonymity of the best system for anonymous communication system, each node will be a mix public and private key encryption and decryption. Consider of balance between performance and anonymity, Tor's OP constructs circuits incrementally, negotiating a symmetric key with each OR on the circuit, one hop at a time. Our system only needs to use the asymmetric key encryption and decryption once during the key exchange period. In the communication process, sender or receiver use symmetric key encryption or decryption once while the relay nodes just decode the packets. Our system also can be fixed to meet the requirement of anonymous level by adjust the length of mix ring, the lower anonymous applications can gain lower latency.

7 Conclusion

In this paper, we present a enhance mix rings solution which provides sender- and sender-receiver anonymity. Our system is built on top of hybrid peer-to-peer network which is completely distributed, semi-self-organized, and scalable with thousands of active participants who may communicate simultaneously.

From our analysis on anonymity, we show that our system provides high level of anonymity while reducing the latency of data delivery and the link utilization. In performance side, we use asymmetric encryption in key exchange period and symmetric encryption in send and receive period. In relay period, we only repackage the packet without encryption. These will decrease the overhead compare with Tor. It means that our system not only resilient to various passive and active attacks from outsider nodes and insider peers, but also achieve high level anonymity in a cheap, scalable and efficient way.

References

1. Reiter, M.K., Rubin, A.D.: Anonymous Web transactions with crowds. Commun. Acm 42, 32–38 (1999)
2. Reed, M.G., Syverson, P.F., Goldschlag, D.M.: Anonymous connections and onion routing. IEEE J. Sel. Area Comm. 16, 482–494 (1998)
3. Jia, Z., Haixin, D., Wu, L., Jianping, W.: A light-weighted extension of anonymous communications in IPv6 Network. In: International Conference Green Circuits and Systems (ICGCS), pp. 404–408 (2010)
4. Burnside, M., Keromytis, A.D.: Low latency anonymity with mix rings. Information Security. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 32–45. Springer, Heidelberg (2006)
5. Chaum, D.L.: Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. Commun. Acm 24, 84–88 (1981)
6. Freedman, M.J., Sit, E., Cates, J., Morris, R.: Introducing Tarzan, a peer-to-peer anonymizing network layer. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 121–129. Springer, Heidelberg (2002)
7. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. J. Cryptol, 1, 65–75 (1988)
8. Dolev, S., Ostrobsky, R.: Xor-trees for efficient anonymous multicast and reception. ACM Transactions on Information and System Security (TISSEC) 3, 63–84 (2000)
9. Sherwood, R., Bhattacharjee, B., Srinivasan, A.: P5: A Protocol for Scalable Anonymous Communication. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, p. 58. IEEE Computer Society, Los Alamitos (2002)
10. Dingledine, R., Mathewson, N., Syverson, P.: Tor: the second-generation onion router. In: Proceedings of the 13th conference on USENIX Security Symposium, vol. 13, p. 21. USENIX Association, San Diego (2004)
11. Serjantov, A., Sewell, P.: Passive attack analysis for connection-based anonymity systems. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 116–131. Springer, Heidelberg (2003)

12. Berthold, O., Federrath, H., Köhntopp, M.: Project "anonymity and unobservability in the Internet. In: Proceedings of the tenth conference on Computers, freedom and privacy: challenging the assumptions, pp. 57–65. ACM, Toronto (2000)
13. Han, J., Liu, Y.: Rumor Riding: Anonymizing Unstructured Peer-to-Peer Systems. In: Proceedings of the 2006 IEEE International Conference on Network Protocols, pp. 22–31. IEEE Computer Society, Los Alamitos (2006)
14. Levine, B., Reiter, M., Wang, C., Wright, M.: Timing Attacks in Low-Latency Mix Systems. In: Juels, A. (ed.) FC 2004. LNCS, vol. 3110, pp. 251–265. Springer, Heidelberg (2004)
15. Wright, M.K., Adler, M., Levine, B.N., Shields, C.: The predecessor attack: An analysis of a threat to anonymous communications systems. ACM Transactions on Information and System Security (TISSEC) 7, 489–522 (2004)
16. Back, A., Möller, U., Stiglic, A.: Traffic Analysis Attacks and Trade-Offs in Anonymity Providing Systems. In: Moskowitz, I.S. (ed.) IH 2001. LNCS, vol. 2137, pp. 245–257. Springer, Heidelberg (2001)
17. Shields, C., Levine, B.N.: A protocol for anonymous communication over the Internet. In: Proceedings of the 7th ACM conference on Computer and communications security, pp. 33–42. ACM, Athens (2000)

Game-Based Distributed Resource Allocation in Horizontal Dynamic Cloud Federation Platform

Mohammad Mehedi Hassan, Biao Song, and Eui-Nam Huh

Dept. of Computer Engineering, Kyung Hee University
Global Campus, South Korea
{hassan,bsong,johnhuh}@khu.ac.kr
http://khu.ac.kr

Abstract. In this paper, we propose a game-theoretic solution to the problem of distributed resource allocation in emerging horizontal dynamic cloud federation (HDCF) platform. It differs from the existing vertical supply chain federation (VSCF) models in terms of establishing federation and dynamic pricing. We study two resource allocation games - non cooperative and cooperative games to analyze interaction among CPs in a HDCF environment. We use price-based resource allocation strategies and present both centralized and distributed algorithms to find optimal solutions which have low overhead and robust performance. Various simulations were carried out to validate and verify the effectiveness of the proposed resource allocation games. The simulation results demonstrate that a cost effective resource allocation with robust performance is achieved in the cooperative scheme.

Keywords: Horizontal dynamic cloud federation, vertical supply chain federation, distributed resource allocation, non-cooperative game, cooperative game.

1 Introduction

In recent years, horizontal dynamic cloud federation (HDCF) models are emerging [1] [2] [3] where various CPs (smaller, medium, and large) collaborate themselves dynamically to gain economies of scale and an enlargement of their virtual machine (VM) infrastructure capabilities (e.g. enlargement of Infrastructure-as-a-Service (IaaS) capability) to meet quality of service (QoS) targets of heterogeneous cloud service requirements without individually increasing the amount of virtual resources. It differs from existing vertical supply chain federation (VSCF) model [4] [5] [6], in which CPs leverage cloud services from other CPs for seamless provisioning and a priori agreements among the parties are needed to establish the federation [2].

In a HDCF environment, there are two types of participants: a buyer CP called primary CP (pCP) and a seller or cooperating CP called cCP. We consider the scenario for IaaS CPs. A IaaS CP could be at the same time both pCP and/or cCP. The pCPs initiate a HDCF platform and can pay cCPs for VM resource

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 194–205, 2011.

consumption to complete jobs. The Fig. 1 shows a formed horizontal dynamic cloud federation platform. The HDCF platform may dissolve as soon as the demand has been completed. We assume that all the CPs (pCPs and cCPS) are rational (self-interested and welfare maximizing) and they will refuse to offer their VM resources to each other unless they can recover their costs. So there is a need to define an effective VM resource allocation mechanism among IaaS CPs (pCPs and cCPs) having heterogeneous cost functions that motivates them to form the platform. Such a mechanism needs to be fair and ensures mutual benefits which is unexplored in previous works [1] [2] [3].

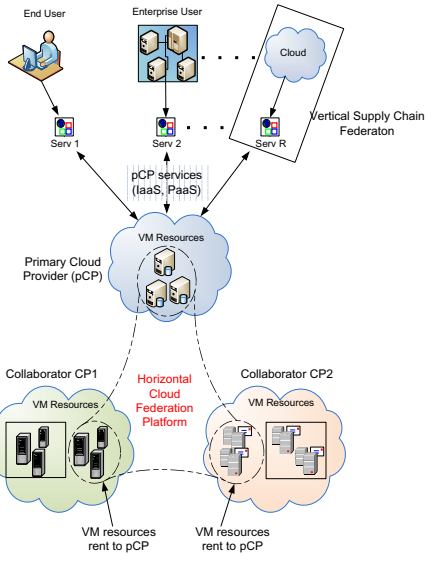


Fig. 1. A formed horizontal dynamic cloud federation Platform

Recently, game-theory based distributed resource allocation mechanisms have received a considerable amount of attention in different areas like grid computing [7] [8] [9], and also in cloud computing area [10] [11]. However, most of these works in cloud computing area mainly focus on optimal resource allocation using game theory in a single provider scenario except [12] where the authors focus on using a coalition game theory to find the profit share and the notion of diversity in a existing static cloud federation (PlanetLab) scenario. Besides, in computational grids environment, game theory is also used for resource allocation [7] [8] [9]. For example, L. He *et al* [7] proposed a coalition formation-based resource allocation mechanism using game theory. They used automated multi-party explicit negotiation for resource allocation. However, they did not evaluate the social welfare among the agents. Other approaches [8], [9] also have high computational complexity and difficult to implement. Hence, there is still need of a practicable

solution for resource allocation which effectively encourages CPs to participate in a HDCF platform.

In this paper, we analyze game theory based distributed resource allocation mechanisms for self-interested IaaS CPs that motivate them to participate in a HDCF platform. The objective of a distributed resource allocation game in HDCF platform is to maximize the utility of the system, defined as the sum of the buyer pCPs' utilities, without exceeding the resource capacity and expense price. We propose two resource allocation games - non-cooperative and cooperative games to analyze the utility of a pCP in a HDCF platform. We used price-based resource allocation strategies and develop both centralized and distributed algorithms for the games to achieve optimal solutions. These algorithms have low overhead and robust performance against dynamic pricing and stability. Various simulations were conducted to measure the effectiveness of these algorithms.

The paper is organized as follows: In section 2, we present the mathematical problem formulation. In Section 3, we describe the two resource allocation games in detail. In Section 4, we evaluate the effectiveness of the proposed resource allocation games in a HDCF environment and finally Section 5 concludes the paper.

2 Mathematical Problem Formulation

The notations used in the paper are summarized in Table 1. Consider a pCP requires VM resources with specific QoS requirements during a certain period t to continue providing services to its clients. A set of cCPs $P = \{P_i^t \,|i = 1...m\}$ is available during that period which can form a HDCF platform with pCP by providing VM resources with required QoS. Let R_{VM}^t be the total VM resources supplied in a HDCF platform in period t, r_i^t be the units of VM resource supplied by a CP i in period t, and $\tilde{C}_i^t$ be its maximum capacity in that period. The sum of the VM resources supplied to any pCP should be $\sum_{i=1}^{m} r_i^t = R_{VM}^t$. We know that the pCP can buy these VMs cheaper than the revenue it obtains for selling them to clients [13]. Now, we present various definitions used for the mathematic model formulation.

Definition 1 (Profit): The expected profit of a pCP obtained from executing tasks on R_{VM}^t resources from cCPs is defined as follows:

$$\text{Profit}_{pCP}^t(R_{VM}^t) = \text{Rev}_{cCP}^t(R_{VM}^t) - R_{VM}^t \cdot \underset{cCP}{\overset{t}{\text{Pr}}}(R_{VM}^t) \tag{1}$$

As shown in eq. (1), the total profit is determined by the total VM resource R_{VM}^t supplied in the HDCF platform and a pCP can only influence the value of R_{VM}^t by setting a proper price function $\text{Pr}_{cCP}^t(R_{VM}^t)$. So the pCP can strategically define the price function $\text{Pr}_{cCP}^t(R_{VM}^t)$ such a way that it can motivate available cCPs for contributing resources in a HDCF platform as well as get profit.

Table 1. Summary of Notations

Parameters	Description
R_{VM}^t	Total VM resources supplied in a HDCF platform in period t
$P = \{P_i^t \mid i = 1...m\}$	Total Number of Cloud providers present in period t
r_i^t	VM resource supplied by provider i in period t
$\tilde{C}_i^t$	Total VM capacity of provider i in period t
$Cost(r_i^t)$	Cost of supplying r_i^t unit of VM resource by provider i in period t
M_i^t	Cost of the first unit of VM resource by provider i in period t
α_i	Learning factor of provider i where $0.75 < \alpha_i < 0.9$
ω	Parameter defining the rate of revenue in a HDCF platform
$\mathrm{Rev}_{cCP}^t(R_{VM}^t)$	Revenue function estimated by a pCP for cCPs in period t
$\mathrm{Pr}_{cCP}^t(R_{VM}^t)$	Price per hour given to cCPs by a pCP for each unit of VM resource supplied in period t
$Util(r_i^t)$	Utility of any cCP i by providing r_i^t unit of VM resources in period t

Definition 2 (Cost Function): In order to supply r_i^t units of VM resource, any cCP i has to pay $Cost({r_i}^t)$, which is defined as follows [14]:

$$Cost(r_i^t) = \frac{M_i^t \cdot {r_i^t}^{1+\log_2 \alpha_i}}{1 + \log_2 \alpha_i}, \quad 0 \leq r_i^t \leq \tilde{C}_i^t \tag{2}$$

The cost function can be heterogeneous for different cCPs based on α and M. The higher the value of α and M, the higher the production cost for a provider.

Definition 3 (Revenue Function): *Let ω be the increasing rate of revenue. Now, a pCP can estimate the revenue function of the HDCF platform as follows*:

$$\mathrm{Rev}_{cCP}^t(R_{VM}^t) = \frac{M \cdot \left(1 - e^{-R_{VM}^t \cdot \omega}\right)}{\omega} \tag{3}$$

The function $\mathrm{Rev}_{cCP}^t(R_{VM}^t)$ is a non-decreasing and concave function which means that the more resources supplied by cCPs, the higher the revenue. However, the marginal revenue is decreasing as the resource increased.

Definition 4 (Price Function): *Based on the revenue function of eq. (3), the price per hour given to cCPs by a pCP for each unit of VM resource supplied in period t is defined as follows*:

$$\underset{cCP}{\overset{t}{\mathrm{Pr}}}(R_{VM}^t) = M \cdot e^{-R_{VM}^t \cdot \omega} \tag{4}$$

$$s.t. \quad R^t_{VM} > 0 \tag{5}$$

The function $\Pr^t_{cCP}(R^t_{VM})$ is the marginal gain of the HDCF platform. When the amount of VM resource increase, the price per hour of each unit of VM resource decrease. Also this function represents the *proportional fairness* of contributing resources by cCPs.

3 Resource Allocation Games in a HDCF Platform

In this section, we study two resource allocation games in a HDCF platform. These games are repeated and asynchronous games. In one game, cCPs can supply R^t_{VM} resources to a pCP in non cooperative manner and in the other game they supply VM resources in cooperative manner. In both games, a pCP strategically define a price function $\Pr^t_{cCP}(R^t_{VM})$ and publicizes it along with the total amount of VM resources supplied. Each cCP only knows these information and can update its own strategy in each move so to maximize its utility.

3.1 Non-cooperative Resource Allocation Game

In the non-cooperative game of resource allocation, the cCPs make decision to maximize their own utilities regardless of other cCPs. They choose r^t_i based on the public information: the aggregated VM resource R^t_{VM} and the price function $\Pr^t_{cCP}(R^t_{VM})$. Formally, cCP i needs to perform:

$$Max\ Util(r^t_i) = r^t_i \cdot \Pr^t_{cCP}(R^t_{VM}) - Cost(r^t_i) \tag{6}$$

$$s.\ t. \qquad 0 \le r^t_i \le \tilde{C}^t_i \tag{7}$$

If the net utility of a cCP is less than or equal to zero, it will not participate in the game, and it will be removed from the list of cCPs. Note that R^t_{VM} implicitly depends on r^t_i. If the value of r^t_i is changed, the value of R^t_{VM}, as well as $\Pr^t_{cCP}(R^t_{VM})$, will be changed accordingly. Thus, in the optimization, the value of R^t_{VM} would be better presented in terms of r^t_i. Let r^t_{-i} be the amount of VM resource collectively supplied by the cCPs except cCP i, then $r^t_{-i} = R^{t'}_{VM} - r^{t'}_i$, where $R^{t'}_{VM}$ and $r^{t'}_i$ are the total amount of VM resource and the amount of VM resource supplied by cCP i respectively in the previous round. The equivalent optimization problem of eq. (6) can be re-written by using eqs. (4) and (2) as follows:

$$Max\ Util(r^t_i) = r^t_i \cdot M \cdot e^{-(r^t_i + r^t_{-i})\cdot\omega} - \frac{M^t_i \cdot {r^t_i}^{1+\log_2\alpha_i}}{1+\log_2\alpha_i} \tag{8}$$

$$s.\ t. \qquad 0 \le r^t_i \le \tilde{C}^t_i \tag{9}$$

Now to obtain the optimal value of r^t_i) of the game, we take the derivative of eq. (8) with respect to r^t_i as follows:

$$Util'(r^t_i) = M \cdot e^{-(r^t_i + r^t_{-i})\cdot\omega} - r^t_i \cdot M \cdot \omega \cdot e^{-(r^t_i + r^t_{-i})\cdot\omega} - M^t_i \cdot {r^t_i}^{\log_2\alpha_i} = 0 \tag{10}$$

Now from eq. (10), it is difficult to find the close form solution of optimal r_i^t. We can use direct search method with linear constraint like pattern search method [15] with multiple initial guesses to the optimal VM quantity r_i^t.

However, for a defined price function, the Nash equilibrium of this non-cooperative game may not be *unique* as the order of move will influence the equilibrium point. The game may converge to different equilibrium, depending on the sequence of move of the cCPs. So the non-cooperative game is not desired in maximizing a pCP's profit because it does not lead to a unique Nash equilibrium.

3.2 Cooperative Resource Allocation Game

In the cooperative game of resource allocation, we jointly consider the benefits of both the pCP and cCPs. Being a pCP, the objective is to maximize its total profit in the HDCF platform. However, for a defined price function $\Pr_{cCP}^t(R_{VM}^t)$, the system may converge to different equilibrium, depending on the sequence of move of the cCPs. There is no guarantee for the pCP to set a particular marginal pricing function that leads to a desirable outcome which maximizes its total profit. So a pCP can set a initial constant price and can choose a proper price Pr_{cCP}^t to maximize its total profit.

For cCPs, they try to maximize their benefits based on r_i^t and the initial constant price Pr_{cCP}^t. So the objective of any cCP i is defined as follows:

$$Max\ Util(r_i^t) = r_i^t \cdot Pr_{cCP}^t - Cost(r_i^t) \tag{11}$$

$$s.t. \qquad 0 \leq r_i^t \leq \tilde{C}_i^t \tag{12}$$

Since there is a boundary constraint for the variable r_i^t, that is, $0 \leq r_i^t \leq \tilde{C}_i^t$, the problem in eq. (11) can be formulated as a constrained optimization, which can be solved by the method of Lagrangian Multiplier.

$$L = Util(r_i^t) - \sum_{i=1}^{m} \gamma r_i^t + \sum_{i=1}^{m} \varphi(r_i^t - \tilde{C}_i^t) \tag{13}$$

where γ and φ are the Lagrangian constant. The Karush Kuhn Tucker (KKT) condition is as follows:

$$\frac{\partial L}{\partial r_i^t} = Util'(r_i^t) - \gamma + \varphi = 0, \quad i = 1, ..., m \tag{14}$$

$$Util'(r_i^t) = M_i^t \cdot {r_i^t}^{\log_2 \alpha_i} = Pr_{cCP}^t \tag{15}$$

$$r_i^t = \left[\frac{M_i^t}{Pr_{cCP}^t} \right]^{-\frac{1}{\log_2 \alpha_i}} \tag{16}$$

By solving r_i^t of eq. (15), we can obtain the solution of a cCP's optimization problem of choosing r_i^{t*} as follows:

$$r_i^{t*} = \begin{cases} r_i^t, & 0 \le r_i^t \le \tilde{C}_i^t, \\ \tilde{C}_i^t, & r_i^t > \tilde{C}_i^t, \\ 0, & r_i^t \le 0 \end{cases} \tag{17}$$

Thus, given the value of proper price $\Pr_{cCP}^t$, a pCP can predict the total VM resource R_{VM} contributed to the system, that is

$$R_{VM}^{t*} = \sum_{i=1}^{m} r_i^{t*} \tag{18}$$

Now, let's consider the optimization problem of a pCP. If the pCP knows the parameters M and α of all the cCPs, it can formulate its own maximization, which aims at maximizing the total profit with respect to R_{VM}^{t*}

$$Max\ \text{Profit}_{pCP}^t(R_{VM}^{t*}) = \text{Rev}_{cCP}^t(R_{VM}^{t*}) - R_{VM}^{t*} \cdot \overset{t}{\underset{cCP}{\text{Pr}}} \tag{19}$$

$$s.t. \quad \overset{t}{\underset{cCP}{\text{Pr}}} \ge 0, \quad 0 \le r_i^t \le \tilde{C}_i^t \tag{20}$$

Since the total amount of VM resource R_{VM}^t solely depends on the value of price $\Pr_{cCP}^t$ through eqs. (17 and (18)), one can rewrite the objective function by substituting R_{VM}^t in terms of $\Pr_{cCP}^t$ as follows:

$$\begin{aligned} &Max\ \Pr ofit_{pCP}^t(\overset{t}{\underset{cCP}{\text{Pr}}}) \\ &= \text{Rev}_{cCP}^t \left(\sum_{i=1}^{m} \left[\frac{M_i^t}{\Pr_{cCP}^t} \right]^{-\frac{1}{\log_2 \alpha_i}} \right) - \left(\sum_{i=1}^{m} \left[\frac{M_i^t}{\Pr_{cCP}^t} \right]^{-\frac{1}{\log_2 \alpha_i}} \right) \cdot \overset{t}{\underset{cCP}{\text{Pr}}} \end{aligned} \tag{21}$$

$$= \frac{M \cdot \left(1 - e^{-\sum_{i=1}^{m} \left[\frac{M_i^t}{\Pr_{cCP}^t} \right]^{-\frac{1}{\log_2 \alpha_i}} \cdot \omega} \right)}{\omega} - \left(\sum_{i=1}^{m} \left[\frac{M_i^t}{\Pr_{cCP}^t} \right]^{-\frac{1}{\log_2 \alpha_i}} \right) \cdot \overset{t}{\underset{cCP}{\text{Pr}}} \tag{22}$$

$$s.t. \quad 0 \le r_i^t \le \tilde{C}_i^t \tag{23}$$

Although it is difficult to find the close-form solution of $\Pr_{cCP}^{t*}$ for eq. (22), we can solve this optimization efficiently using direct search method, for example, pattern search method is applied to solve the value of the optimal price $\Pr_{cCP}^{t*}$. Once the pCP finds the optimal price, it can calculate the value of all r_i^{t*} using eq. (17). However, it may happen that the boundary constraints in eq. (23) may violate and in that case the problem becomes more complicated. Still we can find the solution mathematically using Lagrangian multiplier. Without the

constraints it can be shown that the objective function in eq. (22) is a concave function. So there exists a unique solution that satisfies the KKT-conditions of eq. (22) as follows:

$$L = \Pr ofit^t_{pCP}(\underset{cCP}{\overset{t}{\Pr}}) - \sum_{i=1}^{m} \gamma r_i^t + \sum_{i=1}^{m} \varphi(r_i^t - \tilde{C}_i^t) \tag{24}$$

$$\frac{\partial L}{\partial \Pr^t_{cCP}} = \frac{\partial \left[\Pr ofit^t_{pCP}(\Pr^t_{cCP})\right]}{\partial \Pr^t_{cCP}} - \frac{\partial \left[\sum_{i=1}^{m} \gamma r_i^t\right]}{\partial \Pr^t_{cCP}} + \frac{\partial \left[\sum_{i=1}^{m} \varphi(r_i^t - \tilde{C}_i^t)\right]}{\partial \Pr^t_{cCP}} = 0 \tag{25}$$

$$\gamma \geqslant 0, \quad \varphi \geqslant 0, \ \gamma r_i^t = 0, \ \varphi(r_i^t - \tilde{C}_i^t) = 0, \ 0 \leqslant r_i^t \leqslant \tilde{C}_i^t, \ i = 1, ..., m \tag{26}$$

From the KKT conditions, if $\gamma = 0, \quad and \ \varphi = 0$, then all the r_i^t lie between $[0, \tilde{C}_i^t]$. When the boundary constraints are violated ($\gamma \neq 0, \quad or \ \varphi \neq 0$), the value of r_i^t are forced to be the boundary value [either 0 *or* $\tilde{C}_i^t$]. If r_i^t is less than or equal to zero for certain cCP i, we are sure that the cCP is not eligible for contributing as the cost of supplying the VM is comparatively high. Similarly, if r_i^t is greater than $\tilde{C}_i^t$, we are sure that the cCP has optimal value of r_i^t. This cCP should provide as much VM resource as possible since the cost is comparatively low. Thus, we can eliminate some cCPs, whose value of r_i^t is known already, from the problem formulation and resolve the r_i^t for the remaining cCPs.

Until now, we assume the pCP knows the characteristic of the cost function of each cCP such that it can determine the behavior of the cCPs, and it can construct its own objective function. However, in distributed environment, the pCP can only observe the action of each cCP by setting a probing price. The cCPs choose the best r_i^t to maximize their net utility. The pCP keeps adjusting the price gradually until a desirable profit is obtained. Now we present a distributed algorithm to find the optimal value of $\Pr^t_{cCP}$.

The algorithm is described step by step as follows:

Step 1: Initialize the probing price $\Pr^t_{cCP} = 0.1$ and $\{r_i^t\}_{i=1}^{P} = 0$.

Step 2: Send $\Pr^t_{cCP} = 0.1$ to all cCPs and receive corresponding $R^t_{VM} = \sum_{i=1}^{P} r_i^t$.

Step 3: if $\text{Profit}^t_{pCP}(R^{t*}_{VM}) = \text{Rev}^t_{cCP}(R^{t*}_{VM}) - R^{t*}_{VM} \cdot \underset{cCP}{\overset{t}{\Pr}}$ is maximized or $\frac{\partial \left[\Pr ofit^t_{pCP}(\Pr^t_{cCP})\right]}{\partial \Pr^t_{cCP}} = 0$, the optimal $\Pr^t_{cCP}$ is found and break. Otherwise update $\Pr^t_{cCP}$ based on old price and the percentage change of the net profit.

Step 4: If $0 \leq r_i^t \leq \tilde{C}_i^t$ for all $i \in P$, then break.

Step 5: Now for some cCP, $i \in P$, If $r_i^t \leq 0$, remove those cCPs from the list of P. Also for some cCP, $i \in P$, If $r_i^t \geq \tilde{C}_i^t$, set $r_i^t = \tilde{C}_i^t$.

4 Simulation and Discussion

In this section, we focus on evaluating the effectiveness of the proposed resource allocation games in a HDCF platform. We focus on the case of one pCP and six

cCPs in a HDCF platform where at anytime any CP can be a pCP and/or cCP. Both the non-cooperative and cooperative games, the performance measures are the social welfare, total profit, cost effectiveness and scalability. The experimental parameters are shown in Table 2. Using Amazon (EC2) as the example, we assume the range of production cost of first unit varies from 2$/hr to 3$/hr and service availability for all provider is 99.95% [14]. For simplicity, we consider each cCP has almost same amount of VM resource capacity in period t. Also the evaluation of two resource allocation games were done based on mathematical simulation, which was implemented in MATLAB 7.0.

Table 2. Parameters Used in the Resource Allocation Games for cCPs

cCPs i	Production cost of first unit per hr M_i^t	Learning factor α_i	Total capacity $\tilde{C}_i^t$
1	2.8	0.79	300
2	2.7	0.84	302
3	2.0	0.83	305
4	2.3	0.80	304
5	2.9	0.78	303
6	2.4	0.78	301

4.1 Convergence of the Resource Allocation Games

Convergence is a basic requirement that the resource allocated by the cCPs should converge in each game. In all the experiments, we consider $\omega = 0.01$, $M = 3$ and $\xi = 0.3$. We analyze the behavior of each game based on the VM resource supplied at the steady state. Fig. 2 depicts the quantity of VM resource supplied by the six cCPs in each iteration of two games. It demonstrates that the resource allocation games converge to a steady state after a number of iterations. As shown in the graph, the non-cooperative game converges fast, while it takes more iterations for the cooperative game to stabilize. However, the converging speed does not affect the performance of the games.

4.2 Performance Analysis of Resource Allocation Games

In this subsection, we first evaluate the total profit in the proposed resource allocation games. Fig. 3 plots the total profit of the pCP in the two resource allocation games. We can see that the cooperative game generated the highest total profit (204) as compared to the non-cooperative game (140).

Now we evaluate the social welfare in the resource allocation games. Fig. 4 shows the social welfare in the two resource allocation games. It can be seen that the social welfare achieved by the non-cooperative game is much higher (135) as compared to the cooperative game (81). The reason is that the cooperative game is designed to maximize the total profit in a HDCF platform by trading off the

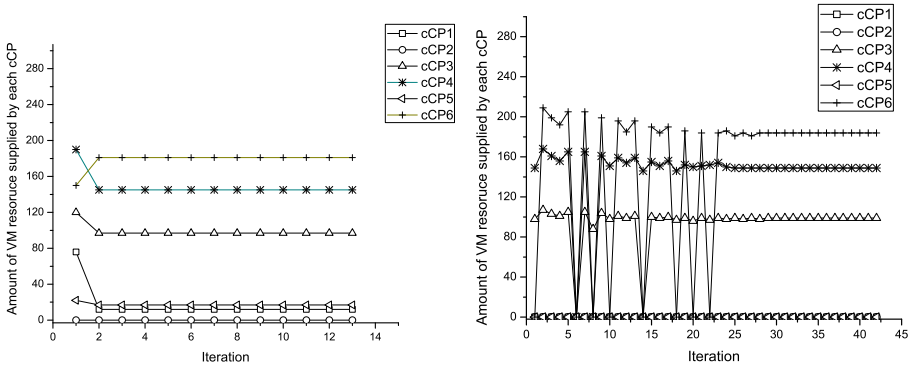


Fig. 2. VM resources supplied by each cCP in (a) non-cooperative resource allocation game and (b) cooperative resource allocation game

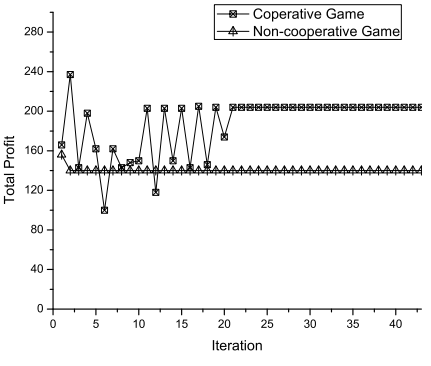


Fig. 3. Total profit in each resource allocation game

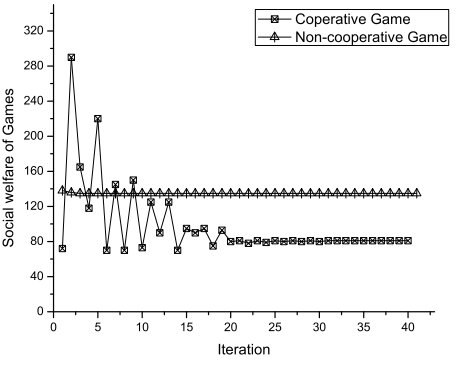


Fig. 4. Social welfare of cCPs in each resource allocation game

social welfare. Now the key discussion here is which approach, the cooperative or the non-cooperative is better. To compare the performance of these games, we can use the total utility achieved in a HDCF platform that is the sum of the total profit and social welfare. For this, we evaluate the total utility in the two games having different revenue function ω as shown in Fig. 5. Note that larger the ω, the higher the revenue is for the same quantity of VM resource. It can be seen that the cooperative game performs well as compared to the non-cooperative game. Also the cooperative game is cost effective as few low-cost cCPs provide more VM resources (see Fig. 2 (b)). Hence, we conclude that the cooperative game provides a cost-effective resource supply to a HDCF platform and thus admits the best set of cCPs to participate.

Also to evaluate the effect of HDCF system size (scalability) in two games, we vary the number of cCPs in the HDCF system from 6 to 24 for ω value 0.01. Note that the small ω implies more quantity of VM resource is required

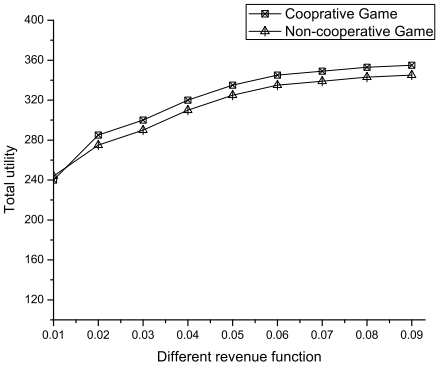


Fig. 5. Total utility in a HDCF platform for each resource allocation game under different revenue function

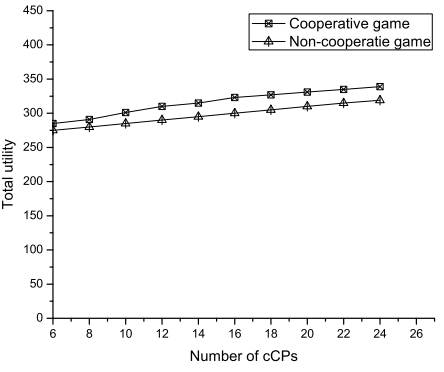


Fig. 6. Performance of two resource allocation games in terms of total utility with different number of cCPs

to obtain the same amount of revenue. And large number of cCPs means they can supply VM resource with less cost. The result is shown in Fig. 6. We can see that in cooperative game the total utility increases with the number of cCPs as compared to the non-cooperative game. Also in steady state, only the cCPs that provide cost-effective VMs are remained in cooperative game.

5 Conclusions

In this paper, we analyze game theory based optimal VM resource allocation mechanisms in a HDCF environment. We propose two resource allocation games-non-cooperative and cooperative. It is shown that desirable outcome (e.g. total utility, cost effectiveness etc.) cannot be achieved under a non-cooperative environment. Under the cooperative resource allocation game, the CPs have a strong motivation to contribute VM resources. Also, this game is cost-effective and scalable as only the collaborators with low-cost participate in a HDCF platform. In future, we will study the performances of these games in a simulated environment where hundreds of clouds will dynamically join and leave the federation.

Acknowledgments. This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency)" (NIPA-2011-(C1090-1121-0003)

References

1. Bittman, T.: The evolution of the cloud computing market. Gartner Blog Network (November 2008), http://blogs.gartner.com/thomasbittman/2008/11/03/theevolution-of-the-cloud-computing-market/

2. Celesti, A., Tusa, F., Villari, M., Puliafito, A.: How to enhance cloud architectures to enable cross-federation. In: IEEE Intl. Conf. on Cloud Computing, pp. 337–345 (2010)
3. Dodda, R.T., Smith, C., Moorsel, A.: An architecture for cross-cloud system management. In: Ranka, S., Aluru, S., Buyya, R., Chung, Y.-C., Dua, S., Grama, A., Gupta, S.K.S., Kumar, R., Phoha, V.V. (eds.) IC3 2009. Communications in Computer and Information Science, vol. 40, pp. 556–567. Springer, Heidelberg (2009)
4. Rochwerger, B., Breitgand: The reservoir model and architecture for open federated cloud computing. IBM J. Res. Dev. 53, 535–545 (2009)
5. Buyya, R., Ranjan, R., Calheiros, R.: InterCloud: Utility-oriented federation of cloud computing environments for scaling of application services. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010. LNCS, vol. 6081, pp. 13–31. Springer, Heidelberg (2010)
6. Maximilien, E.M., Ranabahu, A., Engehausen, R., Anderson, L.: Ibm altocumulus: a cross-cloud middleware and platform. In: Proc. of the 24th ACM SIGPLAN Conference OOPSLA 2009, pp. 805–806 (2009)
7. He, L., Ioerger, T.R.: Forming resource-sharing coalitions: a distributed resource allocation mechanism for self-interested agents in computational grids. In: Proc. of the 2005 ACM Symposium on Applied Computing (SAC 2005), pp. 84–91 (2005)
8. Carroll, T.E., Grosu, D.: Formation of virtual organizations in grids: a game-theoretic approach. Concurr. Comput.: Pract. Exper. 22, 1972–1989 (2010)
9. Khan, S.U., Ahmad, I.: Non-cooperative, semi-cooperative, and cooperative games-based grid resource allocation. In: Proc. of the 20th Intl. Conf. on Parallel and Distributed Processing (IPDPS 2006), p. 121 (2006)
10. Wei, G., Vasilakos, A.V., Yao, Z., Xiong, N.: A game-theoretic method of fair resource allocation for cloud computing services. J. Supercomput. 54, 252–269 (2010)
11. Jalaparti, V., Nguyen, G.D., Gupta, I., Caesar, M.: Cloud Resource Allocation Games. Technical Report, University of Illinois (2010), http://hdl.handle.net/2142/17427
12. Antoniadis, P., Fdida, S., Friedman, T., Misra, V.: Federation of virtualized infrastructures: sharing the value of diversity. In: Proc. of the 6th Int. Conf. Co-NEXT 2010, pp. 12:1–12:12. ACM, New York (2010)
13. Goiri, I., Guitart, J., Torres, J.: Characterizing cloud federation for enhancing providers' profit. In: IEEE Intl. Conf. on Cloud Computing, pp. 123–130 (2010)
14. Amit, G., Xia, C.H.: Learning Curves and Stochastic Models for Pricing and Provisioning Cloud Computing Services. Service Science 3, 99–109 (2011)
15. Kolda, T.G., Lewis, R.M., Torczon, V.: Optimization by direct search: New perspectives on some classical and modern methods. SIAM Review 45, 385–482 (2003)

Stream Management within the CloudMiner

Yuzhang Han[1], Peter Brezany[1], and Andrzej Goscinski[2]

[1] Department of Scientific Computing, Faculty of Computer Science, University of Vienna, Vienna, Austria
{han,brezany}@par.univie.ac.at
[2] School of Information Technology, Deakin University, Geelong, Australia
ang@deakin.edu.au

Abstract. Nowadays cloud computing has become a major trend that enterprises and research organizations are pursuing with increasing zest. A potentially important application area for clouds is data analytics. In our previous publication, we introduced a novel cloud infrastructure, the CloudMiner, which facilitates data mining on massive scientific data. By providing a cloud platform which hosts data mining cloud services following the Software as a Service (SaaS) paradigm, CloudMiner offers the capability for realizing cloud-based data mining tasks upon traditional distributed databases and other dataset types. However, little attention has been paid to the issue of data stream management on the cloud so far. We have noticed the fact that some features of the cloud meet very well the requirements of data stream management. Consequently, we developed an innovative software framework, called the StreamMiner, which is introduced in this paper. It serves as an extension to the CloudMiner for facilitating, in particular, real-world data stream management and analysis using cloud services. In addition, we also introduce our tentative implementation of the framework. Finally, we present and discuss the first experimental performance results achieved with the first StreamMiner prototype.

Keywords: data analytics, data stream, sensor, service architecture, service data flow, StreamMiner, CloudMiner.

1 Introduction

Cloud computing is a computing paradigm in which high throughput/performance IT-related services are provided on demand via the Internet to multiple customers on the pay as you use basis [1]. To combine cloud technology with data mining, we proposed CloudMiner - a cloud-based infrastructure designed to support on-demand data mining related to traditional business analytics and novel e-Science analytics [2]. The main functions of CloudMiner are summarized as follows [3]: (i) it establishes a powerful and flexible cloud environment for executing data mining tasks via cloud services; and (ii) it simplifies service development, deployment and publishing.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 206–217, 2011.

The first CloudMiner infrastructure prototype operates on traditional databases scattered in the cloud. Nevertheless, myriad problems in business and e-Science domains deal with real-time dynamic data, in other words, data streams. Hence, it is tempting to enhance CloudMiner with the capability of data stream management and analysis. The following paragraphs of this section analyze the primary problems faced by general stream management applications as well as the potential of solving these problems using cloud computing.

1.1 Challenges of Data Stream Management Applications

A data stream is a sequence of data packets continuously sent from one device to another during a certain period of time. The concept *data stream management* refers to all the activities that are related to the construction, transmission, persistence, and processing of data streams. These activities could bring about intricate issues. By nature, most data stream management applications face the challenges which can be characterized as follows.

They require huge computing power: data stream-oriented applications differ from permanent-data applications in that they deal with data packets available only for a rather limited time slot. Hence, the packet processing devices of such applications must be guaranteed with professional-level computing power, particularly when the processing procedures are complex - like those in genetics or multimedia production. An apt example would be a Web TV application: a millisecond of delay in every packet could jeopardize the program's overall performance and distort fluent live broadcast into discontinuous frames.

A single stream might be processed by multiple users on different locations in differing ways: a single stream can yield different results if used by different users cherishing different intentions. Thus, geographically dispersed users would wish to establish a virtual organization in which they all have access to the same source streams as well as the processing results of others. A striking example refers to the Japan nuclear crisis in 2011. As the situation at the nuclear reactors deteriorated, several organizations and myriad scientists in different countries were closely focusing on the updates of environmental data streams gathered from the reactor surroundings. On this occasion, a virtual organization could, on the one hand, help them effectively share the raw streams, and on the other hand allow them to access, validate, and practice further analysis upon the results of their colleagues.

1.2 Cloud-Enabled Stream Processing

Taking into account the foregoing challenges, we propose to use the cloud technology to response them. The benefits of combining streams and clouds are analyzed below.

Clouds provide ample computing power: clouds provide users a virtual computer with huge and scalable computing power. This responds precisely to stream processing's need for compute capability. Besides, the user also benefits from lowered hardware costs by running her applications on the cloud.

Clouds allow every user to develop distinct applications: as a matter of fact, every user of a cloud, disregarding her physical location, can deploy her unique cloud service to address her own need. Therefore, people can develop different stream-based applications as cloud services, which access the single source stream as well as output streams from other services. Under this circumstance, by developing cloud services, every individual concerned with the nuclear reactor crisis, as mentioned in Subsection 1.1, can do real-time processing on the streams either coming directly from the reactor environment or generated by other observers' services.

Moreover, every cloud service can easily invoke other services to run their program logic as a part of itself. Thanks to this feature, different stream-oriented applications - encapsulated in cloud services - can share the software tools which are frequently used in stream processing, such as the Fourier transform and many data mining tools. Finally, the cloud also simplifies stream transmissions among applications. As a cloud is a space of full connectivity, any data including streams can be efficiently transferred among cloud services.

Propelled by these benefits, we propose the cloud-based software framework, the StreamMiner, which is composed of a dynamic set of interconnected cloud services. These services extend the CloudMiner in the direction of stream-oriented functionalities. The relation between the StreamMiner and the CloudMiner illustrated in Fig. 1 is as follows: (i) the CloudMiner (upper left in the figure) consists of the ServiceCloud and DataCloud; (ii) the StreamMiner includes a group of services (the light-colored boxes) located in the ServiceCloud; and (iii) while StreamMiner services utilize existing services (the dark-colored boxes) in the ServiceCloud, both kinds of services communicate with services in the DataCloud to access data.

1.3 Contributions and Organization of the Paper

The main contributions of the paper can be characterized as follows. First, to help users to manage and analyze streams new cloud-based software framework, called the StreamMiner, which is composed of a dynamic set of interconnected cloud services, is proposed and elaborated. Second, StreamMiner services are proposed and the programming language-independent description of each type of cloud service that is included the framework is formulated. Third, a proof of concept in the form of its prototype application developed is demonstrated. It is a public cloud dedicated to data stream sharing and analysis. Fourth, a proof of performance of the StreamMiner is presented based on the experiments carried out on the implemented platform.

The rest of this paper is organized as follows. Section 2 expatiates on the StreamMiner framework, its operation model and workflows, as well as the cloud services it provides; Section 3 describes the prototype application of the framework together with the technologies underlying the prototype; Section 4 demonstrates the experimental testing of the performance of the prototype. Moreover, Section 5 discusses the related work, and finally, Section 6 concludes with a vision of the future research.

2 StreamMiner Framework

2.1 Architecture

The StreamMiner allows the user to define hierarchical complex cloud-based applications, which process sensor-produced data streams. Aiming at a particular task, each application is essentially a network of interconnected cloud services, denoted as a *production flow*. The lower right part of Fig. 1 illustrates a work scenario of the StreamMiner. It can be recognized in the figure that there are four types of cloud services forming a single production flow. The names and functions of these services are as follows.

Management service (MS): one instance of the StreamMiner only employs a single management service (the *MS* in the case of Fig. 1), which serves as a bridge between the user and the world of data streams. All users and other services must register to it, so that they become visible to one another. The management service organizes production flows of services according to the user's need.

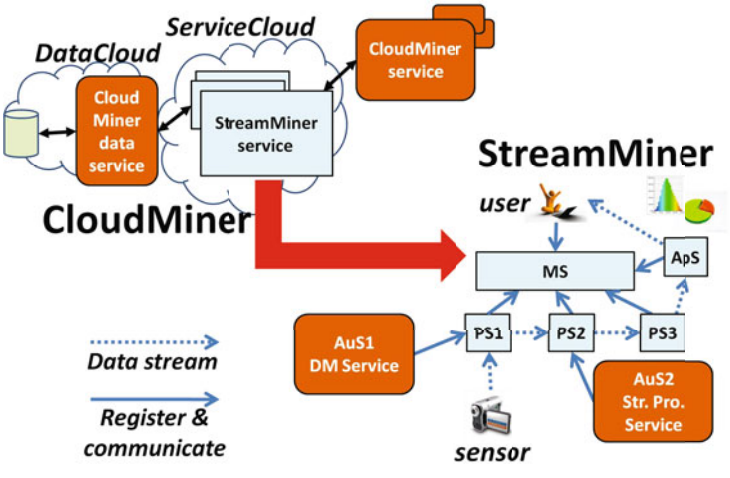


Fig. 1. The CloudMiner and StreamMiner

*Production service (*PS*):* a StreamMiner application can have one or more *PSs* (*PS1* to *PS3* in Fig. 1). The production services account for the productive part of StreamMiner and undertake the major activity of production flows stream processing. The particular processing task of a *PS* is defined by the user. During its operation, each *PS* typically receives the stream produced by other *PSs*, processes it, and outputs the resulting stream to other services. Under the coordination of the management service, multiple *PSs* - as long as they are registered - can form a production flow to practice complex tasks of data stream management and analysis. In the production flow, each *PS* assumes one sub-task of the entire production flow process.

Auxiliary service (AuS): a StreamMiner implementation can contain any number of auxiliary services (in Fig. 1, *AuS1* and *AuS2*), which are computing services assisting *PSs* to complete their tasks. After registering to a certain *PS*, a group of *AuSs* expose libraries of algorithms and processing functions to the *PS*. The owner *PS*, in turn, can invoke its *AuSs* to use their programs during stream processing. This relationship is reflected in the figure that *AuS1* provides certain data mining libraries to *PS1*, for instance, for clustering and classification; while *AuS2* offers certain stream processing operations to *PS2*, such as data buffering and filtering.

Application service (ApS): a StreamMiner environment also contains one or more application services (in Fig. 1 the *ApS*) each of which is connected to certain production services and users. Typically developed by the end user, an *ApS* provides the program logic that analyzes and utilizes the data stream produced by the *PS(s)* of a certain or more than one production flow. The utilization includes storing, visualization, and knowledge exploration upon the incoming data stream. Only those application services, production services, and users that are registered to the management service can view one another and cooperate. In the figure, *ApS* receives the stream produced by *PS3* and generates meaningful visualization of the stream such as pie charts and histograms for the user.

2.2 StreamMiner Services

The StreamMiner framework is a systematic definition of a complex mechanism, which can be implemented to build cloud-based stream-oriented applications. To provide a guide for implementing the framework, we formulate the programming language-independent description of each type of cloud service enumerated in Section 2.1. Such description sheds light on (i) the general characteristics of each service type, and (ii) the details of remote methods this type of service should provide.

As far as the management service is concerned, this type of service functions as the single organizer of production flows. As such, it must assume the task of user/service registration. Thus, a valid implementation of the management service must include the following remote method to carry out user registration:

Integer registerUS (String username, String password)

Called by the user, the method uploads the user's private data via the two arguments, and returns her an integer for user ID that is unique in the scope of the current instance of management service. Another central task of the management service is production flow construction. To assume this task, a service implementation must provide the following method:

Integer setupPF (Integer[] service_id, String[] desps)

This method is called by the user to indicate which services - represented by their service IDs - are to be employed in the to-be-established production flow, and how they should be connected with one another - specified by the argument *desps*. It generates and returns a unique ID for the established production flow.

Besides, the production service is another major service type. For this type of service, stream transmission is a vital function. Two approaches are defined for stream transmission among *PSs*: one is transmission per packet, another is transmission via connection. One can choose either of them to implement depending on her particular hardware and software condition.

The packet-oriented approach only applies to the situation that the sending and receiving *PSs* reside in the same cloud which features high internal transmission rate. To realize this approach, the method

Boolean receivePacket (String timestamp, String value)

must be implemented by the receiving *PS*, and then be called by the sending *PS*. In the view of the StreamMiner, receiving/processing/sending a data stream is decomposed into periodical packet receiving/processing/sending operations executed throughout a certain time interval. Once called, the method *receivePacket*, on the one hand, transmits the packet value and timestamp of the packet; on the other hand, it notifies the receiver that a packet is to be received.

By contrast, the connection-oriented approach functions in a way that the sender and receiver located in the same cloud establish a virtual connection for transmission. The advantage of this approach is that it does not require a service invocation for every packet sending. Instead, the sender calls the following method implemented on the receiver only once for transferring an entire stream:

Boolean receiveStream (Integer cache_size)

After this method is called, the transmission is performed discontinuously if no disturbance occurs, and before the method returns, both partners of the transmission retain a connection. The argument cache_size is used to specify the size of cache the receiver should prepare for stream reception.

Apart from the foregoing ones, the StreamMiner also defines tens of other service methods for miscellaneous uses. They will be addressed in future publications.

3 Technology and Application

At the time of writing, a prototype application of the StreamMiner framework has been developed. It is a public cloud, named the StreamMiner Cloud, dedicated to data stream sharing and analysis. This section starts with introducing the technologies underlying the StreamMiner Cloud, then describes it in details.

3.1 Cloud Services and Stream Transmission

Administrated by the cloud middleware Nimbus [4], all the cloud services implemented in the StreamMiner Cloud and CloudMiner environment are REpresentational State Transfer or RESTful Web services [5]. The RESTful Web service is characterized by implementing all service-client communication using HTTP operations. Due to this characteristic, RESTful services display not only

better usability for clients, but lower overhead on client-service communication in comparison with conventional SOAP Web services. Apart from that, we chose RESTful services in the hope that all StreamMiner Cloud services can be compatible with the worldwide major Web services, such as Twitter, Yahoo, Flickr, and Ebay, which are also RESTful.

In addition, we chose to take the connection-oriented approach mentioned in Section 2.2 to implement the production services in the StreamMiner Cloud. In support of that, a third-party tool, the DataTurbine engine [6], is employed. The engine is a server-client platform specializing in real-time stream transmission. In a DataTurbine system, all configured DataTurbine clients can set up virtual connections and transmit streams among them via a central DataTurbine stream server. In the StreamMiner Cloud, every production service and application service encapsulates a DataTurbine client in its program logic. By calling the client, cloud services can transmit streams among them via connections, conforming to the connection-oriented concept.

3.2 A Real-World Application: The StreamMiner Cloud

The StreamMiner Cloud is a tentative application of the StreamMiner framework, aiming at testing the feasibility and practicability of the framework. This cloud, together with the CloudMiner infrastructure, is located in the GridLab of the University of Vienna, a computer laboratory dedicated to cloud and high-performance computing research. The StreamMiner Cloud hosts a central management service and multiple production/auxiliary services. It reflects the scalability of clouds in that the number of *PSs* and *AuSs* can be changed on the fly according to the users' demand.

The major goal of the StreamMiner Cloud lies in two respects. First, it enables stream sharing among users scattered worldwide: any quantity of remote users and sensors can be connected to the cloud; the users can access any streams produced by the sensors or other users. Second, it supports stream processing and analysis: the user can process and analyze streams by defining her own production flows; in such flows, both existing services or user-developed services can be included.

The accessibility of the StreamMiner Cloud is realized based on the Web technology. A Web site is developed where a user can make use of all the functions of the cloud. Fig. 2 shows a snapshot of the production flow construction page of the Web site. On this page, the user first chooses all the production services to include in the production flow - below the label *choose production service* in the lower left corner. Then she specifies which of these services should send streams - below *choose data source*, and which of them should receive streams - below *choose data sink*. Further, the auxiliary services used by this production flow are selected - below *choose auxiliary service*, while the production services, to which these *AuSs* belong, are chosen - below *choose production service* in the lower right corner. Meanwhile, the topology of the constructed production flow is displayed on the left. This What You See Is What You Get manner simplifies the process of production flow construction.

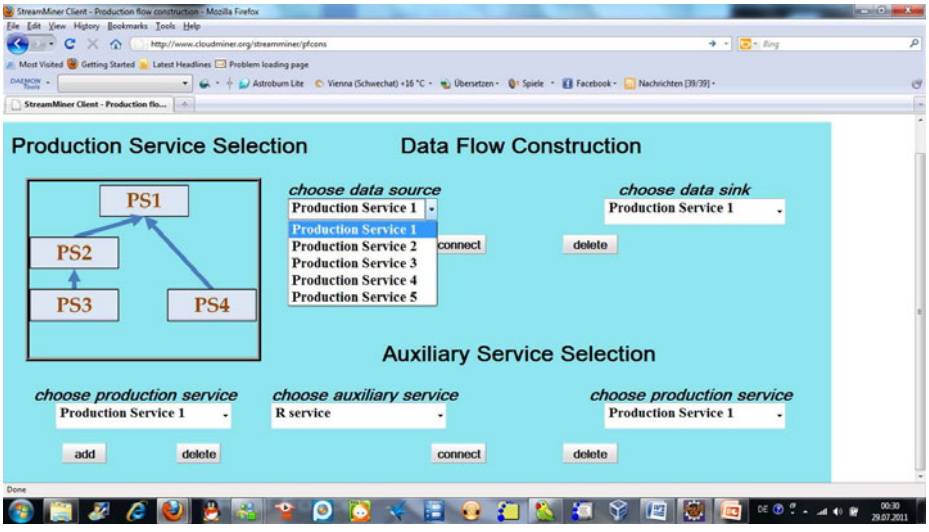


Fig. 2. StreamMiner Cloud Web site

4 Performance Experiments

A series of performance tests were carried out, intended to demonstrate the performance of the StreamMiner Cloud under various conditions, as well as to explore limitations of this prototype. All of the tests share the following experiment configuration. First, a number of identical production services and a DataTurbine server are in use. Each of these components is deployed on a separate compute node with Intel 2.66GHz quad-core CPU and 3.6GB memory. Secondly, as for the source data stream, we employ the MOA [7] waveform generator, which generates waveform-like data sequence with synthesized noise signals.

In each test, five production services (*PS1* to *PS5*) are employed to construct five different production flows (*PF1* to *PF5*). All these production flows share the common precondition. First, *PS1* invariably receives the raw stream containing 100 identical data packets sent periodically; each packet consists of eight real numbers. Second, all *PSs* within a production flow carry out the same task upon the received stream, or no task at all. Furthermore, each *PFn* contains *PS1* to *PSn*; whereas *PSn-1* outputs processed streams to *PSn*. This means that, e.g., *PF5* is composed of *PS1*, *PS2*, ..., *PS5*; *PS1* sends streams to *PS2*, *PS2* to *PS3*, etc.

4.1 Experiment 1: Transmission Delays

The first experiment seeks to investigate the pure transmission delay incurred by the StreamMiner Cloud. The transmission delay of a packet in a certain production flow refers to the total time consumed in transmitting this packet

among the production services of the production flow. This value merely contains the time of network transmission, but not that of packet processing. In this experiment *PF1* to *PF5* are executed one after another, while each production service in these production flows does not perform any processing task upon the received stream. This means that *PSn* receives every packet and sends it to *PSn+1* immediately. Further, the packet transmission delay of every packet is recorded. Take *PF5* as an example: the time difference between the generation of every packet and its arrival at *PS5* is recorded.

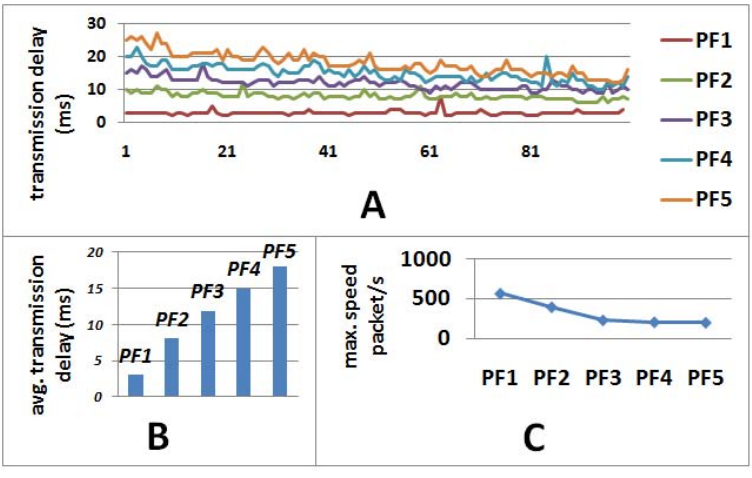


Fig. 3. Transmission delay and maximu sending speed

Chart A of Fig. 3 demonstrates the transmission delay in milliseconds of each packet (indexed from 1 to 100 on the horizontal axis) transmitted in each production flow. It can be observed that production flows with more services, such as *PF5*, tends to suffer from larger delays compared to those with less services, such as *PF2*, since the curve of, for example, *PF5* is higher than that of *PF2*. This behavior is also reflected in Chart B, which illustrates the average transmission delay of each production flow of 100 packets. As can be seen, the height of bars shows a linear increment from *PF1* to *PF5*. The occurrence of the behavior is conceivable: as the number of services increases, packets need to be transmitted more often, and every transmission no doubt contributes to the total packet transmission delay of a production flow. Furthermore, all nodes used in the experiment are identical, resulting in the identical transmission time between any two of them. All these factors are conducive to the linear increment of average transmission delay we observed.

4.2 Experiment 2: Transmission and Processing

The second experiment is to explore the relative significance of transmission delay and stream processing time in each production flow. For this purpose,

every production service is designed to execute a time-consuming processing task - the data classification based on an extensive neural network - upon every received packet. Again, *PF1* to *PF5* are started separately. When a data packet has passed through a production service, both the time consumed in transmission and the time of task execution are recorded. All such records are summed up into the pie charts shown in Fig. 4. In each pie chart, the processing time of every production service in *PFn* is illustrated, together with the length and percentage of transmission delay in this production flow.

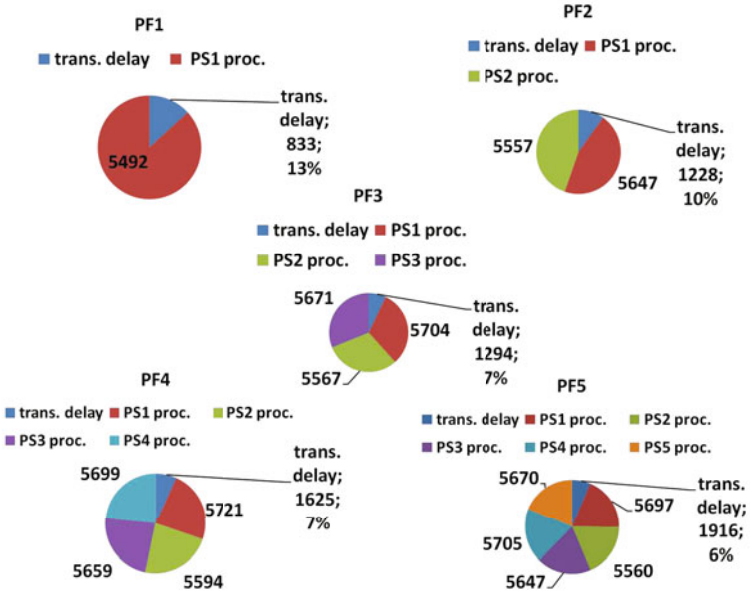


Fig. 4. Transmission delay and processing time

There are two examples of behavior shown in this figure. For one thing, the processing time contributed by any individual service is nearly constant through *PF1* to *PF5*, i.e., around 5500 ms. This behavior reflects the fact that every service is assigned the identical processing task for any packet. For another, the transmission delay increases from *PF1* to *PF5*. This result is caused by the same factor affecting the first experiment, namely the more services a production flow contains, the higher the transmission delay is. Moreover, a decline in the percentage of transmission delay is observed through *PF1* to *PF5*, where the percentage decreases from 13% to 6%. This is to say, the more complicated a production flow is, the less significance the transmission delay presents. This fact lends support to our initial claim that the StreamMiner should address complicated compute-intensive tasks.

4.3 Experiment 3: Limitation on Transmission Speed

In the third experiment we try to detect the upper limit in packet sending speed, namely the highest speed the source stream can send without inflicting a packet lost. Again, *PF1* to *PF5* are tested. Chart C of Fig. 3 records the detected maximum speed of each production flow. For instance, the value 400 of *PF2* means, if the stream generator sends a stream faster than 400 packets per second, then some services in *PF2* might fail to receive packets sent by its sender. A declining trend of the curve, as we observe in the figure, is justifiable, since *PFn* contains one more service than *PFn-1* and thus requires more time to let a packet past through. To this end, *PFn* might not be functional at a speed which *PFn-1* can accept.

5 Related Work

So far, there was only a small number of publications addressing stream data processing in clouds; we mention three of them. Feng et al. [8] put forward the concept of elastic stream cloud. As such, a cloud is the datacenter hardware and software that provides a stream as a service on demand. Based on this, the characteristics of general-purposed clouds and stream-oriented clouds are compared. They claim, for instance, that both types of clouds can operate on 100s to 1000s nodes; whereas the former requires no dedicated links between cloud nodes, but the latter requires low-latency dedicated inter-node links.

Vijayakumar et al. [9] introduce in their work the design of some algorithms that handle unexpected data rates in cloud-based streaming applications. Aiming at matching the stream processing rate with the rate of stream arrival, these algorithms enable an existing streaming application to achieve the optimal CPU allocation with regard to the incoming data stream.

Kleiminger et al. [10] demonstrate a particularly ingenious approach to adaptively balance the workload of a stream processing system. They construct a combined stream processing system that employs a local stream processor to handle average loads while using cloud processors when confronted with peak demand. By doing this, they seek to ensure efficient utilization of resources for differing workloads, whereas providing stable throughput and reliable processing of data.

6 Conclusions and Future Work

This work reflects our tentative research endeavor into the intersection of two promising areas, clouds and data stream management. In the beginning we explored the requirements of stream management applications. Then, we identified particular features the cloud displays to satisfy these requirements. In light of that, we moved to construct the image of a software framework, the StreamMiner, which is our approach to combining those two areas. Being on an abstract level, this framework can be implemented differently by any individual user to

handle her practical problems. Furthermore, we briefly portrayed the prototype application of StreamMiner - the StreamMiner Cloud - that is, a public cloud dedicated to stream sharing and analysis. We also presented an in-depth analysis of the performance tests carried out on this cloud.

We are confident that this work has opened up a path into a broader realm of research for further endeavor. Future works can be undertaken in the following - as well as other directions. (i) The persistence and replay of streams should be addressed. (ii) The framework should be extended to operate on streams of different types, for instance, periodic v.s. aperiodic streams. (iii) More research effort is needed to enhance transmission and processing rates.

References

1. Buyya, R., Broberg, J., Goscinski, A.: Cloud Computing: Principles and Paradigms. Wiley, Chichester (2011)
2. Perrott, R., Harmer, T., Lewis, R.: e-Science Infrastructure for Digital Media Broadcasting. Computer, 67–72 (2008)
3. Goscinski, A., Janciak, I., Han, Y., Brezany, P.: The CloudMiner: Moving Data Mining into Computational Clouds. In: Grid and Cloud Database Management. Springer, Berlin (2011)
4. Sempolinski, P., Thain, D.: A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pp. 417–426 (2010)
5. Laitkorpi, M., Selonen, P., Systa, T.: Towards a Model-Driven Process for Designing ReSTful Web Services. In: IEEE International Conference on Web Services, pp. 173–180 (2009)
6. Tilak, S., Hubbard, P., Miller, M., Fountain, T.: The Ring Buffer Network Bus (RBNB) DataTurbine Streaming Data Middleware for Environmental Observing Systems. In: IEEE International Conference on e-Science and Grid Computing, pp. 125–133 (2007)
7. Bifet, A., Holmes, G., Pfahringer, B., Kranen, P., Kremer, H., Jansen, H., Seidl, T.: MOA: Massive Online Analysis, a Framework for Stream Classification and Clustering. In: Journal of Machine Learning Research (JMLR) Workshop and Conference Proceedings (2010)
8. Feng, J., Wen, P., Liu, J., Li, H.: Elastic stream cloud (ESC): A stream-oriented cloud computing platform for Rich Internet Application. In: 2010 International Conference on High Performance Computing and Simulation, pp. 203–208 (2010)
9. Vijayakumar, S., Zhu, Q., Agrawal, G.: Dynamic Resource Provisioning for Data Streaming Applications in a Cloud Environment. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pp. 441–448 (2010)
10. Kleiminger, W., Kalyvianaki, E., Pietzuch, P.: Balancing load in stream processing with the cloud. In: 2011 IEEE 27th International Conference on Data Engineering Workshops, pp. 16–21 (2011)

Security Architecture for Virtual Machines*

Udaya Tupakula[1], Vijay Varadharajan[1], and Abhishek Bichhawat[2]

[1] Information & Networked Systems Security Research, Department of Computing, Faculty of Science, Macquarie University, Sydney, Ausralia
{udaya.tupakula,vijay.varadharajan}@mq.edu.au
[2] Department of Electronics and Computer Engineering, IIT Roorkee, India
abhibpec@iitr.ernet.in

Abstract. We propose security architecture based on virtual machine monitor to efficiently deal with attacks on virtual machines. We will show that our model is capable of detecting suspicious processes running in the virtual machine, can detect and prevent different types of attacks including zero day attacks by monitoring the virtual machine traffic and the processes that are generating or receiving the traffic. The architecture also makes use of sharing information about the suspicious behaviour among multiple Intrusion detection systems deployed in different virtual machine monitors. We describe the implementation of the proposed architecture and present a detailed analysis of how our architecture can be used to detect zero day attacks.

Keywords: Virtual Machine Monitors, Intrusion Detection, hidden processes.

1 Introduction

The current Internet environment is vulnerable to a range of different types of attacks [1, 2] such as malware, phishing, spam, and denial of service; several new types of attacks are appearing on a daily basis. The attackers are able to exploit vulnerabilities in software such as operating systems and applications as well as inherent weaknesses in the Internet protocol stack. As a result there is an increased activity of zero day attacks in the Internet.

Although there are several tools such as intrusion detection systems, honey pots, antivirus and anti malware, the dynamic nature of the attacks makes it difficult to detect and prevent attacks. The host based tools have good visibility of internal state of the monitored system and can detect the attacks more efficiently. However since the tools are implemented on the monitored system itself, they are vulnerable to attacks by the attacker. The network based tools detect the attacks by monitoring the incoming and outgoing traffic from the monitored machines. They have less visibility into the state of monitored machines but offer high attack resistance. For efficient

* The authors would like to thank Departments of the Prime Minister and Cabinet (PM&C) and Defence Signals Directorate (DSD), Australia, for their financial support of the research project on Secure Virtualization Systems. The PM&C and DSD funding should not be taken to imply endorsement of the content or conclusions of the research project.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 218–229, 2011.

detection of attacks, it is desirable for the tools to have good visibility of the monitored system while at the same time offering high resistance to attacks. Several limitations of the existing tools can be overcome by implementing the security tools using Virtual Machine Monitors (VMM) [3]. A VMM is an additional software layer which has complete control on the physical resources and enables to run multiple operating systems on a scalable computer. Since the VMM can have complete control of the resources, good visibility of the internal state of the virtual machines, while being isolated from the virtual machines themselves, they can be used for improving the attack detection/prevention efficiency of the security tools. Such VMM based security tools are sometimes referred to as Virtual Machine Introspection (VMI) tools [4]. In this paper, we propose security architecture based on virtual machine monitor to efficiently deal with attacks on virtual machines.

The paper is organized as follows. In Section 2, we propose security architecture for virtual machine monitor (VMM) based systems and present the operation of our model. Section 3 presents the implementation and analysis of our architecture. In particular it illustrates how the architecture has been used to detect Slammer type attacks and present some performance characteristics. Section 4 presents some of the related work and Section 5 concludes the paper.

2 Our Model

In this section we will first present an overview of our model and then operation of our model.

2.1 Overview

Let us consider a scenario where services are provided by the virtual machines residing on top of virtual machine monitors (VMMs). Our aim in this paper is the development of an intrusion detection architecture that enables efficient detection and prevention of different types of attacks on virtual machines and the isolation of the malicious entities generating these attacks. We will collectively refer to different types of attacks such as worms, viruses, Trojan horses as malware attacks. The intrusion detection system (IDS) is integrated into the VMM. In this paper, we will assume that the VMM is trusted and secure. This is a common trust assumption that is made with many VMM based systems. This assumption is based on the premise that compared to operating systems, a VMM is smaller in size and hence in principle can be designed (and verified) to be secure. If this assumption is not valid, then vulnerabilities in the VMM can be exploited by the attacker to attack any of the virtual machines that are running on top of it.

Figure 1 shows the architecture of the IDS system which can be integrated into the VMM or the host operating system. Since the VMM has complete control on the physical resources, our model can determine the state of the virtual machines by monitoring the usage of allocated resources to the virtual machines for intrusions. An important function of our model is to identify the malicious entity that is generating the attack traffic and dynamically isolate the malicious entity. The entity can be very broad such as a compromised virtual machine or an application or a process that is

running in a virtual machine. In order to deal with the attacks efficiently, the entity has to be defined at a fine granular level. If the whole virtual machine is isolated due to an attack arising from one application, then it can cause denial of service for the other legitimate applications that are running in that virtual machine.

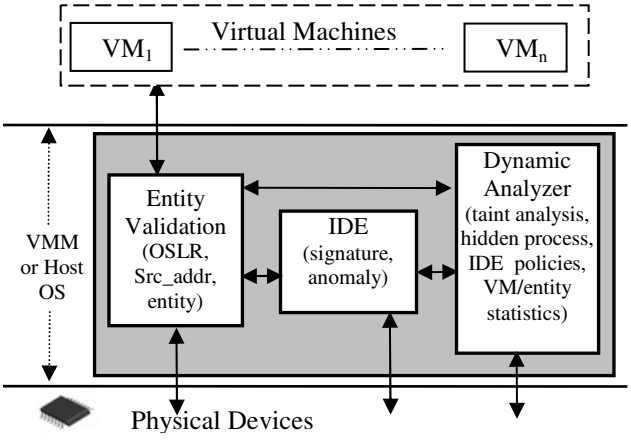


Fig. 1. Security Architecture

Our architecture consists of the following components: Entity Validation (EV), Intrusion Detection Engine (IDE), and Dynamic Analyzer (DA). The Entity Validation component is used for detection of attack traffic with spoofed source address, fine granular detection of process (which is generating or receiving traffic in the virtual machine), secure logging of new entities in the virtual machines. The Intrusion Detection Engine component is used for detection of known attacks and detection of suspicious behaviour by monitoring the incoming and outgoing traffic of virtual machines. Dynamic Analyzer is used for detection of suspicious processes running in the virtual machine, detection of zero day attacks, fine granular isolation of malicious entities (processes/applications) that are generating the attack traffic, and sharing of information about suspicious behaviour and attack signatures between multiple Analyzers in a distributed environment.

We will now describe how the security architecture is used to detect the intrusions; we will consider first at the source end, then at the destination.

2.2 Operation at Source

Whenever a new virtual machine is installed on the VMM, the OS Library and Repository (OSLR) module in the Entity Validation component is initialized with information on the operating system configuration running in the virtual machine. For instance, in the case of Windows XP, OSLR is initialized with configuration information on Windows XP image, its service pack version, Internet Explorer version, detail of drivers and any additional applications installed on Windows. As new applications are installed, the OSLR captures information about these applications when they start interacting with other hosts. The information stored in the

OSLR is used by the Dynamic Analyzer in defining the attack signatures that are specific to each virtual machine. In addition, as the information in the OSLR increases, we can use machine learning techniques such as Bayesian logic to differentiate between legitimate and suspicious traffic for each application or virtual machine, which are useful for detecting zero day attacks.

The virtual machine reports the application and the process that is generating the traffic. The EV logs the packet in the Traffic Store, validates the application (or process) that generated the traffic and the source IP address of the traffic. The details of the entity that are recorded in the database are the process name, process ID, user object, session id and priority of the process. The component Traffic Store keeps a record of all traffic passing between from and to virtual machines. Depending on the resources, we can store either the complete packets or hash of the packets. One approach is to maintain the complete traffic for a certain time period (say for 1 hr or 2hr) and then later only maintain heuristics such as the source, destination, the length of communication, total bytes sent or received and the protocols used form communication.

If the record for the application or process exists in the OSLR database, then it is updated with traffic details. Otherwise a new record is created for the reported entity and a flag is used to indicate that this is a new record, which can be subsequently queried by the IDE or the Dynamic Analyzer if the traffic is considered to be suspicious or malicious.

If the virtual machine has a public IP address, then the entity validation component ensures that the packet generated by the virtual machine has correct IP address. If multiple virtual machines are sharing a single IP address then the entity validation module replaces the private IP address with the shared public IP address. Note that since the source address of all the packets is validated by the entity validation component, it is not possible for the virtual machines to generate attack traffic with spoofed source address. However at this stage, it is still possible to generate attack traffic with correct source address and we will see later how this is detected and prevented in our architecture.

Since EV has access to the physical resources, it is able to validate if the process or application reported by the VM actually generated the packet. If the process reported by the virtual machine is found in the memory, then the information can be trusted and the traffic is passed to the IDE. If the source address of the traffic is spoofed or if the process/application reported by the VM is not found in the memory, then the virtual machine is considered to be suspicious and reported to the Dynamic Analyzer for further analysis.

The main goals of the IDE is to ensure that any traffic that is entering or leaving the VMM does not contain any traffic that is matching with the known attack patterns and to detect if the incoming or outgoing traffic is suspicious. The relevant security policies that need to be enforced are determined by the administrator and/or the Dynamic Analyzer. Since the OSLR has the details of resources allocated to each virtual machine and applications running on each virtual machine, the Dynamic Analyzer can use this information to specify security policies by considering the unique properties of the virtual machine.

We have created a database in the IDE of known attack signatures; this database will be continually updated as and when new attack signatures are discovered. We

have organized the database in such a way that the VMM administrator configures the attacks per virtual machine. Hence the IDE is used to detect the known attacks in the incoming or outgoing traffic. However the traffic may also contain zero day attacks. So the traffic is validated (both at regular intervals and randomly) against statistical patterns of data. For instance, flooding is a common behaviour [1,2] in most of the emerging attacks. We use these techniques to identify packets as suspicious in the following cases: if there is a sudden burst of traffic or if we detect abnormal usage of resources (such as CPU or network) by a single entity, or random packets to the same or multiple destinations, packets with spoofed source address, and packets destined to non standard port numbers. In some cases, there is need for communication between the anomaly module and OSLR to detect if some of the traffic is suspicious. For example if a burst of reply packets are generated by the virtual machine the IDE can query the OSLR to determine the corresponding request packets. If no request packets were received by the virtual machine, then the suspicious reply packets are considered to be malicious.

When the CPU is idle, the anomaly based module in the IDE requests the dynamic analyzer to update the statistical behaviour of the processes in the VM. The dynamic analyzer applies the Bayesian based learning technique on the OSLR data and updates the statistical data in anomaly based module to differentiate between legitimate and suspicious behaviour for each virtual machine. We are not describing this algorithm here in this paper due to space restrictions. Essentially this algorithm enables the IDE to capture the dynamic changes for each virtual machine and identify the attacks. The evaluation process of the IDE works as follows: If the traffic is not matching with any of the attack signature and found to be legitimate by the anomaly based detection module, then the traffic is forwarded to the destination. If the traffic is matching with a known attack signature or found to be suspicious by the anomaly engine then the virtual machine is considered as suspicious and reported to the Dynamic Analyzer for further analysis.

The Dynamic Analyzer (DA) performs further analysis on the virtual machine that is considered suspicious by the EV or the IDE to detect if there is ongoing zero day attack. Most of the zero day attacks exploit the vulnerabilities in the operating system and databases by creating buffer overflows, rewriting parts of the memory and manual jump to addresses. The attacks can also come from hidden processes collecting sensitive information in an unauthorized manner or generating attack traffic to random hosts in the Internet. In case of zero day attacks, the attack signatures are identified by specifying the behaviour of the suspicious entity or by identifying the similarities of the packets generated by the suspicious entity.

Our architecture provides the following mechanisms in the DA to validate if the suspicious behaviour of the virtual machine is malicious or benign. First, it validates if there is any suspicious process running in a virtual machine or if some of the critical process are not running in the virtual machines. For example, attacks such as conficker [3] disable important services such as error reporting, auto updates, windows defender and background intelligent transfer services. The virtual machine is considered as suspicious if such important services are not running in the virtual machine. In addition to this there can be some malicious hidden process running in the virtual machine. The DA queries the virtual machine to report the processes running in it. When the virtual machine reports the running processes, the DA obtains

the list of processes that are actually running in the memory assigned to the virtual machine and compares it with the VM report. If there is any variation in the list of processes reported by the VM and list of processes observed by the dynamic analyzer then the VM can be considered as malicious. However since the queries are done during different time intervals there is possibility for some new process to be initiated or for an existing process to be terminated after the report is generated by the virtual machine and before the query is performed by the dynamic analyzer. Hence to minimize risk of false positives and false negatives, if there is variation in the number of processes then the complete process is repeated. If there is a variation in the number of processes after repeated validations then the virtual machine is considered as malicious and the malicious process/entity can be detected by comparing the process reports.

Now the malicious entity can be dynamically isolated from the virtual machine or further analyzed in an isolated environment using taint analysis or statistical analysis to determine the behaviour of malicious entity and identify attack signatures.

Since the dynamic analyzer has access to the physical resources, it can monitor all the interactions of the malicious entity with other entities in the virtual machine and how the subsequent packets will be generated by the malicious entity. For example, if the malicious process is accessing the inputs from the keyboard, then it can be considered as collecting sensitive information of the user and sending it to the attacker without the user's knowledge. Alternatively if the malicious process is generating some malicious/suspicious packets to one or more destination addresses then it can be considered as sending attack traffic without user's knowledge. Now a detailed analysis of the payload of the outgoing packets is carried out such as what type of data is being sent and to which destination.

While a decision is being made by the DA on the suspicious virtual machine or suspicious traffic, we have a choice as to either dropping the packets (pessimistic approach) or passing or rate limiting the packets (optimistic approach) based on the characteristics of the packets. Our current implementation uses rate limited transfer of packets, if the security policies such as validate source address function (src_addr), and validate hidden processes in virtual machine function (vm_hid_prc) are satisfied. In this case even if the rate limited traffic is found to be attack traffic, then the destination host can easily trace the attacking source (since attack traffic has correct source address) and inform the source DA. Although there can be false positives and false negatives in this case, the impact will be minimal on legitimate traffic and significant impact on the attack traffic. For example, since we are rate limiting the suspicious traffic, this will reduce the impact of the spread of the malware. On the other hand if the rate limited traffic is legitimate, this will only cause some delays to the legitimate traffic. However if the suspicious packet exhibits serious properties such as spoofed source address, sending reply packets without receiving requests then the packets are dropped. This is because sending packets with spoofed traffic will eliminate the possibility for the traceback and in other case there is no legitimate use for sending reply packet for which no request was received. Hence in this case, attacks can be efficiently detected and prevented with no false positives or false negatives.

After the analysis, if the DA identifies the suspicious traffic to be malicious, the architecture identifies the malicious entity that generated the packet by querying the

OSLR and isolates the malicious process. Then the dynamic analyzer develops a new attack signature based on the properties exhibited by the malicious packet/flow and updates the signature database. This will prevent any other virtual machine sending similar attack traffic in the future. On the other hand if the suspicious flow is identified to be benign then the statistical data is updated. This will prevent false alarms from similar traffic in the future.

2.2 Operation at Destination

Now let us consider how the attacks are detected at the destination IDS in our architecture. At the receiving end, the traffic destined to the virtual machines is received by the IDE. The traffic is monitored against security policies of the destination virtual machine such as known attack signatures and anomaly based detection. If the traffic does not match with any of the known attack signatures and identified to be legitimate by the anomaly based module then the traffic is passed to the Entity Validation component at the destination. If the packet matches with any of the attack signatures or found to be suspicious then the traffic is reported to the Dynamic Analyzer at the destination.

The EV logs the traffic in the OSLR in the destination VMM and forwards the traffic to the appropriate VM. The VM reports the entity that is receiving the traffic and this is updated in the database. If the reported entity is not found in the memory allocated to the virtual machine then the virtual machine is considered as suspicious and reported to the dynamic analyzer for further analysis.

As explained at the source, the DA at the destination uses similar techniques to detect malicious traffic. It matches with the attack signatures and to validate traffic that is considered as suspicious. If traffic is matching with the attack signature then the packet is dropped and there is an option to notify the source IDS. If a notification is sent to source IDS, the source DA then determines the malicious entity that generated the attack traffic by querying the OSLR and isolates it from sending similar packets in the future.

The DA at the destination end checks if there is any suspicious process running in the destination virtual machine using a similar method as discussed at the source dynamic analyzer. If a suspicious process is detected in the hosted virtual machine then it is run in an isolated environment and the suspicious traffic is passed to the virtual machine. If the suspicious traffic is received by the suspicious process in the virtual machine and responding to the suspicious traffic by sending attack traffic then it can be considered as control command to the suspicious process. If no suspicious process are detected in the hosted virtual machine then the traffic that is considered to be suspicious can be either dropped, rate limited or forwarded to the destination virtual machine as before.

If the DA decides to forward the suspicious traffic to the destination then the virtual machine is considered as suspicious. One of the important reasons to identify the hosted virtual machine (at the destination) as suspicious in this case is to analyze the impact of the received suspicious packet on the hosted virtual machine. Hence future packets from the hosted virtual machine will also be monitored by the DA.

In our architecture, the destination virtual machine will be considered to be "questionable" until a decision is made on the received suspect packet by the dynamic

analyzer. To minimize the risk of crash of the destination virtual machine, the dynamic analyzer can copy the image of the destination virtual machine and perform further analysis in an isolated environment by passing the suspicious traffic to the isolated virtual machine. Any identified attack signatures via this analysis is used to update the database in the IDE. In addition the attack signature can also be sent to the source dynamic analyzer. Moreover the source and destination IDS can share further information on the suspicious packets which can further help to deal with zero day attacks. We will see below in Section 3.1 how such sharing of information is useful in the case of Slammer attacks.

3 Analysis

We have implemented the IDS architecture shown in Figure 1 on Xen Virtual Machine Monitor. We have conducted performance analysis with varying number of virtual machines hosted on each physical server and validated against different types of attacks. We have used Xen 3.1.2 VMM with virtual machines running different operating systems and applications. The device drivers in Xen have a front end module which are implemented in the virtual machine and a back end module in the Dom 0. The guest VM send the packets using the front end drivers and the host machine sends the packets to the guest virtual machines using the back end drivers. The policy engine is placed between the front end and back end drivers. In this paper we present the analysis of our model with slammer attack and some performance results.

3.1 Slammer Analysis

Here we present a detailed analysis of how our model was able to deal with the Slammer worm [1] on a virtual machine running unpatched SQL server. Slammer is an interesting case because it only required a single malicious packet to severely degrade the service running in many hosts in the Internet during January 2003. Although several worms have been witnessed during past few years, Slammer, which exploited the buffer overflow vulnerability in unpatched SQL server and MSDE, is considered to be one of the fastest spreading worms. The worm achieved the peak scan rate of 55 million scan within 3 minutes of infection. The attack consisted of a single UDP packet destined to port number 1434 with the payload size of 374 bytes and total size including headers was 404 bytes. Following scanning, any machine which was running un-patched SQL became vulnerable to the worm and started infecting other vulnerable hosts in the Internet. The other interesting behaviour of the attack is that it exists as a running process and does not write itself to the disk. In this section, we use Slammer to illustrate how our architecture can deal with such sophisticated attacks.

It has to be noted that our model can deal with the emerging attacks such as conficker [2] and torpig. For example, attacks such as conficker, and torpig perform several additional activities that can be easily detected as malicious by the IDS components. For example, the conficker worm disables all the important services in the windows operating systems security related process. This can be detected by

model during validation of the process running in the virtual machine. Even if the emerging malware uses advanced techniques such as domain flux to evade detection, the attacks can be detected since all the interactions of the VM are securely logged in the OSLR. Also since each component of our model deals with different types of attack behaviour, as the malicious behaviour exhibited by the worms increase, the attacks can be efficiently detected by our model. Since Slammer is a single packet attack, the following discussion is applicable to other attacks also.

We captured the traffic from a production SQL server and developed several statistical policies for the traffic such as max-min-avg for total traffic, protocols at different layers, port numbers, and packet sizes. We have implemented an unpatched SQL server on virtual machine and snapshot of the virtual machine is taken for every 30 minutes interval and a maximum of 5 snapshots were maintained. The total count for anomaly detection module for each virtual machine was set to 1 minute. Now the UDP traffic containing Slammer worm is sent to the virtual machine running unpatched SQL server. Let us now discuss how our model detected the attack in different cases.

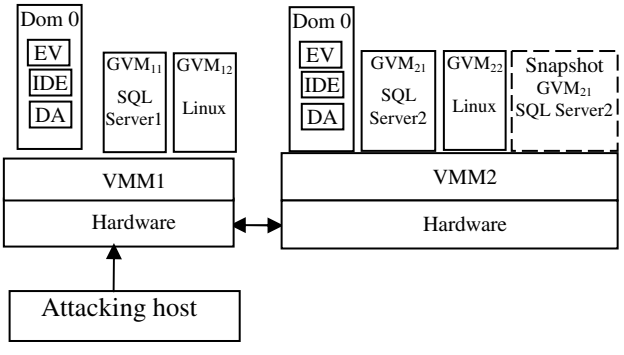


Fig. 2. Slammer Scenario

As shown in Figure 2, the Slammer attack traffic was destined to the guest machine (GVM_{11}) which was running unpatched SQL server on VMM1. The malicious UDP traffic was received by the IDE in host virtual machine on VMM1. Since the traffic did not match with any of the known attack signature and since it did not exhibit any anomaly, the packet was passed to the Entity Validation component. Now the packet is logged in the OSLR and passed to the virtual machine and the entity that was receiving the packet is the SQL server. As soon as the packet is processed by the GVM_{11} it gets infected and starts generating UDP flood. These packets are received by the Entity Validation component and logged in the OSLR, and the traffic is validated for the source address and entity. Note that even if a single UDP packet was generated with spoofed source address, the validate source address function (src_addr) would raise the flag (flag_VM) and the attack is easily detected by the EV. However since the source address of the UDP traffic was valid the EV did not detect any suspicious behaviour and the traffic was forwarded to the IDE. Hence we can conclude that attack traffic was generated with correct source address. Since the UDP

traffic did not match with any of the attack signatures, they were passed to the anomaly detection module. The total count of the traffic was increasing with every outgoing packet and the packets were randomly validated against some sub modules of VM statistics (such as packet size (pkt_sze), no of open connections (open_con), TCP/IP protocols (proto)) and/or Entity statistics (such as entity protocols (ent_proto), entity port numbers (ent_port)). The flag_VM was enabled when the total traffic was above the threshold and a report was sent to the Dynamic Analyzer. The debug mode was enabled for the VM and all the incoming and outgoing traffic from the virtual machine is monitored by the DA. First the dynamic analyzer validated the number of processes running in the virtual machine (using (vm_hid_prc) function) to check any suspicious process in the virtual machine. The DA queried the OSLR and determined that the entity that was generating the suspicious traffic was SQL server and the variations in the UDP traffic which will be discussed further enabled the DA to consider the traffic as malicious and prevent the attack traffic.

Here we present one of the techniques used for the automatic attack signature detection. The variance function compares the statistical behaviour of the suspicious entities with the legitimate behaviour and returns the variance of different parameters in decreasing order. This is followed by similarity function which takes the output of the variance function and identifies the TOP N similarities as the attack signature. The attack signature was identified to be UDP traffic destined to port number 1434 by considering the TOP 2 similarities in the attack traffic. The signature database was updated with the new attack signature to the SQL server.

Now let us discuss different case scenarios outlining how the attacks were triggered in our security architecture.

Case 1: The standard port for SQL server assigned by the IANA is 1433 and TCP is the standard protocol used for the communication. UDP port 1434 is used to report the dynamically assigned port in case of multiple instances of database running on the same server. Even if multiple instances of SQL server are running on the server, the corresponding port can be only used as a request-reply protocol where the remote client request for the port number of the dynamically assigned port and the corresponding reply responds with the dynamically assigned port. Hence the requests can be in the form of 1 or 2 UDP packets and the reply packets can be in 1 or 2 packets. In the case of Slammer there is considerable variation where a single request will trigger multiple response packets. The statistics of the legitimate SQL server revel that the ratio of TCP: UDP protocol to be in the order of > 99 : < 1. However during the attack the ratio of the TCP:UDP protocol was found to be <1: >99.

In some cases, the traffic to port number 1434 was identified as attack traffic. The statistics of the port number 1434 for incoming: outgoing traffic in a regular situation was found to be of the order of 1: 2. However during the time of attack, the incoming: outgoing traffic was found to be <1: >25000. Hence the traffic destined to port number 1434 was identified to be malicious. The following suspicious cases were also detected by the anomaly detection module. In some case the UDP traffic was destined to broadcast address. In some cases, the LAN was congested by the UDP traffic and the analysis of the network traffic confirmed UDP flood. This was triggered as suspicious network behaviour. When the packets were destined to a virtual machine

that was not running SQL server or MSDE, the UDP traffic was identified to be malicious since the port 1434 was closed on the destination virtual machine. This information was shared between the dynamic analyzer and was useful for detecting the Slammer worm.

Case 2: When the UDP traffic was destined to another virtual machine hosted on other physical server, the source DA informed the destination DA regarding the suspicious traffic and forwarded the suspicious UDP traffic. In some cases (GVM_{22} in Figure 2) the destination port was closed on the virtual machines that were not running SQL server or MSDE and in the case where the virtual machine was running SQL server (GVM_{21} in Figure 2), since the source analyser has informed the destination analyser prior to sending the suspicious traffic, the destination analyser forwarded the UDP traffic to the snapshot of the virtual machine and the snapshot virtual machine on VMM2 in Figure 2 was infected and started flooding UDP traffic.

3.2 Performance Analysis

We have conducted performance analysis of our model for different components. Figure 3 shows the average time (in sec) for correlating the network flows observed at the entity validation module with processes running in the virtual machine. The results are an average for 10 runs. As the number of virtual machines increase, there is minor variation in correlation time. Similarly, Figure 4 shows the average time (in sec) for validation of hidden processes in the virtual machines.

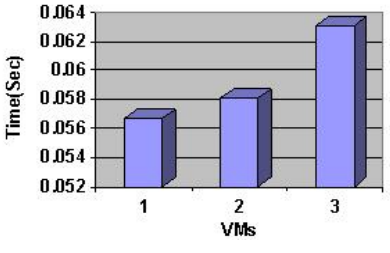


Fig. 3. Traffic to process mapping

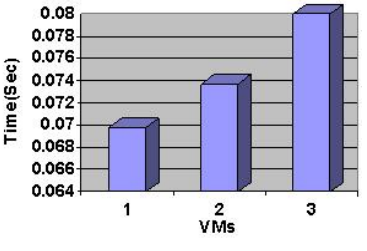


Fig. 4. Hidden process validation

4 Related Work`

Recently there is considerable research interest for developing security tools that are based on Virtualization. Garfinkel [4] proposed a Livewire intrusion detection system which makes use of the VMM to obtain the state of the virtual machine and detect in there is any ongoing attack. Antfarm [5] can be used for the detection of hidden processes in the virtual machine. Vigilante [6] is a collaborative approach where each host runs specific software which captures the information regarding the exploit of the worm and distributes a Self Certifying Alert (SCA) to warn other hosts regarding the spread of the worm. The end hosts can use the information in the SCA to identify if it is vulnerable to the worm and apply a host based filter to prevent the worm.

5 Conclusion

In this paper we have proposed security architecture based on virtual machine monitor to efficiently deal with attacks on virtual machines. Our model is capable of detecting suspicious processes running in the virtual machine, can detect and prevent different types of attacks including zero day attacks by monitoring the virtual machine traffic and the processes that are generating or receiving the traffic. The architecture also makes use of sharing information about the suspicious behaviour among multiple Intrusion detection systems deployed in different virtual machine monitors to detect the attacks.

References

1. Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the Slammer worm. IEEE, Security & Privacy 1(4), 33–39 (2003)
2. Shin, S., Gu, G.: Conficker and Beyond: A Large-Scale Empirical Study. In: 26th Annual Computer Security Applications Conference, Austin, Texas, USA, December 6-10, pp. 151–160. ACM, New York (2010)
3. Smith, J.E., Nair, R.: The architecture of virtual machines. Computer 38(5), 32–38 (2005)
4. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: 10th Network and Distributed System Security Symposium, California. Internet Society, USA (2003)
5. Jones, S., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: VMM-based Hidden Process Detection and Identification using Lycosid. In: 4th International Conference on Virtual execution environments, Seattle, WA, March 5-7, pp. 91–100. ACM SIGPLAN/SIGOPS, USA (2008)
6. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-End containment of Internet Worms. In: Proceedings of the 20th ACM symposium on Operating systems principles, SOSP 2005, Brighton, UK, October 23-26, pp. 133–147. ACM, New York (2005)

Fast and Accurate Similarity Searching of Biopolymer Sequences with GPU and CUDA

Robert Pawłowski, Bożena Małysiak-Mrozek,
Stanisław Kozielski, and Dariusz Mrozek

Institute of Informatics, Silesian University of Technology
Akademicka 16, 44-100 Gliwice, Poland
robertp86@gmail.com, {Dariusz.Mrozek,Bozena.Malysiak,
Stanislaw.Kozielski}@polsl.pl

Abstract. CUDA is an architecture introduced by NVIDIA Corporation, which allows software developers to take advantage of GPU resources in order to increase the computational power. This paper presents an approach to accelerate the similarity searching of DNA and protein molecules through parallel alignments of their sequences with the use of GPU and CUDA. In order to optimally align two biopolymer sequences, such as amino acid or nucleotide sequences, we employ the Smith-Waterman algorithm. We present the optimization steps leading to achieve a very good efficiency of our implementation on GPU and we compare results of efficiency tests with other known implementations. The results show that it is possible to search bioinformatics databases accurately within a reasonable time.

Keywords: bioinformatics, DNA, proteins, sequence alignment, parallel computing, GPU, CUDA.

1 Introduction

Biological databases are very valuable tool in the research carried out in bioinformatics, molecular biology and related fields. For example, comparing an unidentified sequence of nucleotides or amino acids, which was obtained during experimental research, to the known sequences stored in a database, we can conclude about evolutionary relationships of the unidentified sequence, molecular structure encoded by the sequence, and even the biological role or function of the encoded structure.

The role of biological databases is constantly growing. Over the last few years, the number of amino acid and nucleotide sequences added to these databases almost doubles every year [4], [5]. Bioinformatics is constantly looking for more efficient methods to search the collected information. Due to the inhibition of the growth rate of CPUs clock speed and the increasing availability of multiprocessor and multicore systems, it is worth to develop algorithms that work in parallel and collectively in order to search large databases.

The paper presents the possibility of using the great potential of Graphics Processing Units (GPUs) employing NVIDIA CUDA (*Compute Unified Device Architecture*)

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 230–243, 2011.

and SIMT architecture (*Single Instruction, Multiple Thread*) to accelerate the similarity searching in databases collecting DNA/RNA or protein sequences. CUDA has several unique features that determine its usefulness for this purpose, i.e. availability and large number of devices supporting this architecture, approachable and well-documented application programming interface (API), portability and scalability of developed solutions. With the use of CUDA and SIMT architecture, we have improved the speed of similarity searching performed with the Smith-Waterman alignment algorithm, which has a great practical importance, but whose computational complexity does not allow for its widespread use in processing large data repositories.

2 Theoretical Background

2.1 Alignment Algorithm

The algorithm of T.F. Smith and M.S. Waterman [11] finds the most similar, relatively short sections for pairs of compared biomolecular sequences. This allows to capture conserved sequence motifs and structural mutations arising in the process of evolution, such as translocations or duplications of fragments of sequences. In contrast to algorithms that use heuristics, like the most popular BLAST [1], the Smith-Waterman algorithm is characterized by a higher sensitivity. This results in the ability to detect more distant similarities in compared sequences.

In order to align two sequences P and Q, the Smith-Waterman algorithm uses the similarity matrix M (sometimes called alignment matrix). The size of the matrix M is $(p+1)\times(q+1)$, where p and q are the lengths of the matched sequences. Let $P(i)$ and $Q(j)$, $i=2,\ldots,p+1$, $j=2,\ldots,q+1$, are elements of compared sequences P and Q, located at the position $i-1$ and $j-1$, respectively. Then, starting from $i=2$ and $j=2$, successive elements of the similarity matrix M are determined according to the formula:

$$M_{ij} = \max\{S_{ij} + M_{(i-1)(j-1)}, \max_{1<k\le i-1, k\in N}\{M_{kj} - |G_P|\}, \max_{1<l\le j-1, l\in N}\{M_{il} - |G_P|\}, 0\}, \tag{1}$$

$$\text{where: } S_{ij} = \begin{cases} M_A, & if \quad P(i) = Q(j) \\ M_P, & if \quad P(i) \neq Q(j) \end{cases}, \tag{2}$$

For the first row and first column of the matrix:

$$M_{i1} = M_{1j} = 0 \cdot \tag{3}$$

The M_A is an award for matching, and the M_P is a penalty for mismatching elements $P(i)$ and $Q(j)$. Mismatching elements of two sequences are interpreted as the effect of mutation – substitution in the evolution process. The values of M_A and M_P can be chosen arbitrarily or according to one of the known substitution matrix, e.g. PAM or BLOSUM. The G_P is a gap penalty, which results from the possible deletions or insertions. The G_P may be expressed as a constant value or a linear function, e.g.:

$$G_P = G_O + G_E n, \tag{4}$$

where: G_O is a penalty for opening a gap, and $G_E n$ is a penalty for gap extension, proportional to the gap length n; G_E is a constant of proportionality. The $|G_P|$ means the absolute value of the G_P.

The value of the similarity measure (*Score*) for aligned pair of sequences is the highest value in the filled matrix *M*. The larger the value of the *Score*, the more two sequences are structurally similar. The complexity of the Smith-Waterman algorithm is *O(pq)*. Although modern computers posses a considerable computing power, the complexity is still too high to apply the Smith-Waterman algorithm as a routine method for searching large collections of sequences in databases.

2.2 CUDA Programming Model and Architecture of Hardware Accelerator

In graphics cards with CUDA technology the high scalability was achieved by the hierarchical organization of *threads*, which are basic execution units. Each thread has its own index, the vector of coordinates corresponding to its location in one-, two-or three-dimensional organizational structure called the *block*. Thread blocks form one- or two-dimensional structure called the *grid*. Each thread block is processed by a Streaming Multiprocessor (SM). The number of available multiprocessor depends on the graphics card. However, in devices with CUDA 1.1 that was used in our research, each SM is composed of eight Scalar Processor cores (SP), two special function units, a multithreaded instruction unit (IU), and high-speed shared memory (Fig. 1).

Each multiprocessor has also access to the on-chip, read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from the *constant memory* space, and also to the read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from the *texture memory* space. Texture memory is a special type of memory providing high performance in rendering images.

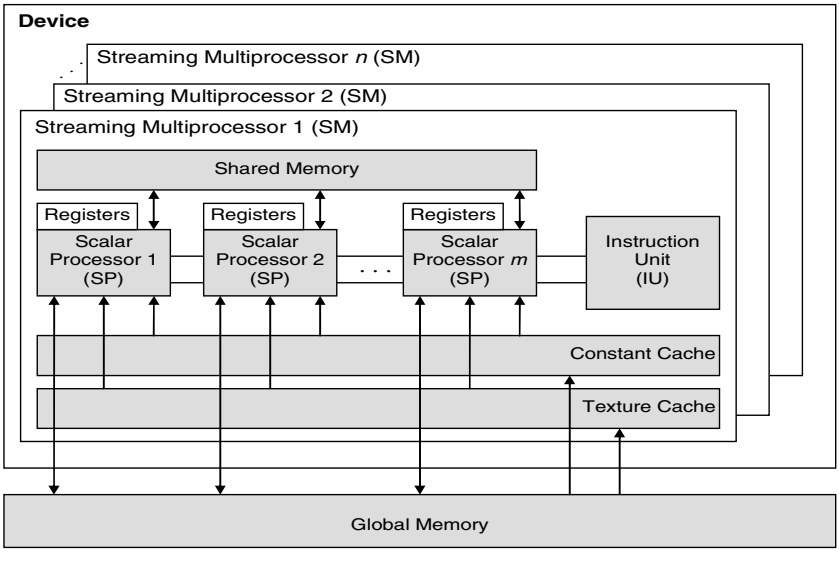


Fig. 1. Architecture of the GPU computing device (based on [9])

Multiprocessors employ a new architecture, called SIMT (*Single Instruction, Multiple Thread*). In this architecture, a multiprocessor maps each thread to a scalar processor core, where each thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Threads in the warp perform the same instructions, but operate on different data, as it is in the SIMD architecture. Full performance in SIMT architecture is achieved when all 32 threads have the same execution path [9].

3 Related Works

Parallel implementation of the Smith-Waterman algorithm using CUDA technology is relatively new in a scientific literature. One of the first works, which achieved a considerable acceleration, is SW-CUDA [8]. SW-CUDA was implemented on the NVIDIA GeForce 8800 GTX, reaching an efficiency of 1.8 GCUPS (*Giga Cell Updates Per Second*). It seems that the burden of this method is the overhead associated with the determination and use of so-called *query-profile*. It eliminates the necessity of free access to the global memory to read the values of the substitution matrix. Query profile is stored in the texture memory. This generates a significant overhead in the case of long input sequences, since cache misses may occur very often. These observations were confirmed by G.M. Striemer and A. Akoglu in [12]. In our paper, we show that avoiding the overhead caused by the use of *query-profiles*, and placing the substitution matrix directly in the texture memory results in higher performance.

One of the most efficient implementation of the Smith-Waterman algorithm for GPUs with CUDA 1.1 is CUDASW++ 1.0 [6], which reached up to 10 GCUPS on a single graphics card with NVIDIA GeForce GTX 280. The weak point of the implementation is that it stores the substitution matrix in the shared memory, which is affected by occasional conflicts of banks. In the CUDASW++ 1.0, texture memory is used for buffering reads of sequence elements from a database. However, performed tests showed that efficiency improvement resulting from the local cache is relatively small. A better solution is to provide consistent reads from the global memory. Then the property of local caching of texture memory can be used to achieve other objectives. Moreover, the CUDASW++ 1.0 stores the input sequence in constant memory, whose size is significantly reduced. As a result, the CUDASW++ 1.0 works for input sequences no longer than 59 000 elements [6].

At present, the most efficient implementation published for devices with the CUDA technology in version 1.2 or higher, is CUDASW++ 2.0 [7], reaching about 17 GCUPS on NVIDIA's GeForce GTX 280. In [7] Y. Liu et al. describe a completely new approach, inspired by the work of Farrar [3], and improved implementation known from [6]. In the revised implementation, they returned to the idea of *query-profile*, which is used in the SW-CUDA, but with the revised organization of the data structure. Additional efficiency was achieved by the code optimization and storing four successive elements of compared sequences in 32-bit words. A similar solution is used in our implementation presented in the following section.

4 Implementation of Parallel Alignment on GPU and CUDA

In order to quickly search large repositories of biopolymer sequences we assumed that the process will be carried out in parallel by threads performing simultaneous pairwise alignments for a query sequence and successive database sequence. In the proposed method accelerating the similarity searching of biopolymer chains each pair of sequences is aligned with the use of the Smith-Waterman algorithm by only one thread. Therefore, the implemented algorithm will be described as SW-CUDA-STSA (*STSA – Single-Thread-Single-Alignment*). We also made the implementation, in which a pair of sequences is aligned by multiple threads (SW-CUDA-MTSA). However, performed tests have shown that synchronization of threads performing single alignment badly affect the performance. In SW-CUDA-STSA particular alignments are not dependant on each other in any way, since there is no need to exchange information between any threads and the synchronization of threads is not necessary. This has a positive impact on performance. In the following sections we present the optimization steps leading to achieve a very good efficiency of the SW-CUDA-STSA.

4.1 Calculation of Alignment Matrix

Parallel and independent alignments give us a kind of freedom in determining elements of the alignment matrix. With such a freedom we can seek a better method than the simple calculation of the subsequent rows or columns, as in the original Smith-Waterman method. In order to reduce access demands to the global memory, which has a low bandwidth, the most preferred method is to perform calculations in *areas*, e.g. squares of $n \times n$ elements [6]. Given the number of registers available on our GPU, we decided to calculate areas of 4×4 elements. In order to determine the element values of such an area, it is necessary to know values of 9 elements along the left and top edge of the area, as shown in Fig. 2a-d.

Assume that successive areas in the alignment matrix are calculated by processing columns (Fig. 3), and elements of the areas are calculated by processing rows (Fig. 2). In Fig. 3a we can see that for the first area in the column the values along the top edge, which are needed in the calculation of the next row, are always equal to 0, and for next areas in the column (Fig. 3b) they are equal to the last row of the previous area. The storage of items of the upper edge in the global memory or the shared-memory would not be a good solution, since the access to the global memory is very slow and the shared-memory can be affected by conflicts of memory banks. Therefore, the best solution is to store these values in registers. The value of the upper-left corner of the area is also transferred between the iterations of the algorithm by using the register.

The values of the left edge of the area are different for each *column of areas* and equal to the values of the last column of the previous *column of areas* (Fig. 3c and Fig. 3d). The total number of these values is equal to the length of the sequence located along the vertical edge of the alignment matrix. It is advisable that, for each thread matching a pair of sequences, the number was the same, which could easily arrange for coalesced access to the global memory.

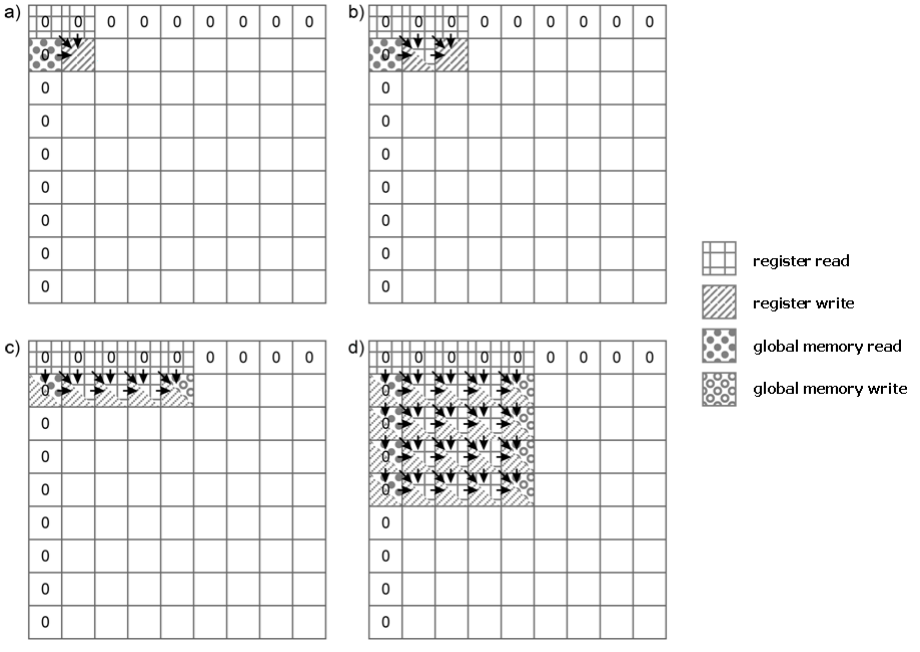


Fig. 2. Order of elements determination for a particular *area* (4×4) in the alignment matrix for the proposed method

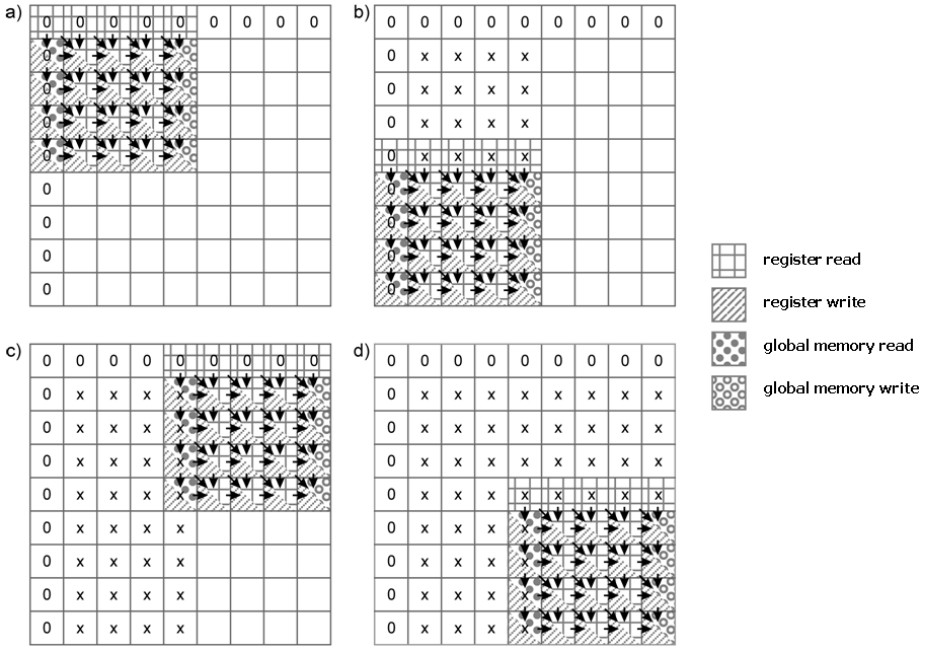


Fig. 3. Order of areas determination in the alignment matrix for the proposed method

While calculating successive *columns of areas*, we must always locate the same sequence, i.e. query sequence, along the vertical edge of the alignment matrix.

4.2 Reducing the Number of Transactions

In Fig. 2 and Fig. 3 we can also observe read and write operations to particular resources for each phase during the calculation of the alignment matrix.

When calculating values of the alignment matrix by processing areas of 4 x 4 elements, we have limited the number of accesses to the global memory, i.e. 4 reads and 4 writes for all 16 elements of the matrix area. To these 8 global memory accesses, we have to add another 8 accesses associated with reading and saving the data necessary to determine the gap penalty function G_P.

In our implementation, data transfers involving the global memory are further reduced. We just keep the four values that are read or written from/to the global memory, as values of the 4-byte data type (*int* or *float*) in the C language (128-bit word). Then the transaction mechanism of the CUDA technology ensures that such 128-bit words will be read or written by the warp threads in only two transactions, rather than four. Provided that the access is coalesced – more threads read or write subsequent 128-bit words. In summary, in our implementation the number of required accesses to the global memory is optimized.

4.3 Reducing Idle Time of Threads

In presented implementation of the Smith-Waterman algorithm, each thread performs single alignment. For concurrent operation of possible large number of threads, it is necessary to provide a large amount of data at the same time to the memory of the graphics card. Since threads are executed in blocks, the amount of required data is very roughly proportional to the number of threads in block. After the whole program is executed on the graphics card, blocks remain idle for a certain period before they act, which results from the necessity of transferring a large amount of data.

Idle time when running successive blocks can be effectively reduced by performing data transfer to the device in batches using streams and properly selecting the number of threads in the block. During tests, the best results were obtained for 64 threads in the block and a batch size of approximately 20 MB. The division of data into parts of appropriate size is completely independent of the length or form of a given query sequence. Therefore, the division can be made in advance and we can load the divided data from a file and then store these data in RAM of the host, using them to perform alignments.

Another problem is related to the organization of data that is transferred to the device. The vast majority of sequences stored in the database are relatively short. Very long sequences are usually only a small percentage of the database. This hypothetical situation is shown very simply in Fig. 4. We must remember that threads working in a block run concurrently. If for some reason the executed task takes less time for some threads in the block, these threads will not become a part of another block. Prematurely terminated threads continue to hold resources, but do not have any instructions to run. In Fig. 4a an idle interval for a short, sample database sequence is

marked with the use of a solid arrow. Therefore, sequences from the database processed by threads in one block should be of similar length. In this way, the time between the completion of the fastest and the slowest thread is shorter. We can achieve this by pre-processing a database through sorting sequences according to their lengths in descending or ascending order before the data is divided into batches and set to blocks. For simplicity, let the data packet in Fig. 4a be a subset of sequences from the database, which is processed by only one block of threads.

As shown in Fig. 4b, after sorting sequences and setting a smaller number of threads in the block (smaller batches of data) the difference in length between the longest and the shortest sequence is smaller and the resource locking time is shorter than for the case presented in Fig. 4a.

4.4 Data Arrangement

Data arrangement and appropriate access to resources of the graphics card significantly affect the performance of the algorithm.

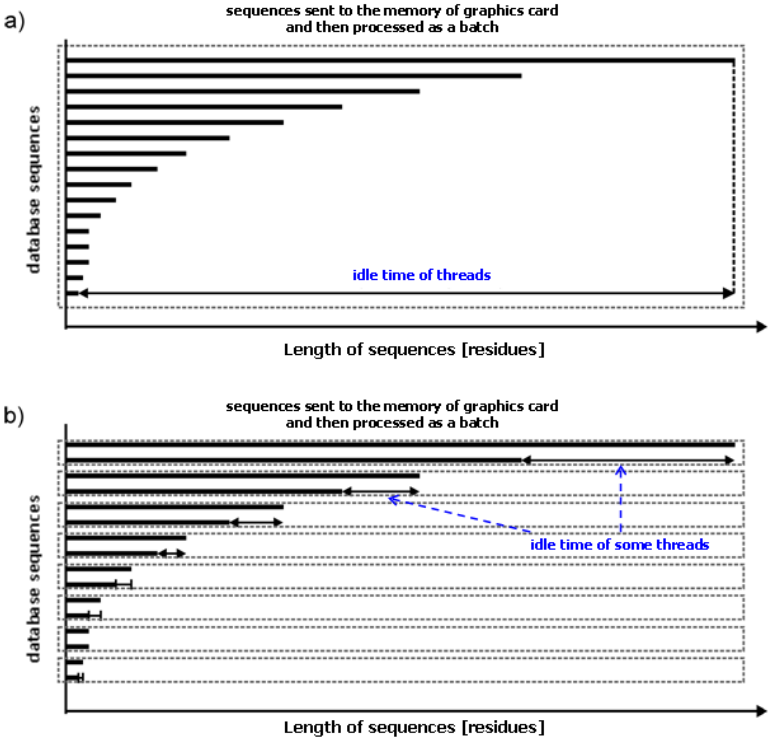


Fig. 4. Dependence of sequences length in a set of processed data for the idle time of some threads in a block: a) entire data set processed at once in one block, b) entire data set divided into smaller subsets, processed independently. The number of sequences in the subset is proportional to the number of threads in the block

In our implementation, data processed by threads in block (database sequences) are placed in the global memory, due to their amount. Access to these data can be enhanced by choosing appropriate structure of their storage in computer's memory before they are sent to the global memory of the graphics card.

In our solution, we store sequences in a two dimensional array with a row addressing, as it is presented in Fig. 5. Each cell of the array stores 32-bit words and the number of cells in row is equal to the number of block threads. Each memory cell can store four symbols of database sequence, since each symbol takes 8 bits (*char* data type in the C language). Particular database sequences are placed in columns of the array. Consecutive rows store four subsequent symbols of database sequences. Each column provides a set of data (database sequence) to be processed by a single thread. If a sequence is shorter than the longest sequence in the array, the sequence is completed by using arbitrarily chosen empty symbol with the scoring value of 0.

With a row addressing, each thread reads appropriate cell storing four characters of a sequence. In this way, we implement coalesced reads of data by all threads in a block. Moreover, in spite of reading only one single symbol of a sequence, threads read four symbols during a single access to the global memory.

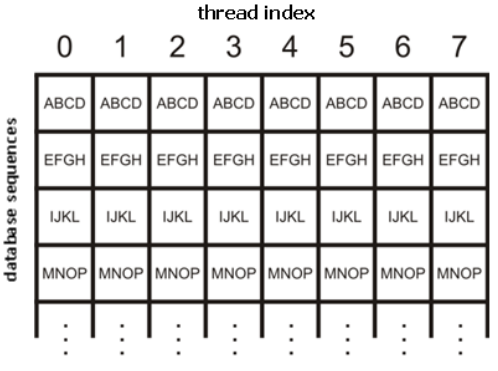


Fig. 5. Arrangement of database sequences in the global memory

4.5 Storing the Input Sequence and the Substitution Matrix

Substitution matrices are widely used in the comparison of two biopolymer sequences, especially proteins, since they evaluate the conservativeness between two elements of these sequences. In our solution, a substitution matrix that is used in the alignment process as well as the given, query sequence, are both stored in the texture memory. Texture memory is best suited for storing spatial data, and the substitution matrix can be considered as this sort of data. Input sequence is also placed in texture memory, since this type of memory is locally cached. When loading element of the sequence or array into the cache, we also load the neighboring elements. As a consequence, after reading the first symbol of the query sequence from the texture memory and loading it into the cache, the next symbol is immediately loaded in the background.

6 Efficiency Tests

We tested performance of the developed SW-CUDA-STSA algorithm and we compared its results to results obtained for different publicly available algorithms. The SW-CUDA-STSA was compared with (1) its sequential, direct implementation of the Smith-Waterman algorithm, (2) the CUDA-SW++ 1.0 algorithm [6] and running locally: (3) alignment algorithm for the SSEARCH program in the FASTA package version 3.5.4.11 [10] and (4) NCBI BLAST algorithm 2.2.23. CUDA-SW++ 2.0 [7] is currently the most popular published implementation of the Smith-Waterman algorithm, which is dedicated to the SIMT architecture. However, we have selected the CUDA-SW++ 1.0 to our studies, since the newer version of CUDA-SW++ 2.0 requires devices supporting CUDA technology version 1.2 or higher. The SSEARCH program is available in two variants. First, basic version has the optimized Smith-Waterman algorithm, about two times faster than the direct implementation [3], which also is confirmed by the results of measurements presented in Fig. 6. Second version is dedicated to Intel processors, possessing the Streaming SIMD Extensions 2 (SSE2) instruction technology.

Algorithms were evaluated on the basis of test results for 20 given sequences of the length between 127 and 2 999 amino acids, against the Swiss-Prot 2010_08 database released on 13 July 2010, consisting of 518 415 records (sequences). For testing purposes we used Zotac graphics card with the NVIDIA GeForce 8800 GT having:

- 14 multiprocessors SM (96 scalar processor cores, SP),
- 8 192 32-bit registers per multiprocessor, shared between all block threads,
- 16 kB shared memory, physically located in 16 banks with parallel access,
- 8 kB cache for constant memory, attributable to one multiprocessor,
- 6÷8 kB cache for texture memory, attributable to one multiprocessor,
- 512 MB global memory, containing areas of constant memory (64 kB), texture memory and local memory.

The graphics card that we used in tests supports the CUDA technology version 1.1. The device was installed on a PC with the Intel Core 2 Quad Q9300 2.5 GHz processor and 8 GB of RAM, running under the Microsoft Windows 7® 64-bit operating system. For all tested implementations, measurements were made with the use of the same BLOSUM62 substitution matrix and the same values of the penalty for opening a gap (-10) and gap extension (-2).

Fig. 6 shows execution times for various implementations of the Smith-Waterman algorithm for various lengths of the query sequence. Results presented in Fig. 6 show that single CPU implementation of the Smith-Waterman alignment algorithm is the slowest among other tested implementations. On the other hand, the popular BLAST algorithm appears to be the fastest. The SW-CUDA-STSA presented in the paper is faster than other implementations, like CUDASW++ 1.0, and slower only than BLAST. However, BLAST is a heuristic method, oppositely to other tested implementations that use dynamic programming. In Fig. 6 we can also observe that for all tested algorithms the execution time grows with the rising length of the given, input sequence.

Fig. 7 shows the efficiency of tested algorithms and their implementations. To remove the factor of different size of the problem, the measurements in Fig. 7 are expressed in MCUPS (*Million Cell Updates Per Second*):

$$MCUPS = \frac{p \times q}{t} \times 10^{-6}, \tag{5}$$

where: p and q are the lengths of the matched sequences, and t is the runtime in seconds.

Conclusions are similar to those presented formerly. In terms of performance, the SW-CUDA-STSA is second only to the BLAST algorithm and better than competitive CUDASW++ 1.0 and other implementations.

Analyzing the efficiency of tested algorithms presented in Fig. 7 it is also worth noting that, in contrast to other algorithms, the performance of SW-CUDA-STSA remains stable regardless the length of the input sequence.

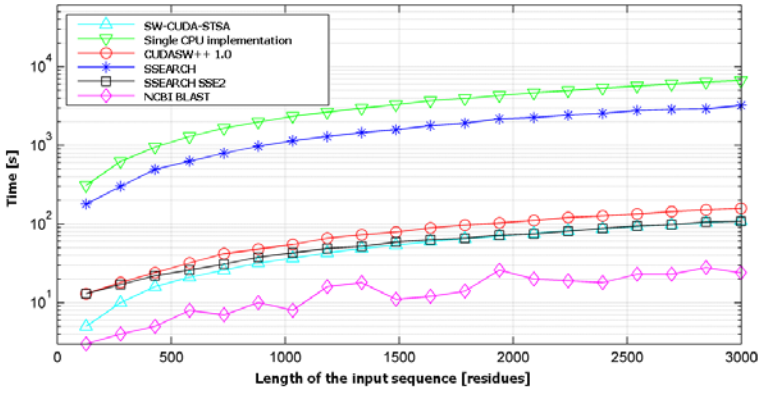


Fig. 6. Execution times for the SW-CUDA-STSA algorithm and referential algorithms for various lengths of the query sequence

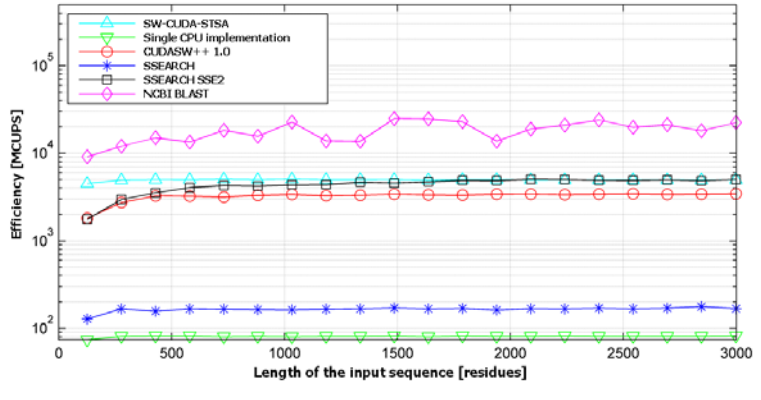


Fig. 7. Efficiency [MCUPS] of the SW-CUDA-STSA algorithm and referential algorithms for various lengths of the query sequence

This is especially visible for short input sequences (up to 300 elements). Constant performance is achieved by providing data to the GPU device on an almost continuous basis during ongoing parallel alignments. In the competing solutions, the measurements were made by first loading the entire database into memory of the graphics card, and then processing the data. In our implementation, data processing is carried out in two-stage pipeline – while the first portion of data is processed, another portion is read, prepared in the RAM of the host, and transferred to the global memory of GPU device, when needed. Processing delay, resulting from the transfer of the first batch of data is so small that we have managed to maintain a good performance even for short sequences. On the occasion of such a pipelined processing, it has another important advantage. It allows us to run the algorithm on graphics cards that do not have enough memory to load and hold the entire database of sequences.

We also calculated the acceleration of particular implementations with respect to one, chosen implementation. Implementation of the Smith-Waterman algorithm available in the SSEARCH program can be seen as a universal reference point to calculate the acceleration of other algorithms. As one of the component program of the FASTA package, this algorithm was optimized through many years and certainly takes into account most of the methods to accelerate the implementation of the Smith-Waterman algorithm for the SISD architecture. The acceleration of particular implementations referenced to the Smith-Waterman algorithm from the SSEARCH program is illustrated in Fig. 8.

The results of measurements presented in Fig. 8 show that, for the used hardware configuration, the SW-CUDA-STSA implementation described in this paper is almost thirty times faster than the implementation from the basic version of SSEARCH program and sixty times faster than standard implementation on single CPU. The implementation from the SSEARCH program using the SSE2 technology has proved to be only slightly less efficient. Moreover, for shorter input sequences SW-CUDA-STSA is much more effective than a faster version of SSEARCH. Both algorithms, SW-CUDA-STSA and SSEARCH SSE2, are about 20% faster than the fastest of the published versions of the CUDASW++ 1.0.

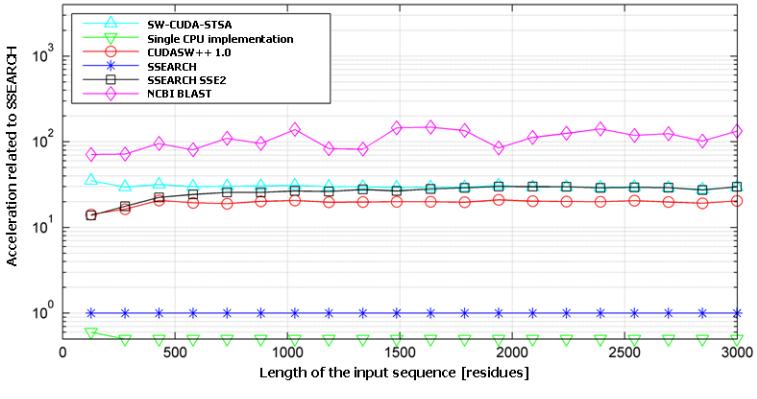


Fig. 8. Acceleration of the SW-CUDA-STSA algorithm and referential algorithms for various lengths of the query sequence. The acceleration is relative to the SSEARCH algorithm.

Such a good result of our method rises from the independence of working threads, complete elimination of synchronization, and from the data structure chosen for this algorithm, which in turn is a consequence of an attempt to reconcile the specific architecture of graphics cards supporting CUDA technology with the strategy of dynamic programming.

5 Concluding Remarks

The SW-CUDA-STSA described in the paper, although not completely novel, because it combines already known solutions, has a number of features that result in increased flexibility of the implementation and further reduction of the execution time of the Smith-Waterman algorithm.

Designing the implementation, we tried to include the best features of known solutions, minimizing the costs of the initial data processing and avoiding operations, which could provide an additional overhead.

We also noticed that there is a stronger dependency between the performance and the length of the input sequence in the CUDASW++, than in the proposed SW-CUDA-STSA. CUDASW++ reveals a clear performance decrease especially for short sequences (Fig. 7), while the proposed SW-CUDA-STSA preserves performance stability in a wide range of input sequence length. Moreover, the competitive CUDASW++ 1.0 is limited by the size of constant memory, which stores the input sequence, and which is only 64 kB in devices with CUDA 1.1. The developed solution SW-CUDA-STSA theoretically works for any sequence length. The limitation is only the amount of global memory installed on the device, which in our case is 512MB. This allows us to compare a given sequence even with whole genomes, making the SW-CUDA-STSA a powerful and effective tool in a post-genome era.

Future works will cover further acceleration of the similarity searching of biomolecular sequences by using more efficient GPUs supporting CUDA 2.1 and NVIDIA Unified Virtual Addressing on 64 bit devices. The algorithm can be also implemented on clusters with computing nodes containing general purpose GPUs for massive and parallel scanning of large biological databases. Moreover, currently our development team works on implementation of heuristic BLAST on GPU-based devices, which should additionally speed up the similarity searching and allow us to compare whole genomes to each other, as does the MUMmerGPU [13].

Acknowledgments. This scientific research was partly financed from funds for science in years 2010-2012 by a research and development project, under Grant No. O R00 0113 12, and supported by the European Union from the European Social Fund within the EkDan project.

References

1. Altschul, S.F., et al.: Basic Local Alignment Search Tool. Journal of Molecular Biology 215, 403–410 (1990)
2. Boyer, M., Skadron, K., Weimer, W.: Automated Dynamic Analysis of CUDA Programs. University of Virginia, USA (2008), http://www.nvidia.com/docs/IO/67190/stmcs08.pdf

3. Farrar, M.: Striped Smith–Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23(2), 156–161 (2007)
4. GenomeNet, http://www.genome.jp/en/db_growth.html
5. Gough, E.S., Kane, M.D.: Evaluating Parallel Computing Systems in Bioinformatics. In: Proceedings of the Third International Conference on Information Technology: New Generations, Las Vegas, NV, pp. 233–238 (2006)
6. Liu, Y., Maskell, D., Schmidt, B.: CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Research Notes 2(73), 1–10 (2009)
7. Liu, Y., Maskell, D., Schmidt, B.: CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. BMC Research Notes 3(93), 1–12 (2010)
8. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9, 1–9 (2008)
9. NVIDIA CUDA programming guide 2.3, http://developer.download.nvidia.com/com-pute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf
10. Pearson, W.R., Lipman, D.J.: Improved tools for biological sequence analysis. Proceedings of the National Academy of Sciences 85, 2444–2448 (1988)
11. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology 147, 195–197 (1981)
12. Striemer, G.M., Akoglu, A.: Sequence Alignment with GPU: Performance and Design Challenges. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS, pp. 1–10 (2009)
13. Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A.: High-Throughput Sequence Alignment Using Graphics Processing Units. BMC Bioinformatics 8(474) (2007)

Read Invisibility, Virtual World Consistency and Probabilistic Permissiveness are Compatible

Tyler Crain[2], Damien Imbs[2], and Michel Raynal[1,2]

[1] IUF,
[2] IRISA,
Université de Rennes 1, France
{Tyler.Crain,Damien.Imbs,Michel.Raynal}@irisa.fr

Abstract. The aim of a Software Transactional Memory (STM) is to discharge the programmers from the management of synchronization in multiprocess programs that access concurrent objects. To that end, an STM system provides the programmer with the concept of a transaction. The job of the programmer is to design each process the application is made up of as a sequence of transactions. A transaction is a piece of code that accesses concurrent objects, but contains no explicit synchronization statement. It is the job of the underlying STM system to provide the illusion that each transaction appears as being executed atomically. Of course, for efficiency, an STM system has to allow transactions to execute concurrently. Consequently, due to the underlying STM concurrency management, a transaction commits or aborts.

This paper studies the relation between two STM properties (read invisibility and permissiveness) and two consistency conditions for STM systems, namely, opacity and virtual world consistency. Both conditions ensure that any transaction (be it a committed or an aborted transaction) reads values from a consistent global state, a noteworthy property if one wants to prevent abnormal behavior from concurrent transactions that behave correctly when executed alone. A read operation issued by a transaction is invisible if it does not entail shared memory modifications. This is an important property that favors efficiency and privacy. An STM system is permissive (respectively probabilistically permissive) with respect to a consistency condition if it accepts (respectively accepts with positive probability) every history that satisfies the condition. This is a crucial property as a permissive STM system never aborts a transaction "for free". The paper first shows that read invisibility, probabilistic permissiveness and opacity are incompatible, which means that there is no probabilistically permissive STM system that implements opacity while ensuring read invisibility. It then shows that read invisibility, probabilistic permissiveness and virtual world consistency are compatible. To that end the paper describes a new STM protocol called IR_VWC_P. This protocol presents additional noteworthy features: it uses only base read/write objects and locks which are used only at commit time; it satisfies the disjoint access parallelism property; and, in favorable circumstances, the cost of a read operation is $O(1)$.

Keywords: Asynchronous system, Commit/abort, Opacity, Permissiveness, Serializability, Software transactional memory, Transaction, Virtual world consistency.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 244–257, 2011.

1 Introduction

1.1 Software Transactional Memory (STM) Systems

The aim of an STM system is to simplify the design and the writing of concurrent programs by discharging the programmer from the explicit management of synchronization entailed by concurrent accesses to shared objects. This means that, when considering synchronization, a programmer has to concentrate on where atomicity is required and not on the way it is realized.

More explicitly, an STM is a middleware approach that provides the programmers with the *transaction* concept [9,15]. This concept is close but different from the notion of transactions encountered in databases [4,7,8]. A process is designed as (or decomposed into) a sequence of transactions, each transaction being a piece of code that, while accessing any number of shared objects, always appears as being executed atomically. The job of the programmer is only to define the units of computation that are the transactions. He does not have to worry about the fact that the objects can be concurrently accessed by transactions. Except when he defines the beginning and the end of a transaction, the programmer is not concerned by synchronization. It is the job of the STM system to ensure that transactions execute as if they were atomic.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is irrelevant from an efficiency point of view. So, an STM system has to do "its best" to execute and commit as many transactions per time unit as possible. Similarly to a scheduler, an STM system is an on-line algorithm that does not know the future. If the STM is not trivial (i.e., it allows several transactions that access the same objects in a conflicting manner to run concurrently), this intrinsic limitation can lead it to abort some transactions in order to ensure both transaction atomicity and object consistency. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered a new transaction). Abort is the price that has to be paid by transactional systems to cope with concurrency in absence of explicit synchronization mechanisms (such as locks or event queues).

1.2 Consistency Criteria for STM Systems

In databases, the classical consistency criterion for transactions is serializability [14] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that commit. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting.

In contrast to database transactions that are usually produced by SQL queries, in an STM system the code encapsulated in a transaction is not restricted to particular patterns. Consequently a transaction always has to operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b-c$ is different from 0 in all consistent states (intuitively, a consistent state is a global state that, considering only the committed transactions, could have existed

at some real time instant). If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b = c$ (and $b = c$ defines an inconsistent state). If this occurs, the transaction throws an exception that has to be handled by the process that invoked the corresponding transaction. Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops. Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction always has to see a consistent state of the data it accesses. The aborted transactions have to be harmless.

In order to ensure that aborted transactions are harmless, consistency criteria have been defined specifically for STM systems. The first criterion aimed at STM systems is *opacity*. It was first informally suggested in [3], and then formally introduced and investigated in [6]. Another criterion, called *virtual world consistency* [12], has been designed after opacity. Virtual world consistency also prevents bad behaviors by aborted transactions, but is strictly weaker than opacity.

1.3 Desirable Properties for STM Systems

Invisible read operation. A read operation issued by a transaction is *invisible* if it does not entail the modification of base shared objects used to implement the STM system [13]. This is a desirable property for both efficiency and privacy.

Base operations and underlying locks. The use of expensive base synchronization operations such as Compare&Swap() or the use of underlying locks to implement an STM system can make it inefficient and prevent its scalability. Hence, an STM systems should use synchronization operations sparingly (or even not at all) and the use of locks should be as restricted as possible.

Disjoint access parallelism. Ideally, an STM system should allow transactions that are on distinct objects to execute without interference, i.e., without accessing the same base shared variables. This is important for efficiency and restricts the number or unnecessary aborts.

Permissiveness. The notion of permissiveness has been introduced in [5] (in some sense, it is a very nice generalization of the notion of *obligation* property [11]). It is about aborting transactions. Intuitively, an STM system is *permissive* "if it never aborts a transaction unless necessary for correctness" (otherwise it is *non-permissive*). More precisely, an STM system is permissive with respect to a consistency condition (e.g., opacity) if it accepts every history that satisfies the condition.

Some STM systems are randomized in the sense that the commit/abort point of a transaction depends on a random coin toss. Probabilistic permissiveness is suited to such systems. A randomized STM system is *probabilistically permissive* with respect to a consistency condition if every history that satisfies the condition is accepted with positive probability [5].

As indicated in [5], an STM system that checks at commit time that the values of the objects read by a transaction have not been modified (and aborts the transaction if true) cannot be permissive with respect to opacity.

1.4 Content of the Paper

This paper is on permissive STM systems with invisible reads. It has several contributions.

- It first shows that an STM system that satisfies read invisibility and opacity cannot be permissive (or probabilistically permissive).
- The paper then presents an STM system (called IR_VWC_P) that satisfies read invisibility, virtual world consistency and probabilistic permissiveness. The IR_VWC_P protocol presents additional noteworthy properties.

– It uses only base read/write operations and locks, each associated with a shared object. Moreover, a lock is used at most once by a transaction at its end (when it executes an operation called try_to_commit()).
– It satisfies the disjoint access parallelism property.

2 STM Computation Model and Base Definitions

Processes and atomic shared objects. An application is made up of an arbitrary number of processes and m shared objects. The processes are denoted p_i, p_j, etc., while the objects are denoted $X, Y, \ldots$, where each id X is such that $X \in \{1, \cdots, m\}$. Each process consists of a sequence of transactions (that are not known in advance).

Each of the m shared objects is an atomic read/write object. This means that the read and write operations issued on such an object X appear as if they have been executed sequentially, and this "witness sequence" is legal (a read returns the value written by the closest write that precedes it in this sequence) and respects the real time occurrence order on the operations on X (if $op1(X)$ terminates before $op2(X)$ starts, $op1$ appears before $op2$ in the witness sequence associated with X).

Transaction. A transaction is a piece of code that is produced on-line by a sequential process (automaton), that is assumed to be executed atomically (commit) or not at all (abort). This means that (1) the transactions issued by a process are totally ordered, and (2) the designer of a transaction does not have to worry about the management of the base objects accessed by the transaction. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or write any shared object.

The set of the objects read by a transaction defines its *read set*. Similarly the set of objects it writes defines its *write set*. A transaction that does not write shared objects is a *read-only* transaction, otherwise it is an *update* transaction. A transaction that issues only write operations is a *write-only* transaction.

Transaction are assumed to be dynamically defined. The important point is here that the underlying STM system does not know in advance the transactions. It is an on-line system (as a scheduler).

Operations issued by a transaction. We denote operations on shared objects in the following way. A read operation by transaction T on object X is denoted $X.\mathsf{read}_T()$. Such an operation returns either the value v read from X or the value $abort$. When a value v is returned, the notation $X.\mathsf{read}_T(v)$ is sometimes used. Similarly, a write

operation by transaction T of value v into object X is denoted $X.\mathsf{write}_T(v)$ (when not relevant, v is omitted). Such an operation returns either the value ok or the value $abort$. The notations $\exists\ X.\mathsf{read}_T(v)$ and $\exists\ X.\mathsf{write}_T(v)$ are used as predicates to state whether a transaction T has issued a corresponding read or write operation.

If it has not been aborted during a read or write operation, a transaction T invokes the operation $\mathsf{try\_to\_commit}_T()$ when it terminates. That operation returns $commit$ or $abort$.

Incremental snapshot. As in [1], we assume that the behavior of a transaction T can be decomposed in three sequential steps: it first reads data objects, then does local computations and finally writes new values in some objects, which means that a transaction can be seen as a software $\mathsf{read\_modify\_write}()$ operation that is dynamically defined by a process. (This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model.)

The read set is defined incrementally, which means that a transaction reads the objects of its read set asynchronously one after the other (between two consecutive reads, the transaction can issue local computations that take arbitrary, but finite, durations). We say that the transaction T computes an *incremental snapshot*. This snapshot has to be *consistent* which means that there is a time frame in which these values have coexisted (as we will see later, different consistency conditions consider different time frame notions).

If it reads a new object whose current value makes its incremental snapshot inconsistent, the transaction is directed to abort. If the transaction is not aborted during its read phase, T issues local computations. Finally, if the transaction is an update transaction, and its write operations can be issued in such a way that the transaction appears as being executed atomically, the objects of its write set are updated and the transaction commits. Otherwise, it is aborted.

Read prefix of an aborted transaction. A read prefix is associated with every transaction that aborts. This read prefix contains all its read operations if the transaction has not been aborted during its read phase. If it has been aborted during its read phase, its read prefix contains all read operations it has issued before the read that entailed the abort. Let us observe that the values obtained by the read operations of the read prefix of an aborted transaction are mutually consistent (they are from a consistent global state).

3 Consistency Conditions: Opacity and Virtual World Consistency

The opacity consistency condition. Informally suggested in [3], and formally introduced and investigated in [6], the *opacity* consistency condition requires that no transaction reads values from an inconsistent global state where, considering only the committed transactions, a *consistent global state* is defined as the state of the shared memory at some real time instant. Let us associate with each aborted transaction T its execution prefix (called *read prefix*) that contains all its read operations until T aborts (if the abort is entailed by a read, this read is not included in the prefix). An execution of a set of transactions satisfies the *opacity* condition if (i) all committed transactions plus each aborted transaction reduced to its read prefix appear as if they have been executed sequentially and (ii) this sequence respects the transaction real-time occurrence order.

Virtual world consistency. This consistency condition, introduced in [12], is weaker than opacity while keeping its spirit. It states that (1) no transaction (committed or aborted) reads values from an inconsistent global state, (2) the consistent global states read by the committed transactions are mutually consistent (in the sense that they can be totally ordered) but (3) while the global state read by each aborted transaction is consistent from its individual point of view, the global states read by any two aborted transactions are not required to be mutually consistent. Said differently, virtual world consistency requires that (1) all the committed transactions be serializable [14] (so they all have the same "witness sequential execution") or linearizable [10] (if we want this witness execution to also respect real time) and (2) each aborted transaction (reduced to a read prefix as explained previously) reads values that are consistent with respect to its causal past only.

As two aborted transactions can have different causal pasts, each can read from a global state that is consistent from its causal past point of view, but these two global states may be mutually inconsistent as aborted transactions have not necessarily the same causal past (hence the name *virtual world* consistency). This consistency condition can benefit many STM applications as, from its local point of view, a transaction cannot differentiate it from opacity.

In addition to the fact that it can allow more transactions to commit than opacity, one of the main advantages of virtual world consistency lies in the fact that, as opacity, it prevents bad phenomena (as described previously) from occurring without requiring all the transactions (committed or aborted) to agree on the very same witness execution. Let us assume that each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.) when, executed alone, it reads values from a consistent global state. As, due to the virtual world consistency condition, no transaction (committed or aborted) reads from an inconsistent state, it cannot behave incorrectly despite concurrency, it can only be aborted. This is a first class requirement for transactional memories.

4 Invisible Reads, Opacity and Permissiveness are Incompatible

Theorem 1. *Read invisibility, opacity and permissiveness (or probabilistic permissiveness) are incompatible.*

Proof: Let us first consider permissiveness. The proof follows from a simple counter-example where three transactions T_1, T_2 and T_3 issue sequentially the following operations (depicted in Figure 1).

1. T_3 reads object X.
2. Then T_2 writes X and terminates. If the STM system is permissive it has to commit T_2. This is because if (a) the system would abort T_2 and (b) T_3 would be made up of only the read of X, aborting T_2 would make the system non-permissive. Let us notice that, at the time at which T_2 has to be committed or aborted, the future behavior of T_3 is not known and T_1 does not yet exist.
3. Then T_1 reads X and Y. Let us observe that the STM system has not to abort T_1. This is because when T_1 reads X there is no conflict with another transaction, and similarly when T_1 reads Y.

4. Finally, T_3 writes Y and terminates. Let us observe that T_3 must commit in a permissive system where read operations (issued by other processes) are invisible. This is because, due to read invisibility, T_3 does not know that T_1 has previously issued a read of Y. Moreover, T_1 has not yet terminated and terminates much later than T_3. Hence, whatever the commit/abort fate of T_1, due to read invisibility, no information on the fact that T_1 has accessed Y has been passed from T_1 to T_3: when the fate of T_3 has to be decided, T_3 is not aware of the existence of T_1.

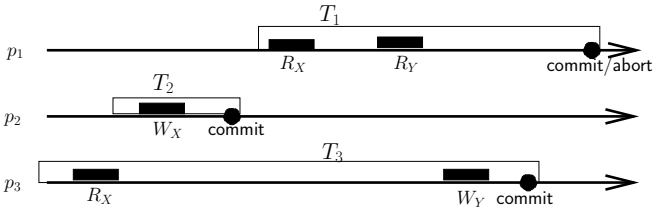


Fig. 1. Invisible reads, opacity and permissiveness are incompatible

The strong transaction history $\widehat{PO} = (\{T_1, T_2, T_3\}, \rightarrow_{PO})$ associated with the previous execution is such that:

- $T_3 \rightarrow_{PO} T_2$ (follows from the fact that T_2 overwrites the value of X read by T_3).
- $T_2 \rightarrow_{PO} T_1$ (follows from the fact that T_1 reads the value of X written by T_2). Let us observe that this is independent from the fact that T_1 will be later aborted or committed. (If T_1 is aborted it is reduced to its read prefix "X.read(); Y.read()" that obtained values from a consistent global state.)
- Due to the sequential accesses on Y that is read by T_1 and then written by T_3, we have $T_1 \rightarrow_{PO} T_3$.

It follows from the previous item that $T_1 \rightarrow_{PO} T_1$. A contradiction from which we conclude that there is no protocol with invisible read operations that both is permissive and satisfies opacity.

Let us now consider probabilistic permissiveness. The same counter-example and the same reasoning as before applies. As none of T_2 and T_3 violates opacity, a probabilistic STM system that implements opacity with invisible read operations has a positive probability of committing both of them. As read operations are invisible, there is positive probability that both read operations on X and Y issued by T_1 be accepted by the STM system. It then follows that the strong transaction history $\widehat{PO} = (\{T_1, T_2, T_3\}, \rightarrow_{PO})$ associated with the execution in which T_2 and T_3 are committed while T_1 is aborted has a positive probability to be accepted. It is trivial to see that this execution is the same as in the non-probabilistic case for which it has been shown that this history is not opaque. From this we have that read invisibility, permissiveness, and opacity are incompatible.

□ $_{Theorem\ 1}$

Remark 1. Let us observe that any opaque system with invisible reads would be required to abort T_3. When T_3 performs the try_to_commit() operation detecting that its read

of X has been overwritten, it must abort (this is because T_3 has no way of knowing whether or not T_1's read exists at this point, so T_3 must abort in order to ensure safety). From this we have that read invisibility, permissiveness, and opacity are *incompatible* in the sense that any pair of properties can be satisfied only if the third is omitted.

Remark 2. The previous proof shows that opacity is a too strong consistency condition when one wants both read invisibility and permissiveness. Differently, when considering the previous execution, the virtual world consistency protocol IR_VWC_P presented in this paper will abort transaction T_1. It is easy to see that the corresponding weak transaction history is virtual world consistent: The read prefix "$X.\mathsf{read}_{T_1}()$; $Y.\mathsf{read}_{T_1}()$" of the aborted transaction T_1 can be ordered after T_2 (and T_3 does not appear in its causal past).

5 Step 1: Ensuring Virtual World Consistency with Read Invisibility

As announced in the Introduction, the protocol IR_VWC_P is built is two steps. This section presents the first step, namely, a protocol that ensures virtual consistency with invisible read operations. The second step (Section 6) will enrich this base protocol to obtain probabilistic permissiveness.

5.1 STM Interface, Incremental Reads and Deferred Updates

The underlying system on top of which is built the STM system is made up of base shared read/write variables (also called registers) and locks. Some of the base variables are used to contain pointer values. As we will see, not all the base registers are required to be atomic. There is an exclusive lock per shared object.

The STM system provides the process that issues a transaction T with four operations. The operations $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$ have been already presented. The operation $\mathsf{begin}_T()$ is invoked by a transaction T when it starts. It initializes local control variables.

The proposed STM system is based on the incremental reads and deferred update strategy. Each transaction T uses a local working space. When T invokes $X.\mathsf{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later $X.\mathsf{read}_T()$ invocations (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory may have changed in between. As already explained, this is the *incremental snapshot* strategy.

When T invokes $X.\mathsf{write}_T(v)$, it writes v into its working space (and does not access the shared memory) and always returns ok. Finally, if T is not aborted while it is executing $\mathsf{try\_to\_commit}_T()$, it copies the values written (if any) from its local working space to the shared memory. (A similar deferred update model is used in some database transaction systems.)

5.2 The Underlying Data Structures

Implementing a transaction-level shared object. Each transaction-level shared object X is implemented by a list. Hence, at the implementation level, there is a shared array $PT[1..m]$ such that $PT[X]$ is a pointer to the list associated with X. This list is made up of cells. Let $CELL(X)$ be such a cell. It is made up of the following fields (see Figure 2).

- $CELL(X).value$ contains the value v written into X by some transaction T.
- $CELL(X).begin$ and $CELL(X).end$ are two dates (real numbers) such that the right-open time interval $[CELL(X).begin..CELL(X).end[$ defines the lifetime of the value kept in $CELL(X).value$. Operationally, $CELL(X).begin$ is the commit time of the transaction that wrote $CELL(X).value$ and $CELL(X).end$ is the date from which $CELL(X).value$ is no longer valid.
- $CELL(X).last\_read$ contains the commit date of the latest transaction that read object X and returned the value $v = CELL(X).value$.
- $CELL(X).next$ is a pointer that points to the cell containing the first value written into X after $v = CELL(X).value$. $CELL(X).prev$ is a pointer in the other direction.

It is important to notice that none of these pointers are used in the protocol (Figure 3) that ensures virtual world consistency and read invisibility. $CELL(X).next$ is required only when one wants to recycle inaccessible cells. Differently, $CELL(X).next$ will be used to obtain permissiveness (see Section 6).

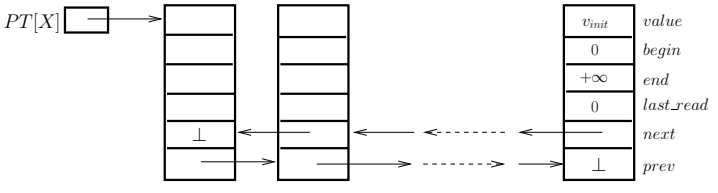


Fig. 2. List implementing a transaction-level shared object X

No field of a cell is required to be an atomic read/write register of the underlying shared memory. Moreover, all fields (but $CELL(X).last\_read$) are simple write-once registers. Initially $PT[X]$ points to a list made up of a single cell containing the tuple $\langle v_{init}, 0, +\infty, 0, \bot, \bot\rangle$, where v_{init} is the initial value of X.

Locks. A exclusive access lock is associated with each read/write shared object X. These locks are used only in the try_to_commit() operation, which means that neither $X.\mathsf{read}_T()$ nor $X.\mathsf{write}_T()$ is lock-based.

Variables local to each process. Each process p_i manages a local variable denoted $last\_commit_i$ whose scope is the entire computation. This variable (initialized to 0) contains the commit date associated with the last transaction committed by p_i. Its aim is to ensure that the transactions committed by p_i are serialized according to their commit order.

In addition to $last\_commit_i$, a process p_i manages the following local variables whose scope is the duration of the transaction T currently executed by process p_i.

- w_bot_T and w_top_T are two local variables that define the time interval during which transaction T could be committed. This interval is $]w\_bot_T..w\_top_T[$ (which means that its bounds do not belong to the interval).
  It is initially equal to $]last\_commit_i..+\infty[$. Then, it can only shrink. If it becomes empty (i.e., $w\_bot_T \geq w\_top_T$), transaction T has to be aborted.
- lrs_T (resp., lws_T) is the read (resp., write) set of transaction T. Incrementally updated, it contains the identities of the transaction-level shared objects X that T has read (resp., written) up to now.
- $lcell(X)$ is a local cell whose aim is to contain the values that have been read from the cell pointed to by $PT[X]$ or will be added to that list if X is written by T. In addition to the six previous fields, it contains an additional field denoted $lcell(X).origin$ whose meaning is as follows. If X is read by T, $lcell(X).origin$ contains the value of the pointer $PT[X]$ at the time X has been read. If X is only written by T, $lcell(X).origin$ is useless.

Notation for pointers. $PT[X]$, $cell(X).next$ and $lcell(X).origin$ are pointer variables. The following pointer notations are used. Let PTR be a pointer variable. $PTR\downarrow$ denotes the variable pointed to by PTR. Let VAR be a non-pointer variable. $\uparrow VAR$ denotes a pointer to VAR. Hence, $PTR \equiv \uparrow (PTR\downarrow)$ and $VAR \equiv (\uparrow VAR)\downarrow$).

5.3 The read$_T$() and write$_T$() Operations

When a process p_i invokes a new transaction T, it first executes the operation begin$_T$() which initializes the appropriate local variables.

The X.read$_T$() operation. The algorithm implementing X.read$_T$() is described in Figure 3. When p_i invokes this operation, if $lcell(X)$ exists, it returns the value locally saved in $lcell(X).value$ (lines 02 and 13). If $lcell(X)$ has not yet been allocated, p_i does it (line 03) and updates its fields $value$, $begin$ and $origin$ with the corresponding values obtained from the shared memory (lines 04-07). Process p_i then updates w_bot_T and w_top_T as follows.

- The algorithm defines the commit time of transaction T as a point of the time line such that T could have executed all its read and write operations instantaneously at that time. Hence, T cannot be committed before a committed transaction T' that wrote the value of a shared object X read by T. According to the algorithm implementing the try_to_commit$_T$() operation (see line 27), the commit point of such a transaction T' is the time value kept in $lcell(X).begin$. Hence, p_i updates w_bot_T to $\max(w\_bot_T, lcell(X).begin)$ (line 08). X is then added to $lrst_T$ (line 09).
- Then, p_i updates w_top_T (the top side of T's commit window, line 10). If there is a shared object Y already read by T (i.e., $Y \in lrs_T$) that has been written by some other transaction T'' (where T'' is a transaction that wrote Y after T read

operation $\mathsf{begin}_T()$:
(01) $w\_bot_T \leftarrow last\_commit_i$; $w\_top_T \leftarrow +\infty$; $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$.
==========
operation $X.\mathsf{read}_T()$:
(02) **if** ($\nexists$ local cell associated with the R/W shared object X) **then**
(03) allocate local space denoted $lcell(X)$;
(04) $x\_ptr \leftarrow PT[X]$;
(05) $lcell(X).value \leftarrow (x\_ptr\downarrow).value$;
(06) $lcell(X).begin \leftarrow (x\_ptr\downarrow).begin$;
(07) $lcell(X).origin \leftarrow x\_ptr$;
(08) $w\_bot_T \leftarrow \max(w\_bot_T, lcell(X).begin)$;
(09) $lrs_T \leftarrow lrs_T \cup X$;
(10) **for each** ($Y \in lrs_T$) **do**
$w\_top_T \leftarrow \min\big(w\_top_T, (lcell(Y).origin\downarrow).end\big)$ **end for**;
(11) **if** ($w\_bot_T \geq w\_top_T$) **then** $\mathsf{return}(abort)$ **end if**
(12) **end if**;
(13) return $(lcell(X).value)$.
==========
operation $X.\mathsf{write}_T(v)$:
(14) **if** ($\nexists$ local cell associated with X) **then** allocate local space $lcell(X)$ **end if**;
(15) $lws_T \leftarrow lws_T \cup X$;
(16) $lcell(X).value \leftarrow v$;
(17) $\mathsf{return}(ok)$.
==========
operation $\mathsf{try\_to\_commit}_T()$:
(18) lock all the objects in $lrs_T \cup lws_T$;
(19) **for each** ($Y \in lrs_T$) **do** $w\_top_T \leftarrow \min\big(w\_top_T, (lcell(Y).origin\downarrow).end\big)$ **end for**;
(20) **for each** ($Y \in lws_T$) **do** $w\_bot_T \leftarrow \max((PT[Y]\downarrow).last\_read, w\_bot_T)$ **end for**;
(21) **if** ($w\_bot_T \geq w\_top_T$) **then**
release all locks and disallocate all local cells; $\mathsf{return}(abort)$ **end if**;
(22) $ct\_time_T \leftarrow$ select a (random/heuristic) time value $\in\ ]w\_bot_T..w\_top_T[$;
(23) **for each** ($X \in lws_T$) **do** $(PT[X]\downarrow).end \leftarrow ct\_time_T$ **end for**;
(24) **for each** ($X \in lws_T$) **do**
(25) allocate in shared memory a new cell for X denoted $CELL(X)$;
(26) $CELL(X).value \leftarrow lcell(X).value$; $CELL(X).last\_read \leftarrow ct\_time_T$;
(27) $CELL(X).begin \leftarrow ct\_time_T$; $CELL(X).end \leftarrow +\infty$;
(28) $PT[X] \leftarrow\ \uparrow CELL(X)$
(29) **end for**;
(30) **for each** ($X \in lrs_T$) **do**
(31) $(lcell(X).origin\downarrow).last\_read \leftarrow \max((lcell(X).origin\downarrow).last\_read, ct\_time_T)$
(32) **end for**;
(33) release all locks and disallocate all local cells; $last\_commit_i \leftarrow ct\_time_T$;
(34) $\mathsf{return}(commit)$.

Fig. 3. Algorithm for the operations of the protocol

Y), then w_top_T has to be set to $ct\_time_{T''}$ if $ct\_time_{T''} < w\_top_T$. According to the algorithm implementing the $\mathsf{try\_to\_commit}_T()$ operation, the commit point of such a transaction T'' is the date kept in $(lcell(Y).origin \downarrow).end$. Hence, for each $Y \in lrs_T$, p_i updates w_bot_T to $\min\big(w\_top_T, (lcell(Y).origin \downarrow).end$ (line 10).

Then, if the window becomes empty, the $X.\mathsf{read}_T()$ operation entails the abort of transaction T (line 11). If T is not aborted, the value written by T' (that is kept in $lcell(X).value$) is returned (line 13).

The $X.\mathsf{write}_T(v)$ operation. The algorithm implementing this operation is described at lines 14-17 of Figure 3. If there is no local cell associated with X, p_i allocates one (line 14) and adds X to lws_T (line 15). Then it locally writes v into $lcell(X).value$ (line 16) and returns ok (line 17). Let us observe that no $X.\mathsf{write}_T()$ operation can entail the abort of a transaction.

5.4 The $\mathsf{try\_to\_commit}_T()$ Operation

The algorithm implementing this operation is described in Figure 3 (lines 18-34). A process p_i that invokes $\mathsf{try\_to\_commit}_T()$ first locks all transaction-level shared objects X that have been accessed by transaction T (line 18). The locking of shared objects is done in a canonical order in order to prevent deadlocks.

Then, process p_i computes the values that define the last commit window of T (lines 19-20). The update of w_top_T is the same as described in the $\mathsf{read}_T()$ operation. The update of w_bot_T is as follows. For each register Y that T is about to write in the shared memory (if T is not aborted before), p_i computes the date of the last read of Y, namely the date $(PT[Y] \downarrow).last\_read$. In order not to invalidate this read (whose issuing transaction has been committed), p_i updates w_bot_T to $\max((PT[Y] \downarrow).last\_read, w\_bot_T)$. If the commit window of T is empty, T is aborted (line 21). All locks are then released and all local cells are freed.

If T's commit window is not empty, it can be safely committed. To that end p_i defines T's commit time as a finite value randomly chosen in the current window $]w\_bot_T..w\_top_T[$ (let us remind that the bounds are outside the window, line 22). This time function is such that no two processes obtain the same time value.

Then, before committing, p_i has to (a) apply the writes issued by T to the shared objects and (b) update the "last read" dates associated with the shared objects it has read.

a. First, for every shared object $X \in lws_T$, process p_i updates $(PT[X] \downarrow).overwrite$ with T's commit date (line 23). When all these updates have been done, for every shared object $X \in lws_T$, p_i allocates a new shared memory cell $CELL(X)$ and fills in the four fields of $CELL(X)$ (lines 25-28). Process p_i also has to update the pointer $PT[X]$ to its new value (namely $\uparrow CELL(X)$) (line 28).
b. For each register X that has been read by T, p_i updates the field $last\_read$ to the maximum of its previous value and ct_time_T (lines 30-32). (Actually, this base version of the protocol remains correct when $X \in lrs_T$ is replaced by $X \in (lrs_T \setminus lws_T)$. As this improvement is no longer valid in the final version of the $\mathsf{try\_to\_commit}_T()$ algorithm described in Section 6, we do not consider it in this base protocol.)

Finally, after these updates of the shared memory, p_i releases all its locks, frees the local cells it had previously allocated (line 33) and returns the value $commit$ (line 34).

On the random selection of commit points. It is important to notice that, choosing randomly commit points (line 22, Figure 3), there might be "best/worst" commit points for committed transactions, where "best point" means that it allows more concurrent conflicting transactions to commit. Random selection of a commit point can be seen as an inexpensive way to amortize the impact of "worst" commit points (inexpensive because it eliminates the extra overhead of computing which point is the best).

Proof of the algorithm for VWC and read invisibility. Due to space limitations, the proof is not included here. It can be found in [2].

Improving the base protocol described in Figure 3. The base protocol presented previously can be improved on the following three points: how the useless cells can be collected, how read operations can be made fast and how serializability can be replaced by linearizability. Due to space limitations, these improvements are not included here. They can be found in [2].

6 Step 2: Adding Probabilistic Permissiveness to the Protocol

The final IR_VWC_P protocol ensures virtual world consistency, read invisibility and probabilistic permissiveness. It is proved correct. Due to space limitations, this final protocol is not presented here. It can be found in [2].

7 Conclusion

This paper has investigated the relation linking read invisibility, permissiveness and two consistency conditions, namely, opacity and virtual world consistency. It has shown that read invisibility, probabilistic permissiveness and virtual world consistency are compatible. To that end an appropriate STM protocol has been designed and proved correct. Interestingly enough, this new STM protocol has additional noteworthy features: (a) it uses only base read/write operations and a lock per object that is used at commit time only and (b) satisfies the disjoint access parallelism property.

Acknowledgements. This research is part of the TRANSFORM project, a Marie Curie project funded by the European Community's Seventh Framework Programme (grant agreement n° 238639), devoted to the theory of software transactional memories. We also thank Hagit Attiya and Sandeep Hans for their constructive comments on a draft of this paper.

References

1. Bernstein, P.A., Shipman, D.W., Wong, W.S.: Formal Aspects of Serializability in Database Concurrency Control. IEEE Transactions on Software Engineering SE-5(3), 203–216 (1979)
2. Crain, T., Imbs, D., Raynal, M.: Read Invisibility, Virtual World Consistency and Permissiveness are Compatible. Tech Report #1958, IRISA, Univ. de Rennes 1, France (November 2010)

3. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
4. Felber, P., Fetzer, C., Guerraoui, R., Harris, T.: Transactions are coming Back, but Are They The Same? ACM Sigact News, DC Column 39(1), 48–58 (2008)
5. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in Transactional Memories. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305–319. Springer, Heidelberg (2008)
6. Guerraoui, R., Kapałka, M.: On the Correctness of Transactional Memory. In: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2008), pp. 175–184. ACM Press, New York (2008)
7. Harris, T., Cristal, A., Unsal, O.S., Ayguade, E., Gagliardi, F., Smith, B., Valero, M.: Transactional Memory: an Overview. IEEE Micro 27(3), 8–29 (2007)
8. Herlihy, M.P., Luchangco, V.: Distributed Computing and the Multicore Revolution. ACM SIGACT News, DC Column 39(1), 62–72 (2008)
9. Herlihy, M.P., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-free Data Structures. In: Proc. 20th ACM Int'l Symposium on Computer Archictecture (ISCA 1993), pp. 289–300 (1993)
10. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
11. Imbs, D., Raynal, M.: Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 67–78. Springer, Heidelberg (2008)
12. Imbs, D., Raynal, M.: A versatile STM protocol with Invisible Read Operations that Satisfies the Virtual World Consistency Condition. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 266–280. Springer, Heidelberg (2010)
13. Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisentatt, D., Scherer III, W.N., Scott, M.L.: Lowering the Overhead of Software Transactional Memory. In: Proc. 1st ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing, TRANSACT 2006 (2006)
14. Papadimitriou, C.H.: The Serializability of Concurrent Updates. Journal of the ACM 26(4), 631–653 (1979)
15. Shavit, N., Touitou, D.: Software Transactional Memory. Distributed Computing 10(2), 99–116 (1997)

Parallel Implementations of Gusfield's Cut Tree Algorithm

Jaime Cohen[1,2], Luiz A. Rodrigues[1,3], Fabiano Silva[1], Renato Carmo[1], André L.P. Guedes[1], and Elias P. Duarte Jr.[1]

[1] Federal University of Paraná, Department of Informatics
Curitiba, Brazil
[2] Paraná State University at Ponta Grossa, Department of Informatics
Ponta Grossa, Brazil
[3] Western Paraná State University, Department of Computer Science
Cascavel, Brazil
{jaime,larodrigues,fabiano,elias,renato,andre}@inf.ufpr.br

Abstract. This paper presents parallel versions of Gusfield's cut tree algorithm. Cut trees are a compact representation of the edge-connectivity between every pair of vertices of an undirected graph. Cut trees have many applications in combinatorial optimization and in the analysis of networks originated in many applied fields. However, surprisingly few works have been published on the practical performance of cut tree algorithms. This paper describes two parallel versions of Gusfield's cut tree algorithm and presents extensive experimental results which show a significant speedup on most real and synthetic graphs in our dataset.

Keywords: Graph edge-connectivity, Cut tree algorithms, MPI, OpenMP.

1 Introduction

A cut tree is an important combinatorial structure able to represent the edge-connectivity between all pairs of nodes of undirected graphs. Cut trees have many direct applications, e.g. [21,2,19,23], and algorithms for cut tree construction are used as subroutines to solve other important combinatorial problems in areas such as routing, graph partitioning and graph connectivity, e.g. [22,9,18,14].

Despite of the numerous applications of cut trees, few studies were done on the practical performance of cut tree algorithms. Distributed or parallel implementations of cut tree algorithms are not available and experimental studies of such implementations have not yet been published.

This paper describes two parallel versions of Gusfield's cut tree algorithm and presents experimental results that show a significant speedup on most real and synthetic graphs in our dataset.

The sequential Gusfield's algorithm consists of $n-1$ calls to a maximum flow algorithm and the parallel version optimistically makes those calls in parallel. Even though the iterations may depend on previous ones, the experiments show that this a priori assumption rarely affects the performance of the algorithm and

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 258–269, 2011.

a significant speedup can be achieved. The implementations were tested using real and synthetic graphs representing potential applications.

This paper is organized as follows. In Section 2 we present an overview of the previous research on cut trees. Section 3 gives basic graph theory definitions including the definition of cut trees. Section 4 describes Gusfield's algorithm in its sequential and parallel versions. Section 5 defines the environment and the parameters used in the experiments. The results are presented and discussed in Section 6. Finally, in Section 7, we present the conclusions and future work.

2 Related Work

The concept of cut trees and a cut tree construction algorithm were first discovered by R. E. Gomory and T. C. Hu in 1961 [13]. Another cut tree algorithm was discovered by D. Gusfield in 1990 [15]. Let us call them *GH algorithm* and *Gus algorithm*, respectively. Both algorithms require the computation of $n-1$ minimum s-t-cuts (or, equivalently, maximum flows). We will discuss their differences in Section 4. The fastest deterministic maximum flow algorithm, by A. V. Goldberg and S. Rao [10], runs in time $\tilde{O}(\min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\}m)$. An extra $O(n)$ factor gives the worst case time complexity of the best deterministic algorithm to find a cut tree of a weighted undirected graph.

The only published experimental study on cut tree algorithms is the paper "Cut Tree Algorithms" by A. V. Goldberg and K. Tsioutsiouliklis [12]. They compared the GH algorithm and Gus algorithm. They concluded that an optimized version of the GH algorithm is more robust than Gus algorithm, justified by the fact that at a few instances, Gus algorithm had a much worse running time. We note, however, that most of those instances belong to synthetic classes of graphs for which balanced cuts exist by construction. Despite of that, their implementation of Gus algorithm was the fastest one more times than any of their implementations of GH algorithm.

3 Definitions

A *graph* G is a pair $(V(G), E(G))$ where $V(G)$ is a finite set of elements called vertices and $E(G)$ is a set of unordered pairs of vertices called edges. A *capacitated graph* is a graph G associated with a function $c : E(G) \to \mathbb{Z}_+$ defining the *capacities* of the edges in $E(G)$.

Let G be a capacitated graph. A *cut* of G is a bipartition of $V(G)$. The cut *induced* by a set $X \subset V(G)$ is the bipartition $\{X, \overline{X}\}$ of $V(G)$ induced by X, where $\overline{X} = V(G) - X$. The set $E_G(X, \overline{X}) = \{\{u, v\} \in E(G) \colon u \in X, v \in \overline{X}\}$ contains the edges that *cross* the cut $\{X, \overline{X}\}$. The *capacity* of the cut $\{X, \overline{X}\}$ is $c(X, \overline{X}) = \sum_{e \in E_G(X,\overline{X})} c(e)$.

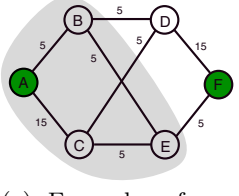
Let s and t be two vertices of G. An *s-t-cut* of G is a cut $\{X, \overline{X}\}$ such that $s \in X$ and $t \in \overline{X}$. A *minimum s-t-cut* is an s-t-cut of minimum capacity. A cut $\{\{s\}, V - \{s\}\}$ is called a *trivial cut*. The *local connectivity between s and t* in

$V(G)$, denoted by $\lambda_G(s,t)$, is the capacity of a minimum s-t-cut. Any maximum flow algorithm for directed graphs can be used to compute the local connectivity in undirected graphs using the reduction that transforms each undirected edge into two antiparallel edges.

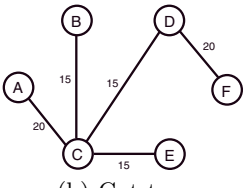
All Pairs Minimum Connectivity. Consider the problem of finding the local connectivity between all pairs of vertices of an undirected graph. The naive solution consists of running $\binom{n}{2}$ maximum flow algorithms, one for each pair of vertices. R. E. Gomory e T. C. Hu [13] showed that only $n-1$ maximum flow computations are necessary. The solution to the problem consists of constructing a weighted tree that represents the values of all pairwise local connectivities.

A *flow equivalent tree* of a graph G is a capacitated tree T with vertex set $V(G)$ such that for all $u, v \in V(G)$ the minimum capacity of an edge on the path between u and v in T equals the local connectivity $\lambda_G(u,v)$, i.e., $\lambda_T(u,v) = \lambda_G(u,v)$, for all $u, v \in V(G)$.

A *cut tree* is a flow equivalent tree T such that the cut induced by removing an edge of minimum weight from the path between u and v is a minimum u-v-cut of G, for all $u, v \in V(G)$. Cut trees are also called *Gomory-Hu trees* [20].



(a) Example of an undirected weighted graph and a minimum A-F-cut



(b) Cut tree

Fig. 1. Examples of an undirected graph, a minimum cut and a cut tree

Fig. 1(b) shows a cut tree of the graph of Fig. 1(a). Figure 1(a) shows a minimum cut between vertices A and F induced by the removal of the edge $\{C, D\}$ from the tree.

4 Cut Tree Algorithms

Two cut tree algorithms for weighted undirected graphs are known: the *Gomory-Hu's* algorithm [13] and *Gusfield's* algorithm [15]. Both algorithms make $n-1$ calls to a maximum flow algorithm and they use a divide and conquer approach. The algorithms differ in their data structure: while the former algorithm contracts the original graph, the latter computes all cuts on the input graph. After describing Gus algorithm and comparing it with GH algorithm, we discuss the choice to parallelize Gus algorithm.

4.1 Gusfield's Algorithm - Sequential Version

The sequential Gus algorithm consists of $n-1$ iterations, each of them containing a call to a maximum flow algorithm on the input graph. See the pseudocode in Algorithm 1. Initially, the tree is a star with all vertices pointing to node 1 (lines 1-2). At each iteration (lines 3-6), the algorithm chooses a different source vertex s, $s \geq 2$. This choice determines the destination vertex t as the current neighbor of s in the tree. Then, using a maximum flow algorithm, a minimum s-t-cut is found. The tree is reshaped as follows: every neighbor t' of t, $t' > s$, that sits on the side of s of the cut gets disconnected from t and gets connected to s. The algorithm ends when each node from 2 to n has been the source of an iteration. The implementation of the algorithm is simple and requires no changes in the maximum flow algorithm. This version of the algorithm finds a flow equivalent tree. A small change in the algorithm causes it produce a cut tree: in line 8, allow any neighbor of t that belongs to X to become a neighbor of s.

Algorithm 1. Sequential Gusfield's Algorithm

Input: $G = (V_G, E_G, c)$ is a weighted graph
Output: $T = (V_T, E_T, f)$ is a flow equivalent tree of G
1: **for** $i = 1$ to $|V_G|$ **do**
2: $tree_i \leftarrow 1$
// $|V_G| - 1$ maximum flow iterations
3: **for** $s \leftarrow 2$ to $|V_G|$ **do**
4: $t \leftarrow tree_s$
5: $flow_s \leftarrow$ max-flow(s, t)
6: $\{X, \overline{X}\} \leftarrow$ minimum s-t-cut
// update the tree
7: **for** $u \in V_G,\ u > s$ **do**
8: **if** $tree_u = t$ **and** $u \in X$ **then**
9: $tree_u \leftarrow s$
// return the flow equivalent tree
10: $V_T \leftarrow V_G$
11: $E_T \leftarrow \emptyset$
12: **for** $s \leftarrow 2$ to $|V_G|$ **do**
13: $E_T \leftarrow E_T \cup \{s, tree_s\}$
14: $f(\{s, tree_s\}) \leftarrow flow_s$
15: **return** $T = (V_T, E_T, f)$

4.2 Parallelization of Cut Tree Algorithms

A parallel solution to the cut tree problem involves two choices: the algorithm to implement (Gus or GH algorithm) and the level of parallelism to explore.

First, we will argue that Gus algorithm is a better choice for parallelization. GH algorithm is similar to Gus algorithm but after finding a minimum s-t-cut, it contracts each side of the cut and recurse on each of the graphs obtained.

The existence of balanced minimum cuts favors the GH algorithm because the graph size is reduced at each iteration. Most real graphs and graphs generated by random models such as Erdös-Rényi (ER) model and the preferential attachment model rarely have balanced cuts. That means not only that the subproblems do

not get much smaller, but also that load balancing among processors does not occur. Indeed, in our preliminary experiments with the sequential implementations used in [12], Gus algorithm outperformed GH algorithm on all large real graphs in our dataset and all graphs generated by the ER model and the BA model. This can only happen when balanced cuts are rare or nonexistent.

Gus algorithm always executes maximum flow algorithms on the same input graph. Therefore, each process can simply start with a copy of the graph and no further transmissions of the input graph are necessary. The interprocess communication only comprises of source and destination pairs, cuts and connectivity values. On the other hand, a parallel GH algorithm would require more interprocess communication because either the graphs or the contracted subsets of vertices should be sent from the master to the slaves threads or processes.

For all these reasons, it seems that Gus algorithm is more amenable to parallelization than GH algorithm. Nonetheless, further research can be done to explore possible ways to parallelize GH algorithm to produce a faster algorithm for graphs that contain balanced cuts. A hybrid algorithm is another possibility.

With respect to the level of parallelism to explore, we note that the maximum flow problem is hard to parallelize. Despite of extensive research on the problem, experimental studies of parallel max flow algorithms report very modest speedups [3,16] due to the synchronization requirements. On the other hand, Gus algorithm can run one sequential maximum flow per thread or process, without synchronization, and achieve high speedups, as we will see in Section 6.

4.3 MPI Version

MPI has emerged in the early 1990s as a set of libraries for process management and message exchange in distributed memory architectures [6]. The advantage of this solution is scalability, because it uses independent computers that can be easily connected through the network. However, in comparison with OpenMP, MPI requires a larger reorganization of the code sequence to obtain the parallel solution, usually based on the master/slave model.

The pseudocode of the MPI implementation appears in Algorithm 2. The master process is $proc_0$ and the slaves are $proc_1, .., proc_{p-1}$. Each process maintains a copy of the input graph. We assume that $V_G = \{1, 2, 3, \ldots, |V_G|\}$. The master creates the tasks and sends them to the slaves (lines 5, 15 and 18). Each task contains the source and the destination nodes, s and t, inputs of the maximum flow algorithm. When a slave finishes a task, it sends the value of the maximum flow and the cut to the master. Based on these data, the master may update the tree if s is still a neighbor of t (line 9). This is done in the same way as the sequential algorithm. If s and t are not neighbors by the time of processing the task result, then we say that the task "failed" and another task having s as the source is produced (line 18). The stop condition of the `while` loop in line 7 guarantees that $|V_G| - 1$ successful receives are executed.

The structure of the graph influences the number of failed tasks. If the s-t-cut $\{X, \overline{X}\}$ is such that X is small, the tree suffers few changes. The speedup of the parallel execution depends on the number of tasks that fails.

Algorithm 2. MPI Gusfield's Algorithm

Input: $G = (V_G, E_G, c)$, $proc_j$ processors $(0 \leq j < P)$
Output: $T = (V_T, E_T, f)$ is a flow equivalent tree of G

```
 1: if proc_j = 0 then // master process
 2:    for i ← 1 to |V_G| do
 3:       tree_i ← 1
 4:    for s ← 2 to P do
 5:       send Task(s, tree_s) to proc_{s-1}
 6:    s ← P + 1
 7:    while s < P + |V_G| do
 8:       receive result s', t', flow, {X, X̄} from proc_j
 9:       if tree_{s'} = t' then // update the tree
10:          flow_{s'} = flow
11:          for all u ∈ V_G, u > s' do
12:             if tree_u = t' and u ∈ X then
13:                tree_u ← s'
14:          if s ≤ |V_G| then
15:             send a new Task(s, tree_s) to proc_j
16:          s ← s + 1
17:       else // failed, try again
18:          send Task(s', tree_{s'}) to proc_j
      // Build T as in lines 10-14 of the sequential algorithm
19:    return T
20: else // slave processes
21:    while more tasks do
22:       receive Task(s, t)
23:       flow, {X, X̄} ← MaxFlow(s, t)
24:       send s, t, flow, {X, X̄} to proc_0
```

4.4 OpenMP Version

OpenMP is an API (Application Programming Interface) designed for parallel programming on shared memory architectures (SMP). This API offers policies that can be added to code sequences in Fortran, C and C++ which define how work is shared among threads to be executed on different processors/cores and how data on shared memory is accessed [6].

The adaptation of Gusfield's algorithm to OpenMP was done by the parallelization of the main loop which performs the $n-1$ calls to the maximum flow routine. See Algorithm 3. Let k be a predefined maximum number of threads. The algorithm uses an optimistic strategy and finds k minimum s-t-cuts in parallel. Each of these cuts is a candidate for changing the tree. After computing its s-t-cut, each thread verifies if the destination t is still a neighbor of s in the tree. If it is not, meaning that the cut of a previous successful thread separated s from t, then we say that the thread "failed" and another s-t-cut is computed in order to separate s from its new neighbor. If the thread succeeds in separating s from t, it proceeds to update the tree. The test and the operations on the tree are done inside a critical region, what guarantees the correctness of the algorithm.

Algorithm 3. OpenMP Gusfield's Algorithm

Input: $G = (V_G, E_G, c)$, P the number of processors
Output: $T = (V_T, E_T, f)$ is a flow equivalent tree of G

```
 1: for i ← 1 to |V_G| do
 2:    tree_i ← 1
    // |V_G|−1 maximum flow iterations
 3: for s ← 2 to |V_G| in parallel do
 4:    proc_j is an idle process
 5:    succeed ← false
 6:    repeat
 7:       t ← tree_s
 8:       proc_j executes flow, {X, X̄} ← MaxFlow(s, t)
          // omp critical region
 9:       if tree_s = t then
10:          proc_j updates the tree
11:          succeed ← true
12:    until succeed
    // Build T as in lines 10-14 of the sequential algorithm
13: return T
```

5 Experimental Setup

The experiments with MPI ran on a cluster with 15 Intel Core 2 Quad 2.4 GHz, 2 Gbyte memory and 4096 Kbyte cache, interconnected by a Gigabit Ethernet network. The experiments with OpenMP used a Quad-Core 2.8 GHz AMD Opteron workstation with 8 cores, 8 Gbyte memory and 512 Kbyte cache. The code was written in C language and compiled with `gcc` (optimization level `-O3`). Our implementations are based on the push-relabel maximum flow algorithm [11] code HIPR[1], developed by B.V. Cherkassky and A.V. Goldberg [8]. We implemented the flow equivalent tree version of Gus algorithm.

The dataset was composed of 10 graphs as shown in Table 1. The first four graphs come from real data: 2 collaboration networks [17,4], a power grid network [24] and a peer-to-peer network [17]. Two networks were generated by random models: the Erdös-Rényi (ER) model [5] and the preferential attachment model [1]. The other 4 graphs are synthetic graphs of different types that have been used as benchmarks for min cut and cut tree algorithms [7,12].

Table 1. Sizes of the graphs in the dataset

Graph	$\|V\|$	$\|E\|$
CA-HepPh	11,204	235,238
GeoComp	3,621	9,461
PowerGrid	4,941	6,594
P2P-Gnutella	10,876	39,994
BA	10,000	49,995

Graph	$\|V\|$	$\|E\|$
ER	10,000	49,841
Dblcyc	1,024	2,048
NOI	1,500	562,125
Path	2,000	21,990
Tree	1,500	563,625

[1] Owned by IG Systems, Inc. Copyright 1995-2004. Freely available for research purposes.

Speedups were calculated as $S = T_S/T_P$, where T_S is the time of the sequential implementation of Gusfield's algorithm and T_P is the execution time in parallel on P processes. The efficiency was calculated using $E = S/P$.

6 Experimental Results

Initial tests were performed in order to define the best scheduling strategies for the tasks. For MPI, the best results were achieved running the master process and one slave in one machine and the other slaves in individual machines. For OpenMP, the loop scheduling with best performance was the OpenMP dynamic scheduling that distributes tasks during runtime in order to balance the load. All further experiments used these strategies.

Results with MPI. Let P be the number of processes. For $P = 2$, the algorithm runs the master and 1 slave process. Because the slave waits for a task, the execution is sequential and, therefore, $S_2 \leq 1$ and $E_2 \leq 0.50$. For $P > 2$, at least one of the $P-1$ slaves waits while the master tries to update the tree. Let us approximate the sequential time by $T_s = T_F + T_T$, where T_F is the time to compute $|V|-1$ maximum flows and T_T is the time to update the tree. A lower bound on the parallel time can be computed by assuming the best case where the maximum flow work is divided evenly by the $P-1$ slave processes. Each slave takes $\frac{T_F}{P-1}$ time to compute maximum flows and waits idle approximately $\frac{T_T}{P-1}$ time for new tasks. Therefore, the parallel running time with P processes is bounded by $T_P \geq \frac{T_F+T_T}{P-1}$. The speedup can be upper bounded by $P-1$, because

$$S_P = \frac{T_s}{T_P} \leq \frac{T_F + T_T}{\frac{T_F+T_T}{P-1}} = P - 1.$$

An upper bound for the efficiency is $E_P = \frac{S_P}{P} \leq \frac{P-1}{P}$.

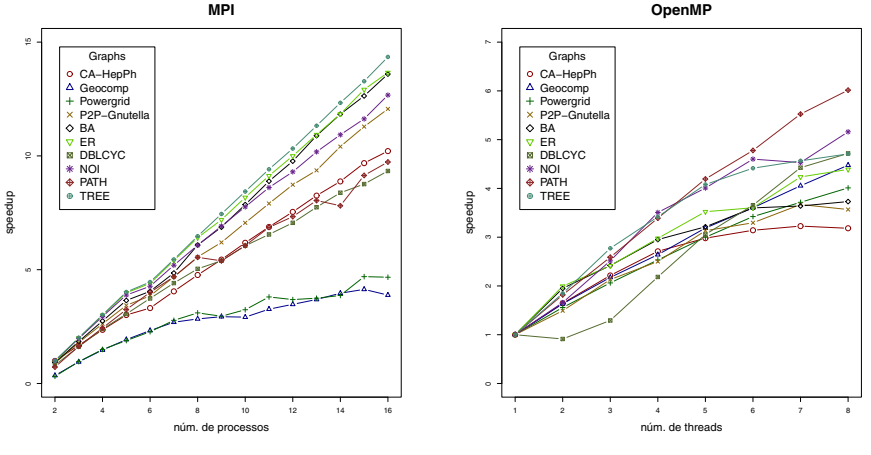


Fig. 2. Speedups with MPI e OpenMP

Table 2. MPI results with the running times, speedups (S) and efficiency (E) of each graph in the dataset

Num procs	CA-HepPh time	S	E	Geocomp time	S	E	Powergrid time	S	E	P2P-Gnutella time	S	E	BA time	S	E
sequential	475.01	-	-	2.25	-	-	3.85	-	-	65.23	-	-	87.71	-	-
2	479.93	0.99	0.49	6.45	0.35	0.17	12.31	0.31	0.16	73.24	0.89	0.45	95.49	0.92	0.46
3	283.42	1.68	0.56	2.37	0.95	0.32	4.01	0.96	0.32	36.25	1.80	0.60	47.58	1.84	0.61
4	201.56	2.36	0.59	1.53	1.47	0.37	2.57	1.50	0.37	24.88	2.62	0.66	31.94	2.75	0.69
5	157.89	3.01	0.60	1.17	1.93	0.39	2.04	1.89	0.38	19.01	3.43	0.69	24.04	3.65	0.73
6	143.35	3.31	0.55	0.97	2.32	0.39	1.69	2.27	0.38	16.94	3.85	0.64	21.66	4.05	0.67
7	117.33	4.05	0.58	0.84	2.69	0.38	1.39	2.77	0.40	13.92	4.69	0.67	18.07	4.85	0.69
8	99.64	4.77	0.60	0.79	2.84	0.35	1.24	3.10	0.39	11.73	5.56	0.69	14.42	6.08	0.76
9	87.25	5.44	0.60	0.77	2.94	0.33	1.30	2.96	0.33	10.52	6.20	0.69	12.77	6.87	0.76
10	76.84	6.18	0.62	0.77	2.91	0.29	1.19	3.24	0.32	9.24	7.06	0.71	11.16	7.86	0.79
11	69.05	6.88	0.63	0.69	3.27	0.30	1.01	3.80	0.35	8.25	7.91	0.72	9.87	8.89	0.81
12	63.01	7.54	0.63	0.65	3.48	0.29	1.04	3.69	0.31	7.47	8.73	0.73	8.98	9.77	0.81
13	57.54	8.26	0.64	0.61	3.69	0.28	1.03	3.75	0.29	6.97	9.36	0.72	8.06	10.89	0.84
14	53.50	8.88	0.63	0.57	3.97	0.28	0.99	3.87	0.28	6.27	10.41	0.74	7.41	11.83	0.85
15	49.06	9.68	0.65	0.54	4.14	0.28	0.82	4.70	0.31	5.78	11.29	0.75	6.94	12.63	0.84
16	46.52	10.21	0.64	0.58	3.88	0.24	0.82	4.67	0.29	5.41	12.06	0.75	6.45	13.60	0.85

Num procs	DBLCYC time	S	E	ER time	S	E	NOI time	S	E	PATH time	S	E	TREE time	S	E
sequential	11.13	-	-	104.23	-	-	384.84	-	-	5.42	-	-	236.78	-	-
2	13.83	0.81	0.40	109.90	0.95	0.47	385.61	1.00	0.50	7.52	0.72	0.36	237.10	1.00	0.50
3	6.88	1.62	0.54	52.40	1.99	0.66	194.59	1.98	0.66	3.32	1.63	0.54	117.89	2.01	0.67
4	4.66	2.39	0.60	34.82	2.99	0.75	130.75	2.94	0.74	2.22	2.44	0.61	78.53	3.02	0.75
5	3.65	3.05	0.61	26.17	3.98	0.80	99.07	3.88	0.78	1.67	3.25	0.65	58.90	4.02	0.80
6	2.98	3.74	0.62	23.70	4.40	0.73	90.11	4.27	0.71	1.35	4.00	0.67	53.11	4.46	0.74
7	2.52	4.41	0.63	19.30	5.40	0.77	74.08	5.20	0.74	1.16	4.69	0.67	43.45	5.45	0.78
8	2.21	5.04	0.63	16.27	6.40	0.80	63.36	6.07	0.76	0.98	5.54	0.69	36.59	6.47	0.81
9	2.06	5.39	0.60	14.49	7.19	0.80	55.65	6.92	0.77	1.01	5.39	0.60	31.79	7.45	0.83
10	1.84	6.05	0.60	12.75	8.17	0.82	49.59	7.76	0.78	0.89	6.06	0.61	28.07	8.43	0.84
11	1.70	6.55	0.60	11.42	9.13	0.83	44.70	8.61	0.78	0.79	6.86	0.62	25.16	9.41	0.86
12	1.58	7.06	0.59	10.43	10.00	0.83	41.38	9.30	0.77	0.74	7.33	0.61	22.94	10.32	0.86
13	1.44	7.75	0.60	9.53	10.93	0.84	37.81	10.18	0.78	0.67	8.04	0.62	20.91	11.32	0.87
14	1.33	8.38	0.60	8.80	11.84	0.85	35.21	10.93	0.78	0.69	7.81	0.56	19.20	12.33	0.88
15	1.27	8.76	0.58	8.07	12.92	0.86	33.10	11.63	0.78	0.59	9.14	0.61	17.83	13.28	0.89
16	1.19	9.34	0.58	7.63	13.66	0.85	30.37	12.67	0.79	0.56	9.73	0.61	16.50	14.35	0.90

Results of the MPI implementation appear in Table 2 and in Fig. 2. The running times are the average of 10 runs. Efficiencies for the instances TREE and ER achieve the upper bound $\frac{P-1}{P}$ for P between 2 and 5 and they are not far from the maximum for greater values of P.

The speedups are consistently high for all graphs but POWERGRID and GEOCOMP which are the easiest instances for the sequential algorithm. The efficiency drops as the number of fails increases as shown in Fig. 3.

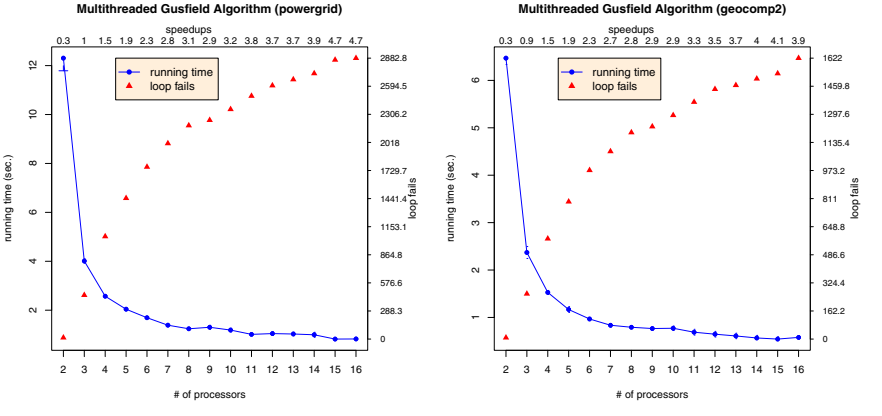
Results with OpenMP. Results for the parallel Gusfield's algorithm with OpenMP on a 8-core computer are reported in Table 3 and in Fig. 2. Efficiency was above 0.50 on most executions.

The best speedup obtained with OpenMP and 8 threads in real graphs was 4.48 and in synthetic graphs was 6.01. The worst speedup in real and synthetic graphs for 8 threads were 3.18 and 3.73, respectively. We report other positive results on a 16-core computer where the implementation achieved the best speedup of 9.4 running 16 threads on the NOI5 graph. An experiment on an ER

Table 3. OpenMP results with the running times, speedups (S) and efficiency (E) of each graph in the dataset

Num threads	CA-HepPh time	S	E	Geocomp time	S	E	Powergrid time	S	E	P2P-Gnutella time	S	E	BA time	S	E
1	517.07	-	-	2.10	-	-	3.03	-	-	65.79	-	-	87.66	-	-
2	314.32	1.65	0.82	1.29	1.63	0.82	1.94	1.56	0.78	44.10	1.49	0.75	44.94	1.95	0.98
3	233.42	2.22	0.74	0.97	2.17	0.72	1.47	2.06	0.69	30.87	2.13	0.71	36.24	2.42	0.81
4	190.77	2.71	0.68	0.80	2.64	0.66	1.20	2.53	0.63	26.31	2.50	0.63	29.71	2.95	0.74
5	173.56	2.98	0.60	0.66	3.19	0.64	1.01	3.00	0.60	21.00	3.13	0.63	27.29	3.21	0.64
6	164.59	3.14	0.52	0.58	3.60	0.60	0.88	3.42	0.57	19.96	3.30	0.55	24.35	3.60	0.60
7	160.22	3.23	0.46	0.52	4.05	0.58	0.82	3.72	0.53	17.93	3.67	0.52	24.08	3.64	0.52
8	162.48	3.18	0.40	0.47	4.48	0.56	0.76	4.01	0.50	18.44	3.57	0.45	23.50	3.73	0.47

Num threads	DBLCYC time	S	E	ER time	S	E	NOI time	S	E	PATH time	S	E	TREE time	S	E
1	10.12	-	-	115.16	-	-	585.37	-	-	5.60	-	-	315.09	-	-
2	11.09	0.91	0.46	57.71	2.00	1.00	352.84	1.66	0.83	3.08	1.82	0.91	170.51	1.85	0.92
3	7.83	1.29	0.43	47.71	2.41	0.80	232.72	2.52	0.84	2.16	2.59	0.86	113.66	2.77	0.92
4	4.63	2.18	0.55	38.69	2.98	0.74	166.95	3.51	0.88	1.65	3.39	0.85	92.18	3.42	0.85
5	3.31	3.06	0.61	32.70	3.52	0.70	146.06	4.01	0.80	1.33	4.19	0.84	77.23	4.08	0.82
6	2.77	3.66	0.61	31.96	3.60	0.60	127.23	4.60	0.77	1.17	4.78	0.80	71.38	4.41	0.74
7	2.29	4.43	0.63	27.19	4.23	0.60	129.23	4.53	0.65	1.01	5.52	0.79	69.02	4.57	0.65
8	2.14	4.72	0.59	26.23	4.39	0.55	113.44	5.16	0.65	0.93	6.01	0.75	66.93	4.71	0.59



(a) Powergrid (b) Geocomp

Fig. 3. Some graphs with results of the MPI executions. Dots represent real time. Triangles represent the number of loop fails. The speedups appear on the top margin of each graph.

graph with 3000 nodes and average degree 5, achieved 9.2 of speedup running 10 threads with 0.92 of parallel efficiency.

The OpenMP implementation is not as scalable as the MPI implementation because on the multi-core computers loop fails are not the only factor affecting performance and memory access becomes a bottleneck as the number of threads increases. Also, since each thread requires its own copy of the graph, memory

usage increases. Our implementation with OpenMP scales well up to 10 to 16 threads depending on the instance.

7 Conclusion

Cut trees are a widely used combinatorial structure. Two sequential cut tree algorithms are known, but no parallel implementation of them has been reported in the past. This paper presents results of parallel implementations of Gusfield's cut tree algorithm using OpenMP and MPI. The results show that parallel versions of the algorithm can achieve high speedups. The parallel solution is relatively simple, requiring few changes in the original code, particularly with OpenMP. While OpenMP allows a greater control over the running threads, MPI provides more scalability. The implementations are complementary as they can explore the benefits of multi-core machines and computer clusters.

Future work includes a formal analysis of the running times and the scalability of the solutions and an experimental comparison of Gusfield's Algorithm with Gomory-Hu's Algorithm. Heuristics to improve efficiencies and the scalability of the solutions can be explored.

Acknowledgments. This work was partially supported by FINEP through project CT-INFRA/UFPR. Jaime Cohen was on a paid leave of absence from UEPG to conclude his Ph.D. and was supported by a Fundação Araucária/SETI fellowship under Order No. 16/2008. Luiz A. Rodrigues was also supported by Fundação Araucária/SETI, project 19836. This work was partially supported by grants 304013/2009-9 and 308692/2008-0 from the Brazilian Research Agency (CNPq).

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. 74(1), 47–97 (2002)
2. Backstrom, L., Dwork, C., Kleinberg, J.: Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In: Proceedings of the 16th int'l conference on World Wide Web. WWW 2007. ACM, NY (2007)
3. Bader, D.A., Sachdeva, V.: A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In: ISCA PDCS (2005)
4. Batagelj, V., Mrvar, A.: Pajek datasets (2006), http://vlado.fmf.uni-lj.si/pub/networks/data/
5. Bollobás, B.: Random Graphs, 2nd edn. Cambridge University Press, Cambridge (2001)
6. Chapman, B., Jost, G., Van der Pas, R.: Using OpenMP: portable shared memory parallel programming. MIT Press, Cambridge (2008)
7. Chekuri, C.S., Goldberg, A.V., Karger, D.R., Levine, M.S., Stein, C.: Experimental study of minimum cut algorithms. In: SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms. pp. 324–333. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997)

8. Cherkassky, B.V., Goldberg, A.V.: On Implementing the Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19(4), 390–410 (1997)
9. Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K.: Graph clustering and minimum cut trees. Internet Mathematics 1(4) (2003)
10. Goldberg, A.V., Rao, S.: Beyond the flow decomposition barrier. J. ACM 45, 783–797 (1998)
11. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM 35, 921–940 (1988)
12. Goldberg, A.V., Tsioutsiouliklis, K.: Cut tree algorithms. In: SODA 1999: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, USA (1999)
13. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics 9(4), 551–570 (1961)
14. Görke, R., Hartmann, T., Wagner, D.: Dynamic Graph Clustering Using Minimum-Cut Trees. In: Dehne, F., Gavrilova, M., Sack, J.R., Tóth, C. (eds.) WADS 2009. LNCS, vol. 5664, pp. 339–350. Springer, Heidelberg (2009)
15. Gusfield, D.: Very simple methods for all pairs network flow analysis. SIAM J. Comput. 19, 143–155 (1990)
16. Hong, B., He, Z.: An asynchronous multi-threaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic. IEEE Transactions on Parallel and Distributed Systems 99 (2010)
17. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: Densification and shrinking diameters. In: ACM Trans. on Knowledge Discovery from Data, ACM TKDD (2007)
18. Letchford, A., Reinelt, G., Theis, D.: Odd minimum cut-sets and b-matchings revisited. SIAM Journal on Discrete Mathematics 22(4) (2008)
19. Mitrofanova, A., Farach-Colton, M., Mishra, B.: Efficient and robust prediction algorithms for protein complexes using gomory-hu trees. In: Altman, R.B., Dunker, A.K., Hunter, L., Murray, T., Klein, T.E. (eds.) Pacific Symposium on Biocomputing, pp. 215–226 (2009)
20. Nagamochi, H., Ibaraki, T.: Algorithmic Aspects of Graph Connectivity. Cambridge University Press, New York (2008)
21. Rao, G., Stone, H., Hu, T.: Assignment of tasks in a distributed processor system with limited memory. IEEE Transactions on Computers 28, 291–299 (1979)
22. Saran, H., Vazirani, V.V.: Finding k cuts within twice the optimal. SIAM J. Comput. 24(1), 101–108 (1995)
23. Tuncbag, N., Salman, F.S., Keskin, O., Gursoy, A.: Analysis and network representation of hotspots in protein interfaces using minimum cut trees. Proteins: Structure, Function, and Bioinformatics 78(10), 2283–2294 (2010)
24. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature 393(6684), 440–442 (1998)

Efficient Parallel Implementations of Controlled Optimization of Traffic Phases

Sameh Samra[1], Ahmed El-Mahdy[1,*], Walid Gomaa[1,*],
Yasutaka Wada[2], and Amin Shoukry[1,*]

[1] Egypt-Japan University of Science and Technology (E-JUST), Egypt
[2] Faculty of Science and Engineering, Waseda University, Japan

Abstract. Finding optimal phase durations for a controlled intersection is a computationally intensive task requiring $O(N^3)$ operations. In this paper we introduce cost-optimal parallelization of a dynamic programming algorithm that reduces the complexity to $O(N^2)$. Three implementations that span a wide range of parallel hardware are developed. The first is based on shared-memory architecture, using the OpenMP programming model. The second implementation is based on message passing, targeting massively parallel machines including high performance clusters, and supercomputers. The third implementation is based on the data parallel programming model mapped on Graphics Processing Units (GPUs). Key optimizations include loop reversal, communication pruning, load-balancing, and efficient thread to processors assignment. Experiments have been conducted on 8-core server, IBM BlueGene/L supercomputer 2-node boards with 128 processors, and GPU GTX470 GeForce Nvidia with 448 cores. Results indicate practical scalability on all platforms, with maximum speed up reaching 76x for the GTX470.

Keywords: parallel processing, high performance computation, traffic phases.

1 Introduction

Traffic control requires real-time adjustment of the traffic signal [1,2] in order to ensure smooth and safe passage of vehicles. It is a key application in modern cities and has a strong effect on the quality of life and economy of these cities. However, optimal traffic control is computationally intractable, NP-hard problem, generally requiring an exponential order of computational steps. Moreover, traffic control requires real-time response, as it controls live traffic flows.

This paper focuses on the control of a single traffic intersection, as well as many independent intersections. This problem asks for obtaining an optimal sequence of green durations for every possible traffic flow (phase) across the intersection. It is essentially a sequential decision making problem that requires solution paradigms such as dynamic programming.

[*] Currently on-leave from Alexandria University.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 270–281, 2011.

With the increasing trend in utilizing multicore in today's processors, a tremendous increase in processing power have become possible at the expense of a substantial change in the programming model. Harnessing this processing power requires devising explicitly parallel algorithms that match the characteristics of these processors.

In this paper, we study the utility of modern parallel architectures for the single traffic intersection problem. A parallel algorithm is developed based on a sound serial algorithm (developed by Sen and Head [3]). The serial algorithm is based on dynamic programming, and has time complexity of $O(N^3)$ for one intersection, where N is the net number of discrete time steps that represent the temporal window over which the global decision making problem is solved. Our parallel algorithm is cost-optimal achieving a time complexity of $O(N^3/p)$, where p is the number of processors. The paper investigates the implementation of the algorithm on a variety of platforms that include a multicore server (8-core), a GPU (448-core), and the IBM BlueGene/L supercomputer (2-node boards, 128 processors). The study identifies the scalability and the minimum problem size for a given platform, aiding in possible design and implementation of future traffic control systems on a variety of platforms.

More specifically the paper has the following contributions:

- A parallel cost-optimal control algorithm for a single intersection based on Sen and Head's algorithm [3] with:
    - A corresponding a shared-memory (OpenMP) implementation.
    - A Message-passing (MPI) implementation that incorporates loop reversal, communication messages pruning, and load-balancing optimizations.
    - A Data-parallel (CUDA) implementation that incorporates loop reversal, efficient use of caches, and efficient assignment of threads to hide latency, and utilize GPU parallelism.
- Experimental performance evaluation of the above implementations on different parallel platforms.

The paper is organized as follows: Section 2 briefly reviews related work. Section 3 provides background and definition of the intersection control problem including the serial intersection algorithm. Section 4 introduces our parallel algorithm and develops its the time complexity. Sections 5 and 6 explain our message passing and data parallel implementations, respectively; each section provides necessary background on the programming model, and the implementation algorithm. Section7 provides the experimental setup and results. Section 8 concludes the paper.

2 Related Work

In the following, a brief literature survey about parallelization of dynamic programming and the traffic control problem is given.

GPUs have been used to accelerate the execution of an instance of dynamic programming called the Smith-Waterman algorithm [4]. In [5, 6, 7] the authors give different parallel implementations of this algorithm on GPUs.

Heunget et al. [8] give a dynamic programming based algorithm for traffic control. Their approach is decentralized and based on installing local controllers at the junctions of traffic lights. These controllers are physically and functionally independent employing fuzzy logic and genetic algorithms to handle the local control and the learning process, respectively. Coordination is introduced among the local controllers to derive optimal green time decisions using a global dynamic programming algorithm. The algorithm is a conventional serial CPU based implementation. A parallel solution of the traffic control problem is given in [9]. The authors employ a game-theoretic approach of a fictitious play to iteratively find coordinated traffic split plan.

The work done in [10] views the traffic control problem as an online optimization problem that is characterized both as nonlinear and non-convex. The authors reformulate the problem as a mixed-integer linear programming MILP (an LP mathematical program where some of the variables have to be integers). MILP solvers have already been around in the literature. Though, the authors propose an approach (non-parallel) to reduce the complexity of the MILP optimization problem in order to increase the real-time feasibility of the optimization problem.

The serial dynamic algorithm we utilize is serial polyadic dynamic programming algorithm; though computations happen in regular epochs, the communication pattern is variable and different than existing solutions in the literature.

3 Serial Intersection Algorithm

Algorithm 1 presents the serial algorithm Controlled Optimization of Traffic Phases (COP) for traffic control given in [3]. First, the basic conventions and notions are given, followed by the algorithm itself. Let r be a constant integer denoting the effective clearance is the amount of time necessary to make a safe phase transition. Let γ be a constant integer denoting the minimum green time. The control/decision variable x_j denotes the amount of green time allocated for stage (phase) j. The state variable s_j represents the total number of discrete time steps that have been allocated so far after stage j has been completed. The space of all feasible control decisions at state s_j is denoted $X_j(s_j)$. The value function and the performance measure at stage j are $v(s_j)$ and $f(s_j,x_j)$ respectively. T denotes the total number of discrete time steps (the temporal window over which the decision making problem is solved). This parameter plays an essential role in the parallel version of the algorithm as will be seen later.

1 procedure COP(T,r,γ)
2 begin
3 Initialize j:= 1, s_0:= x_0:= v(s_0) = 0;
4 **for** $s_j = r$ **to** T do // *running* over all possible values of the state variable at stage j
5 **begin**
6 **If** $s_j - r < \gamma$ **then**// the case where the allocated green time is less than the min. threshold (γ)
7 $X_j(s_j)$:= {0}; // in such case no green time should be allocated for phase j
8 **else**
9 $X_j(s_j)$ = {0,γ ,γ + 1, … , $s_j - r$};// otherwise, these are the feasible values for green time at state sj

10 **For each** $x_j \in X_j(s_j)$ **do** // choose the best value for the decision variable (green time) according to the value fn
11 **begin**
12 **If** $x_j == 0$ **then**
13 $s_{j-1} := s_j$; // phase j is allocated zero time
14 **else**
15 $s_{j-1} := s_j - x_j - r$;
// forward recursion, clearly sj = sj-1 + xj + r (for the current suggested value of xj)
16 $v_j'(s_j,x_j) := f_j(s_j,x_j) \bullet v_{j-1}(s_{j-1})$; // evaluating the value fn at the particular state sj with the control variable set to xj
17 **end**// end for xj
18 $x_j^* := \text{argmin}_{xj} v_j'(s_j,x_j)$;// computing the optimal value of the control variable at state sj
19 $v_j(s_j) := v_j'(s_j,x_j^*)$;// computing the optimal value fn at state sj given the optimal value of the control var xj*
20 **end** // end for sj
21 $j := j + 1$; // next stage
22 **Repeat** the above loop (line 4) until the stopping criterion is satisfied
23 **end**// end procedure COP

Algorithm 1. Serial Algorithm

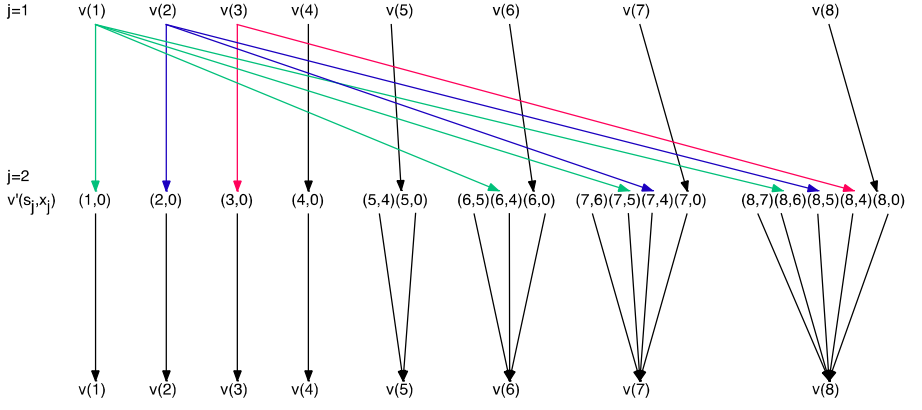


Fig. 1. Task Dependence Graph (T=8, γ=4, r=1)

The stopping criterion in line 22 is reached when no change occurs in the value function for the last ρ stages, where ρ is the total number of phases for the given intersection. In such a case, no further phase change would incur any difference in the value function; hence the optimization process is stopped.

The time complexity of calculating the value function in line 16 depends on the particular choice of the performance measure $f_j(s_j,x_j)$ and the particular implementation, especially considering the time/space trade-off (for example, whether or not to use a lookup table to store old values instead of re-computing them).

Let N be the total number of processed time steps per stage ($N=T-r+1$) and assume line 16 takes O(m) steps, where m is a function of N that depends on the particular choice of the performance measure. Then the time complexity of computing each

stage j is $O(mN^2)$. To compute the complexity of the whole algorithm we need to compute the maximum possible number of stages. Based on the stopping criterion mentioned in the previous paragraph and the worst-case scenario that each phase with non-zero duration takes only 1 time step (γ=1), and assuming zero clearance interval (r=0), we can have at most ρ N stages. ρ is a constant, hence the total number of stages is $O(N)$ and the whole algorithm takes $O(mN^3)$ computation steps. In our implementations, we consider the cost function to be the cars queue length as defined in the COP algorithm [3]. Computing this cost function is O(1), thus the overall complexity is $O(N^3)$.

Fig. 1 shows the task dependence graph of a stage of the serial algorithm described in the previous section. The figure assumes r=1, T=8, and γ=4. The top row lists the obtained $v(sj)$ values from the first iteration (j=1). Obtaining these values only depend on the initial conditions. The middle line shows the required computations $v'(s_2,x_2)$. For T higher than 8, the graph can be easily extended to the right to $v(T)$ by:

1) Extending the first row from $v(8)$ to $v(T)$.
2) Extending the second row with T-8 new clusters of $v(s_j,x_j)$ such that each cluster corresponds to an s_j with entries $(s_j,0)$, (s_j,γ), $(s_j,\gamma+1)$,..., (s_j,s_j-r).
3) Connect a directed edge from $v(sj)$ to each new $v(s_j,x_j)$, where $l_j = s_j-r-\gamma$ for $sj>r+\gamma$, and $l_j = 0$, otherwise.
4) Similarly, extend the last row from $v(8)$ to $v(T)$.

The $v(sj)$ values are obtained as a reduction operation on the $v'_j(sj)$ values. For higher values of j, the graph would simply require repeating the j=2 portion of the dependence graph.

4 Shared Memory Parallel Intersection Control Algorithm

Our parallel formulation of the algorithm is to assign processor p_j to state s_j for j = 1, ... , N. Each processor is, therefore, responsible for computing the $v'_j(s_j,x_j)$ value as well as the corresponding optimal parameters x_j^* and $v_j(s_j)$. The execution time complexity for each stage would be $O(mN)$ giving a total parallel time complexity of $O(mN^2)$, which when multiplied with the number of processors N would give the serial time complexity $O(mN^3)$. It is worth noting that an execution time of $O(mN \log N)$ can be achieved using N^2 processors. This can be done by requiring each processor to compute exactly one $v'_j(s_j,x_j)$ value, and this can be done in $O(m)$ time. Then computing their minimum, that is the $v_j(s_j)$ value, by arranging the processors into a balanced binary tree, where at level i of the tree a total of $N^2/2^i$ different partial minimum values are computed. The depth of the tree is $O(\log N)$, and an upper bound of computation cost $O(N)$ is needed at each level of the tree. Hence, a total of $O(mN \log N)$ is achieved. However, such algorithm is not cost optimal, since in such a case the product of the parallel complexity $O(mN \log N)$ with the number of processors N^2, giving $O(mN^3 \log N)$, is not equal to the serial complexity of $O(N^3)$. The degree of parallelism in the problem exceeds the number of cores in commodity[1]. It is also

[1] In the GPU case, more thread is favoured, but the control-flow may not be suitable for data-parallel nature of the GPU.

worth noting that the latter algorithm will be useful when computing several intersections in a pipelined fashion.

The shared memory implementation is done by parallelizing the loop at lines 4-20 (Algorithm 1). $v()$, x^*, are designated as shared variables.

5 Message Passing Implementation

The message-passing programming model has the advantage of allowing more scalability than the shared-memory programming model. This is due to the fact that shared memory architectures are economically more expensive in addition to the overhead of memory coherence. On the other hand, message passing programming is much more complex.

Since the degree of parallelism in the COP algorithm is quite large, and to allow for scalability studies, we designed the parallel version such that the actual number of processors can be smaller than the theoretical number of processors identified in our main parallel formulation. This has introduced the problem of mapping many logical processors into one physical processor. It is worth noting that simulating parallel execution of the logical processors lead to very poor results due to the fact of sending/receiving many unnecessary messages. The communication pattern can negatively affect the performance by having the processors remaining in idle state for long periods of time. This case will be discussed after presenting the algorithm.

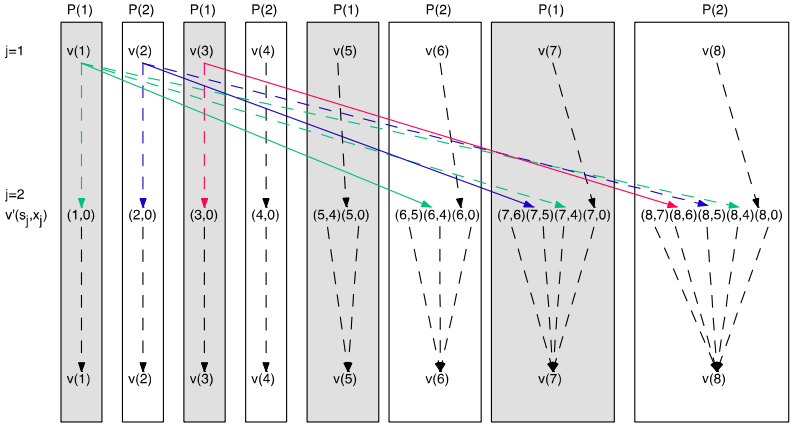


Fig. 2. Processor Assignment Effect on the Task Dependence Graph (T=8,γ=4,r=1)

Fig. 2 illustrates our main idea about the mapping of logical processors into physical ones, pruning redundant messages. The figure shows an interleaving degree of 2. The solid arrows show actual communication/messages between processors, whereas the dashed arrows indicate redundant communication (the messages have already been sent to the same physical processor). The interleaving of the logical processors among the available pair of physical processors (the odd-numbered logical tasks are assigned to physical processor 1 and the even-numbered logical tasks are

assigned to physical processor 2) allows for uniform distribution of the messages among the processors, thereby decreasing idle time and avoiding deadlocks. If logical processors are to be grouped as contiguous blocks where each block is assigned to some physical processor, the processors on the left-hand side will only send to processors to the right side resulting in a serial execution of tasks. Deadlocks are prevented by executing from left-hand side to right-hand side.

Messages among logical processors that are assigned to the same physical processor are avoided. For example, P(1) does not send $v(1)$ to the state-decision pair (7,5) as it is already assigned to the same physical processor P(1). Also, redundant communication is avoided. For example, P(1) does not send $v(1)$ to P(2) in the computation of the pair (8,6) since P(1) has already sent the same message to P(2) when computing the pair (6,4).

```
1 procedure Solve(rank, p, r, γ)
\\ rank: processor id
\\ p: number of processors
2 begin
3     j := 1; sj_1 := 0;  v := 0;        \\ v is the
value function
3     while checkStoppingCriteria(v,j-1)
4     begin
5        for sj := r + rank to T step p
6        begin
7           TargetProcIds := {sj + r + γ, … ,Min(sj + r
             + γ + p, T)};
8           if j > 1 then
9              AllToAll v [si][j-1], xj_star[sj][j-1]
10       end
11    end
12    v [sj][j] = Min_xj_star (fj(sj,xj) . v [sj_1][j-1])for
all xj in { sj, sj-1, …,  , 0};
13    xj_star[sj][j] = xj_star;
14end
15j:=j+1;
16end
17 ReportSolution();
```

Algorithm 2. MPI Solve

Algorithm 2 gives the pseudo-code of our MPI Solve procedure using MPI communication method of **AllToAll()**. AllToAll() send given data to all processors in the cluster and it is cost optimized to be O(p) only [14]. Each processor executes the Solve procedure with *rank* set to the processor's id. The algorithm proceeds similar to the serial one till line 4. Line 5 distributes the loop iterations among p processors such that iteration k is given to processor $k - r \bmod p$.

The first stage of computation (j=1) relies on the initial conditions and no message is communicated among processors. For j>1, line 9 sends the previous stage messages to the corresponding physical processors (the list of processors in *TargetProcIds*). It is

worth noting that the number of messages sent is at most p. Also line 9 receives the previous stage data from corresponding physical processors. The function $h(x_j)$ in line 14 is defined as follows:

$$h(x_j) = \begin{cases} 0 & x_j = 0 \\ x_j + r & \text{Otherwise} \end{cases} \quad (1)$$

The implementation of parallel checkStoppingCriteria procedure differs from the serial algorithm in that processor with rank p-1 is responsible for checking the stopping condition when $j > \rho$. The processor broadcasts its decision to other processors. Other processors receive the decision from that processor.

It is worth noting that the stopping criteria computational complexity is $O(\rho) = O(1)$, each computational step requires $O(\log p)$ communication steps to broadcast the decision, therefore it does not interfere with the overall complexity of the algorithm$<O(N^3/p)$. Also, practically, ρ is much smaller than the number of processed time slots (N); therefore its contribution is negligible.

6 Data Parallel Implementation

General Purpose Graphics Processing Units (GPGPUs) favour data parallel programming models. OpenCL [11] and CUDA [12] are popular data parallel models targeting GPGPUs, that are similar; this paper focuses on CUDA being more mature and easier to program than OpenCL as the latter is targeting diverse GPUs from different vendors.

In the CUDA programming model, data parallelism is expressed in a form of Single-Program Multiple Data streams (SPMD) model. A group of programs (i.e. threads) is executed in a lock-step fashion. Such a group is called 'warp' that is of size 32 in current GPUs. Warps are collectively executed on a Streaming Multiprocessor (SM). In CUDA notation, collections of threads are called 'thread blocks'. One noticeable feature of thread blocks is the availability of thread synchronization operation, as well as shared memory. Thread blocks share global memory with high bandwidth but of high latency. The latter is potentially hidden using the underlying multithreading execution engine of the GPU.

Recently, the current generation of CUDA enabled GPUs (code named Fermi) supports data caches [13]. There are L1 data cache for each SM and one L2 data cache for global memory. In our algorithm, we, therefore, maximize data reuse to increase cache hit ratios.

An important performance aspect of our algorithm is load-balancing the stage iteration loop. Mapping many iterations into a thread is not plausible here as it would decrease multithreading and, therefore, expose GPU's high latency. Moreover, threads will have different control-flow patterns, which are not plausible for the underlying data-parallel execution engine, resulting in serialization in the execution of threads. Our algorithm deals with this case by mapping each iteration linearly to threads; since the warp size is small with respect to the total number of threads, we only waste negligible cycles (provided N is much bigger than warp size).

CUDA GPUs support memory coalescing, where same memory access from threads from the same warp is coalesced saving memory bandwidth. To achieve this, the loop reversal in the earlier algorithm (MPI), in addition to the linear mapping of sj's into threads result in all threads having similar data accesses (access the same element v).

Another aspect is the high latency of GPUs, which should be hidden by using multithreading. Generally, the latency results from accessing global memory (100s of cycles in current GPUs) as well as dependent read-after-writes register operations (10 cycles); since the algorithm optimizes cache performance, the latter smaller latency is required to hide, which requires number of threads proportional to the latency value. The algorithm, therefore, assigns each thread to *sj*; assigning an *v'*() would result in higher degree of parallelism by that would be traded off with synchronization among threads and possibly blocks (for larger *T*) which would significantly reduce the regularity of control-flow and synchronization overhead among blocks. Increasing the number of threads can however increase the number of communication messages among threads; however, the GPU has high communication bandwidth to global memory (1 word per 32 instructions on average for the GTX470) that exceeds the required global memory access (approx. N^2 global memory accesses/ N^2 computations = 1 word per operation, where the operation easily exceeds 32 instructions).

The use of data caches in CUDA GPUs allows for reducing the communication from global memory. Our linear mapping achieves the same interleaving behavior we utilized in the MPI algorithm, but this time among the blocks, thereby each block behaves as a faster shared memory parallel subsystem, and the blocks operate in slower shared memory system.

Algorithm 3 is the data parallel version of the algorithm. The stopping criterion is similar to Algorithm 3 of the MPI, but thread synchronization is implemented by a call to a host function, as the ratio of synchronizations to computations is 1 : *N*, amortizing the overhead of synchronization.

```
1 procedure Solve(blockidx, blockwidth, threadidx)
\\ blockidx: block index
\\ blockwidth: number of threads per block
\\ threadidx: thread number within a block
2 begin
3     j:=1;sj_1:=0;v:=0;      \\v is the value function
4     sj := blockidx × blockwidth + threadidx + r
5     while checkStoppingCriteria(v,j-1)
6     begin
7       Compute Solution as per lines 6-19 of algorithm1
8       j:=j+1;
9     end
10     if sj = T
11     begin
12        ReportSolution();
13     end
14 end
```

Algorithm 3. Data Parallel Algorithm

7 Experimental Study

We have implemented the algorithm using three parallel programming models: (1) OpenMP, which is a shared memory model, (2) MPICH, which is a message passing

model, and (3) CUDA, which is a data parallel model. The platforms used for our algorithms are the following: OpenMP runs on a machine, named fuji, with two Quad-Core processor (Intel Xeon), hence an overall of eight processing units (8 cores). MPICH runs over the same fuji machine as well as the IBM supercomputer BlueGene/L, the latter with 128 processors. The machine used for CUDA is an Intel Core I7 machine with one quad-core processor, though the algorithm is run using its GPU which is a GTX470 GeForce Nvidia with 448 cores.

According to the nature of the particular problem we investigate, traffic control, three input parameters control the workload: the number of phases, and the total number of discrete time steps T which represents a finite future horizon over which the optimization algorithm considers for decision making, and the traffic load itself (for example, the rate of cars entering the intersection). As can be seen from the parallel algorithm, only T would control the degree of parallelism, hence it is the only parameter varied in our experiments as seen below. The other parameters would just add more parallel iterations and hence would scale the obtained curves. Therefore, they are kept fixed throughout all our experimentation (we have used the values from the original paper [3]) : the number of phases is 3 and the car generation is scheduled according to Table 1 in [3]. In the experiments, we vary workload and number of processors and measure the corresponding execution speed, Intersections/Sec.

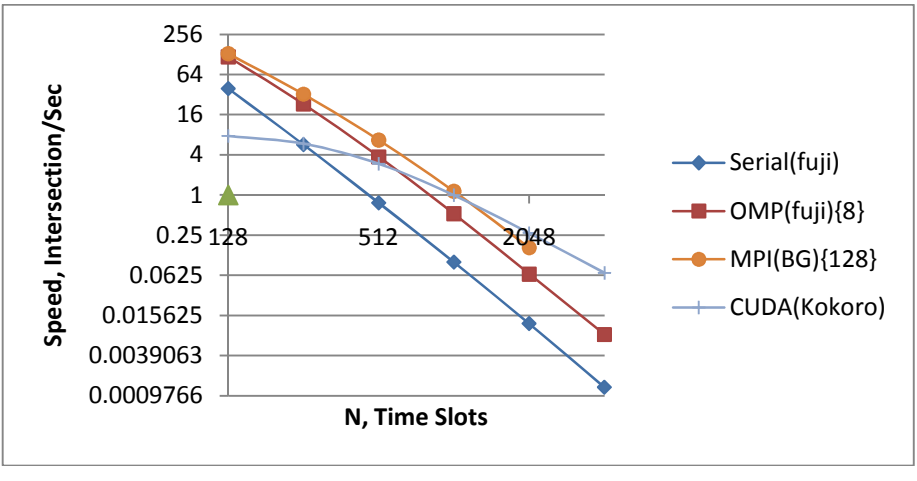


Fig. 3. Best Parallel Execution for a Single Intersection

Fig. 3 plots the execution speed (in Intersection/Sec) while changing the number of 'Time Slots', N. The 'Serial' case represents the serial algorithm run on fuji machine; with the increase of N, the execution speeds decrease quadratically, reaching 0.1 Intersection/Sec for $N = 1024$.

The OMP(fuji){8} represents the case of OpenMP algorithm over 8 cores, on the machine fuji. It shows a speedup of 5 to 6 with respect to the 'serial' case, with increasing of N. The MPI(BG){128} represents the case of MPI algorithm over 128 cores. It shows speedup of 4 to 13 times over the serial case.

The CUDA(kokoro){448} represents the case for the CUDA algorithm over 448 cores on the machine kokoro. The performance for small *N* is smaller than all other cases; this is mainly due to the lack in the number of threads to hide latency and make sure of GPU parallelism. With increasing *N*, the GPU significantly improves. The GPU becomes dominant after *N*=1024, reaching a speedup factor of 52 over the serial processor for *N*=4096.

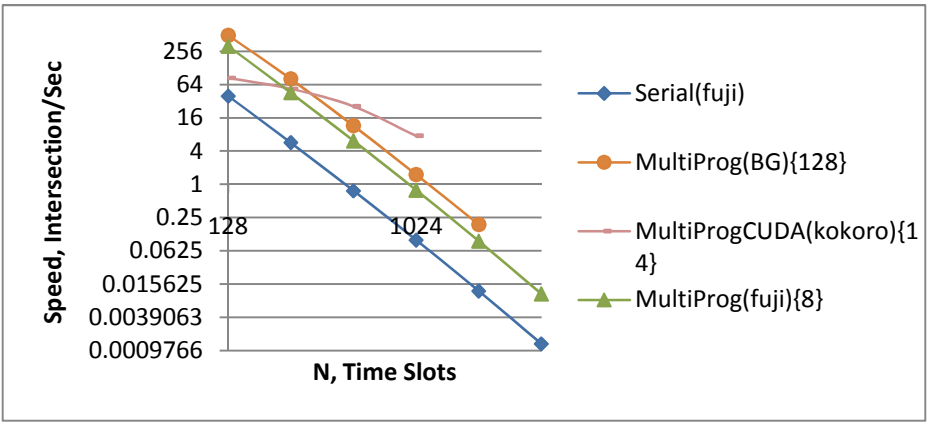


Fig. 4. Best Execution Speeds with Many Intersections

The analysis above focused on improving the execution time of a single intersection. However, if processing a single intersection time is not critical, and the throughput of processing many intersections is critical, simple multiprogramming would suffice and provide good scalability. Fig. 4 explores that aspect. We still include the 'Serial' case as a baseline reference. The MultiProg(BG){128} and MultiProg(fuji){8} are scaled up execution of the serial case for each platform using the maximum available processors; the former using 128 processors, and the latter using 8 processors. The MultiProgCUDA(kokoro){14} allow for running 14 intersections on the same GPU, each in a single SM. That case combines parallelism within an intersection, and among other intersections. The results show that CUDA is up to 76 times faster than the serial case. However, *N* cannot be 2048 or higher due to reaching the memory capacity. The CUDA case is up to 5 times faster than the BG.

8 Conclusion

This paper introduced a cost-optimal, parallel version of an optimal traffic control for an intersection [3]. Given the diversity of existing parallel architectures, the paper developed three different parallel implementations, using well-known parallel programming models: shared memory, message passing, and the recently introduced CUDA data parallel models. Experimentations on typical high-end servers, BlueGene/L supercomputer, and CUDA GPU, have shown that the parallel algorithm is scalable on all of the parallel programming models; comparing with the serial algorithm, speedups up to 76x has been achieved.

CUDA GPU performance is superior to BlueGene/L server. But, on the other hand, BlueGene/L is more scalable than CUDA GPU.

Future work would target other parallel programming models such as MapReduce and n-body models. Also, we would extend the algorithm to optimize traffic control on multiple intersections.

Acknowledgments. The authors would like to acknowledge their fruitful discussions with Dr Hisham El-Shishiny, manager of IBM Center for Advanced Studies in Cairo. The authors would like also to acknowledge the support of Mohamed Baddar, from IBM Center for Advanced Studies in Cairo, who helped with running the developed codes on IBM BlueGene/L super computer. This work is partially funded by IBM PhD Fellowship and Faculty Award.

References

1. Zhou, G., Gan, A., Shen, L.: Optimization of Adaptive Transit Signal Priority Using Parallel Genetic Algorithm. Tsinghua Science & Technology 12(2), 131–140 (2007)
2. Machemehl, R., Shenoda, M.: Development of a Phase-by-Phase, Arrival-Based, Delay-Optimized Adaptive Traffic Signal Control Methodology with Metaheuristic Search. Center for Transportation Research, University of Texas at Austin (2006)
3. Sen, S., Head, K.: Controlled optimization of phases at an intersection. Transportation science 31(1), 5–17 (1997)
4. Waterman, M.S., Smith, T.F.: Identification of common molecular subsequences. J. Mol. Biol. 147, 195–197 (1981)
5. Manavski, S., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9(2), S10 (2008)
6. Xiao, S., Aji, A.M., Feng, W.: On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In: International Conference on Parallel and Distributed Systems, pp. 26–33 (2009)
7. Siriwardena, T.R.P., Ranasinghe, D.N.: Accelerating global sequence alignment using CUDA compatible multi-core GPU. In: 5th International Conference on Information and Automation for Sustainability (ICIAFs), pp. 201–206 (2010)
8. Heung, T.H., Ho, T.K., Fung, Y.F.: Coordinated Road-Junction Traffic Control by Dynamic Programming. IEEE Transactions on Intelligent Transportation Systems 6(3), 341–350 (2005)
9. Cheng, S., Epelman, M.: CoSIGN: A Parallel Algorithm for Coordinated Traffic Signal Control. IEEE Transactions on Intelligent Transportation Systems 7(4), 551–564 (2006)
10. Lin, S., De Schutter, B., Xi, Y., Hellendoorn, H.: Fast Model Predictive Control for Urban Road Networks via MILP. IEEE Transactions on Intelligent Transportation Systems (2011)
11. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering 12(3), 66–73 (2010)
12. Kirk, D., Hwu, W.: Programming Massively Parallel Processors. Morgan Kaufmann, San Francisco (2010)
13. NVIDIA CUDA C Programming Guide. 3rd edn. NVIDIA Corporation (2010)
14. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison Wesley, Reading (2003)

Scheduling Concurrent Workflows in HPC Cloud through Exploiting Schedule Gaps

He-Jhan Jiang[1], Kuo-Chan Huang[1], Hsi-Ya Chang[2],
Di-Syuan Gu[1], and Po-Jen Shih[1]

[1] Department of Computer and Information Science
National Taichung University of Education
No. 140, Min-Shen Road, Taichung, Taiwan
{mice.jiang,ejeu67195,frontpageserver}@gmail.com,
kchuang@mail.ntcu.edu.tw
[2] National Center for High-Performance Computing
National Applied Research Laboratories
P.O. Box 19-136, Hsinchu, Taiwan
jerry@nchc.narl.org.tw

Abstract. Many large-scale scientific applications are usually constructed as workflows due to large amounts of interrelated computation and communication. Workflow scheduling has long been a research topic in parallel and distributed computing. However, most previous research focuses on single workflow scheduling. As cloud computing emerges, users can now have easy access to on-demand high performance computing resources, usually called HPC cloud. Since HPC cloud has to serve many users simultaneously, it is common that many workflows submitted from different users are running concurrently. Therefore, how to schedule concurrent workflows efficiently becomes an important issue in HPC cloud environments. Due to the dependency and communication costs between tasks in a workflow, there usually are gaps formed in the schedule of a workflow. In this paper, we propose a method which exploits such schedule gaps to efficiently schedule concurrent workflows in HPC cloud. The proposed scheduling method was evaluated with a series of simulation experiments and compared to the existing method in the literature. The results indicate that our method can deliver good performance and outperform the existing method significantly in terms of average makespan, up to 18% performance improvement.

Keywords: HPC cloud, workflow scheduling, distributed gap search.

1 Introduction

Many large-scale scientific and engineering applications are usually constructed as workflows due to large amounts of interrelated computation and communication. Common types of workflows can be represented by Directed Acyclic Graphs (DAG) for describing the inter-task precedence constraints [7]. Each node represents a task

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 282–293, 2011.

which executes a specific program. The number next to each node means the execution time of the task. The edges represent the dependence between tasks and the number next to an edge means the inter-task data transmission time. A job scheduler has to schedule and allocate each task according to the dependence specified in the workflow definition.

Traditionally, few users have access to high performance computing platforms for executing large-scale workflows. Now, the situation has changed since the cloud computing model [23][24] emerged. Cloud computing has been thought as a promising next-generation computing platform and service model recently. It is usually divided into three types of services: SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service) [25]. Common examples of these services include Salesforce.com [26] and Gmail [27] for SaaS, Google App Engine [28] and Microsoft Azure [29] for PaaS, and Amazon EC2 for IaaS [30]. Among the various cloud services proposed, HPC cloud service [31][32] has recently become a promising one. It provides on-demand high-performance computing services for compute-intensive scientific and engineering applications. Since multiple workflows from different users might run simultaneously, it becomes an important issue how to schedule concurrent workflows on HPC cloud efficiently.

Workflow scheduling in parallel and distributed environments, in general, is a NP-complete problem [11] [12], therefore many heuristic methods have been proposed [2][3][6][7][8][9]. Most of them were designed for scheduling a single workflow only. Recently, several approaches [1][5][13][14][15] have been proposed to deal with multiple workflow scheduling. In [1] Bittencourt and Madeira proposed an approach to scheduling multiple workflows on computational grids. In their approach, a Path Clustering Heuristic (PCH) [10] is used to partition a workflow into several task groups first, and then the list scheduling heuristic is applied to allocate these task groups onto processors. During task group allocation, some time gaps may form in the schedules of each processor. A *gap search* algorithm was proposed to find suitable gaps for allocating subsequent task groups [4]. In their approach, the gap search algorithm tries to allocate an entire task group onto a single time gap. We call it continuous gap search hereafter in this paper. However, sometimes a big task group cannot fit into any gaps. To further improve resource utilization and workflow performance, in this paper, we propose a distributed gap search scheme which can allocate the tasks within the same group onto different time gaps on different processors. The proposed scheme has been evaluated with a series of simulation experiments and compared to the continuous gap search scheme. The results indicate that the proposed distributed gap search scheme outperforms the previous continuous scheme significantly in terms of average makespan, up to 18% performance improvement.

The remainder of this paper is organized as follows. Section 2 discusses related works on workflow scheduling. We describe the PCH and continuous gap search algorithm [1][4] in section 3. Section 4 presents our distributed gap search scheme. Section 5 evaluates the proposed distributed scheme and compares it with the continuous approach. Section 6 concludes this paper.

2 Related Work

Heuristic-based workflow scheduling algorithms usually can be classified into four types: (1) list-based, (2) clustering-based, (3) duplication-based, (4) level-based. List-based heuristic approaches [16][17][18] maintain a list of all tasks of a workflow application according to their priorities and then schedule the tasks based on the list.

The main idea of clustering-based heuristic methods [19] is to reduce communication delay by grouping the tasks of heavy communication into the same labeled cluster. In general, a clustering-based heuristic method has two phases: clustering and merging. In the clustering phase, the original workflow application is partitioned into clusters, and the merging phase merges the clusters so that the remaining number of clusters equals to the number of resources.

A duplication-based heuristic method [20] helps a task to transmit the data to the resource of succeeding task(s) through duplicating the task on the destination processor. This duplication arrangement can reduce the communication cost from a task to a successor in order to minimize the overall makespan of the entire workflow. A level-based heuristic method, e.g. LHBS (Levelized Heuristic Based Scheduling) [21], divides the workflow into levels of independent tasks. Within each level, LHBS can use the Greedy, Min-Min, Min-Max, or Sufferage [22] heuristics to map the tasks onto resources.

Both the method in [1] and our method in this paper adopt a hybrid approach to workflow scheduling. At the first step a clustering-based scheme is used to partition a workflow into several task groups. Then, in the second step the list-based heuristic is applied to allocate these task groups onto processors.

3 PCH and Gap Search

This section introduces the Path Clustering Heuristic (PCH) approach and the gap search algorithm [1][4]. The entire workflow scheduling process is divided into two phases. In the first phase, PCH is used to cluster tasks in a workflow into different groups, and then priorities are assigned to the task groups based on their dependence. In the second phase, the gap search algorithm is used to allocate each task group onto a specific processor.

The workflows discussed in this paper can be represented by Directed Acyclic Graphs (DAG), G (V, E), where:

- V is the set of tasks, $t_n \in V$, $|V|$ = number of tasks;
- E is the set of directed edges, $e_n \in E$, $|E|$ = number of edges.

Each workflow starts at one node, named the front node, and finishes at one node, named the end node. Each node in the workflow is a task representing a specific job or program to execute and each edge represents the data dependence between nodes. Each task starts its execution only after receiving all data from his parent nodes.

The Path Clustering Heuristic (PCH) is a DAG scheduling heuristic which clusters the tasks into different groups and uses the list scheduling technique to schedule the

tasks within a group onto the same resource. PCH focuses on reducing the communication costs between tasks and has been shown good performance in [1][4].

We first define several node attributes which will be used in the following.

- Computation cost:

$$w_{i,r} = \frac{instructions_i}{power_r}$$

$w_{i,r}$: the computation cost of task i on resource r
$instructions_i$: the amount of instructions in task i
$power_r$: processing power, in instructions per second.

- Communication cost:

$$c_{i,j} = \frac{data_{i,j}}{bandwidth_{r,t}}$$

$c_{i,j}$: The communication cost between task i and j, using the link between resource r and t. If r = t, $c_{i,j}$ =0.

- Priority:

$$P_i = \begin{cases} w_i & \text{, if i is the last task} \\ w_i + \max\limits_{t_j \in succ(n_i)} (c_{i,j} + P_j) & \text{,otherwise} \end{cases}$$

$succ(n_i)$: the set of immediate successors of task t_i.

- Earliest start time:

$$EST(t_i, r_k) = \begin{cases} Time(r_k) & \text{,if i =1} \\ \max\{Time(r_k), EST_{pred}\} & \text{,otherwise} \end{cases}$$

representing the earliest start time of the task on resource k,
where $Time(r_k)$ is the time when the resource k is ready for task execution and $EST_{pred} = max_{t_h \in pred(t_i)}(EST_h + w_h + c_{h,j})$ as defined in [1].

- Estimated finish time:

$$EFT(t_i, r_k) = EST(t_i, r_k) + \frac{instructions_i}{power_k}$$

representing the estimated finish time of task i on resource k.

After allocating the task groups of a single workflow, there are some gaps formed between the allocated task groups. Therefore, a gap search algorithm [1][4] was proposed to improve the resource utilization by trying to find the unused gaps to allocate subsequent task groups from other workflows.

Figure 1 illustrates an example of how to apply the gap search algorithm [1][4] in multiple-DAG scheduling. In Figure 1, there are two workflows for scheduling at the same time. In the first step, PCH is used to cluster the tasks, resulting in four task groups for the upper DAG and three task groups for the second DAG. Then, the upper DAG is scheduled first with the traditional list scheduling heuristic. When scheduling the second DAG, we first calculate the computation cost of each task group, *e.g.* {A, B, C, and D}, through summing up the computation costs of all the tasks within the task group. Then, the gap search algorithm is used to check whether there is a suitable gap to allocate the task group. A suitable gap for a task group means a gap with a time period equal to or larger than the computation cost of the task group.

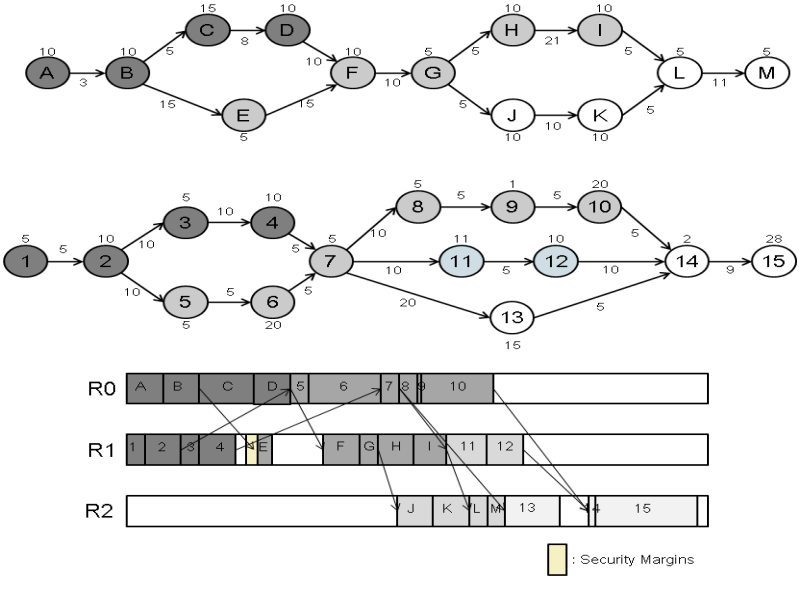


Fig. 1. Example of scheduling two workflows using PCH and gap search

To consider the issue that a gap might dynamically shrink due to the unexpected slowdown of a resource, the gap search algorithm adopts a security margin mechanism [4], which calculates the time gap by assuming a specific percentage of shrink caused by resource slowdown. In this paper, we assume this percentage to be 10%, which means if there is a task group of 90-second computation cost, we have to find a time gap larger than 100 seconds to accommodate the task group since 100*(100%-10%) = 90.

Algorithm 1 in the following describes the gap search algorithm in details. For a resource with k tasks already scheduled on it, there are k candidate time gaps to allocate a subsequent task group. In Algorithm 1, Lines 1 to 9 tries to allocate the task group into the first time gap, indexed with 0. Lines 10 to 19 iteratively tries to allocate the task group into the remaining k-1 time gaps until a suitable gap is found. If no suitable gap is found after trying all k time gaps, the task group will be allocated to immediately follow the last task on current schedule.

Algorithm 1. gap search (grp, S_r)

Input: grp: the task group to be scheduled
S_r: current schedule of resource r
m: number of tasks in grp; tasks are indexed from 1 to m
Output: $g_{grp,r}$: the index of the gap found, starting from 0

1: $size_{grp,r} \leftarrow EFT(t^{grp}_m,r) - EST(t^{grp}_1,r)$
2: k ←number of tasks in S_r
3: **if** $(EST(t_1,r) * s\_margin) \geq size_{grp,r}$ **then**
4: Compute ESTs and EFTs for $t^{grp}_j \in grp$ in the first gap
5: $D_{t^{grp}_m} \leftarrow$ tasks ahead the gap which depends on t^{grp}_m
6: **if** $(EST(t_1,r) - EST(t^{grp}_1,r) \geq size_{grp,r})$ and $D_{t^{grp}_m} = \varnothing$ **then**
7: $g_{grp,r} = 0$; **return** $g_{grp,r}$
8: **end if**
9: **end if**
10: **for** i=1 to k-1 **do**
11: **if** $((EST(t_{i+1},r) - EFT(t_i,r)) * s\_margin) \geq size_{grp,r}$ **then**
12: Compute ESTs and EFTs on current gap $\forall\, t^{grp}_j \in grp$
13: $D_{t^{grp}_m} \leftarrow$ tasks ahead the gap which depends on t^{grp}_m
14: **if** $(EST(t_{i+1},r) - EST(t^{grp}_1,r) \geq size_{grp,r})$ and $D_{t^{grp}_m} = \varnothing$ **then**
15: $g_{grp,r} = i$; return $g_{grp,r}$
16: **end if**
17: **end if**
18: **end for**
19: $g_{grp,r} = k$; **return** $g_{grp,r}$ //no gap found

As shown in Figure 1, there are some gaps left idle because they are not large enough to accommodate any task group. To further improve the resource utilization, we propose a distributed gap search scheme to resolve the issue in the next section.

4 Distributed Gap Search

The original gap search algorithm in [1][4] tries to allocate an entire task group into a single gap on a specific resource. If a gap cannot accommodate any task group, it is left idle, resulting in both degraded resource utilization and delayed task completion time. This section proposes a distributed gap search scheme, which allows for allocating the tasks of the same group into different gaps on different resources. This

distributed scheme has the potential to further improve resource utilization, leading to a better workflow execution performance in terms of makespan.

Figure 2 is an example illustrating how the distributed gap search scheme works. The two workflows to be scheduled in Figure 2 are the same as those in Figure 1, while the distributed gap search scheme is applied now. In Figure 2, when trying to allocate the task group {5, 6, 7, 8, 9, 10} from the second workflow, it is found that no single gaps on any resources can accommodate this task group. In the original gap search algorithm in [1][4], this task group would then be allocated to the end of the schedule on resource zero. On the other hand, our distributed gap search scheme partitions the task group into two subgroups, {5, 6, 7, 8, 9} and {10}, where the first subgroup can fit into a gap on resource two and the second subgroup can be allocated onto another resource, in this case resource zero, as shown in Figure 2. This leads to a better resource utilization and workflow execution performance.

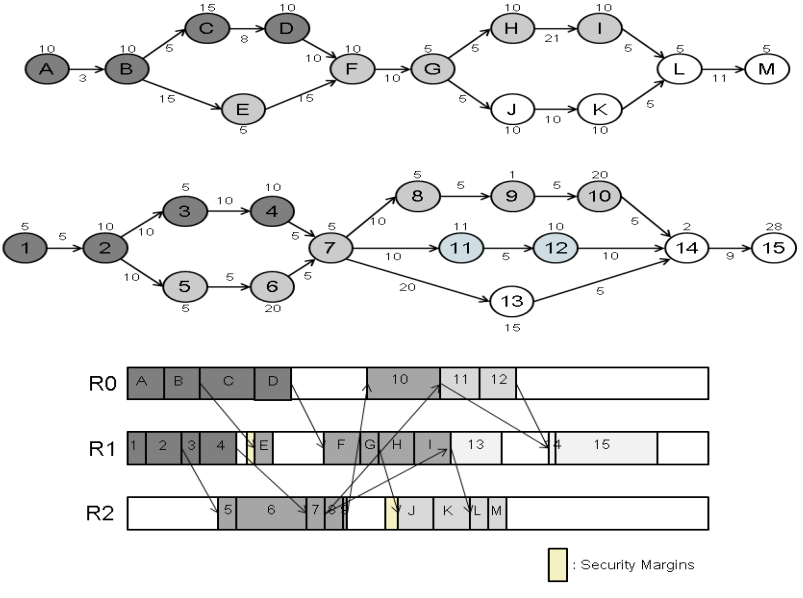


Fig. 2. Example of distributed gap search

Algorithm 2 in the following describes the distributed gap search scheme in details. Lines 14 to 26 deal with the case that a task group cannot fit into current gap. The while loop through lines 16 to 25 performs the re-clustering, which partitions the task group into two subgroups and ensures that the first subgroup can fit into current gap and contains as more tasks as possible. Lines 19 and 20 allocate the first subgroup into current gap, while line 21 recursively calls the distributed gap search scheme to allocate the second subgroup.

Algorithm 2. Distributed Gap Search: DGS(grp)

grp: task group to be scheduled
G: the set of time gaps on all resources
S_r: current schedule of resource r
m: number of tasks in grp; tasks are indexed from 1 to m

```
1: sort the gaps in G into a list L with a non-decreasing order of beginning time
2: k ←number of tasks in S_r
3: EFT(t_0,r) ← 0;
4: foreach time gap g in L do
5:         r ←the resource where g resides
6:         size_grp,r ← EFT( t^grp_m,r )-EST( t^grp_1,r )
7:         size_g← the time duration of gap g
8:         end_g← the end time of gap g
9:         Compute ESTs and EFTs on current gap ∀ t^grp_j ∈ grp
10:      D_{t_m^grp} ← tasks ahead the gap which depends on t^grp_m
11:      if (end_g - EST(t^grp_1,r ) ≥ size_grp,r ) and D_{t_m^grp} = ∅ then
12:            grp is allocated in current gap g;
13:               update S_r;
14:      else
15:            n = m-1;
16:            while n > 0 do
17:                  size_grp(1,n),r←EFT ( t^grp_n,r )-EST( t^grp_1,r )
18:                  if (end_g - EST( t^grp(1,n)_1,r )) ≥ size_grp(1,n),r then
19:                        grp(1,n) is allocated in current gap g;
20:                        update s_r;
21:                        call DGS(grp(n+1,m)); //recursive call
22:                        return;
23:                  else
24:                        n--;
25:            end while
26:      end if
27: end foreach
28: compare the schedule ends of all resources and allocate grp onto the end
    of the schedule on resource r which leads to the earliest start time of grp
    compared to other resources. //no gap found update s_r;
29: return;
```

5 Experiments and Discussions

This section presents a series of experiments which compare the proposed distributed gap search scheme with the original gap search algorithm in terms of average makespan through simulation studies. We implemented a DAG generator to produce workflows for the following simulation experiments. The DAG generator works as follows:

1. It generates a DAG with one front node and one end node.
2. Each DAG contains one to four fork-join structures randomly.
3. Each fork operation produces two or three branches.
4. Each branch contains two to four nodes randomly.
5. The generator can generate DAG's with different CCR values.
6. It assigns a random weight to each node and edge.

In the following experiments,

1. We simulate a workload with 500 DAGs.
2. Each DAG contains 10 to 50 nodes.
3. Each node has the computation cost ranging from 5 to 20 seconds.
4. Each edge has the communication cost ranging from 5 to 20 seconds.
5. Each experiment was conducted for 20 times and the average makespan value was calculated.

The following presents the experimental results. Figure 3 shows the performance results of 500 DAG's running on 30 resources. Experiments were conducted with workflows of two different CCR values, 0.1 and 10. Under all the CCR values, the proposed distributed gap search scheme outperforms the original gap search algorithm. Figure 4 illustrates that the proposed distributed gap search scheme delivers better performance through improving the overall resource utilization. Up to 18% performance improvement can be obtained as shown in the above experiments.

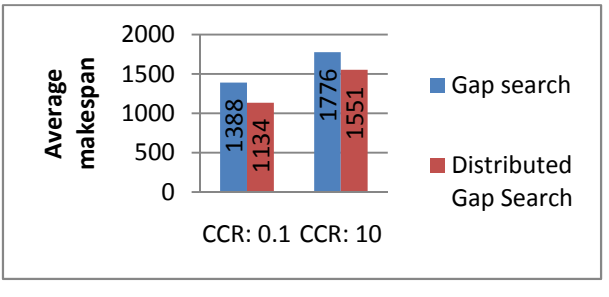


Fig. 3. Average makespan of 500 DAGs on 30 resources

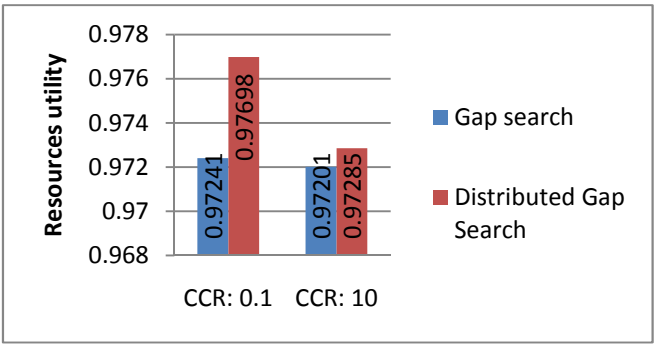


Fig. 4. Resource utilization of 500 DAGs on 30 resources

6 Conclusions

Concurrent workflow scheduling becomes a crucial issue in HPC cloud environments. This paper deals with this issue through a hybrid scheduling approach. At the first step, a clustering based PCH approach [1][4] is applied to cluster the tasks within a workflow into different task groups. At the second step, the list-based scheduling heuristic is accompanied with a gap search mechanism to allocate the task groups onto the resources. A distributed gap search scheme is proposed in this paper to further improve resource utilization and workflow completion time. The proposed scheme were evaluated with a series of simulation experiments on workflows of various structures and compared to the continuous gap search algorithm in [1][4]. The results indicate that the proposed distributed gap search scheme outperforms the previous continuous approach significantly in terms of average makespan. Up to 18% performance improvement was obtained in the experiments.

References

1. Bittencourt, L.F., Madeira, E.R.M.: Towards the Scheduling of Multiple Workflows on Computational Grids. Journal of Grid Computing 8, 419–441 (2009)
2. Kwok, Y.K., Ahmad, I.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999)
3. Adam, T.L., Chandy, K.M., Dickson, J.R.: A Comparison of List Schedules for Parallel Processing Systems. Communications of the ACM 17(12), 685–690 (1974)
4. Bittencourt, L.F., Madeira, E.R.M.: Fulfilling Task Dependence Gaps for Workflow Scheduling on Grids. In: 3rd IEEE International Conference on Signal-Image Technology and Internet Based Systems, pp. 468–475 (2007)
5. Stavrinides, G.L., Karatza, H.D.: Scheduling Multiple Task Graphs in Heterogeneous Distributed Real-Time Systems by Exploiting Schedule Holes with Bin Packing Techniques. Simulation Modelling Practice and Theory, vol 19(1), 540–552 (2011)
6. Bittencourt, L.F., Sakellariou, R., Madeira, E.R.M.: DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In: 18th 'Conference on Parallel, Distributed and Network-based Processing, pp. 27–34 (2010)

7. Wieczorek, M., Prodan, R., Hoheisel, A.: Taxonomies of the Multi-Criteria Grid Workflow Scheduling Problem. In: Grid Middleware and Services, pp. 237–264 (2008)
8. Rahman, M., Ranjan, R., Buyya, R.: Cooperative and Decentralized Workflow Scheduling in Global Grids. Future Generation Computer Systems 26, 753–768 (2010)
9. Ding, F., Zhang, R., Ruan, K., Lin, J., Zhao, Z.: A QoS-based Scheduling Approach for Complex Workflow Applications. In: 5th Annual ChinaGrid Conference, pp. 67–73 (2010)
10. Bittencourt, L.F., Madeira, E.R.M.: A Performance-Oriented Adaptive Scheduler for Dependent Tasks on Grids. Concurrency and Computation: Practice and Experience 20, 1029–1049 (2008)
11. Gary, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co, New York (1979)
12. Ullman, J.D.: NP-Complete Scheduling Problems. Journal of Computer and Systems Sciences 10, 384–393 (1975)
13. Zhao, H., Sakellarious, R.: Scheduling Multiple DAGs onto Heterogeneous Systems. In: 15th Heterogeneous Computing Workshop, 14 pp (2006)
14. Yu, Z., Shi, W.: A Planner-Guided Scheduling Strategy for Multiple Workflow Applications. In: 37th International Conference on Parallel Processing Workshops, pp. 8–12 (2008)
15. N'takpé, T., Suter, F.: Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations. In: IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8 (2009)
16. Kwok, Y., Ahmad, I.: Dynamic Critical-Path Scheduling: An Effective Technique for Allocation Task Graphs to Multi-processors. IEEE Transactions on Parallel and Distributed Systems 7(5), 506–521 (1996)
17. Sih, G.C., Lee, E.A.: A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. IEEE Transactions on Parallel and Distributed Systems 4(2), 175–186 (1993)
18. EI-Rewini, H., Lewis, T.G.: Scheduling Parallel Program Tasks onto Arbitrary Target Machines. Journal of Parallel and Distributed Computing 9, 138–153 (1990)
19. Yang, T., Gerasoulis, A.: DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Transactions on Parallel and Distributed Systems 5(9), 951–967 (1994)
20. Park, G., Shirazi, B., Marquis, J.: DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multi-processor Systems. In: International Conference on Parallel Processing, pp. 157–166 (1997)
21. Mandal, A., Kennedy, K., Koelbel, C., Marin, G., Mellor-Crummey, J., Liu, B., Johnsson, L.: Scheduling Strategies for Mapping Application Workflows onto the Grid. In: 14th IEEE Symposium on High Performance Distributed Computing, pp. 125–134 (2005)
22. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing 61(6), 810–837 (2001)
23. Hofmann, P., Woods, D.: Cloud Computing: The Limits of Public Clouds for Business Applications. IEEE Internet Computing, 90–93 (November 2010)
24. Wei, Y., Blake, M.B.: Service-Oriented Computing and Cloud Computing: Challenges and Opportunities. IEEE Internet Computing, 72–75 (November 2010)

25. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. Technical report no. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (2009)
26. salesforce.com, http://www.salesforce.com
27. Gmail, http://gamil.com
28. Google App Engine, http://code.google.com/intl/en/appengine
29. Microsoft Azure Platform, http://www.microsoft.com/windowsazure/
30. Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/
31. Akioka, S., Muraoka, Y.: HPC Benchmarks on Amazon EC2. In: 24th IEEE International Conference on Advanced Information Networking and Applications Workshops, pp. 1029–1034 (2010)
32. Kim, H.: el-Khamra, Y., Jha, S., Parashar, M.: An Autonomic Approach to Integrated HPC Grid and Cloud Usage. In: 5th IEEE International Conference on e-Science, pp. 366–373 (2009)

Efficient Decoding of QC-LDPC Codes Using GPUs

Yue Zhao, Xu Chen, Chiu-Wing Sham, Wai M. Tam, and Francis C.M. Lau

Department of Electronic and Information Engineering,
Hong Kong Polytechnic University, Hong Kong
{9901931r,chenxu,encwsham,tamwm,encmlau}@polyu.edu.hk

Abstract. In this work, we propose an efficient quasi-cyclic LDPC (QC-LDPC) decoder simulator which runs on graphics processing units (GPUs). We optimize the data structures of the messages used in the decoding process such that both the read *and* write processes can be performed in a highly parallel manner by the GPUs. We also propose a highly efficient algorithm to convert the data structure of the messages from one form to another with very little latency. Finally, with the use of a large number of cores in the GPU to perform the simple computations simultaneously, our GPU-based LDPC decoder is found to run at around 100 times faster than a CPU-based simulator.

Keywords: Belief propagation, CUDA, graphics processing unit (GPU), low-density parity-check codes, LDPC decoder.

1 Introduction

Low-density parity-check (LDPC) codes were invented by Robert Gallager [6] but had been ignored for years until Mackay rediscovered them [8]. They have attracted much attention recently because they can achieve excellent error performance based on the belief propagation (BP) decoding algorithm. Although the BP decoding algorithm involves intensive computations, it possesses a high data-parallelism feature. Since GPUs are highly parallel structures with many processing units they can provide a cheap, flexible and efficient solution of simulating a LDPC decoder.

In [4,3], the sum-product LDPC decoder and the min-sum decoder have been implemented with GPUs. Moreover, by combining sixteen fixed-point 8-bit data to from one 128-bit data, the LDPC decoder in [3] decodes sixteen codewords simultaneously and achieves a high throughput. The drawback is that the decoding process has to be continued until all the codewords have been correctly decoded or the maximum number of iterations is reached. Although the method in [3] allows coalesced memory access in *either* the read *or* write process, coalesced memory access in *both* the read *and* write processes is yet to be achieved.

In this paper, we develop a flexible and highly parallel quasi-cyclic LDPC (QC-LDPC) decoder simulator running on GPUs. Moreover, the BP decoding algorithm is implemented with floating-point precision. Our results have shown

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 294–305, 2011.

that compared with CPU-based simulators, our GPU-based LDPC decoder simulator improves the speed by around 100 times.

In the remainder of the paper, Section 2 reviews the LDPC codes and the BP decoding algorithm. Section 3 introduces the GPUs and Section 4 describes in detail the implementation of the GPU-based LDPC decoder. Section 5 shows our simulation results and we conclude the paper in Section 6.

2 Review of QC-LDPC Codes and the Belief Propagation Decoding Algorithm

2.1 QC-LDPC Codes

A binary (N, K) LDPC code is a linear block code defined by a sparse $M \times N$ parity-check matrix $\mathbf{H}$, where N represents the code length and $M = N - K$ denotes the number of parity checks. Moreover, the $M \times N$ matrix $\mathbf{H}$ contains mostly 0's and a relatively small number of 1's. The $\mathbf{H}$ matrix of a LDPC code with row weight 6 and column weight 3 is shown as follows.

$$\mathbf{H} = \begin{bmatrix} 1\,1\,1\,0\,0\,1\,0\,1\,0\,1 \\ 1\,0\,1\,1\,0\,0\,1\,0\,1\,1 \\ 0\,1\,0\,1\,1\,1\,1\,1\,0\,0 \\ 1\,0\,0\,0\,1\,1\,0\,1\,1\,1 \\ 0\,1\,1\,1\,1\,0\,1\,0\,1\,0 \end{bmatrix} \tag{1}$$

The code rate of the LDPC code is given by $R \geq 1 - M/N$ and the equality holds when $\mathbf{H}$ is full rank.

In a QC-LDPC code, the parity-check matrix $\mathbf{H}$ is composed of square sub-block matrices (also called sub-matrices) of size $p \times p$ [5,12]. We denote the (j, l)th sub-block matrix of $\mathbf{H}$ by $\mathbf{I}(p_{j,l})$ where $-1 \leq p_{j,l} \leq p - 1$ ($j = 0, 1, \ldots, J - 1$; $l = 0, 1, \ldots, L - 1$), $J = M/p$ and $L = N/p$. Moreover, $\mathbf{I}(p_{j,l})$ represents (i) a zero matrix when $p_{j,l} = -1$, (ii) an identity matrix when $p_{j,l} = 0$ and (iii) a circulant permutation matrix obtained by circularly shifting the identity matrix to the right by $p_{j,l}$ when $1 \leq p_{j,l} \leq p - 1$. Thus, we can represent a QC-LDPC code with J block rows and L block columns as

$$\mathbf{H} = \begin{bmatrix} \mathbf{I}(p_{0,0}) & \mathbf{I}(p_{0,1}) & \cdots & \mathbf{I}(p_{0,L-1}) \\ \mathbf{I}(p_{1,0}) & \mathbf{I}(p_{1,1}) & \cdots & \mathbf{I}(p_{1,L-1}) \\ \vdots & & \ddots & \vdots \\ \mathbf{I}(p_{J-1,0}) & \mathbf{I}(p_{J-1,1}) & \cdots & \mathbf{I}(p_{J-1,L-1}) \end{bmatrix}. \tag{2}$$

Figure 1 illustrates the structure of a QC-LDPC code with three block rows (i.e., $J = 3$) and four block columns (i.e., $L = 4$).

2.2 Belief Propagation Algorithm

LDPC codes are most commonly decoded using the belief propagation (BP) algorithm [10],[11]. Let μ_n be the initial log-likelihood ratio (LLR) that variable

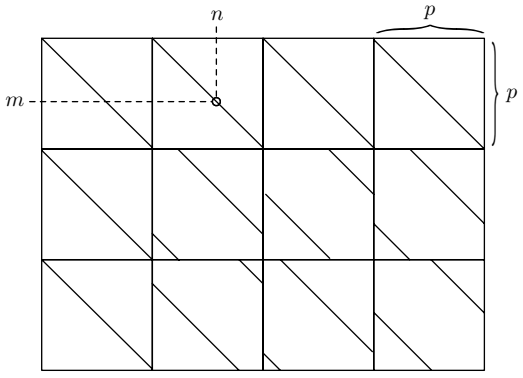


Fig. 1. The parity-check matrix of a QC-LDPC code with three block rows (i.e., $J = 3$) and four block columns (i.e., $L = 4$). Each square represents one sub-block matrix of size $p \times p$ and the solid line shows the positions of 1's in each sub-block matrix.

node n is a "0" to that it is a "1", i.e.,

$$\mu_n = \ln\left[\frac{\Pr(c_n = 0|y_n)}{\Pr(c_n = 1|y_n)}\right] \tag{3}$$

Initially, μ_n is calculated based on the channel parameter N_0 and the received signal y_n using $\mu_n = (4/N_0) \cdot y_n$ [7].

Define $\mathcal{N}(m)$ as the set of variable nodes that participate in check node m and $\mathcal{M}(n)$ as the set of check nodes connected to variable node n. At iteration i, let $\beta_{mn}^{(i)}$ be the LLR messages passed from variable node n to check node m; $\alpha_{mn}^{(i)}$ be the LLR messages passed from check node m to variable node n; and $\beta_n^{(i)}$ be the *a posteriori* LLR of variable node n. Then the standard BP algorithm can be described as follows [8,2].

1. Initialization: $\beta_{mn}^{(0)} = \mu_n$ for $0 \le n \le N-1$ and $m \in \mathcal{M}(n)$. We set the iteration number $i = 1$ and the maximum number of iterations to I_{max}.
2. Iteration:
   (a) Horizontal Step: For $0 \le m \le M-1$ and for $n \in \mathcal{N}(m)$, the check-to-variable messages are updated by

$$\alpha_{mn}^{(i)} = 2\tanh^{-1}\left[\prod_{n\prime \in \mathcal{N}(m)\backslash n} \tanh\left(\frac{\beta_{mn\prime}^{(i-1)}}{2}\right)\right] \tag{4}$$

   where $\mathcal{N}(m)\backslash n$ denotes the set $\mathcal{N}(m)$ excluding the variable node n.
   (b) Vertical Step: For $0 \le n \le N-1$ and $m \in \mathcal{M}(n)$, the variable-to-check messages are calculated by

$$\beta_{mn}^{(i)} = \mu_n + \sum_{m\prime \in \mathcal{M}(n)\backslash m} \alpha_{m\prime n}^{(i)} \tag{5}$$

   where $\mathcal{M}(n)\backslash m$ denotes the set $\mathcal{M}(n)$ with check node m excluded.

3. Finally, we compute the *a posteriori* LLRs $\beta_n^{(i)}$ and perform hard decisions.

$$\beta_n^{(i)} = \mu_n + \sum_{m' \in \mathcal{M}(n)} \alpha_{m'n}^{(i)} \tag{6}$$

3 Parallel Computations Using GPUs

3.1 GPU Architecture

A graphics processing unit (GPU) consists of multi-threaded, multi-core processors. Its high parallelism and large memory bandwidth offer very good performance for processing computer graphics. Compared with CPU, GPUs are especially effective when performing intensive and highly parallel computations. GPUs follow the single-instruction multiple-data (SIMD) paradigm.

Figure 2 shows the architecture of a typical GPU. It contains a number of multiprocessors called streaming multiprocessors (SMs). The SMs are executing functions in parallel asynchronously and each SM has a group of stream processors or cores. The device memory called the *global memory* can be access by all SMs, as shown in Fig. 2. There is also on-chip memory called *shared memory* shared among cores within each SM. The shared memory has a small capacity (tens of KB) but it has a low-latency. In our work, the GPU used has 7 stream multiprocessors (SMs) and each SM contains 48 cores [1]. Moreover, each SM has a limited shared memory of 48 KB and 768 MB global memory.

4 Implementation of a LDPC Decoder on GPUs

In the BP decoding algorithm, the most computation intensive processes are the check-node updating and the variable-node updating. In (4), it can be observed that the computations for each check node is independent of the computations for other check nodes. Furthermore, the computations performed for all check nodes are identical, but on different sets of data, so are the computations for the variable nodes (see (5)). So in the horizontal step of the BP algorithm, we assign one separate thread to process one check node, which is corresponding to one row of the parity-check matrix $\mathbf{H}$ in (1). Similarly, we assign a separate thread to the processing for one variable node in the vertical step of the BP algorithm, which is updating the messages in each column in $\mathbf{H}$. The horizontal step and vertical step are then mapped onto two separate kernels and are executed alternatively in one decoding iteration.

An important consideration for an efficient CUDA implementation is to ensure coalesced global memory access. The accesses (loads and stores) to the global memory by threads of a half warp (for devices of compute capability 1.x) or of a warp (for devices of compute capability 2.x) are coalesced into as few as one transaction when certain access requirements are met [9]. To meet these access requirement, the global memory should be viewed as aligned segments of 16 and 32 words (a word is 8-byte for float). In a simple coalesced access pattern, the

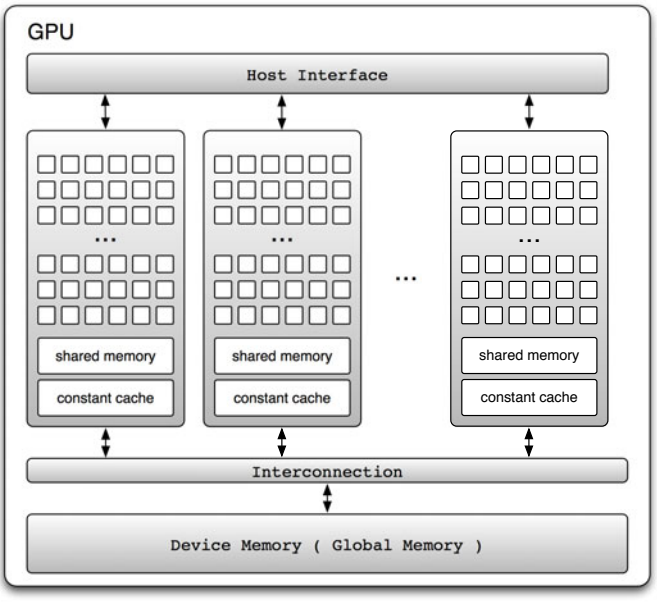


Fig. 2. Hardware architecture of a typical GPU [1]

k-th ($k = 0, 1, \ldots, 31$) thread in a warp accesses the k-th data in a memory segment. For example, if each data represents a 4-byte floating-point number, the segment size will be $32 \times 4 = 128$ bytes, and the start address of each segment should be a multiple of 128 bytes.

4.1 Data Structure to Represent the Messages

Each edge in a Tanner graph is associated with two messages — check-to-variable message and variable-to-check message. Recall that each edge also corresponds to a non-zero entry in the parity-check matrix $\mathbf{H}$, we only store the non-zero elements.

Referring to Fig. 3 in which a QC-LDPC code with 3 block rows ($J = 3$), 4 block columns ($L = 4$) and a sub-matrix size 6×6 ($p = 6$) is shown, we first consider the horizontal step which evaluates the check-to-variable messages based on the variable-to-check messages. As mentioned in the previous section, we assign $M = pJ(= 18)$ threads to process the M rows. To ensure that coalesced global memory access can be performed, the M variable-to-check messages in each block column will be stored in consecutive memory locations to form a "message segment". The data structure for the vertical step is similar, and we denote this check-to-variable message array by $\mathbf{H}_v$. Consequently, the kth ($k = 0, 1, \ldots, N-1$) thread can access its required data which corresponds to the data in the kth column of $\mathbf{H}_v$.

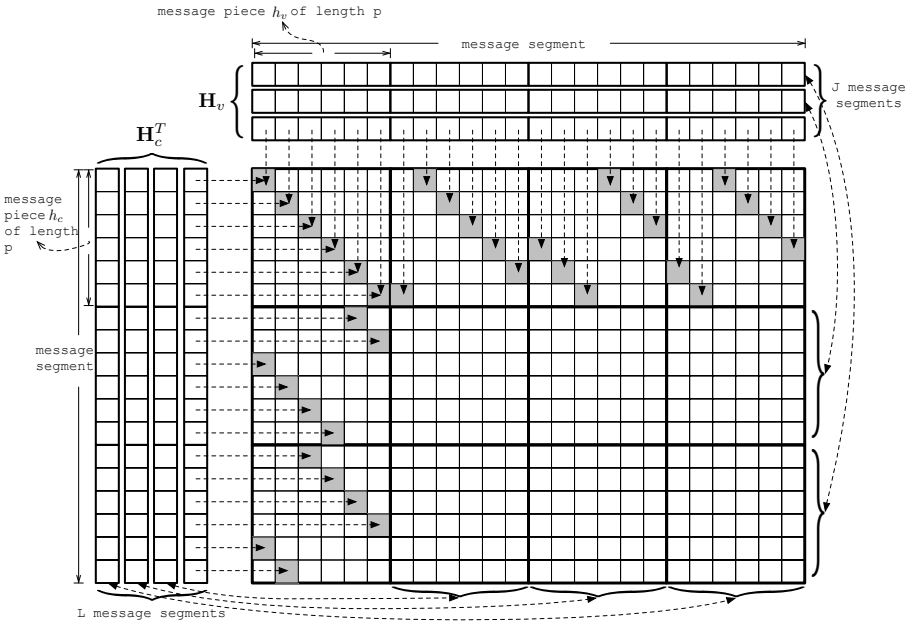


Fig. 3. The corresponding message pieces $\mathbf{h}_c$ and $\mathbf{h}_v$, message segments, two-dimensional arrays $\mathbf{H}_c$ and $\mathbf{H}_v$ for a QC-LDPC code. The gray squares denote $1's$ in the $\mathbf{H}$ matrix. Only 6 sub-matrices are illustrated in detail.

As we have mentioned, the aim of forming the two-dimensional message arrays $\mathbf{H}_c$ and $\mathbf{H}_v$ is to facilitate the coalesced global memory access. With the use of coalesced memory access, different threads will be able to process simultaneously different column vectors in $\mathbf{H}_c$ or $\mathbf{H}_v$. Recall that there are pJ and pL columns in $\mathbf{H}_c$ and $\mathbf{H}_v$, respectively, and that threads are executed in groups of 32 . Thus, we propose appending γ dummy messages to each *message piece*, formed by the variable-to-check messages or check-to-variable messages in one sub-matrix, such that the number of messages in each *modified message piece* becomes a multiple of 32 units . For example, if $p = 422$, γ can be set to 26 such that $p' = p + \gamma = 448 = 32 \times 14$. Note that the total numbers of threads required in the horizontal step and the vertical step are now $M' = p'J$ and $N' = p'J$, respectively. Since the dummy messages are appended to the end of the valid messages, we can easily control the computations of the threads such that the dummy messages will not affect the original calculations.

Before the check-node-updating kernel and variable-node-updating kernel, the data structure should be in $\mathbf{H}'_c$ format and $\mathbf{H}'_v$ format, respectively. For QC-LDPC code, the conversion between $\mathbf{H}'_c$ and $\mathbf{H}'_v$ is just permutation of the elements, as shown in Fig. 4. Note that during the conversions, the dummy messages are kept untouched and will not affect the original iteration results.

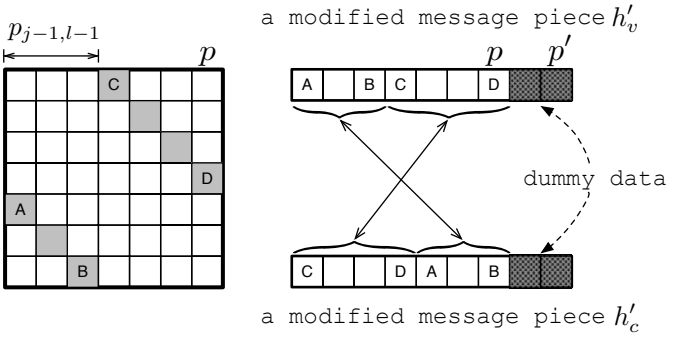


Fig. 4. Conversion of modified message pieces $\mathbf{h}'_v$ (in $\mathbf{H}'_v$) and $\mathbf{h}'_c$ (in $\mathbf{H}'_c$)

4.2 Decoding Procedures in GPU with the Use of Shared Memory

In the initialization kernel, $N' = p' \times L$ threads are assigned to compute the initial LLRs μ_n based on the received messages and broadcast them to the neighboring check nodes of each of the variable nodes. The variable-to-check messages in $\mathbf{H}'_v$ format so formed is then converted into $\mathbf{H}'_c$ format.

Check-Node-Updating Kernel. In the check-node updating kernel that follows, each thread processes one column of the $\mathbf{H}'_c$ matrix (size $L \times p'J$). Totally there are $M' = p'J$ threads allocated and L data in each column. First, the threads load/copy the required data from the global memory to the shared memory in a coalesced way. Once the data are in the shared memory, they can be further accessed extremely fast.

At the end of the check-node updating kernel, the updated check-to-variable messages are copied from the shared memory to the global memory in a coalesced way. The memory access during storing is the same as the memory access during loading. Thus the check-to-variable data are still represented in the $\mathbf{H}'_c$ structure, which will be converted to $\mathbf{H}'_v$ format by the conversion kernel described below.

Conversion from $\mathbf{H}'_c$ to $\mathbf{H}'_v$. Recall that each *modified message piece* (of size $p' = p + \gamma$) in $\mathbf{H}'_c$ has a corresponding sub-matrix $\mathbf{I}(p_{j-1,l-1})$ ($j = 1, 2, \ldots, J$; $l = 1, 2, \ldots, L$) in the parity-check matrix $\mathbf{H}$. Therefore, there is a total of $J \times L$ *modified message pieces* in $\mathbf{H}'_c$. In the $\mathbf{H}'_c$-to-$\mathbf{H}'_v$ conversion kernel, we assign each *modified message piece* with a thread-block and we place $J \times L$ thread-block in a grid. If the number of elements in one *modified message piece* (i.e., p') is no larger than the maximum thread-block size $\Omega_{\max}$ (currently equals 1024), we can assign a sufficient number of threads (e.g., p' threads) in each thread-block such that each thread only needs to process one element in the *modified message piece*. On the other hand, if the number of elements in one *modified message piece* is larger than the maximum thread-block size, each thread must process multiple elements. In summary, the number of elements to be processed by each thread depends on the relative size of a thread-block to a *modified message piece*.

As in Sect. 4.1, we denote a *modified message piece* in $\mathbf{H}'_c$ by $\mathbf{h}'_c$ and assume that it corresponds to the sub-matrix $\mathbf{I}(p_{j-1,l-1})$. In each thread-block, the threads load $\mathbf{h}'_c$ from the global memory and store them in the shared memory in a coalesced way. Then, the threads in the same BLOCK look up a table[1] and find the corresponding value $p_{j-1,l-1}$ to be used. Based on $p_{j-1,l-1}$, each thread copies one or multiple element(s) in $\mathbf{h}'_c$ and saves it/them accordingly in the modified message piece $\mathbf{h}'_v$ in the shared memory. Finally, $\mathbf{h}'_v$ is copied from the shared memory to the global memory in a coalesced way and is stored as part of $\mathbf{H}'_v$.

Variable-Node-Updating Kernel. The variable-node-updating kernel is similar to the check-node-updating kernel. It is worth noting that the initial LLRs μ_n, which are stored in the global memory, are also required for updating the variable-to-check messages.

Conversion from $\mathbf{H}'_v$ to $\mathbf{H}'_c$. Since the operations of the $\mathbf{H}'_v$-to-$\mathbf{H}'_c$ conversion kernel is very similar to those of the $\mathbf{H}'_c$-to-$\mathbf{H}'_v$ kernel described in Sect. 4.2, they will not be explained here.

5 Results and Discussions

We compare the results of the proposed GPU-based LDPC decoder with that of a CPU-based decoder. The CPU-based decoder is developed using C programming and the commonly used linked-list approach is employed to store and link the messages [8]. Details of the CPU and GPU used in our simulations are presented in Table 1. Note that although there are 8 cores in the CPU, simple C programming (without parallel computing) allows us to use only one of the cores. Table 2 shows the characteristics of the QC-LDPC codes under test.

In the following, we fix the number of decoding iterations to be 30 and the simulation terminates after 100 (codeword) block errors are received.

5.1 Simulation Results

BLER/BER. We plot the bit error rate (BER) and the block error rate (BLER or the codeword error rate) of Code A and Code C in Fig. 5 when the E_b/N_0 ranges from 2.7 dB to 3.3 dB. We observe that the BLER/BER performance given by the CPU-based decoder and the GPU-based decoder are very close. It is because both decoders use the same computation algorithm and the same floating-point precision. The only difference are the noisy vectors being generated and used as different seeds are used in the two decoders.

We also find that the BER/BLER reduces from Code A to Code E. It is reasonable because for codes with the same code rate (Code A to Code D), the

[1] We store the $p_{j-1,l-1}$ values as a two-dimensional $(J \times L)$ look-up table in the *constant cache* of the GPU. The time taken to access the *constant cache* is extremely short.

Table 1. Simulation environments

	CPU	**GPU**
Platform	Intel Xeon	Nvidia GTX460
Number of cores	8 (only one core used)	$7 \times 48 = 336$
Clock rate	2.26 GHz	0.81 GHz
Memory	8 GB DDR3 RAM	768 MB global memory and 48 KB shared memory
Maximum Thread-block size $\Omega_{\max}$	—	1024 threads
Programming Language	C	CUDA C

Table 2. Parity-check matrices of the QC-LDPC codes under test. Each of the sub-matrices in the parity-check matrices is either the identity matrix or a circulant permutation matrix obtained by circularly shifting the identity matrix.

Code	$J \times L$	**Sub-matrix size** $(p \times p)$	**Parity-Check Matrix Size** $(M \times N)$	**Number of Edges** $(M \times L$ or $N \times J)$
A	4×24	422×422	1688×10128	40512
B	4×24	632×632	2528×15168	60672
C	4×24	765×765	3060×18360	73440
D	4×24	1024×1024	4096×24576	98304
E	3×6	3000×3000	9000×18000	54000

BLER/BER generally improves with the code-length. Code E, even at a E_b/N_0 of 2.3 dB, outperforms the other codes because it has a very low code rate (1/2) and relative long code-length.

Decoding Time. Table 3 shows the number of transmitted codewords and the simulation times for different codes when the CPU-based and the GPU-based decoders are used.

We consider the average time for decoding one codeword for the GPU-based decoder, i.e., t_{GPU}. Similar to the CPU-based decoder, t_{GPU} increases from Code A to Code D. The reason is that an increasing sub-matrix size $p \times p$ (and hence $p' \times p'$) creates more computations and hence longer simulation per codeword. We further find that t_{GPU} for Code E is smaller than all those for Code A to

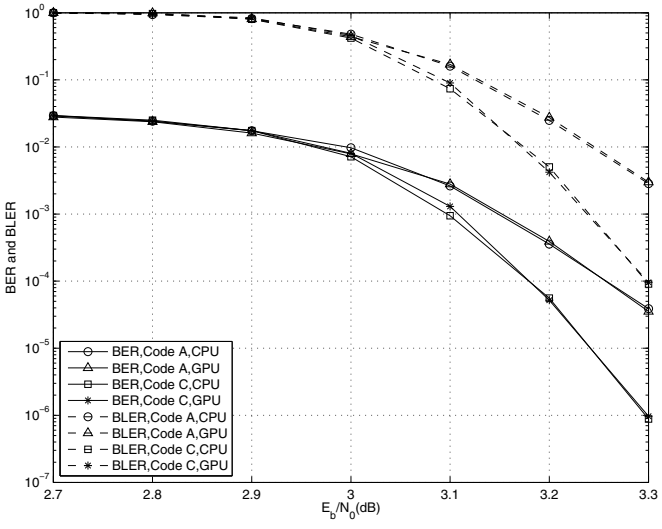


Fig. 5. BER and BLER curves of Code A and Code C. 30 iterations are used for decoding one codeword.

Code D. Although the number of edges for Code E is larger than that in Code A, the higher degree of parallel decoding used in Code E allows Code E to produce a shorter simulation time per codeword compared with Code A.

Finally, we compare the simulation times of the CPU-based decoder and the GPU-based decoder by taking the ratio $t_{\mathrm{CPU}}/t_{\mathrm{GPU}}$. The results in Table 3 indicate that the GPU-based decoder accomplish speedup improvements from 82

Table 3. Comparison of the simulation times using the CPU-based decoder and the GPU-based decoder. C represents the total number of decoded codewords; T denotes the total simulation time; t is the average simulation time per codeword and t' represents the average simulation time per codeword per edge. Code A to code D are decoded at a E_b/N_0 of 3.2 dB and code E is decoded at a E_b/N_0 of 2.3 dB. 30 iterations are used for decoding one codeword.

Code	**CPU**				**GPU**				**Speedup**
	C_{CPU}	T_{CPU} (s)	t_{CPU} (ms)	t'_{CPU} (μs)	C_{GPU}	T_{GPU} (s)	t_{GPU} (ms)	t'_{GPU} (μs)	t_{CPU}/t_{GPU}
A	4058	1270	313	7.726	3670	14	3.8	0.094	82
B	11664	5350	458	7.548	13388	67	5	0.082	92
C	20046	10950	546	7.435	23738	126	5.3	0.072	103
D	70843	51580	728	7.25	77224	613	7.9	0.08	92
E	1103613	428485	388	7.2	944501	3006	3.2	0.06	121

times to 121 times compared with the CPU-based decoder. In another simulation, we decode Code C at a E_b/N_0 of 3.3 dB until 100 block errors are found. The CPU-based decoder takes almost 170 hours (more than a week) to complete the simulation while the GPU-based decoder takes only 1.6 hours. The GPU-based decoder therefore shows a speedup improvement of over 100 times compared with the CPU-based decoder.

6 Conclusion

In this paper, we have developed a GPU-based QC-LDPC decoder. To reduce the memory access latency, we have designed efficient data structures to store the messages passed between the variable nodes and check nodes. These data structures allow coalesced memory access to the global memory. We have also developed a highly data-parallel model to implement the BP decoding algorithm. By using the shared memory more effectively, we reduce the frequency of accessing the global memory. Consequently, the decoding time is further reduced. The proposed GPU-based decoder can decode QC-LDPC codes with long length and high code rate. Besides, the decoder is flexible and scalable, and can be run on the latest or even future GPUs which possess more hardware resources than the current ones. Compared with the traditional CPU-based decoder, results show that the proposed GPU-based decoder is about 100 times faster. A 170-hour CPU-based simulation is now reduced to only 1.6 hours when a GPU-based decoder is used.

Acknowledgements. The work described in this paper was partially supported by a grant from Huawei Technologies Co. Ltd., China (Project No. H-ZG49).

References

1. The current generation CUDA architecture, code named fermi, http://www.nvidia.com/object/fermi_architecture.html
2. Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M., Hu, X.: Reduced-Complexity decoding of LDPC codes. IEEE Transactions on Communications 53(8), 1288–1299 (2005), doi:10.1109/TCOMM.2005.852852
3. Falcao, G., Silva, V., Sousa, L.: How GPUs can outperform ASICs for fast LDPC decoding. In: Proceedings of the 23rd International Conference on Supercomputing, pp. 390–399. ACM, Yorktown Heights (2009), http://portal.acm.org/citation.cfm?id=1542275.1542330
4. Falcao, G., Sousa, L., Silva, V.: Massive parallel LDPC decoding on GPU. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 83–90. ACM, Salt Lake City (2008), http://portal.acm.org/citation.cfm?id=1345206.1345221
5. Fossorier, M.: Quasicyclic low-density parity-check codes from circulant permutation matrices. IEEE Transactions on Information Theory 50(8), 1788–1793 (2004), doi:10.1109/TIT.2004.831841
6. Gallager, R.G.: Low-Density Parity-Check Codes. The MIT Press, Cambridge (1963)

7. Hu, X., Eleftheriou, E., Arnold, D., Dholakia, A.: Efficient implementations of the sum-product algorithm for decoding LDPC codes. In: IEEE Global Telecommunications Conference, GLOBECOM 2001, vol. 2, p. 1036E (2001)
8. MacKay, D.: Good error-correcting codes based on very sparse matrices. IEEE Transactions on Information Theory 45(2), 399–431 (1999)
9. Nvidia, C.: CUDA C best practices guide version 3.2. Tech. rep., NVIDIA Corporation (2010)
10. Richardson, T., Shokrollahi, M., Urbanke, R.: Design of capacity-approaching irregular low-density parity-check codes. IEEE Transactions on Information Theory 47(2), 619–637 (2001)
11. Richardson, T., Urbanke, R.: Efficient encoding of low-density parity-check codes. IEEE Transactions on Information Theory 47(2), 638–656 (2001)
12. Tam, W., Lau, F., Tse, C.: A class of QC-LDPC codes with low encoding complexity and good error performance. IEEE Communications Letters 14(2), 169–171 (2010)

A Combined Arithmetic Logic Unit and Memory Element for the Design of a Parallel Computer

Mohammed Ziaur Rahman

Dept. of Computer Science and Technology
University of Malaya, Kuala lumpur 50603, Malaysia
m.ziaur.rahman@ieee.org, ziaur@um.edu.my

Abstract. Memory-CPU single communication channel bottleneck of the von Neumann architecture is quickly stalling the growth of computer processors. A probable solution to this problem is to fuse processing and memory elements. A simple low latency single on-chip memory and processor cannot solve the problem as the fundamental channel bottleneck will still be there due to the logical splitting of processor and memory. This paper presents that a paradigm shift is possible by combining Arithmetic logic unit and Random Access Memory (ARAM) elements at bit level. This bit level modest ARAM is used to perform word level ALU instructions with minor modifications. This makes the ARAM cells capable of executing instructions in parallel. It is also asynchronous and hence reduces power consumption significantly. A CMOS implementation is presented that verifies the practicality of the proposed ARAM.

Keywords: Computer Architectures, Parallel Architectures, Memory Architectures.

1 Introduction

Brains inspired and guided development of computers including von Neumann computers which have seen a meteoric growth in the past half century. Though the von Neumann architecture provided sufficient opportunity for growth, it is still barely compatible with the abilities of human brains. The von Neumann bottleneck is the single communication channel between memory and processor as identified by Backus in his Turing award lecture [1]. Significant efforts were put following Backus' lecture to overcome this bottleneck. However, liberating computing from von Neumann bottleneck still remains to be a far fetched goal.

On the other hand the bonanza of von Neumann computing was supported by Moore's law that predicts doubling the number of transistors every three years. However, Moore's law is about to come to an end as the size of transistors is approaching molecular dimension. A paradigm shift is necessary if the computing is to grow at similar pace. Similar conclusion is reached from Amdahl's law. This is also recognized by the industry and they shift their focus to multi-core instead of furthering higher clock speed and instruction level parallelism [2].

The brain is a massively parallel system composed of low performance asynchronous *memory and processor cells* often simply referred as *memory cells*.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 306–317, 2011.

Brain components or neurons operate at timescales of a milli-seconds or greater. Biologically inspired computers try to achieve similar results with network processors having asynchronous stimulous. However, the core processor architecture remains unaltered [3].

The massive parallelism available in brain is recognized by early researchers during 70s and the concept of logic-in-memory came into existence [4]. From there-on, many different variants of logic-in-memory systems evolved. Computational RAM (CRAM) is one such system where elementary processing elements are attached with each column of RAM blocks [5]. Similarly, Processor in Memory (PIM) arrays are constructed for building a Terasys workstation [6]. The CRAM and PIM capabilities are quite limited as they integrate single bit processors with memory columns. Harnessing the full processing capability requires merger of individual bit operations which is quite expensive.

On the other hand, due to the advent of billion transistor era, it was conceptualized that RAM can be integrated with processor in a single chip. Thus, Intelligent RAM (IRAM) is proposed that merges a full processor with conventional dynamic RAM (DRAM) in a chip [7]. It reduces memory access latency and supports high bandwidth data access to the CPU. However the memory-CPU logical division is still there and so the von Neumann bottleneck persists.

The present trend in high speed computing measured by the CPU clocks has also reached to the level from where Amdahl's law starts playing a significant role e.g. not much gain is possible by simply increasing the clock rate. Therefore the future lies with clock-less/asynchronous processors [8].

The asynchronous processors have the potential for construction of low power, high-speed and reliable processors [8]. It can solve many problems of synchronous circuits such as usage of worst-case clock cycle, global clock distribution, heat dissipation and noisy behavior due to high interference from simultaneous switching[9]. Moreover, asynchronous processors will also allow for modular circuit design as the individual components can be seamlessly integrated due to the freedom from clock synchronization requirements [5].

On this background, we can summarize the design choices for a futuristic computer architecture to mimic brain characteristics, e.g. it should be asynchronous and should support cell-level parallel processing capabilities. It is a well paid trade off to sacrifice individual processing speed for massive parallelism as evident from the biological example.

An asynchronous, parallel Arithmetic logic and Random Access Memory (ARAM) element is presented here that follows above mentioned design choices. A high-level diagram of an ARAM based computer is shown in Figure 1. A number of ARAM cells are combined into a block having a control unit and a floating point unit. The ARAM blocks are connected to a public bus available across applications and individual application level private bus. The application level private bus is actually a common bus that can be cut-off from another application by using transmission gates in-between ARAM blocks. Due to fitting of applications in a block of ARAM, some ARAM cells may remain unused as highlighted. The first application should perform supervisory actions and hence

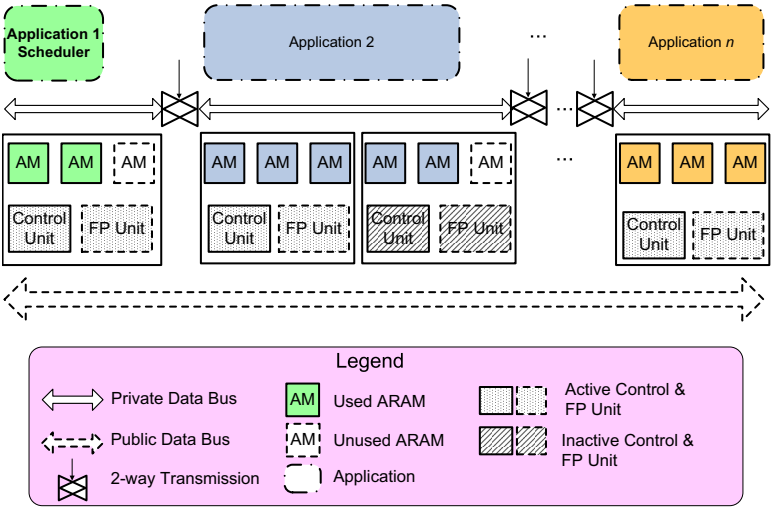


Fig. 1. A High Level Block Diagram of combined ARAM

is mentioned as a scheduler that will allocate/free memories to/from applications and supervise their execution. Some applications (e.g. Application 2 in Figure 1) may be larger than a single ARAM block. They will be extended to adjacent blocks, however only the controller and FP unit of the first block will be active for these larger applications. The memory addressing will be relative for each applications. However, a global addressing will also be active for sharing of data among the applications. This will take place through a separate public bus as shown in the figure.

The design works in truly parallel manner for independent memory elements and unlike CRAM or PIM it incorporates full fledged word-level processing capability to memories. The memory element is static and hence there is no need to refresh the memories optimizing power efficiency. As the instruction sets will be memory-memory only it can significantly reduce number of instructions. Memory access instructions and cache mechanisms will no longer be required as it will be immediate. The multi-threaded operations will be running truly concurrently as internally each thread will have their own processor and hence it is not a conventional time-sharing architecture. The fundamental unit of this architecture is an ARAM cell. The design of an ARAM cell will be shown to be a quite simple one having regular architecture and linear interconnection and area requirements. Thus, it will be quite practical to fabricate in a VLSI chip.

The remainder of the paper is organized as follows. Section 2 presents the fundamental theoretical basis for combined ALU and RAM cell by the introduction of recursive Parallel Self-Timed Adder (PASTA) circuit. Section 3 discusses the transition from PASTA to ARAM with detailed discussion on how a simple adder can be utilized for general purpose ALU operations with very small increase in control overhead. Section 4 provides CMOS implementation of a single

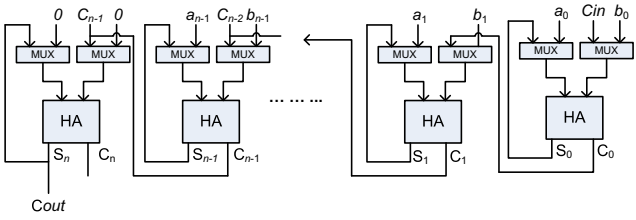


Fig. 2. Architecture of PASTA

bit ARAM cell. A performance comparison of ARAM using SPEC2000 program data is presented in Section 5. Finally, Section 6 draws some conclusions.

2 Theoretical Basis for Parallel Self-Timed Adder

The Asynchronous Arithmetic Logic Unit and Random Access Memory element (ARAM) is based on recursive circuit of [10] which works in a different mode than fundamental mode circuits or Muller circuits. The fundamental principle of recursive Parallel Self-Timed Adder (PASTA) is briefly described here whereas the complete details can be found in [10].

Let $A = (a_{n-1}a_{n-2}\cdots a_0)_2$ and $B = (b_{n-1}b_{n-2}\cdots b_0)_2$ be two n-bit binary operands. The recursive formula for binary addition of A and B is presented as follows.

Let S_i^j and C_i^j be the Sum and Carry respectively for i^{th} bit at the j^{th} recursion. The initial condition for addition is formulated as follows:

$$\begin{aligned} S_i^0 &= a_i \oplus b_i \ . \\ C_i^0 &= a_i b_i \ . \end{aligned} \tag{1}$$

The j^{th} iteration for the recursive addition can be found as follows:

$$S_i^j = S_i^{j-1} \oplus C_{i-1}^{j-1}, 0 \leq i < n \ . \tag{2}$$

$$C_i^j = S_i^{j-1} C_{i-1}^{j-1}, 0 \leq i \leq n \ . \tag{3}$$

The recursion is terminated at the k^{th} iteration when the following condition is met.

$$C_n^k C_{n-1}^k \cdots C_0^k = 0, 0 \leq k \leq n \ . \tag{4}$$

The correctness and further analysis of the above recursive formulations is proved in [10]. Its high-level architecture is shown in Figure 2.

The selection signal for 2 input multiplexers will be a single 0 to 1 transition denoted by *SEL*. It will initially select the actual inputs during "SEL = 0" and will switch on to feedback/carry paths for subsequent iterations (SEL = 1). The feedback paths enable the recursion to continue till the terminating condition is met. The terminating condition will trigger a signal ($TERM$) to go high

and hence will indicate completion of the current operation. PASTA is a logarithmic complexity adder without any lookahead/prefix computation schema. This makes it quite fast even though its design is pretty simple. It also runs in parallel for those number of bits that do not need any carry propagation. The asynchronous operation ensures that it does not waste any time due to synchronization.

The feedback path also makes PASTA a volatile memory element. As long as the condition (SEL = 1) persists, the circuit remembers its present value. In the next section we discuss how this gentle adder can be used to perform all kinds of integer and logical operations.

3 Design of ARAM Memory-cum-Logic Units

PASTA was designed to act as an adder circuit. A key feature of PASTA architecture is the feedback path that gives it memorization capability. Thus, PASTA can perform both as an adder and a memory element. However, the challenge remains to integrate all other operations that a typical ALU performs. If the circuit complexity increases significantly to implement the rest, it might be costly and impractical. However, we will show that the same recursive adder can be used to perform all integer and logical operations of an ALU. Only a small control overhead will be involved to achieve this. In this section we will initially show how the same recursive adder can be used to perform all integer and logical operations of an ALU. Subsequently, we will discuss about the other category of instructions i.e. memory access operations and control instructions. Finally we will discuss about instruction cycles needed for execution of ARAM instructions.

3.1 Logical Operations

During the recursive phase, PASTA has two input signals that change the output of the circuit. These are carry input (C_{in}) and the feedback input (S_i). These inputs are actually enabling the circuit to perform as an adder. An interesting aspect of the architecture is that we can use other input values during the recursive phase to perform different operations than simple addition. This is further explained as follows.

Not Operation: This complementary output is immediately available from single bit implementation of PASTA ($\overline{S_i}$).

Or Operation: Or is logically equivalent to $S_i = a_i \oplus b_i \oplus a_i b_i$ e.g. $S_i = a_i \oplus b_i \oplus C_i$. Therefore if the feedback uses C_i, S_i will hold the result of logical Or operation. As the (S_i, C_i) could not assume the value of (1,1) according to equations (2) and (3) , the C_i during recursion will be zeroed and hence the operation will successfully terminate.

And Operation: The implementation of AND operation is a bit tricky even though it is directly computed by C_i during the initial phase. To retain logical AND in the PASTA memory, the feedback is momentarily broken during

recursive phase to pass C_i in place of S_i while the carry input C_{in} will be held to ground. As the C_{in} is held 0 current input conditions will cause C_i to change to 0. Therefore, to shield the output from present input value and to retain correct AND output, the circuit will switch on to recursive mode as soon as C_i reaches 0 (indicating end of the AND operation). It is to be noted that due to the delay calibration in PASTA the current change in input cannot affect the output and hence it retains the previous stage AND value appropriately.

Xor Operation: As initially $S_i = a_i \oplus b_i$, the feedback will use GND during recursive phase thereby retaining the Xor output. The GND signal will also drive C_i to low value and recursion will be properly terminated.

Shift Left Logical: This functionality is provided by the fact that shift left is indeed doubling the value. Therefore single bit shift left is performed by adding the operand with itself.

Shift Right Logical: This functionality is performed by considering the fact that right shift operation is the opposite of left shift. Conceptually, it is identical to left shift with the position of the bits reveresed. Therefore, the carry outputs from next bit is used instead of carry output from preceding bit as the carry input in current bit adder. This will reverse the bit positions. The same operand is once again applied to both a_i and b_i for right shift operation.

Shift Right Arithmetic: This functionality is performed similar to shift right logical with the exception that C_{in} for the rightmost adder block is set to a_{n-1}. Thereby, during recursive phase rightmost bit is properly set to 1 or 0 to perform the sign extension.

Subtraction: Two's complement subtraction is performed as usual for common logic circuits by the equation $A + \overline{B} + 1$. Therefore, to perform subtraction 1's complement of the 2^{nd} operand is made available in the data bus while C_{in} for the least significant adder block is set to 1.

3.2 Memory Access Operations

A key difference of ARAM is its independence from load-store requirements of conventional processors. The central processing implies data must be transferred to the CPU before an intended operation is performed. On the contrary, in ARAM, processing instructions are made available to data cells for the desired operation. Therefore, load-store instructions will not be required by ARAM. Memory movement instructions, however will still be necessary. This is easily realized by addition operation with one operand being the desired data to be moved and the other being zero.

3.3 Control and Floating Point Operations

There will be a single control unit and Floating Point (FP) unit for a block of memory. The control instructions will be performed by the control unit after instruction decoding for manipulation of Program Counter (PC) and Stack pointer

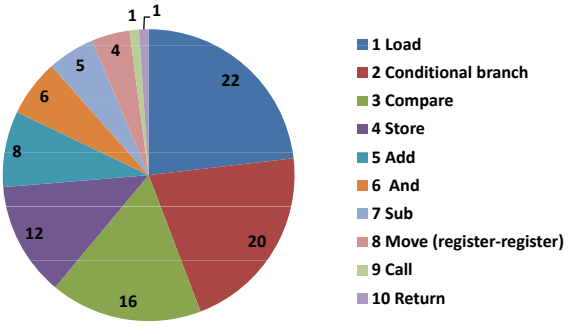


Fig. 3. The average top 10 instructions of the five SPECint92 programs for 80x86 [2]

(SP). The FP unit will be similarly executing floating point instructions. As highlighted in Figure 3, simple instructions use 95% of the CPU time and therefore separate floating point units will not become a bottleneck for program execution.

3.4 Instruction Cycles

The ALU in memory works in a radically different manner as it moves instructions to the data rather than making data available to the proceccsing unit. Thus it has distinct advantages for optimizing instruction execution cycles. In this paradigm, the memory is accessible immediately and hence memory access cycles are not required. Therefore, the classical five stage instruction cycles composed of Instruction Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM) and Write-Back (WB) can be reduced to three stage cycles of IF, ID and EX only. This makes the programs executing faster than conventional computers. Moreover, due to the avoidance of load-store instructions the programs sizes reduces significantly making them further space and time efficient.

Though we compare ARAM with five stage instruction architectures in terms of cycles, it should be noted that ARAM is asynchronous and synchronous clock cycles are not a match for ARAM. The asynchronous cycles will be signalled by completion signals and hence it will be different for different instructions and will be faster than synchronous circuits which are adapted for worst case intstruction cycles. The asynchrnous handshaking using completion detection is also much simpler than micropipelines introduced by Sutherland [11]. However, the complete discussion on this matter is out of scope for this paper due to space limitations and present focus on ALU in RAM related discussions only.

However, this optimization will not be applicable for floating point instructions. The FP unit inside a memory block will perform floating point instructions and for the moment we can consider them similar to current CPUs except that due to their multiplicity with each memory units they will not have any costly optimization. The three stage instruction cycles will not be applicable to floating point routines and five cycle execution is considered in the later performance analysis section.

Overall, considering the chart in Figure 3, it is clear that simple instructions account for most of the CPU time. Moreover, memory load accounts for 22% and combined memory instructions (load, store, register-register mov) accounts for 46% or nearly half of all processing time. It is to be noted that though it is based on integer SPEC-92 profiling, it is indeed representative for average integer and floating point programs as evident in Section 5.

4 Implementation of ARAM

A Complementary Metal Oxide Semiconductor (CMOS) implementation of the ARAM single bit cell is shown in Figure 4(a). The implementation is modified from PASTA to perform all ALU operations as discussed in Section 3. The use of additional multiplexer at the output (T_3) ensures that input is available for the proper duration with further explanation given in [10].

The termination signal ($TERM$) following equation (4) can be generated at word level as shown in Figure 4(d). In addition to the C_i's, the complement of INP_1 and INP_2 signals are also AND-ed together in $TERM$ to ensure that the termination cannot be accidentally turned on during the initial input selection phase.

Though $TERM$ signal is the only block where as many as $n+4$ interconnections are needed, it will not create any fan-in problem as all the connections are parallel. It is also the reason to use inverted logic for C_i's.

The total number of transistors for single bit ARAM cell is 43 without the word level completion detection circuit. The $TERM$ will be constructed for a word of ARAM. Hence for a 64-bit word the total transistors needed is 2820 compared to 384 for a 6T static RAM cell. This is roughly seven times more than usual SRAM cell. With the additional overhead for extra buses and control and floating point units the number of transistor/interconnection requirement will be similarly higher in ARAM. We expect it would not exceed more than ten times than regular SRAMs. Thus a billion transistor cell can represent 16 MB of memory compared to 166 MB in SRAMs.

5 Performance Evaluation

ARAM is an asynchronous, parallel logic and memory system and hence head-to-head comparison with synchronous processors is not possible. Therefore, we adopt the following strategies to create a fare and conservative comparison between these two.

1. Equal cycle duration: We assume asynchronous single task execution will take same time as synchronous single clock cycle. It is to be noted that asynchronous handshaking is very low overhead and hence single operation execution will take nearly equal time the function will take for execution.
2. Single process execution: For the sake of benchmarking, we avoid comparison with parallel-processing available in ARAM. Thus, the benchmark will

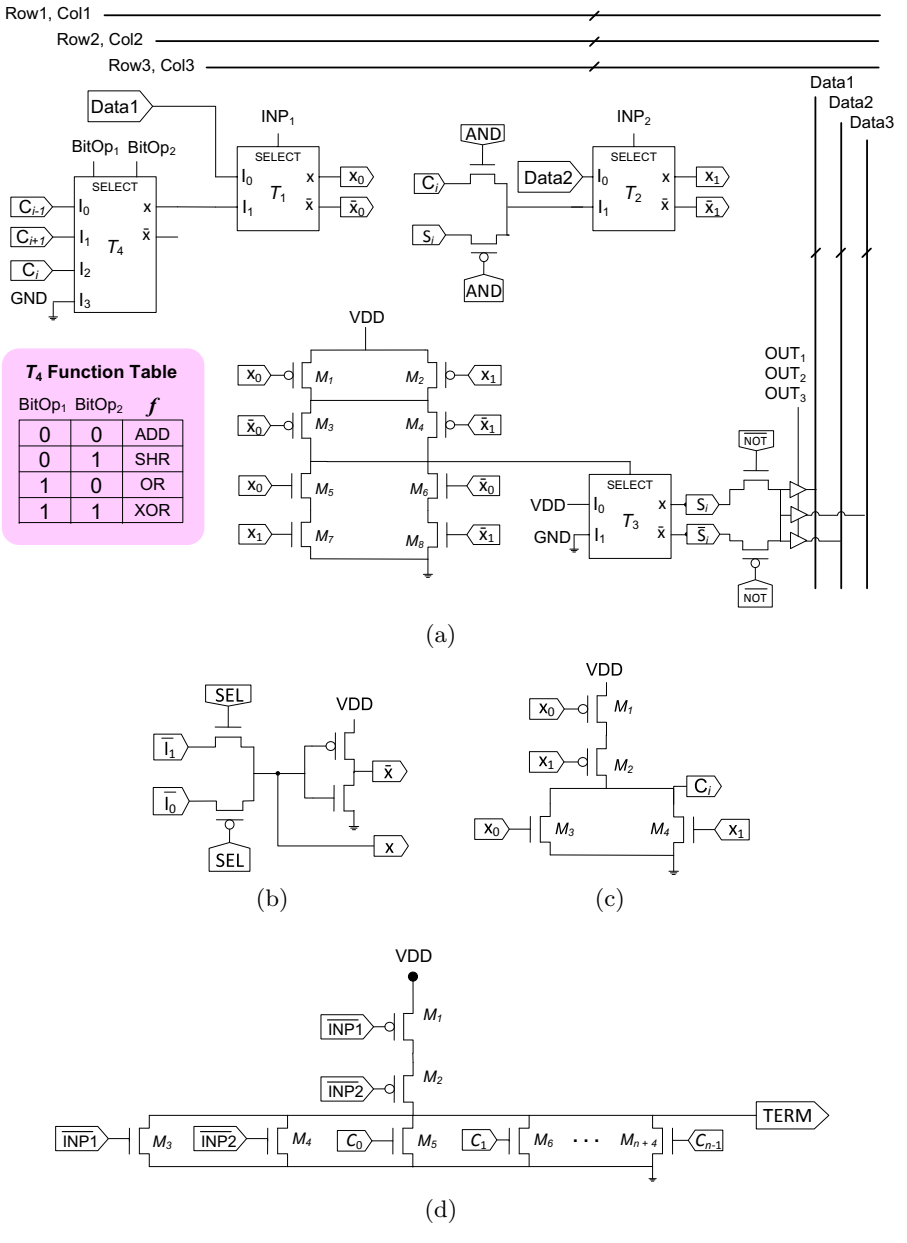


Fig. 4. CMOS Implementation of an ARAM cell. (*a*) Output unit, (*b*) 2×1 MUX and (*c*) Carry module for a single bit ARAM cell. (*d*) The completion detection unit for a word.

establish a minimum performance that could be reaped from ARAM by a single process.

3. Pipelined and Non-pipelined ARAM: The pipelined execution ensures instruction availability at every cycle thereby improving throughput. However, it needs branch predictions module in the controller that is going to increase complexity of the design. For, low-cost embedded solutions branch predictions and pipelining might not be required.

4. Floating point operation cycles: Floating point operations are assumed to take equal cycles as of synchronous processors as it is logically separate in the present design. Hence, 5 cycles are assumed for non-pipelined execution and 2.5 cycles are assumed for pipelined execution.

The Standard Performance Evaluation Corporation (SPEC) CPU2000 benchmarking programs are analysed for MIPS performance and the % running profiles for different instructions are listed in [2]. An average case integer instruction uses 1.54 cycles and floating point instruction uses 2.48 cycles for the CPU2000 programs in the mentioned MIPS machine. Using these data total cycles required per 1000 MIPS instructions and its equivalent for ARAM are computed following the above mentioned comparison strategies .

The instruction profile comparison and relative performance of MIPS and ARAM are depicted in Figures 5 and 6. As clarified before, the CPU-memory bottleneck causes current processors to spend most of the time in memory related load-store type instructions. This is evident in Figure 5. The gain achieved by avoiding memory-access cycles is quite significant. For example *perlbmk* integer benchmark program achieves 10% performance improvement in ARAM even without any pipelining. On the other hand, with pipelining significant improvement is observed in ARAM. The minimum % improvement is 50% with pipelining while it reaches upto 66% for *perlbmk*.

Similar trend is observed in floating point benchmark programs with the exception that the performance improvement is lesser than integer benchmark programs. This is expected as the FP unit cannot totally avoid load-store instructions. In most of the benchmark applications load-FP dominates as the top overhead nearly equal or greater than 'Add' instruction. 'Add-FP', 'store-FP' and 'Mul-FP' are the next three expensive floating point operations. Overall floating point applications will not enjoy equal performance gain as of integer applications. Nevertheless, with pipelining enabled they can run at least equally or 25% faster on ARAM.

The above benchmark results are obtained with the conservative assumptions by not considering any parallelism or clock-cycle reduction due to asynchronous execution. Even then, the results show that ARAM can run applications much faster when pipelining is employed. However, the major benefit that ARAM will bring is parallel execution of separate data-independent processes which is totally omitted in this analysis. Therefore, the actual performance improvement by ARAM will be a multiple of the results presented here.

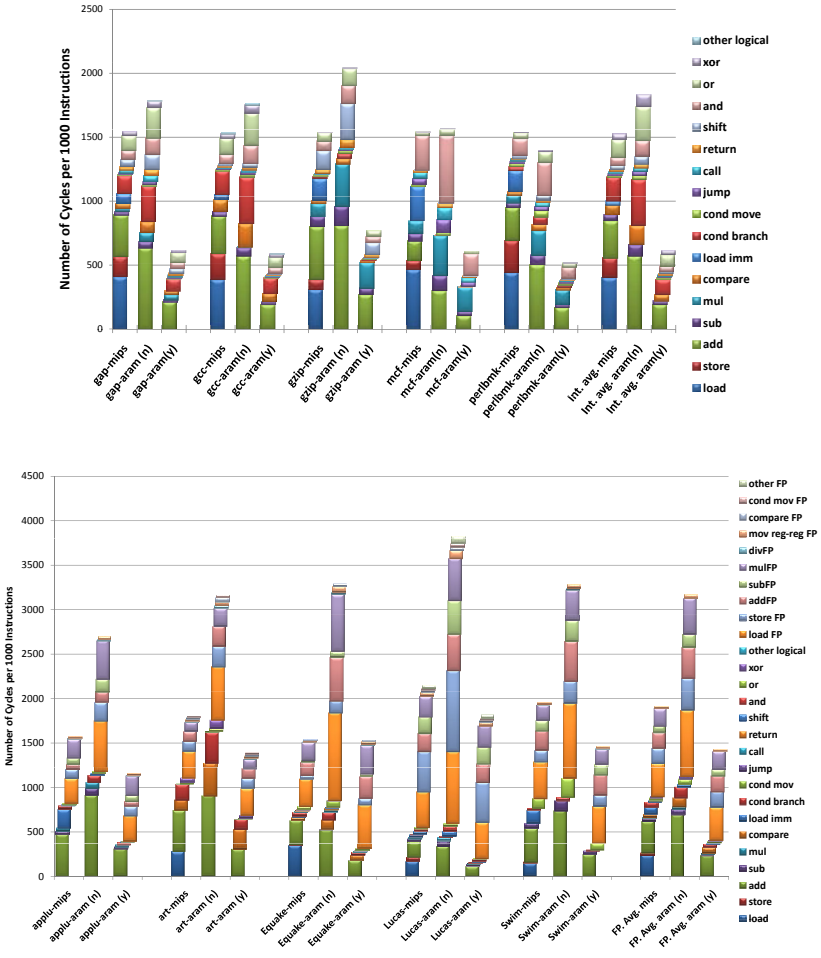
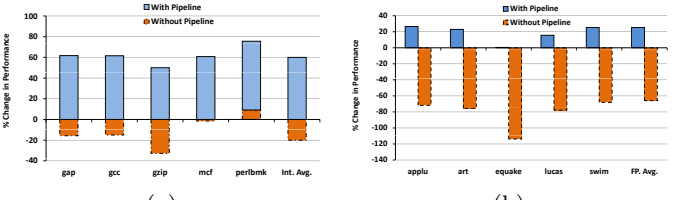


Fig. 5. (*a*) Distribution of integer instructions and (*b*) distribution of floating point instructions for MIPS, ARAM-pipelined (*y*) and ARAM-non-pipelined (*n*) machines



(a) (b)

Fig. 6. Performance comparison of MIPS, ARAM-pipelined and ARAM-non-pipelined architectures. Figure (*a*) shows integer and (*b*) shows floating point benchmark application performances.

6 Conclusion

An asynchronous combined Arithmetic Logic Unit and Random Access Memory (ARAM) element is presented based on parallel self-timed adder. It is also presented that all integer arithmetic and logical operations can be performed by this single adder element. It is not merely a bit-level processing element rather a word level complete ALU. Thus it supports parallel execution of individual processes located in separate memory blocks. The CMOS implementation verified the practicality of the proposed architecture. Though it will consume more transistors per bit of memory, it is not impractical as the required number of transistors is still less than 50 supporting a regular layout. The proposed architecture effectively overcomes the fundamental memory-CPU channel bottleneck of von Neumann architecture and is a step closer towards biological brains.

References

1. Backus, J.: Can programming be liberated from the von neumann style? a functional style and its algebra of programs. Communications of the ACM 21(8), 613–641 (1978)
2. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco (1990)
3. Furber, S., Brown, A.: Biologically-inspired massively-parallel architectures - computing beyond a million processors. In: International Conference on Application of Concurrency to System Design, pp. 3–12 (2009)
4. Stone, H.S.: A logic-in-memory computer. IEEE Transactions on Computers, 73–78 (January 1970)
5. Elliott, D.G.: Computational RAM: A memory-SIMD Hybrid. PhD thesis, University of Toronto, Pasadena, California (December 1997)
6. Gokhale, M., Holmes, B., Iobst, K.: Processing in memory: The terasys massively parallel PIM array. IEEE Computer, 22–31 (April 1995)
7. Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K.: A case for intelligent ram. IEEE Micro, 34–44 (April 1997)
8. Geer, D.: Is it time for clockless chips? IEEE Computer, 18–19 (March 2005)
9. Paver, N.C.: The design and implementation of an asynchronous microprocessor. PhD thesis, University of Manchester (1994)
10. Rahman, M.Z.: A recursive approach to the design of parallel self-timed adders. University of Malaya, Tech. Rep. (2010), http://web.fsktm.um.edu.my/~zia/TR/PASTA-TR-2010.PDF
11. Sutherland, I.E.: Micropipelines. Communications of ACM 32(6), 720–738 (1989)

Parallel Implementation of External Sort and Join Operations on a Multi-core Network-Optimized System on a Chip

Elahe Khorasani, Brent D. Paulovicks, Vadim Sheinin, and Hangu Yeo

IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, U. S. A.
{elkh,ovicks,vadims,hangu}@us.ibm.com

Abstract. In a commercial Relational Database Management System (RDBMS), sort and join are the most demanding operations, and it is quite beneficial to improve the performance of external sort and external join algorithms that handle large input data sizes. This paper proposes parallel implementations of multi-threaded external sort and external hash join algorithms to accelerate IBM DB2, one of leading RDBMSs, using an IBM Power Edge of Network (IBM PowerEN™) Peripheral Component Interconnect Express (PCIe) card as an accelerator. The preliminary results show that the proposed parallel implementation of the algorithms on PowerEN™ PCIe card can speed up the DB2 sort and join performance about two times.

Keywords: Sort, Join, Relational Database.

1 Introduction

The RDBMS now form the majority of database management systems. The RDBMS stores data in the form of a table, and the related data are stored across multiple tables. Each row of the table is called a record, and each record contains fields which are columns of the table. A key can be an individual column or a group of columns within a record and used as a logical way to identify and access a record in a table. The sort and join operations are classic standard relational database operations, and are the most demanding operations of RDBMS for building indexes, binary searches, grouping, aggregation, etc, and hence obviously it is beneficial to improve performance of those operators. There have been efforts to implement parallel versions of sort algorithm using multi-core single instruction multiple data (SIMD) processors or hardware accelerators such as graphics processing units (GPUs) [1]-[5]. There also have been considerable studies on parallel join algorithm, and parallelism was easily exploited for the high performance hash join operation [6]-[9].

Although a single pass in-memory sort or join operations are the fastest, but not always the fastest one with limited resources (limited main memory size). To handle huge input data size, multi-pass operations are more appropriate. External sort algorithm is applicable when the data to be sorted is too large to fit in the primary memory, and external merge sort is one of the most popular algorithms [10]. Hash join

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 318–325, 2011.

algorithm is commonly used in database systems to implement equijoins efficiently. In an equijoin operation, equality is an operator to compare key values from two input tables to generate pairs of matching records. The hybrid hash join algorithm [11] is an external join operation developed to handle the case where the input tables are too big to be stored in the available main memory.

The PowerEN™ chip [12] is a network-optimized processor system on a chip developed by IBM. The chip consists of four A2 chiplets where each chiplet has four 64-bit embedded PowerPC A2 cores and 2 MB shared eDRAM L2 cache (8 MB L2 cache per chip), two memory controllers, four 10 GE Ethernet interfaces and five acceleration engines. Each core supports four hardware threads (sixty four total threads per chip) which share L1 and L2 caches, and provides memory management unit (MMU) and separate instruction and data cache controllers and arrays. The five acceleration engines include Host Ethernet acceleration for network protocol processing, Compression/Decompression engine, Cryptographic coprocessor, Extensible Markup Language (XML) engine and Regular Expression/Pattern-matching engine. The compression engine works as a coprocessor to the A2 core, and uses Lempel-Ziv (LZ77) compression followed by the Huffman Coding. The compression engine runs at 10 Gbps with the compression ratio between two and five when tested with various TPC-H tables. The PowerEN™ PCIe card is an integrated board design based on PowerEN™ chip intended to be used for software developer platform, and IBM System X3650 M2 is used as a host for the card.

In this paper, we implemented efficient parallel external sort algorithm and parallel external hash join algorithm designed to accelerate the performance of DB2 in the area of external sort and external hash join operations by taking advantage of the features of PowerEN™ chip. The DB2 code is altered so that the input records are intercepted to feed the sort and hash join processes that runs on an external multi-core accelerator, where they are implemented in a highly parallel manner and the results are returned to DB2. Since the speed of a single threaded external sort or join process is mainly limited by the speed of the processor and speed of data transfer rate to the external disk, the implemented algorithms are mainly designed to take advantages of multi-threaded execution, compression and decompression units and 10 GE Ethernet interfaces.

2 Algorithm Implementation

The external sort and hybrid hash join algorithms are implemented in two passes, and each pass is implemented using multiple threads to take advantages of multi-core designs and parallelism at the chip level. The sort and join requests are made by the sort or join client (DB2), and the records are streamed from DB2 to the accelerator (PowerEN™ PCIe card) through 10 GE interface. The acceleration unit stores records on the file server after the first pass of the operation through 10 GE interface, and reads back the records when the second pass of the operation starts. Both the DB2 and file server are integrated on IBM System X3650 M2 with two Intel Xeon 5500 (Nehalem) processors (x3650). The diagram of the implementation is depicted in Figure 1.

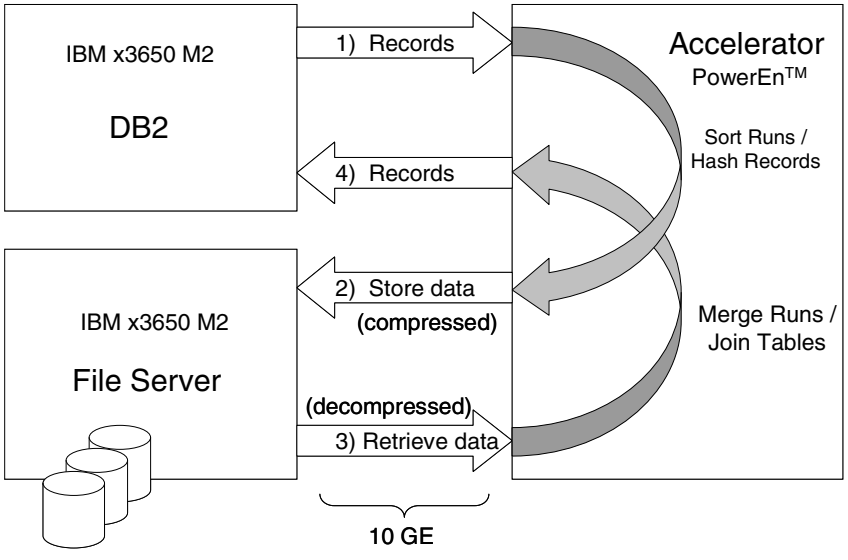


Fig. 1. Overview of external sort and join implementation

2.1 External Sort Algorithm Implementation

The external sort algorithm is implemented based on the quick sort and merge algorithm. The quick sort algorithm is considered to be one of the simplest and fastest algorithms, and the algorithm uses a divide-and-conquer method to sort data. The records to be sorted is segmented into two segments by choosing a comparison element (pivot value) so that all records whose key values are less than the pivot value is assigned to the first segment, and all records whose key values are greater than the pivot value is assigned to the second segment. The two segments are further segmented recursively using the same procedure until each segment consists of a single record only. The quick sort algorithm does not perform well when there is huge discrepancy in size between the two newly created segments, and it is important that each step produces two segments of equal size.

During the first pass, the input records are sent to the accelerator and collected sequentially into run buffers allocated in the main memory (typical run buffer size is 512 MB), and the run buffers are sorted concurrently. When each run buffer has enough records and is ready to be sorted, each run is split into sixteen short, fixed-length segments sub-runs (32 MB). The sub-runs contain a "sort" record generated from each input record. Unlike the input record which might be variable length, a "sort" record is fixed length (multiple of eight bytes in length), and the key values (possibly multiple keys) are extracted from the input record and translated from floating point, character, decimal, etc values into unsigned integer values and packed into the sort record. Each sub-run is then quick sorted in parallel by sixteen threads, and the sixteen sub-runs are merged in parallel by sixteen threads. Using the pointers in the "sort" records, the Asynchronous Data Mover (ADM), which is one of accelerators on PowerEN™, is used to reorder the original input records into output buffers.

The output buffers are passed to the compression engine, and the compressed buffers are written to the file server. During the second pass, all the runs are retrieved from the file server and merged. First, the first buffer (64 KB) from each run is read into memory, and each buffer is passed to the decompression engine. After the decompression is done, the last record in each buffer is examined and the smallest key value is determined. All the buffers are merged in parallel into a single (sorted) intermediate buffer. The intermediate buffers are now merged up to the smallest key value found, and the merged records are returned to the client. The aforementioned process of read and merge buffers from each run is repeated until the runs are exhausted.

2.2 External Hash Join Algorithm Implementation

The external join algorithm implementation is based on the hybrid hash join algorithm, and partitions input tables to accommodate small memory size. During the first pass, the input records are sent to the accelerator from a join client. Two different join clients (a record generator and DB2) are implemented to request join operations and to stream input records to the join server. For example, the record generator generates TPC-H tables and streams input records through the 10 GE interface to the join server implemented on the accelerator, and receives matched records returned by the join server when the join operation is completed. On the join server, the records streamed from the join client are collected in hash buffers allocated in the main memory, multiple threads are implemented to hash the records collected in the hash buffers into a number of partitions concurrently. The number of partitions is pre-estimated using the estimated number of build records and estimated size of the records provided by the join client. Unlike external sort, the join algorithm uses two input tables (build table and probe table) to join, and each input table is partitioned into the same number of disjoint partitions using the same hash function one after another. The hash function converts variable length character strings or four byte key values into four byte hash codes, and records having hash codes within the same range are partitioned to the same partition. It was tested and confirmed that the records are uniformly distributed across the number of partitions. Unbalanced record distribution across the partitions may diminish the performance of parallelized join operation. The hashed records are sent to one of output buffers allocated based on the partition number, and the hashed records are passed to the compression engine when each output buffer is filled with hashed records. The compressed records are streamed to the file server and temporarily stored until the second pass starts. During the second pass, pairs of partitioned files (one build partition and one probe partition) are read from the file server to the accelerator, decompressed and loaded into the main memory. After a pair of partition is loaded into the main memory, each partition is sub-partitioned into the same number of disjoint sub-partitions using a hash function and the smaller sub-partitions are joined in parallel using multiple threads independently. While hashing a build partition into sub-partitions, multiple threads creates hash tables (one hash table for each sub-partition) in the main memory concurrently, and the hash tables are probed concurrently using multiple threads. Since the hashed records are uniformly distributed

across number of sub-partitions, the workloads to join each pair of sub-partitions are evenly distributed among all engaged threads. The joining process of a pair of partitions is repeated until all the partitions are exhausted on the file server.

To reduce the amount of records stored on the file server, during the first pass, Bloom filter is built while hashing the build records. The Bloom filter is a bit filter representation of the set of keys which can be queried to check if a key is present. The corresponding bit for hashed joining key value of record is set while hashing each build record. Then while hashing probe records, the joining key is hashed using the same set of hash functions, and the filter is checked to see whether the corresponding bit was set. Use of Bloom filter reduces the amount of records stored on the file server and reduces "hash probe" time dramatically.

2.3 DB2 Sort and Hash Join Accelerator

Sort and Hash Join are high demanding processes in terms of resources, and accelerating them will improve the overall performance of DB2. In our implementation the DB2 code is altered to export the sort and hash join processes to an external multicore accelerator, where they are implemented in a highly parallel manner and the results are returned to DB2. The DB2's internal sort (SORT) and hash join (HSJN) implementations are bypassed. The SORT and HSJN processes are intercepted and the data flow is directed to and from the accelerator. The changes to the DB2 code are limited to the SORT and HSJN components and do not affect the rest of DB2. The interface between the DB2 and the Accelerator is implemented in a shared library.

In the SORT process, the "insert" phase is intercepted and the record data is directed to the accelerator instead of the DB2 routine. When all the data is sent, the "fetch" routine is intercepted and sorted records are retrieved from the external accelerator instead of the internal buffers that are managed by DB2. Because there is no sorting or merging in DB2, memory allocation, and managing of temporary tables are all bypassed. In the HSJN process, the routines that manage build and probe tables are intercepted and the records for both tables are sent to the accelerator. There is no intermediate retrieval of matched records from the accelerator. After the last row of the probe table is sent, and the hash join process is completed by the accelerator, all of the matched records are retrieved. Similar to the sort process, because DB2 does not implement the hash join process there is no memory management or temporary storing of spilled records. Processing of the input table records and returning the results to the client are not affected.

3 Simulation Results

The simulation results of external sort and hybrid hash join algorithms are compared in Table 1 - Table 4. Table 1 shows the scalability behavior of the multithreaded external sort algorithm implemented on PowerEN™ PCIe card when the number of threads increases. The relative execution times of sort and merge phases are compared using up to 32 threads. Table 2 compares performance (sort rate) of external sort algorithm on DB2 and PowerEN™ PCIe card. The DB2 uses software compression and

the accelerator uses hardware coprocessor to compress/uncompress data. Performance was measured in number of bytes sorted per second, and relative sort performance is listed in the table. The results show that the accelerator sorts the input records two times faster than DB2 sort using a single sort instance. When multiple simultaneous sort requests are made to the accelerator running multiple concurrent sort instances on the accelerator, each sort operation finishes each sort request and the aggregated sort rate measured was about seven times faster than DB2 single sort rate (R_{DB2}) when the number of sort requests form the client was four. Table 3 compares performance of external join algorithm executed on the join accelerator using various numbers of threads. Input table sizes of 26 GB and 30 GB are used for build and probe tables respectively. The "hash build" is an execution time (T_B) to map build records into number of partitions, and the "hash probe" is an execution time (T_P) to map probe records into the same number of partitions as build partitions. A record generator was implemented on IBM System X3650 M2 and was used as a join client that requests external join operation. This implementation is to test the scalability behavior of multithreaded hash join algorithm when the number of threads increases. The test results show that most of the speedup is obtained during the second pass for the actual join process (T_J) which includes both build and probe phases using multiple threads. The first pass (especially T_P) does not scale well as the number of thread increases compared to the second pass. This is not surprising since bloom filter pre-filters out unmatched probe records without storing the records on the file server to optimize the first pass of the external hash join operation. Since there were only small number of matching records between the two input tables (less than one percent of the probe records were having matching records in the build table), most of the records were dropped without being stored on the file server and the amount of records stored on the file server was reduced dramatically. The actual hash probe execution time (T_P) mainly composed of time to receive the input table from the join client, and remains same regardless of the number of threads being used. Table 4 indicates comparison of relative performance of DB2 hash join operation without using an acceleration unit and external join operation being executed on PowerEN™ PCIe card. The join server implemented on the PowerEN™ PCIe card uses hardware compression and decompression coprocessors to reduce data size stored on the file server. The test results show that DB2 hash join operation was accelerated by a factor of two by offloading the hash join operation to the PowerEN™ PCIe card as an accelerator.

Table 1. Test results of external sort algorithm (sort and merge phases) using different number of threads. T_S and T_M are execution times measured for each phase using a single thread.

Number of Threads	Sort Phase	Merge Phase
1	T_S	T_M
2	0.5 x T_S	0.5 x T_M
4	0.25 x T_S	0.26 x T_M
8	0.13 x T_S	0.13 x T_M
16	0.1 x T_S	0.07 x T_M
32	0.07 x T_S	0.05 x T_M

Table 2. Comparison of external sort performance (DB2 vs. accelerator). R_{DB2} is a sort rate measured on DB2 without acceleration unit. The compression was turned on both DB2 and PowerEN™.

Sort Client	Sort Accelerator	# Sort Requests	Sort Rate
DB2	None	1	R_{DB2}
DB2	PowerEN™	1	2.11 x R_{DB2}
DB2	PowerEN™	4	7.22 x R_{DB2}

Table 3. Test results of external join algorithm using different number of threads. T_B, T_P, T_J and T_{Total} are execution times measured for each module using a single thread. T_J includes execution time for build and probe phase.

# Threads	Hash Build	Hash Probe	Join	Total
1	T_B	T_P	T_J	T_{Total}
2	0.92 x T_B	0.93 x T_P	0.7 x T_J	0.82 x T_{Total}
4	0.77 x T_B	0.92 x T_P	0.36 x T_J	0.59 x T_{Total}
8	0.64 x T_B	0.92 x T_P	0.18 x T_J	0.45 x T_{Total}
16	0.5 x T_B	0.88 x T_P	0.1 x T_J	0.35 x T_{Total}
32	0.23 x T_B	0.92 x T_P	0.05 x T_J	0.22 x T_{Total}

Table 4. Simulation results comparison of external join algorithm. $T_{PowerEN}$ is total join execution time measured on PowerEN™.

Join Client	Join Accelerator	Compression	Total Join Time
DB2	PowerEN™	Hardware	$T_{PowerEN}$
DB2	None	Software	1.89 x $T_{PowerEN}$

4 Conclusion

The goal of the work was to accelerate DB2 external sort and hash join operations which cannot be processed using an operation of a single pass due to limitations on resources (main memory size) to hold the huge input tables. The work was focused on efficient parallel implementations of external sort and external hash join algorithms that can execute parallelized operations concurrently as well as alleviate data transfer overhead created by an operation of multiples pass. An IBM PowerEN™ PCIe card was used as an accelerator, and the acceleration unit was integrated with a host machine running DB2 as a client. Our experimental results demonstrate that the performance of individual phase of external sort and external hash join algorithms scales really well in the presence of multiple threads. Although the performance improvement of overall external sort and external hash join operation is little bit shy of that of individual phase due to an overhead, it can be conclude that stand alone external sort and hash join algorithms are good candidates to be offloaded to the accelerator and achieve almost by a factor of two acceleration of the DB2 performance with the help of compression and decompression engine in conjunction with parallel implementation of the algorithms.

References

1. Arefin, A.S., Hasan, M.A.: An Improvement of Bitonic Sorting for Parallel Computing. In: Proceedings of the 9th WSEAS International Conference on Distributed Computing, Athens, Greece (2005)
2. Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y.K., Baransi, A., Kumar, S., Dubey, P.: Efficient Implementation of Sorting on Multi-core SIMD CPU architecture. In: Proceedings of the VLDB Endowment, pp. 1313–1314 (2008)
3. Ramprasad, N., Baruah, P.K.: Radix Sort on the Cell Broadband Engine. In: International Conference on High Performance Computing, HiPC (2007)
4. Zagha, M., Blelloch, G.E.: Radix Sort for Vector Multiprocessors. In: Proceedings Supercomputing, pp. 712–721 (1991)
5. Satish, N., Harris, M., Garland, M.: Designing Efficient Sorting Algorithms for Manycore GPUs. In: The 23rd IEEE Internal Parallel and Distributed Processing Symposium, pp. 1–10 (2009)
6. Azadegan, S., Tripathi, A.: A Parallel Join Algorithm for SIMD Architectures. Journal of Systems and Software, 265–280 (1997)
7. Lu, H., Tan, K.L., Sahn, M.C.: Hash-based Join Algorithms for Multiprocessor Computers with Shared Memory. In: Proceedings of the Sixteenth International Conference on Very Large Database (1990)
8. Garcia, P., Korth, H.F.: Database Hash-Join Algorithms on Multithreaded Computer Architectures. In: Proceedings of the third Conference on Computing Frontiers (2006)
9. Martin, T.P., Larson, P.A., Deshpande, V.: Parallel Hash-Based Join Algorithms for a Shared-Everything Environment. IEEE Transactions on Knowledge and Data Engineering 6 (1994)
10. Knuth, D.E.: The Art of Computer Programming. Sorting and Searching, vol. 3. Addison-Wesley, Reading (1973)
11. Dewitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation Techniques for Main Memory Database Systems. Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data 14 (1984)
12. LaPotin, D.P., Daijavad, S., Johnson, C.L., Hunter, S.W., Ishizaki, K., Franke, H., Achilles, H.D., Dumarot, D.P., Greco, N.A., Davari, B.: Workload and Network-Optimized Computing Systems. IBM Journal of Research and Development 54(1) (2010)

STM with Transparent API Considered Harmful

Fernando Miguel Carvalho[1,2,⋆] and Joao Cachopo[2]

[1] DEETC, ISEL/Polytechnic Institute of Lisbon, Portugal
mcarvalho@cc.isel.ipl.pt
[2] INESC-ID/Technical University of Lisbon, Portugal
joao.cachopo@ist.utl.pt

Abstract. One of the key selling points of Software Transactional Memory (STM) systems is that they simplify the development of concurrent programs, because programmers do not have to be concerned with which objects are accessed concurrently. Instead, they just have to say which operations are to be executed atomically. Yet, one of the consequences of making an STM completely transparent to the programmer is that it may incur in large overheads.

In this paper, we describe a port to Java of the WormBench benchmark, and use it to explore the effects on performance of relaxing the transparency of an STM. To that end, we implemented, in a well known STM framework (Deuce), a couple of annotations that allow programmers to specify that certain objects or fields of objects should not be transactified. Our results show that we get an improvement of up to 22-fold in the performance of the benchmark when we tell the STM framework which data is not transactional, and that the performance of the improved version is as good as or better than a fine-grained lock-based approach.

Keywords: Transactional Memory, Benchmark, Performance.

1 Introduction

The lack of realistic benchmarks is one of the factors that has been hampering the development, testing, and acceptance of Software Transactional Memory (STM) systems. Many of the developments made on STMs are evaluated on micro-benchmarks [8], [10], [14] and [16], which are fairly often criticized in some publications, such as [7]) that question the usefulness of STMs, given their lack of demonstrable applicability to real-world problems.

On the other hand, applying STMs to larger, more realistic benchmarks, such as the STMBench7 [12] and the Lee-TM [2] benchmarks, typically shows very large overheads when compared to the single-threaded sequential version of the benchmark. These overheads are often attributed to the over-instrumentation made on these benchmarks by overzealous STM engines that protect each and every memory access with a barrier that calls back to the STM runtime engine [9].

⋆ This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the RuLAM project (PTDC/EIA-EIA/108240/2008).

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 326–337, 2011.

In this paper we tackle both of these problems. First, we describe a port that we made of the WormBench benchmark [18], from C# to Java. This port extends the original benchmark in several ways, making it more useful as a testbed for evaluating STMs. Moreover, our port, which we called JWormBench, was designed to be easily extensible and to allow easy integration with different STMs.

The second contribution of this paper addresses the second problem by exploring an extension of Deuce [15] that allows programmers to add annotations to their programs specifying that certain objects or fields of objects are not shared among threads, and, therefore, do not need to be instrumented in the same way as are shared data under the control of the STM, which incur in large overheads.

This contrasts with the generally accepted idea that STMs should be completely transparent, meaning that programmers just need to specify which operations are atomic, without knowing which data is accessed within those operations. That is the approach taken by Deuce, which provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior.

Whereas ideally that would be the expected API for an STM, in practice that proves lead to unacceptable overheads. So, in this paper we claim that STMs with transparent APIs are harmful, and that, instead, programmers should have some degree of control on what gets transactified. To show the benefits of this, we used the JWormBench to evaluate the performance of Deuce with and without programmer annotations and show a speedup of up to 22-fold, making the optimized version on par with a very fine-grained lock-based scheme.

In the following section we identify the set of characteristics that we would like to have in a good benchmark for an STM system, and discuss to what extent some of the existing benchmarks satisfy those requirements. Then, in Section 3, we describe in more detail the WormBench benchmark, and the main differences introduced in our port of that benchmark to Java, which we called JWormBench. This new benchmark was used to evaluate the performance of Deuce, and in Section 4 we describe in which tasks the JWormBench operations instrumented by Deuce spend most of their time and how the new proposed API for Deuce can reduce some of that time. Section 5 describes the tested configurations of JWormBench and presents a performance evaluation. Section 6 provides a discussion on alternative approaches to the new API proposed for Deuce. Finally, in Section 7, we conclude with a discussion of our approach and results.

2 Benchmarks for STMs

Benchmarks are essential to test and to compare transactional memory (TM) systems. But, as realistic benchmarks are scarce, TM implementers often resort to micro-benchmarks, which are typically too simple to test their systems properly, leading to fair skepticism about the relevance of their results and the applicability of their approaches. Thus, the TM research community is in dire need of good, realistic benchmarks. But, what makes a benchmark good?

Harmanci et al. [13] distinguish two kinds of experimental evaluations for transactional memory (TM) systems: performance evaluation and semantics evaluation (debugging/testing/verification). Naturally, a good benchmark should allow both types of evaluation. On the other hand, Ansari et al. [2] argue that a TM benchmark should have as desirable features: large amounts of potential parallelism; several types of transactions; complex contention; and transactions with a wide range of durations (transaction length) and amount of data accesses (transaction size).

To the above requirements, we add that a good benchmark should: (1) be flexible enough to allow the integration of new synchronization mechanisms without requiring changes to its source-code; (2) provide a synchronization mechanism based on fine-grained locking approach, which may present a good performance and serves as a reference to achieve by other synchronizations mechanisms and (3) provide a verification test to validate the semantics of the STM and if the overall consistency of the benchmark was not broken by any erroneous synchronization.

To address the lack of realistic benchmarks for TM systems, Guerraoui et al. [12] introduced the STMBench7 benchmark: a benchmark for performance evaluation implemented in Java that models a realistic large scale CAD/CAM application. Typically, STMBench7 operations focus on object manipulation and there are no tasks that apply mathematical functions with different degrees of complexity as exist in other benchmarks, such as STAMP [5] and WormBench [18]. Furthermore, it does not provide a correctness test that is able to verify if an execution has produced correct results.

STAMP is a benchmark suite that attempts to represent real-world workloads in eight different applications. Unlike STMBench7, the STAMP applications are configurable and allow us to vary the level of contention, size of transactions, the percentage of writes, among other parameters. But this benchmark has several drawbacks also. First, not all applications make semantic evaluations of the tested STMs. Second, it is not easy to integrate STAMP applications with some STM algorithms, such as DSTM[14] and JVSTM[4], because it requires us to modify its source code and change the type of all memory locations accessed by a transaction.

In the Java world, LeeTM [2] is an alternative to STMBench7 and it has many of the desirable properties of an STM benchmark: it is based on a real-world application and provides a wide range of transaction durations and sizes. The LeeTM also provides a verifier that validates the correct consistency of the final data structure. One of the limitations of the LeeTM benchmark, however, is that it does not allow extending it to new kinds of operations and research variations of its contention scenarios. Furthermore, there is no possibility of varying the read/write ratio of the benchmark, because all of the transactions write something, unlike most applications.

WormBench [18] is a configurable transactional C# application that was designed to evaluate the performance and correctness of TM systems. The idea behind WormBench is inspired in the Snake game, but in this case the snakes are

worms moving and performing *worm operations* in a *shared world*. The WormBench shares all the benefits of LeeTM, such as providing a verifier algorithm (correctness test) and a wide range of transaction durations and sizes. But, unlike LeeTM, it applies a broad diversity of mathematical operations and it is able to extend to new ones. Moreover WormBench is totally configurable with regard to: the percentage of update operations; the kind of operations and proportion between them; the maximum execution time or number of iterations; the contention level and synchronization strategy. In addition, the WormBench benchmark has a low complexity domain model (compared to STMBench7), making it easy to understand, and has a simple API. Still, as shown in [18], despite this simplicity, the WormBench benchmark can still reproduce STAMP workloads with the same characteristics.

3 JWormBench: A Port of WormBench to Java

The main data structures in the WormBench include *worms* formed by a *body* and a *head*, moving in a shared *world* — matrix of *nodes*. Each *node* has an integer *value* and the total sum of the values of all world's nodes is the *world's state*. For read-only workloads, the world's state should remain unchanged by the execution of the benchmark.

The *worms* perform *worm operations* on the *nodes* under the *worms*'s *head*. In the WormBench application a *worm* object is associated with one thread and is initialized with a stream of *worm operations* and *movements* that will be performed by that thread during the execution of a workload. The tasks that should be performed atomically are annotated with macros that delimit the beginning and the end of the atomic block. Then, each synchronization mechanism should translate those macros to invocations to the corresponding synchronization API.

The WormBench implementation provides 13 types of *worm operations*, which may be grouped in the following categories:

- *read-only* — *Sum*, *Average*, *Median*, *Minimum*, and *Maximum*. Each of these operations reads all the nodes under the worm's head, corresponding to *head's size*^{2} nodes;
- *n-reads-1-write* — *Replace<read-only>With<read-only>*. Each of these operations combines two of the read-only operations described in the previous item: They use the value returned by the first operation to update the node returned by the second. Each operation makes *2*head's size*^{2} reads and one write, updating the *world's state*.
- *n-reads-n-writes* — *Sort* and *Transpose*. When these operations are properly synchronized with other concurrent worm operations, they preserve the *world's state*. Each operation makes the same number of read and write operations corresponding to *head's size*^{2} nodes.

The *worm operation Sort* and those ones based on the *Median* have complexity $O(n^2)$. These algorithms could be implemented with lower complexity, but it was our intention to provide operations computationally intensive.

The JWormBench adds two new features important for the research of new workloads and evaluation of STM scalability: (1) the ability to specify the proportion between different kinds of operations, and (2) the ability to set the number of worms independently of the number of threads. Furthermore, the JWormBench provides a simple API, easy to integrate with any STM implementation in Java. So, anyone may add a new synchronization mechanism (based on STM or other), implementing the appropriate abstract types and providing those implementations to JWormBench via a configuration *module*. In the same way you can also extend JWormBench with new kinds of *worm operations* without modifying the core JWormBench library. A more detailed description about extending JWormBench is provided in JwormBench's wiki, which is available with the source-code repository [6].

To increase the extensibility of JWormBench, we have designed it according to the *inversion of control* (IoC) design pattern . Then, to run a workload on JWormBench we must create an instance of the `WormBench` class and invoke the `RunBenchmark` method. But the `WormBench` class has *dependencies* to several abstract types, whose implementations in turn depend on other abstract types and so on. So we used Guice as the *dependency injection* framework[1] to automatically resolve and inject *dependencies* based on a configuration Guice *module* (a Java class that contributes configuration information — *bindings*).

This new architecture promote the implementation and easier integration of new synchronization mechanisms without the need to interfere and modify the source code of JWormBench. Also note that this modular design does not add any additional overhead to the synchronization mechanism during the execution of the workload and while it is collecting measurements. The additional levels of indirection imposed by IoC and Guice will just delay the setup and will not affect the performance analysis.

Finally the JWormBench also provides a correctness test (i.e. sanity check for the STM system) based on the results accumulated on each threads private buffer. This buffer stores the difference between the new and the old value of every node updated by a worm operation. At the end, if we subtract the accumulated differences on each thread's private buffer to the total value of all nodes, the result must be equal to the initial sum of nodes' values.

4 Annotations to Avoid Over-Instrumentation

One of the goals of Deuce is to provide a transparent API, but the approach followed in the implementation of this feature incurs in over-instrumentation. This happens because all memory locations accessed in the context of a transaction are instrumented, independently of whether those locations are private, or not, to the transaction.

To explore the overheads caused by over-instrumentation and what may be gained by having finer grained control over what to instrument, we propose to extend the Deuce API with two Java annotations — `@NoSyncField` and

[1] Available at: http://code.google.com/p/google-guice/

`@NoSyncArray` — that can be applied on fields and type declarations, respectively, to avoid over-instrumentation in certain scenarios. In Subsection 4.1 we describe those scenarios and the reasons to over-instrumentation. Then, in Subsection 4.2, we introduce the effects and behavior of the new annotations in the Deuce framework, avoiding over-instrumentation on the previously described scenarios.

4.1 Over-Instrumented Tasks

Using profiling analysis we have verified that JWormBench is instrumented by the Deuce framework in five different accesses to memory locations. Table 1 shows the time spent by each operation accessing each memory location: *Coord* — x, y *coordinates* of worm's head; *Worm* — *Worm*'s *coordinate* array; *Node* — Value of the *node*; *World* — *World*'s *node* matrix; *local arr* — *local array* that is auxiliary to the function that defines an operation.

Table 1. Unmodified version of Deuce

#	Operation	Coord	Worm	Node	World	local arr	Flow
0	Sum	680	410	400	730	0	0
1	Average	1.100	420	440	940	0	10
2	Median	820	1.970	690	1.110	44.570	490
3	Minimum	770	650	410	990	0	0
4	Maximum	790	700	320	800	0	0
11	Sort	1.680	1.310	400	2.100	37.270	210
12	Transpose	1.490	1.170	360	1.580	1.060	130

Table 2. Optimized version of Deuce

#	Operation	Coord	Worm	Node	World	local arr	Flow
0	Sum	0	0	380	0	0	0
1	Average	0	0	430	0	0	20
2	Median	0	0	750	0	0	530
3	Minimum	0	0	480	0	0	0
4	Maximum	0	0	520	0	0	0
11	Sort	0	0	630	220	0	200
12	Transpose	0	0	320	0	0	100

Tables 1 and 2: distribution of the operation execution time (in milliseconds) accessing each of the five kinds of memory locations. There is an extra column — *Flow* — that collects the time spent in the execution of the control flow of the transactions.

The results depicted in Table 1 show that *median*, *sort* and *transpose* operations are more time-consuming because they access all kinds of memory locations. On the other hand, the *median* and *sort* operations are so much slower than the others because their algorithms have complexity $O(n^2)$.

These results were collected for JWormBench with Deuce configured to use the TL2 STM[8] and with just one worker thread. The characteristics of *world* and *worms* are the same as in the workloads for performance evaluation described in section 5. As we will see, of all the memory locations depicted in Table 1, only the third location — *Node* — should be instrumented, as it is the only one that is shared and updated by concurrent threads.

In Table 2 we present the results collected for an optimized version of Deuce according to the proposal made in this paper. Comparing the results between Tables 1 and 2 we can confirm that the third memory location is the only one where time is spent executing each *worm operation* (except for *sort*, which also wastes time accessing the *node* matrix of the *world*). Finally, table 3 shows the percentage of time due to over-instrumentation, which represents between 80% and 97% of total execution time of a *worm operation*.

Table 3. Difference between the execution time of each worm operation in unmodified Deuce framework and the optimized one

Operation	Sum	Avg	Med	Min	Max	Sort	Trans
Unmod.	2.220	2.900	49.160	2.820	2.610	42.760	5.660
Optimized	380	450	1.280	480	520	1.050	420
Over-instr	83%	84%	97%	83%	80%	98%	93%

4.2 New Java Annotations for the Deuce API

Our proposal includes two Java annotations — `@NoSyncField` and `@NoSyncArray` — that should be parametrized with a value of the *enum* type — `NoSyncBehavior`. This parameter specifies the behavior of the annotated memory location: `Immutable`, `TransactionLocal` or `ThreadLocal`.

Annotating a field with `@NoSyncField(Immutable)` has a similar effect on Deuce framework to the Java `final` keyword on field declarations. Both make the Deuce framework to avoid instrumentation when accessing those fields. However the `final` keyword has another effect at the Java level, prohibiting changes to the declared field after its initialization. This behavior could be to much restrictive for memory locations that are unmodified inside transactions, but are still target of changes outside them.

Another use of `@NoSyncField` annotation is to annotate fields that are not shared among different threads. So these fields could be private to a single transaction or to a single thread. In the first case the fields should be annotated with `@NoSyncField(TransactionLocal)` meaning that the annotated fields do not need to be synchronized and therefore do not have to be instrumented. In the second case the fields should be annotated with `@NoSyncField(ThreadLocal)` meaning that the most part of the instrumentation incurred by an STM when accessing those fields could be attenuated, excepting the *undo log* that it is still required to revert the updated data if the transaction aborts.

The effects of the `@NoSyncField(TransactionLocal)` on Deuce framework are the same as the `@NoSyncField(Immutable)`, avoiding instrumentation of the annotated memory locations when they are accessed inside a transaction. They just differ under the debug mode and in the way how the Deuce framework verifies if the specified behavior is fulfilled by the transactified program.

The `@NoSyncField(ThreadLocal)` can be applied on the declaration of memory locations such as *Coord* - x, y coordinates of worm's head. These locations belong to *coordinate* objects identifying *nodes* of the *world* that are under the *worm*'s head. The x and y fields of these *coordinate* objects are updated every time a *worm* performs a movement. However, a *worm* has affinity to only one thread and this means that a *worm* is just updated by one and always the same thread. Therefore, these memory locations do not need to be completely instrumented and can be read and updated in-place avoiding the overhead of maintaining a *read set* and *write set*. However the transactions still need to keep an *undo log* where they register the original values of the updated memory locations. Then, when a transaction aborts it uses the *undo log* to revert the data that was updated.

Annotating arrays has an increased challenge in comparison with the same task applied on fields. One difficulty in achieving this goal it is to find a way to attach the intention — *array elements are immutable, or transaction local, or thread local* — to an array declaration. The Java bytecodes for arrays manipulation receives an array reference as parameter and not the declared array variable. Then there is no easy way for the compiler to know the array variable at the moment it processes a bytecode for array access.

An alternative approach is to attach our intention to the array object instead of the array variable. However this strategy postpones the decision from compile-time to run-time and will not totally eliminate the unnecessary over-instrumentation on arrays. As we cannot annotate the array type, then we adopt a different solution: to annotate the type of the array's element with the `NoSyncArray` annotation. This approach has limitations too and may seem a little bit strange because instead of annotating intentions in the array declaration we propose to do that in the declaration of the type of the array's element.

The `@NoSyncArray(Immutable)` can be applied to the type of the array's element, whose elements are immutable during the array's life cycle. Note that the `final` keyword for arrays declaration has a distinct effect from that one that is specified by the `@NoSyncArray(Immutable)`. In the case of the `final` keyword, it just avoids the declared variable (array's reference) from being modified after its initialization and it does not mean anything about the characteristics of its elements. While the `@NoSyncArray(Immutable)` declares that the elements of the array will not change inside a transaction. This behavior is verified for memory locations as those ones referred in section 4.1 for *Worm*'s *coordinate* array and *World*'s *node* matrix.

Finally we can also use `@NoSyncArray` to annotate the declaration of the type of the array's elements as: `@NoSyncArray(TransactionLocal)` or `@NoSyncArray(ThreadLocal)`, to respectively denote that the array's elements are private to a single transaction or to a single thread.

Some *worm operations*, such as *median*, *sort* and *transpose* need an auxiliary local array to perform their algorithm. However, these arrays are private to the transaction, because their references and their elements are not shared across the boundaries of the function that defines the *worm operation*. Given that, there is no concurrent access on these arrays and there is no need to synchronize those accesses.

5 Performance Evaluation

To evaluate the effect of our approach on the performance and the scalability of the JWormBench benchmark, we compared our optimized version of the Deuce framework both against the released version 1.3.0 and against its current implementation[2]. Simultaneously, we also made a comparison with a lock-free implementation of JVSTM that has recently presented a much better performance than Deuce on the STMBench7 benchmark [10]. In our analysis we also

[2] Available at: http://code.google.com/p/deuce/

include an implementation of a fine-grained lock-based *step* for JWormBench, which acquires locks for all the nodes under a worm's head in a pre-specified order to avoid deadlocks.

The testing workload has a *world*'s size of 512 nodes and 48 *worms* with a body's length of one node and the head's size varying between 2 and 16 nodes, corresponding to a number of nodes under the worm's head between 4 and 256. The length of the body affects collisions between worms and we are not interested on that behavior for this analysis. The size of the head directly affects the length of the transaction read-set and write-set, according to the description made in section 3.

In terms of *worm operations*, we have used three different configurations. In all configurations we maintain the proportion between read-write and read-only *worm operations* of 20-80%.The configurations tested are as follows:

1. Combination of *read-only* and *n-reads-1-write worm operations*, excluding operations based on the *median* operation. So, these *worm operations* are not very intensive, having complexity *O(n)*, and the write-set for all read-write transactions has a length of one.
2. Equals to the previous configuration, but including operations based on *median*. So, this configuration is heavier than the previous one, with 4 *worm operations* having complexity $O(n^2)$.
3. Combination of *read-only* and *n-reads-n-write worm operations*. Two of these operations have complexity $O(n^2)$ and the size of the write-set, for read-write transactions, is equals to the number of nodes under *worm's head — head's size*^{2}.

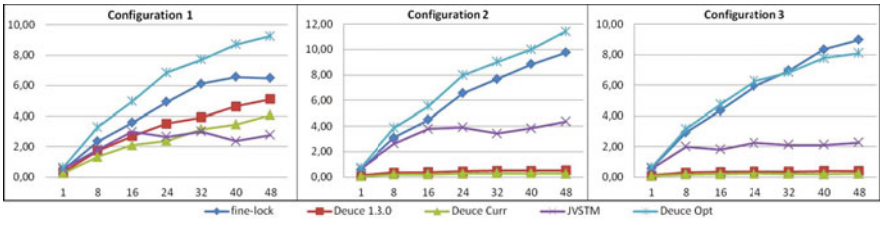


Fig. 1. Configuration 1 - *read-only* and *n-reads-1-write worm operations*, excluding *median*, Configuration 2 - *read-only* and *n-reads-1-write worm operations* and Configuration 3 - *read-only* and *n-reads-n-write worm operations*

The tests were performed on a machine with 4 AMD Opteron(tm) 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the `1.6.0_22-b04`, running on Ubuntu with Linux kernel version 2.6.32.

The charts of figure 1 depict the speedup of each synchronization mechanism over sequential, non-instrumented code. These results show that Deuce 1.3.0 and Deuce's current implementation just scale for the first configuration, without the presence of heavier *worm operations*, such as *median* and *sort*. On the other

hand, our optimization proposal for the Deuce framework scales in the three scenarios and presents a much better performance than the previous versions: 2 times better in the first configuration and 22 times in both configurations 2 and 3.

In comparison with the fine-grained lock approach, the optimized version of the Deuce framework has a better performance for configurations 1 and 2, but is almost equals in performance for configuration 3. The worst performance verified in configuration 3 may happen because the size of the write-set in this case is bigger than in the previous ones, increasing the number of conflicts and causing much more aborted transactions.

The JVSTM presents a similar behavior to that one shown in [10] for the STMBench7 benchmark. It scales until 16 threads and then it stabilizes. Nevertheless for configurations 2 and 3, the JVSTM has a much better performance than either Deuce 1.3.0 or Deuce's current version.

6 Related Work

More recently, Afek et al. [1] propose to solve some cases of over-instrumentation in Deuce by avoiding access barriers for thread-local memory and by using optimizations that eliminate redundant reads and writes. These optimizations pretend to avoid over-instrumentation in the same way as the annotation `@NoSyncField(TransactionLocal)`. But the remaining 5 cases that we propose to resolve with the 5 other forms of annotating memory locations, which are presented in section 4.2, are not covered by the proposal of Afek. In fact the detection of these 5 cases is not easy to implement via a static analysis as proposed by Afeck.

Our approach is also distinct from the option of an heterogeneous API as mentioned in [1], which proposes a specialized `STMReadThreadLocal()` operation beyond the generic `STMRead()`. Unlike the latter, the former assumes that the read location is thread-local and therefore avoids access barriers. But this approach is not applicable to the Deuce framework, which follows an homogeneous API.

The work of Beckman et al. [3] proposes the use of access permissions, via Java annotations, which can be applied to references, affecting the behavior of the object pointed by that reference. Besides having a different semantics, access permissions do not have direct correspondence with the Deuce framework, which is based on conflicts detection at the fields level. Unlike this work, the use of the Java annotations `@NoSyncField` and `@NoSyncArray`, proposed in our work, integrates well with how Deuce makes the instrumentation of accesses to fields.

Yoo et al [17] propose a `tm_waiver` annotation to mark a block or function that would not be instrumented by the compiler for memory access. With a unique annotation they can cover some of the cases that we propose to avoid over-instrumentation, except for thread local data. But their solution is more difficult to manage because it forces the programmer to identify all the blocks or functions that manipulate the memory locations, whose access do not need

to be instrumented. In contrast in our proposal we just have to annotate the memory location on its declaration.

Our proposal also distinguishes between different behaviors of non-instrumented memory locations, allowing us to provide a debug mode in which Deuce framework validates the consistency of each location according to the behavior specified by its annotation.

7 Conclusions

STMs are often criticized for introducing unacceptable overheads when compared with either the sequential version or a lock-based version of any realistic benchmark. Our experience in testing STMs with several realistic benchmarks, however, is that the problem stems from having instrumentation on memory locations that are not actually shared among threads. The solution that we explore in this paper is to give the programmer some mechanisms that allow him to tell to the STM system that some memory locations are not to be manipulated transactionally. This approach reduces the transparency of the STM, which is one of its advantages over lock-based approaches, but it proves to be able to get huge benefits performance-wise. In fact, we have been able to get a 22-fold improvement on the throughput of a realistic benchmark. This result is consistent with other observations made on benchmarks that use a similar approach (e.g., the results of the JVSTM on the STMBench7 reported in [10]).

Actually, not only do we get a huge speedup when we use a less transparent API, our results show that the STM performs better or as good as a fine-grained lock-based approach, which is particularly easy to use in JWormBench, but may not be in other applications. Still, we argue that the lock-based approach is harder to develop and get right than the use of annotations to identify non-transactional memory: To implement a lock-based approach, we need not only to identify the shared resources, as in our approach, but we have to be careful about getting the locks for all of the accessed resources, and doing it in the correct order. So, even if we are losing some of the transparency of the STM approach, we believe that this may be a reasonable tradeoff between easiness of development and performance.

Finally, another contribution of this work is the JWormBench benchmark, which, despite being a port of WormBench, has some key differences from it: (1) Unlike the WormBench, which follows an STM integration approach based on macros, the JWormBench has a new solution based on *inversion of control*, *abstract factory* and *factory method* design patterns [11]; (2) the core engine of the JWormBench benchmark is deployed in a separate and independent library, whose features can be extended with other libraries; (3) unlike JWormBench, the WormBench distribution does not implement the correctness test (i.e. sanity check for the STM system) based on the results accumulated on each thread's private buffer; (4) in WormBench it is not easy to maintain the same contention scenario when varying the number of threads, while in JWormBench the number of threads is totally decoupled from the environment specification and we can

maintain the same conditions along different numbers of worker threads; (5) the operations generator tool in JWormBench allows us to specify the proportion between each kind of operation, which is an essential feature to produce workloads with different ratios of update operations.

References

1. Afek, Y., Korland, G., Zilberstein, A.: Lowering STM overhead with static analysis. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 31–45. Springer, Heidelberg (2011)
2. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Lee-TM: A non-trivial benchmark suite for transactional memory. In: Bourgeois, A.G., Zheng, S.Q. (eds.) ICA3PP 2008. LNCS, vol. 5022, pp. 196–207. Springer, Heidelberg (2008)
3. Beckman, N.E., Kim, Y.P., Stork, S., Aldrich, J.: Reducing stm overhead with access permissions. In: IWACO, pp. 2:1–2:10. ACM, New York (2009)
4. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63, 172–185 (2006)
5. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008 (September 2008)
6. Carvalho, F.M. (2011), https://github.com/inesc-id-esw/JWormBench
7. Cascaval, C., Blundell, C., Michael, M.M., Cain, H.W., Chiras, S., Wu, P., Chatterjee, S.: Software transactional memory: why is it only a research toy? Commun. ACM 51(11), 40–46 (2008)
8. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
9. Dragojevic, A., Ni, Y., Adl-Tabatabai, A.-R.: Optimizing transactions for captured memory. In: SPAA 2009, pp. 214–222. ACM, New York (2009)
10. Fernandes, S.M., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: PPoPP 2011, pp. 179–188. ACM, New York (2011)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley, Reading (1995)
12. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41, 315–324 (2007)
13. Harmanci, D., Felber, P., Gramoli, V., Fetzer, C.: Tmunit: Testing transactional memories. In: TRANSACT (2009)
14. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SIGPLAN Not. 41, 253–262 (2006)
15. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with java stm. In: MultiProg 2010 (2010)
16. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation, pp. 284–298 (2006)
17. Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., Lee, H.-H.S.: Kicking the tires of software transactional memory: why the going gets tough. In: SPAA 2008, pp. 265–274. ACM, New York (2008)
18. Zyulkyarov, F., Cristal, A., Cvijic, S., Ayguade, E., Valero, M., Unsal, O., Harris, T.: Wormbench: a configurable workload for evaluating transactional memory systems. In: MEDEA 2008, pp. 61–68. ACM, New York (2008)

A Global Snapshot Collection Algorithm with Concurrent Initiators with Non-FIFO Channel

Diganta Goswami and Soumyadip Majumder

Indian Institute of Technology Guwahati, Guwahati, Assam, India
{dgoswami,s.majumder}@iitg.ernet.in

Abstract. Taking a global snapshot in the absence of a global clock is a challenging issue in distributed system. The problem becomes more challenging when the communication channel is a non-FIFO one, due to the lack of FIFO properties in transmitting messages. Multiple initiators further complicate the situation. In this paper, we present a global snapshot collection algorithm with multiple initiators in the case of non-FIFO communication channel. We have shown that the algorithm can take a unique global consistent snapshot with non-FIFO channel, and terminates in $O(mn^2)$ message complexity where m is the number of concurrent initiators, and n is the number of processes in the system.

1 Introduction

A distributed computing system comprises of spatially separated processes that do not share a common memory and solely communicate via messages with unpredictable delay. Global Snapshot finds its application in many aspect of distributed system like distributed debugging, termination detection, deadlock detection etc.

A process consists of an initial state which is followed by a series of events. An event e of a process is a 5-tuple $< P_i, ls_i^k, ls_i^{k+1}, \mu, c >$ which indicates that process P_i changes its state from ls_i^k to ls_i^{k+1} when it sends (receives) a message μ along incoming (outgoing) channel c. μ and c can be null which signifies that a transition in local state may happen due to some internal computation despite of any send or receive event. At any instant, the state of process P_i, denoted by ls_i^k, is the result of the linearly ordered sequence of events executed by P_i. For an event e_i and a process state ls_i^k, $e_i \in ls_i^k$ iff e_i belongs to the sequence of events that brings process P_i to the state ls_i^k. Whereas, the state of a channel c_{ij} between process P_i and P_j denoted by cs_{ij}, is actually the set of those messages which are in transit from P_i towards P_j.

A global state of a distributed system is a collection of local states of processes and the state of the channels. Notionally, a global state is represented as

$$\text{GS} = \{ \bigcup_i ls_i, \bigcup_{i,j} cs_{ij} \}$$

We can define consistent global state as a cut that divides a time diagram of a distributed system in two parts in such a way that all receive events that fall in

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 338–348, 2011.

the left side of the partition; corresponding send events of those messages will definitely reside on the same side, i.e on the left side of the partition. It may be possible that a receive event of a message may reside on the right side and its send event on the left side. This situation doesn't hamper the consistency of the snapshot. But whenever a receive event resides at the left side and its send event on right; the cut as well as snapshot becomes inconsistent. In a consistent global state, every message that is recorded as received is also recorded as sent and such a state captures the notion of causality that a message can't be received if it was not sent.

In non-FIFO channel, messages are not necessarily received in orderly fashion i.e a message μ_{ij}^{k} that was sent before a message μ_{ij}^{k+1}, may be received by process P_j after it received μ_{ij}^{k+1}. The challenge is to identify those messages that are sent by process P_i to process P_j after the snapshot has been taken, but received by P_j before it takes the snapshot. If this receive is recorded in the local snapshot ls_j of P_j, then global snapshot will be inconsistent. Most of the snapshot algorithm assume that at any point of time only one initiator exist in the system. But any process can initiate a snapshot collection process at any point of time. There exist significant probability that more than one initiator initiate the process.

In this paper, we have addressed the problem of taking snapshot in a non-FIFO channel by multiple concurrent initiators at the same time. Our objective is to provide a consistent and unique global snapshot to all the initiator. We tried to propose an algorithm that doesn't restrict any process to initiate a snapshot collection process though another process has already initiated a similar task.

2 Related Works

In a seminal paper Chandy and Lamport first provided a marker message based snapshot collection algorithm that is non-inhibitory in nature and requires FIFO channels[1]. In [2], Helary incorporated the concept of wave algorithm in Chandy-Lamport algorithm. In [3], Venkatesan introduced the concept of incremental snapshot and proposed an efficient algorithm by modifying Chandy-Lamport algorithm.the system. But these algorithms work for FIFO channels only. In [4], Lai and Yang came up with a snapshot algorithm for non-FIFO channel. In[5], Mattern gave an algorithm that was based on vector clock. Different topology specific algorithms [6] [7] [8], with non-FIFO channel have also been proposed.

Above mentioned works on snapshot collection algorithms assume that at any point of time only one snapshot collection process is active. Koo and Toueg[9], Spezialetti and Kearns[10] and Prakesh and Singhal[11] have proposed method to handle multiple initiations in FIFO channel. Koo and Toueg algorithm is blocking in nature. The main idea behind Spezialetti-Kearns algoritm is that different nodes may take local snapshots in response to requests from different initiators, but the local snapshots are combined to produce a global consistent snapshot.

Prakash and Singhal[11] permits the full propagation of snapshot requests generated by all initiators. In [12], Hansdah et al's proposed a checkpointing algorithm in non-FIFO channel with multiple initiators. But the algorithm permits only one instance of checkpointing. In this present scenario, we try to provide a solution to a problem to collect a unique consistent global snapshot with multiple initiations in non-FIFO channel.

3 Assumption and System Model

Our system consists of n processes, $P_1, P_2..., P_n$ which are connected to each other directly or indirectly forming a connected graph. Every two processes are connected by at least one path. Links are non-FIFO in nature with finite but unbounded delay. We assume that despite of link failure this connectivity remains uninterrupted and processes can withstand fail-stop errors. Every application message is piggybacked with the current snapshot id and the state of the snapshot. In our model any process has equal likely probability to initiate a snapshot collection and multiple processes can initiate the procedure simultaneously. If a process has already taken a temporary snapshot, it can't initiate snapshot until the snapshot is made permanent.

4 Concurrent Snapshot Collection Strategy

In the present scheme multiple processes can concurrently initiate snapshot. It is a two phase protocol where a process first takes a temporary snapshot and then makes the snapshot permanent. Each process P_i maintains a vector $sendVector_i$ which records the number of messages sent to other processes. Whenever a message has been sent by P_i to P_j, $sendVector_i[j]$ is incremented by 1. A process P_i keeps track of all messages it has received in the counter $receiveCount_i$ and last snapshot id in $currSnapid_i$. Whenever, a message is sent from P_i to P_j, current snapshot id and the state of the snapshot is appended in the message. When the appended snapshot id is equal to the receiver's current snapshot id and receiver's snapshot state is permanent, then it receives the message. On receiving an application message, receiver P_j, increments its $receiveCount_j$ by 1. If the receiver has taken a temporary snapshot, it only includes those messages in its local snapshot with snapshot id less than current snapshot id of the receiver. Other messages will not be included in the snapshot. If a message is received with larger snapshot id then it is kept in the queue until the receiver's snapshot id becomes equal to the message's snapshot id.

When a process P_i wants to initiate a snapshot, it first increases the value of $currSnapid_i$ by 1. It then takes a temporary snapshot. After that it sends snapshot request to neighbours. The snapshot request contains the current snapshot id and the state of the snapshot. The process marks itself as an initiator and also initialize the concurrent initiator list ($concurrent_i$) with its own id.

On receiving a snapshot request a process P_i first checks whether it has already taken a temporary snapshot with the same snapshot id or not. If it has not taken a snapshot then it takes a snapshot and unicasts a response to that initiator. The response contains $sendVector_i$ and $receiveCount_i$. It remembers the process id of the first snapshot request and also puts an entry in its concurrent initiator list($concurrent_i$). If the responder process receives any message with snapshot id less than the current snapshot id it increments counter $transitCount_i$. If the process has already taken a temporary snapshot then it will not take a snapshot but send a 'Dummy-Response' to all subsequent initiators along with the information of its first initiator.

On receiving 'Response' message from P_j, an initiator P_i copies $sendvector_j$ in the j^{th} row of a matrix $sendMatrix_i$ and $receiveCount_j$ at the j^{th} entry of a vector $receiveVector_i$. On receiving 'Dummy-Response', an initiator P_i first updates its $concurrent_i$. When an initiator receives $n-1$ responses(Response and Dummy-Response), it first elects a leader from the initiator list (if there is more than one initiator). The elected leader will be unique throughout the system as all the initiator will have the same concurrent initiator list ($concurrent_i$) for a particular snapshot collection process with same snapshot id. Initiators (including leader) then send their $senMatrix$ and $receiveCount$ to the leader process in 'Compile' messages. After receiving m such 'Compile' messages (m is number of initiator), it merges the data in its own $sentMatrix$ and $receiveVector$, in order to calculate the number of in transit messages for each and every individual process and puts those value in a vector, $transitVector$.

The leader then broadcasts $transitVector$ to all processes. After a process receives all in transit messages, the process includes all those messages in the snapshot and makes the snapshot permanent. Snapshot collection process of the non-initiator process ends, when it receives all in transit messages. It multicasts the final snapshot to all initiators. All the initiators, on receiving $n-1$ final snapshot messages assembles those snapshots and generate the global snapshot. All initiators make their snapshot state permanent and snapshot collection procedure started by those initiator terminates.

5 Algorithm

5.1 Data Structure

- $sendVector_i[1..n]$: It holds the individual count of messages that has been sent by P_i to processes.
- $sendMatrix_i[1..n][1..n]$: It holds the count of messages of that been sent by all processes.
- $receiveCount_i$: It holds the count of messages that been received by process P_i.
- $receiveVector_i[1..n]$ It holds the count of messages that been received by all processes.

- $transitVector_i[1..n]$: An entry, transitVector[j] indicates the number of in-transit messages destined to process P_j.
- $intransitCount_i$: It is used to keep the count of the received message after a process take a temporary snapshot.
- $currSnapid_i$: It holds the current snapshot id. Whenever a process takes a snapshot it increment it by one.
- $currState_i$: It indicates the state of the current snapshot. It may be permanent (P) or temporary (T).
- $concurrent_i$: It holds the ids of the concurrent initiator.
- $firstInitiator_i$: Holds the id of the process whose snapshot request was first reached to the process.
- $receiveTransit_i$:It is a flag that indicates whether a process has received the transitVector.
- $initiator_i$: It is a boolean type of data which indicates whether a Process is an initiator or not.
- $leader$: This variable indicates the id of the leader.

5.2 Messages

- Marker : Marker message is sent when a process want to initiate a snapshot collection procedure.
- Response : Response message is sent by a process to an initiator when it receives a marker message from that initiator and its snapshot state is permanent. Response message that has been sent by process P_i, contains $sendVector_i$ and $receiveCount_i$.
- Dummy-Response : Dummy-Response message is sent by a process to an initiator when it receives a marker message from that initiator and its snapshot state is temporary. Dummy-Response message that has been sent by process P_i, contains $firstInitiator_i$.
- Compile : An initiator sends compile message to the leader. Compile message contains send and receive information that has been collected by the initiator from 'Response' messages.
- In-Transit-Count : This message contains the count of all in transit messages for a particular instance of snapshot collection procedure.
- Final-Response : Final-Response is the actual snapshot message that has been sent by every process to all initiators at the end of the snapshot collection event.

5.3 Algorithms

Algorithm 1. Process P_i : On Sending an Application Message m_{ij} to Process P_j

$sendVector_i[j] \leftarrow sendVector_i[j] + 1$;
$send(m_{ij}, P_i, P_j, currSnapid_i, currState_i)$;

Algorithm 2. Process P_i : On Receiving a Application Message $< m_{ij}, P_j, P_i, currSnapid_j, currState_j >$ from Process P_j

if $currSnapid_j = currSnapid_i - 1$ **then**
  receive the message;
  $receiveCount_i \leftarrow receiveCount_i + 1$;
  if $currState_i = T$ **then**
    put this receive event in the temporary Snapshot;
    $intransitCount_i \leftarrow intransitCount_i + 1$;
if $currSnapid_j = currSnapid_i$ **then**
  if $currState_i = P$ **then**
    receive the message;
  if $currState_i = T$ **then**
    receive the message and don't put this receive event in the temporary snapshot;
  $receiveCount_i \leftarrow receiveCount_i + 1$;
if $currSnapid_j = currSnapid_i + 1$ **then**
  queue the message until $currSnapid_i$ *equals to* $currSnapid_j$;
  don't put this receive event in temporary snapshot;
  on relinquishing the message from the queue do
  $receiveCount_i \leftarrow receiveCount_i + 1$;

Algorithm 3. Process P_i : Snapshot Initiation

if $currState_i = P$ **then**
  $initiator_i \leftarrow true$;
  $firstInitiator_i \leftarrow P_i$;
  $concurrent_i \leftarrow concurrent_i \cup \{P_i\}$;
  take a temporary snapshot;
  $currstate_i \leftarrow T$;
  $currSnapid_i \leftarrow currSnapid_i + 1$;
  $responseCount_i \leftarrow 0$;
  $sendMatrix_i[l][m] \leftarrow -1$, *for l,m = 1,2..,n*;
  $receiveVector_i[l] \leftarrow -1$, *for l = 1,2..,n*;
  $sendMatrix_i[i][k] \leftarrow sendVector_i[k]$, *for k = 1,2..,n*;
  $receiveVector_i[i] \leftarrow receiveCount_i$;
  $send(Marker, P_i, currSnapid_i, currState_i)$ *along all edges e*;

Algorithm 4. Process P_i : On Receiving a Marker $< Marker, P_j, currSnapid_i, currState_i >$ along channel e

if $currSnapid_j = currSnapid_i + 1 \wedge currState_i = P \wedge P_j \notin concurrent$ **then**
  take a temporary snapshot;
  $currSnapid_i \leftarrow currSnapid_j$;
  $currState_i \leftarrow T$;
  $send(Response, P_i, P_j, sendVector_i, receiveCount_i)$ *to initiator* P_j;
  $firstInitiator_i \leftarrow P_j$;
  $concurrent_i \leftarrow concurrent_i \cup \{P_j\}$;
  forward the request along all the edge except e;

if $currSnapid_j = currSnapid_i \wedge currState_i = T \wedge P_j \notin concurrent$ **then**
  $send(Dummy - Response, P_i, P_j, firstInitiator_i)$ to initiator P_j;
  $concurrent_i \leftarrow concurrent_i \cup \{P_j\}$;
  forward the request along all the channel except e;

Algorithm 5. Process P_i : On Receiving a Response from Process P_j

$responseCount_i \leftarrow responseCount_i + 1$;
$sendMatrix_i[j][k] \leftarrow sendVector_j[k]$, *for k = 1,2..,n*;
$receiveVector_i[j] \leftarrow receiveCount_j$;
if $responseCount_i = n - 1$ **then**
  $leader \leftarrow \min_{id}\{concurrent_i\}$;
  $send(Compile, P_i, leader, sendMatrix_i,$
  $receiveVector_i)$ *to leader*;

Algorithm 6. Process P_i : On Receiving a Dummy-Response from Process P_j

$responseCount_i \leftarrow responseCount_i + 1$;
$concurrent_i = concurrent_i \cup \{firstInitiator_j\}$;
if $responseCount_i = n - 1$ **then**
  $leader = \min_{id}\{concurrent_i\}$;
  $send(Compile, P_i, P_{leader}, sendMatrix_i,$
  $receiveVector_i)$ *to leader*;

Algorithm 7. Process P_i : On receiving a Compile Message from Process P_j

$sendMatrix_i[k][l] \leftarrow sendMatrix_j[k][l]$, *if* $sendMatrix_j[k][l] \neq -1$, *for k,l = 1,2..,n*;
$receiveVector_i[k] \leftarrow receiveVector_j[k]$,*if* $receiveVector_j[k] \neq -1$, *for* $k = 1,2..,n$;
Mark $P_j \in concurrent_i$ *as Compiled*;
if *all* $P_k \in concurrent_i$ *is marked* **then**
  $transitVector_i[k] \leftarrow \sum_{l=1}^{n} sendMatrix_i[l][k] - receiveVector_i[k]$, *for* $k = 1,2..,n$;
  $send(In\text{-}Transit\text{-}Count, P_i, P_k, transitVector_i)$ *along all the channel e and to itself*;

Algorithm 8. Process P_i : On Receiving In-Transit-Count Message from Leader P_j along channel e

if $receiveTransit_i = false \wedge currState_i = T$ **then**
  $receiveTransit \leftarrow true$;
  forward the message to all outgoing channel except e;
  wait untill $transitVector_j[i] = intransitCount_i$;
  make the temporary Snapshot Permanent;
  $send(Final - Snapshot, P_i, P_k, currSnapid_i)$ *to all Process* P_k, $P_k \in concurrent_i$;
  if *initiator* $= false$ **then**
    $currstate_i \leftarrow P$;
    $intransitCount_i \leftarrow 0$;
    $receiveTransit_i \leftarrow false$;
    $concurrent_i \leftarrow \phi$;

Algorithm 9. Initiator Process P_i : On Receiving n Final-Snapshot Message

assemble all the snapshot to produce the global snapshot;
$intransitCount_i \leftarrow 0$;
$currState_i \leftarrow P$;
$initiator_i \leftarrow false$;
$receiveTransit \leftarrow false$;
$concurrent_i \leftarrow \phi$

6 Proof of Correctness

Theorem 1. *Concurrent lists at all initiators that exchange information are identical.*

Proof. Each request generated by initiator P_i carries the the initiator id, current snapshot id, and its state. When this request reaches a process P_j, it responses back to the initiator either using 'Response' or 'Dummy-Response message'. If P_j has not taken any temporary snapshot, it responses back with 'Response' message. If it

has already taken a temporary snapshot due to a snapshot request from process P_l, then it responds with a Dummy-response message which also includes the information of the first initiator id, P_l. So the information about the concurrent initiator P_l will reach initiator P_i. Responder P_j puts an entry of the concurrent initiator P_l in its $concurrent_j$ set and then propagate the request to other processes.

When an initiator P_l receives a snapshot request from another initiator P_i, then also it responses back with a 'Dummy-Response' with its own process id as it has already taken a temporary snapshot by the time being. Process P_l will put an entry of process P_i in its concurrent list. So eventually both P_i and P_l know that their initiations are concurrent.

The information about all the concurrent initiators eventually reach to all initiators either by their own Dummy-Response or by the Dummy-Responses of the non-initiator process. So, every initiator after receiving at most $n-1$ response where n is the number of processes in the system, will get a clear count of all initiators and that list will be identical throughout the system.

Theorem 2. *The Elected Leader is unique.*

Proof. The leader amidst the concurrent initiator is chosen from the concurrent list. The process having minimum process id is elected as leader. As proved earlier in theorem 1 that the concurrent lists at all initiators that exchange information are identical, then every process P_i will elected the same process $P_{min} : P_{min} \in concurrent_i$ and $P_{min} = min\{concurrent_i\}$.

Theorem 3. *The Collected Snapshot is consistent.*

Proof. Due to the non-FIFO nature of the channel while taking snapshot two cases may occurs.

Case 1 : It may happen that a message that has been sent by P_i before taking a snapshot, arrives to process P_j after the snapshot request reaches to P_j.
Case 2 : It may also happen that a message that is sent by process P_i after the snapshot but it reaches process P_j before the snapshot request reaches to P_j.

In case 2, the snapshot taken by process P_j will be inconsistent if the receive event of the message is included in the snapshot. The local snapshot of process P_i doesn't contain the sent event of the message but receive event will exist in the snapshot of process P_j. So, for consistent snapshot, these post-snapshot messages that actually come before the snapshot request must not be included in snapshot.

In our proposed algorithm, current snapshot id denoted by $currSnapid_i$ and current snapshot state denoted by $currState_i$ are piggybacked with each application message sent by a process P_i. Whenever, P_i takes a snapshot it increments the snapshot id by 1 and sets the snapshot state as temporary. Messages send before the snapshot, have numerically lesser value and the state of those messages are permanent. Whereas, post snapshot messages have the present snapshot value with snapshot state as temporary until the snapshot is made permanent. Whenever, P_i sends an application message to P_j with $currSnapid_i$ and $currState_i$, process P_j checks it with its own $currSnapid_j$ and $currState_j$. If the $currSnapid_i$ is greater than $currSnapid_j$ by one then it indicates that

the message is sent after taking the snapshot and the snapshot request has not yet reached P_j. In this case this message is queued until both the snapshot id becomes equal. This receive event is not included in the local snapshot of receiver P_j. If both *currSnapid* is same and the snapshot state of P_j is permanent then it receives the message as it already took part in the snapshot taking event and made and its snapshot permanent. If both *currSnapid* is equal but $currState_j$ and $currState_j$temporary, then that receive event is not included in the snapshot. If $currSnapid_i$ is less than $currSnapid_j$ then process P_j receives the message and also put this receive event in the temporary snapshot as this message was sent before P_i took a snapshot and the sent event is in the local snapshot of process P_i.

So, we can see that out of order messages are properly discarded and so no orphan messages are created in the system. A local snapshot is made final when a process receives all messages that were sent before the snapshot is taken. So the state of the channel is empty as there is no in transit message in the system. Hence we can say that the collected snapshot is consistent.

Theorem 4. *Snapshots collected by all initiators are unique.*

Proof. On receiving all in transit messages a process P_i will sent the final response to all initiator processes $P_k \in concurrent_i$. A unique final response is sent to every initiator. So the collected snapshot at every initiator will be trivially identical.

7 Complexity Analysis

Our system comprises of n processes. The connectivity among processes can be assumed as a graph with n nodes and e edges. We assume that m ($m \leq n$) number of processes can simultaneously initiate snapshot collection procedure. When an initiator process sends a snapshot request (Marker), the request is flooded within the entire system. Every process propagates the request to other neighbouring processes. In our algorithm, due to one initiation $O(n^2)$ such messages will be generated. So the message complexity of taking temporary snapshot will take $O(mn^2)$ messages.

On receiving the marker, each process will reply back either with actual 'Response' or 'Dummy-Response' to all initiators. So each process will emit m messages. Overall $O(mn)$ such response messages will be sent. For dissemination of information, every initiator will send one *Compile* message to the leader, resulting $O(m)$ messages. Broadcast of $transitVector$ will again take $O(n^2)$ messages. After receiving all the in-transit messages each and every process multicasts the final snapshot to each initiator. So the overall complexity of our algorithm is $O(mn^2)$ which is same as Spezialetti-Kearns [10] and Prakash-Singhal [11]. Spezialetti-Kearns and Prakash-Singhal algorithm works under the FIFO assumption. Whereas, we are able to keep the same message complexity under more generalized non-FIFO assumption. Like Spezialetti-Kearns and Prakash-Singhal algorithm our algorithm provides a unique snapshot to all initiators. Presence of a spanning tree will bring down the message complexity by a factor of n as the broadcast of the snapshot request only generates $O(n)$ messages. So the message

complexity will dilute to $O(mn)$. We have designed our algorithm in such a way that it will be useful in repetitive snapshot invocation. The space complexity of the initiator process is $O(n^2)$ where non-initiator has the complexity of $O(n)$ only.

8 Conclusion

We have proposed a global snapshot collection algorithm that can handle concurrent initiation of snapshot collection by multiple nodes. In our algorithm, non-initiator nodes send their receive and send counts to different initiators. A leader is selected from the list of concurrent initiators and initiators send their collected information to the leader. Leader's responsibility is to calculate the number of in-transit messages in the system and to inform all nodes. On receiving all in transit messages a node multicasts its final snapshot to all initiators. Consistency of the snapshot in non-FIFO channel is preserved by the piggybacking of snapshot id and snapshot state. Multicasts of final snapshot response ensures that snapshots collected by every initiator is unique. Unlike Spezialetti-Kearns and Prakash-Singhal, we have restricted the communication between initiators by electing a leader that doesn't require any message exchange and can be found in $O(n)$ time in worst case.

References

1. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
2. Hélary, J.-M.: Observing global states of asynchronous distributed applications. In: Bermond, J.-C., Raynal, M. (eds.) WDAG 1989. LNCS, vol. 392, pp. 124–135. Springer, Heidelberg (1989)
3. Chandrasekaran, S., Venkatesan, S.: A message-optimal algorithm for distributed termination detection. J. Parallel Distrib. Comput. 8(3), 245–252 (1990)
4. Lai, T.-H., Yang, T.H.: On distributed snapshots. Inf. Process. Lett. 25(3), 153–158 (1987)
5. Mattern, F.: Virtual time and global states of distributed system. In: Proceddings of the Workshop on Parallel and Distributed Algorithm, pp. 215–226 (1989)
6. Kshemkalyani, A.D.: A symmetric o(n log n) message distributed snapshot algorithm for large-scale systems. In: CLUSTER, pp. 1–4 (2009)
7. Garg, R., Garg, V.K., Sabharwal, Y.: Scalable algorithms for global snapshots in distributed systems. In: ICS, pp. 269–277 (2006)
8. Kshemkalyani, A.D.: Fast and message-efficient global snapshot algorithms for large-scale distributed systems. IEEE Trans. Parallel Distrib. Syst. 21(9), 1281–1289 (2010)
9. Koo, R., Toueg, S.: Checkpointing and rollback-recovery for distributed systems. IEEE Trans. Software Eng. 13(1), 23–31 (1987)
10. Spezialetti, M., Kearns, P.: Efficient distributed snapshots. In: ICDCS, pp. 382–388 (1986)
11. Prakash, R., Singhal, M.: Maximal global snapshot with multiple initiators, pp. 334–351 (1994)
12. Kumar, K.P.K., Hansdah, R.C.: An efficient and scalable checkpointing and recovery algorithm for distributed systems. In: Chaudhuri, S., Das, S.R., Paul, H.S., Tirthapura, S. (eds.) ICDCN 2006. LNCS, vol. 4308, pp. 94–99. Springer, Heidelberg (2006)

An Approach for Code Compression in Run Time for Embedded Systems – A Preliminary Results

Wanderson Roger Azevedo Dias[1], Edward David Moreno[2], and Raimundo da Silva Barreto[1]

[1] Federal University of Amazonas - Department of Computer Science
Manaus, Amazonas, Brazil
[2] Federal University of Sergipe - Department of Computer Science
Aracaju, Sergipe, Brazil
{wradias,edwdavid,xbarretox}@gmail.com

Abstract. Several factors are considered in the development of embedded systems, among which may be mentioned: physical size, weight, mobility, power consumption, memory, safety, all combined with a low cost and ease of use. There are several techniques to optimize the execution time and power consumption in embedded systems. One such technique is the code compression, the majority of existing proposals focuses on decompression assuming the code is compressed in time compilation. This article proposes the development of a new method of compression and decompression code implemented in VHDL and prototyped on an FPGA, called MIC (Middle Instruction Compression). The proposed method was compared with the traditional of Huffman method also implemented in hardware. The MIC showed better performance compared with Huffman for some programs MiBench, widely used in embedded systems, obtaining 17% to less of the logical elements of FPGA, 6% increase in clock frequency (in MHz) and 42% more in compression codes compared the of using Huffman method, and allows the compression and decompression at runtime.

Keywords: Instruction compression, Embedded systems, Computational performance, Power consumption, FPGA.

1 Introduction

Embedded systems are any systems digital are inserted into other systems in order to add or optimize features [12]. Embedded systems have the task to monitor and/or control the environment in which it is inserted. These environments may be present in electronic devices, appliances, vehicles, machinery, engines and many other applications.

The growing demand for the use of embedded systems has become increasingly common, prompting the implementation of complex systems on a single chip, called System-on-Chip (SoC). In this case, the embedded processor is a key component of embedded computer systems [3]. Today, many embedded processors found in the market are based on architectures of high-performance (e.g., RISC architectures of 32 bits) to ensure a better computational performance for the tasks to be performed.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 349–359, 2011.

Therefore, the design of embedded systems for high-performance processors is not a simple task.

It is known that many embedded systems are powered by batteries. For this reason, it is critical that these systems are able to control and manage power, thus enabling a reduction in energy consumption and control of heating. Therefore, designers and researchers focused on developing techniques that reduce energy consumption while maintaining performance requirements. One such technique is the compression of the code of instructions in memory.

Most of the techniques, methodologies and standards for software development, for the control and management of energy consumption, do not seem feasible for development of embedded systems because they possess several limitations of computing resources and physical. Current strategies designed to control and manage energy consumption have been developed for general-purpose systems, where the cost of additional processors or memory are usually insignificant.

The code size increases significantly as the systems become more heterogeneous and complex. In this sense, there was a high technical level that seeks to compress the code at compile time and their relief, in turn, is made at run time [13].

The compression technique was developed in order to reduce the size code [10]. But over time, groups of researchers found that this technique could be of great benefit to the performance and energy consumption in general-purpose systems and embedded systems [10, 11]. Once the code is compressed in memory is possible on each request processor get a much larger amount of instructions contained in memory. So there is a decrease in the activities of transition pins memory access, leading to a possible increase in system performance and a possible reduction in energy consumption of the circuit [11].

Likewise, when storing compressed instructions in the cache increases the number of instructions stored in the same cache and increases its hit rate, reducing search in main memory, increasing system performance and therefore, reducing energy consumption.

This article presents the development of a new method of compressing and decompressing instructions (at runtime), which was implemented in VHDL (Very High Speed Integrated Circuits Hardware Description Language) and prototyped in a FPGA (Field Programmable Gate Array). It is called MIC (Middle Instruction Compression), and we compared it with the traditional method of Huffman which also implemented in hardware, and it was shown to be more efficient than the Huffman method from a comparison using the benchmark MiBench [5].

The rest of the paper is organized as follows: Section 2 explains the PDCCM architecture developed for the MIC method, Section 3 details the description of the method MIC; Section 4 shows the simulations with benchmark MiBench using methods MIC and Huffman; Section 5 presents the related work, and finally, Section 6 presents conclusions and ideas for future work.

2 Architectures for Code Compression

In the literature we have found two basic types of architectures for code compression, CDM and PDC, which indicate the position of the decompressor for the processor and

memory subsystem, as shown in Figure 1. The CDM architecture (Cache Memory Decompressor) indicates that the decompressor is positioned between the cache and main memory, while the PDC architecture (Processor Cache Decompressor) places the decompressor between the processor and cache.

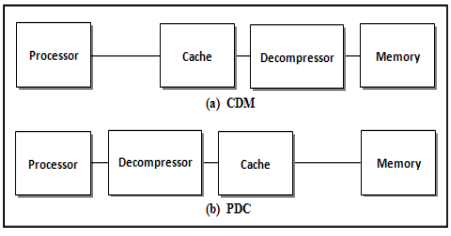


Fig. 1. Architectures for decompression code (a) CDM and (b) PDC [10]

As will be detailed in Section 5, the development of architectures for compression or decompression de instructions done separately, that is, in most of the work undertaken is treated just the hardware decompressor because the compression of the instructions is usually done by modifying the compiler. Thus, the compression is performed at compile time and decompression is done at run time using a specific hardware decompression.

To operate the MIC method proposed in this work, it was necessary to develop a new architecture, and we implemented in FPGA-based hardware, to carry out the compression and decompression of the instruction code at runtime execution. The architecture created was entitled PDCCM (Processor Compressor Decompressor Cache Memory) in which it is shown that hardware compression was inserted between the cache and main memory and the hardware decompression was inserted between the processor and memory cache. The PDCCM architecture was implemented in VHDL and prototyped on a FPGA Cyclone-II model EP2C20F484C7 manufacturer by ALTERA®.

The PDCCM architecture works with instructions size of 32 bits, that is, each line of instruction cache consists of 4 bytes. Thus, the architecture developed is compatible with systems using the ARM processor as the core of the embedded system, because this processor features a set of 32 bits instructions. In PDCCM architecture, using the MIC method of compression/decompression all instructions that are saved in the instruction cache will suffer a 50% compression of its original size.

Figure 2 shows the PDCCM architecture developed to implement the new method (MIC) of compressing and decompressing instructions in hardware, which consists of four basic components, and they are:

- **LAT** (Line Address Table): is a table that has the function for mapping the addresses of the instructions to a new address into the instruction cache;
- **ST** (Sign Table): is a table that contains bits that serve as flags for indicating to the decompressor which pair of bits should be reconstituted, uncompressed;

- **Compressor Unit:** has the function to compress all the codes of the instructions that will be saved in the instruction cache. The compressor is started whenever there is a misses in instruction cache;
- **Decompressor Unit:** has the function to decompress all the instructions that are stored and compressed into the instruction cache and will be relayed to the processor. The decompressor is started whenever there is an access and hit in the instruction cache.

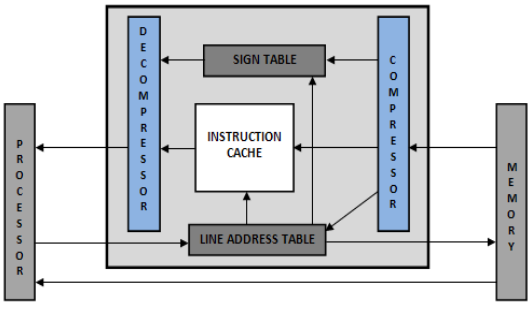


Fig. 2. PDCCM Architecture

3 Code Compression in Run Time

The MIC method (Middle Instruction Compression) is a compression method created for reducing 50% the size of instruction codes that are stored in the instruction cache, and then passing the length of the 32 bits instructions (original size) to 16 bits (compressed size).

Like the Huffman table used by WOLFE and CHANIN [13], into architecture of the CCPR, the instruction table and address translation table used by AZEVEDO [2], into architecture of the IBC, the dictionary of compressed instructions used by BENINI et al. [3] and the dictionary of codes used by LEKATSAS et al. [8, 9] and LEFURGY et al. [7], the MIC method also requires additional tables that are used by components LAT and ST to store the set of flags and the compressed instruction for mapping the new addresses of the instructions in the compressed cache, respectively.

So even with the addition of tables (ST and LAT) in the MIC method, which initially are great to the point does not appear to have had a gain in compression because it requires an additional table equivalent to half the existing cache, and being the compression of 50%, no gain in memory usage, considered as physical resource. Nevertheless, these tables were added to evaluate the structure of the new architecture and the proposed method, which raises the option of having run-time compression, being different from other proposals that are based on the construction of dictionaries at compile time. Right now the group is working with the MIC in order to remove (or reduce) these additional tables and thereby obtain more significant results in the near future.

For compression phase, each instruction should be read into memory and saved to the instruction cache, and it will be split into pairs of bits with each pair consisting of: 00, 01, 10 and 11.

The MIC compressor performs the following logic: pairs with equal values (00 or 11) are replaced by bit 0 (zero) and pairs with different values (10 or 01) are replaced by bit 1 (one). Then, the bits 00 and 11 in compression are replaced by bit 0 and bits 01 and 10 are replaced by bit 1. So, a couple of bits are reduced to a single bit.

An auxiliary table (ST) is used to store the set of flags of double bits compressed. Pairs of bits that start with the value 0 (zero), such as 00 or 10 is saved in the ST bit 0 and bits pairs that start with the value 1 (one), such as 11 or 01, is saved in the ST bit 1. It is noteworthy that the mode of address lines of instruction for this architecture is Big-Endian.

For a better understanding and where possible, the names of components, variables and input and output pins are similar to those used in the code implemented in VHDL.

3.1 Algorithm for Compression/Decompression with MIC Method

In our FPGA-based prototype, the processor requests an instruction to the cache through a special pin, which for this implementation was called `end_inst_proc` (Current PC). After that, the LAT will be searched whether that the address is provided by the processor. If the instruction is found into the cache, then LAT signals with a hit. So, the LAT will provide the new address of the instruction in the instruction cache, the address of the set of flags in ST instruction and the placement of double-byte (first or second), that is, where the instruction and flags in the instruction cache and ST, respectively. All this information is passed to the decompressor unit which decompresses the instruction and returns the uncompressed form to the processor through the variable `returnD_inst_proc`.

The **decompression of instruction codes** is performed as follows:

- The new address of the instruction that was passed by the LAT, should be located in the instruction cache and ST;
- The instruction cache and ST return to the decompressor the 16 bits compressed instruction and 16 bits set of flags;
- If the bit read from the compressed instruction in the instruction cache is 0 (zero), the pair of bits to be reconstructed is 00 or 11. What defines how is the pair of bits is the bit flag, that is, if the flag bit is 0 the pair of bits to be reconstructed is 00 and when it is 1, then the pair of bits to be reconstructed is 11;
- But if the bit read from the compressed instruction in the instruction cache for 1 (one), the pair of bits to be reconstructed is 10 or 01. Then, again the flag bit is the signal that defines how the pair of bits, when it is 0 the pair of bits to be reconstructed is 10 and if the flag bit is 1, the pair of bits to be reconstructed is 01;
- For each instruction to be decompressed, we analyze the 16 bits instruction saved in the instruction cache, thus transforming the instructions compressed 16 bits instructions in 32 bits uncompressed.

Now, if the address provided by the processor is not in the LAT, it means that there is no such instruction into the instruction cache. In this case, the LAT signal gives a miss from the instruction cache. The address provided by the processor will be transferred to the RAM (Random Access Memory), where it will be found and verified whether or not this instruction. If the search in RAM memory detects a miss,

then the instruction should be fetched in external memory, for example, the HD (Hard Disk). Now if the fetch process indicates a hit, it means that the instruction is in RAM. Next, the RAM returns a copy of the instruction in the original format (uncompressed) to the processor through the variable `returnC_inst_proc` and sends another copy to the compressor, which will make the whole process of compression.

The **compression of instruction codes** is performed as follows:

- The instruction is placed in RAM, a copy of it is passed to the processor and one for the compressor;
- The instruction in the compressor is splited into 16 pairs of bits, and each pair is formed at the moment that is read by the compression function. The beginning of the instructions coming from the RAM is the mode MSB (most significant bit);
- The compressor always considers what part of the double-byte (first or second) must be saved in the compressed instruction cache and instruction ST;
- If the pair of bits read for the compression is 00 or 11, then this pair of bits will be replaced by bit 0 and saved in the instruction cache. Now if the pair of bits read for 10 or 01, then this pair of bits will be replaced by bit 1 and saved in the instruction cache;
- The set of flags of ST will be formed through the following logic: if the first bit of double bits being compressed is 0, then the flag bit saved is 0. Now if the first bit of double bits being compressed is 1, then the flag bit will be 1 unless;
- After the compressor does all the compression of the original 32 bits instruction in 16 bits compressed and its bits flags, the compressor will save the double of bytes (first or second) compressed in the instruction cache and instruction set of flags in ST;
- The LAT table is updated with the new instruction address saved in the instruction cache;
- For each instruction that is sought in memory, repeat this process of compression.

It is important to note that this technique of compression/decompression is performed at runtime, via specific hardware that was prototyped in FPGA. In our hardware implementation we found that is similar to the work of LEKATSAS in [8, 9], we found a component that needs only a single cycle for the process of compression or decompression, and the benefits are shown in the next section.

For an analysis of performance method developed in this project (MIC), we have implemented in hardware the traditional method of Huffman compression, because it was used by WOLFE & CHANIN [13] in CCRP architecture and also by BENINI et al. [3] and LEFURGY et al. [7]. Therefore, comparing the MIC with Huffman allow for checking the strengths and weaknesses of this new approach to compression code with a method already used and highly esteemed in scientific community.

4 Simulations with Benchmark MiBench

The benchmark used in the simulations of compression and decompression of MIC methods and Huffman are programs from MiBench package [5]. It is specifically used for embedded systems and has different categories, which are in code assembly of

ARM9 processor, as obtained through the IDA Pro tool [15]. The category and functionality of MiBench benchmark used in the simulations are:

- **Dijsktra** (Network) is an algorithm that calculates the shortest path in a graph;
- **FFT** (Telecommunication): is an algorithm that performs the Fast Fourier Transform which is used in digital signal processing for finding the frequencies in an input signal;
- **MAD** (Consumer Device): is an audio decoder MPEG high quality;
- **QuickSort** (Automotive and Industrial Control): is an algorithm that makes ordering data;
- **SHA** (Security): is an algorithm that generates encryption keys for secure exchange of data and digital signatures;
- **Stringsearch** (Office): is an algorithm that makes the search for a string in a selected text.

We have used the instruction set of processor embedded ARM (ARM9 family, version ARM922T, ISA ARMv4T) to simulate the operation of the compressor and decompressor from MIC and Huffman algorithms into our PDCCM architecture. However, the chosen processor (ARM) is the type RISC and instruction set consists of 32 bits (instruction) which enabled it to be a good platform to simulate the PDCCM architecture.

The only change needed in the PDCCM architecture for using the Huffman method was the replacement of ST component by HT (Table Huffman) that contains the tree of Huffman codes of compressed instructions.

For the simulations of compression and decompression of MIC and Huffman algorithms were selected the 4.096 (4K) first instructions for each MiBench (due to physical limitations of the FPGA used for prototyping), obtained by the compiled code (Assembly) to the embedded processor ARM, forming so the set of instructions sequences that were used to load a piece of RAM memory and instruction cache. For more details, see [14].

The stretch of RAM described in VHDL was used in all simulations with the benchmark MiBench and had fixed size of 4.096 lines of 4 bytes each, thus accounting for 131.072 bits and the instruction cache has a size of 512 lines of 32 bits each. Thus, we observe that there is a 8:1 ratio between the sizes of RAM and the instruction cache.

Table 1 shows the metrics on performance statistics and Table 2 shows the metrics of time for our PDCCM architecture, using both methods MIC and Huffman to compression/decompression of the instructions for some programs from MiBench, executing in FPGA; using both methods compression/decompression MIC and Huffman of the instructions for some programs MiBench.

Table 1. Statistical Perfornance of FPGA

	Compression		**Decompression**	
	MIC	***Huffman***	***MIC***	***Huffman***
Logic elements	1.460 (32%)	1.317 (28%)	3.464 (74%)	4.414 (94%)
Registers	825 (18%)	738 (16%)	1.045 (23%)	1.482 (32%)
Pins	177 (56%)	170 (54%)	177 (56%)	170 (54%)

In Table 1, the compression process, it is possible to observe that the method of Huffman proved to be a little more computationally efficient in our PDCCM architecture, being the amount of logic elements, registers and pins between the two methods not increased from 4% to the method of Huffman. In the process of decompression, the MIC method showed better results, and the amount of logic elements, registers and pins were on average approximately 9% less in favor of the MIC method.

Table 2. Timing Statistics of the FPGA

	Compression		**Decompression**	
	MIC	***Huffman***	***MIC***	***Huffman***
Time in the worst case	9.228 ns	9.712 ns	8.632 ns	10.708 ns
Clock in MHz	75.98 MHz	69.88 MHz	68.42 MHz	66.97 MHz
Clock in time	13.257 ns	14.579 ns	14.668 ns	14.900 ns

Table 2 shows that MIC method has better timing in the FPGA for all benchmark MiBench analyzed. In compression phase, it is observed a difference of 6.1 MHz (the clock frequency in MHz) more to the MIC method being one of the points that makes it more efficient than the method of Huffman. The time, in the worst case, for the two methods was very similar. In decompression, are observed in the clock MHz and time are quite similar for both methods, and the MIC method has a slight advantage compared to the method of Huffman (an improvement of 24% Time in the worst case, 3% Clock in MHz, 1.5% Clock in time).

Based on the 4.096 first instructions of benchmark MiBench obtained from assembly code compiled for ARM platform, observed in Table 3 the MIC method depressed by 50% the size of instructions, that is, 4.096 lines of the stretch of RAM used in the simulation, after the compression process has come to occupy only 2.048 lines in instruction cache. Instructions compressed using the Huffman obtained an average 30% less of compression compared to the size of RAM used in the simulation.

Table 3. Comparasion of the Rate of Compression

MiBench	***MIC***	***Huffman***
Dijsktra	2.048 (50%)	2.622 (36%)
FFT	2.048 (50%)	2.803 (32%)
MAD	2.048 (50%)	3.002 (27%)
QuickSort	2.048 (50%)	3.245 (21%)
SHA	2.048 (50%)	2.785 (32%)
StringSearch	2.048 (50%)	2.913 (29%)
Averages	**2.048 (50%)**	**2.895 (30%)**

Finally, based on these results, we find that for PDCCM architecture using the 4.096 first instructions of the benchmark MiBench (Dijsktra, FFT, MAD, QuickSort, SHA and StringSearch), the MIC method was more efficient in compression, specifically 42% higher when compared with Huffman method.

5 Related Work

This section lists some researches founded in the literature related to compressed instruction codes. Table 4 shows a summary of the major existing work involving code compression and Table 5 the results obtained by the authors of the related work through the simulations and analysis accomplished.

Table 4. Summary of the Main Existing Work in Code Compression

Author	Method	Description
Wolfe and Chanin [13]	Compressed Code RISC Processor (CCRP)	• It was the first hardware decompression implemented in a RISC processor (MIPS R2000); • First technique to use the failures of access to the cache mechanism to trigger the decompression.
Azevedo [2]	Instruction Based Compression (IBC)	• The method divides the set of processor instructions in classes, taking into account the number of occurrences along with number of elements in each class (static analysis); • The technique is to group pairs in the format [prefix, codeword] which replaces the original code.
Benini et al. [3]	Compression of the Cache Line	• The unit used by the method of compression is the line of cache.
Lekatsas et al. [8, 9]	Grouping by Characteristics Own	• The method unites the instructions into 4 groups of characteristics own (immediate jump, quick and uncompressed); • The identification of the group is through a sequence of bits; • The decompressor hardware execute four pipelines at the same time; • Has developed a new architecture capable to decompress one or two instructions per cycle to meet the demand of the processor.
Lefurgy et al. [7]	Compression by Codeword	• The compression technique is based in the coding of the program by using codes dictionary; • After compiling the object code is analyzed and the common sequences of instructions are replaced by codeword; • The final code consists of codewords mixed with uncompressed instructions.

Table 5. Overview of Related Work

Author	Platform	benchmark	Compression ratio	Run time	Energy reduction
Wolfe and Chanin [13]	MIPS	lex, pswarp, matrix25, yacc, eightq, lloop01, xlisp, spim	73%	---	---
Azevedo [2]	SPARC	SPECint95	61,4%	+ 5,89%	---
	MIPS		53,6%	+ 5,89%	---
Benini et al. [3]	DLX	Ptolomy	72%	---	- 30%
Lekatsas et al. [8, 9]	SPARC	Compress, diesel, i3d, key, mpeg, smo	65%	- 25%	- 28%
	Xtensa-1040		65%	- 25%	---
Lefurgy et al. [7]	PowerPC	SPECint95	61%	---	---
	ARM		66%	---	---
	i386		75%	---	---

6 Conclusions and Future Work

This article described a new method of compression, called MIC, which was prototyped in FPGA, and we proved that it may be feasible for embedded systems that use RISC architecture.

We did many simulations with some programs from MiBench benchmark and verified that the MIC method showed the following results, averages values: 17% reduction in the usage of logic elements into FPGA, 6% higher of frequency (MHz) for compression and decompression process of instruction code and 42% more efficient at compression ratio when compared to Huffman method, which also projected in hardware.

Therefore, analyzing the data obtained through the simulations, despite the large memory used by the dictionary (ST) of the MIC, we have concluded that the method developed and presented in this article was computationally more efficient when compared with the Huffman method.

The simulations used programs for different categories, as follows: Dijsktra, FFT, MAD, QuickSort, SHA and StringSearch MiBench benchmark for performance measurements.

So, for the future this technique may become a necessary component in embedded systems projects, since that using code compression techniques, RISC architectures can minimize one of their biggest problems, which is the amount of memory to store programs.

As future work are: perform actual measurements of energy consumption at PDCCM architecture using MIC and Huffman methods; design and implement a RISC processor that already has the hardware built-in compressor and decompressor at its core; testing compression and decompression MIC and Huffman methods using more programs of MiBench. Furthermore, as a future work we like to suggest parallel processing into the compression algorithms and to visualize the impact of those new compression methods in the hierarchy memory and cache performance.

References

1. ARM. Advanced RISC Machines Ltd.: An Introduction to Thumb (March 1995)
2. Azevedo, R.: An architecture for code tablet Dedicated Systems. PhD thesis, Institute of Computing, University of Campinas, Brazil (June 2002)
3. Benini, L., Macii, A., Nannarelli, A.: Cached-Code Compression for Energy Minimization in Embedded Processor. In: Proceedings of the International Symposium on Low-Power, Electronics and Design (ISPLED 2001), Huntington Beach, CA, USA, pp. 322–327 (August 2001)
4. Davis, J., Goel, M., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., Xiong, Y.: Overview of the Ptolemy Project, ERL Technical Memorandum UCB/ERL, Tech. Report no. M-99/37, Dept. EECS, University of California, Berkeley (July 1999)
5. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, Austin, Texas, USA, pp. 3–14 (December 2001)

6. Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the Institute of Radio Engineers (IRE) 40(9), 1098–1101 (1952)
7. Lefurgy, C., Bird, P., Chen, I.-C., Mudge, T.: Improving Code Density Using Compression Techniques. In: Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO 30), Research Triangle Park, NC, USA, pp. 194–203 (December 1997)
8. Lekatsas, H., Henkel, J., Jakkula, V.: Design of One-Cycle Decompression Hardware for Performance Increase in Embedded Systems. In: Proceedings of the 39th Annual Design Automation Conference (DAC 2002), New Orleans, Louisiana, USA, pp. 34–39 (June 2002)
9. Lekatsas, H., Wolf, W.: Code Compression for Embedded Systems. In: Proceedings of the 35th Annual Design Automation Conference (DAC 1998), San Francico, California, USA, pp. 516–521 (June 1998)
10. Netto, E.B.W., Azevedo, R., Centoducatte, P., Araújo, G.: Mixed Static/Dynamic Profiling for Dictionary Based Code Compression. In: Proceedings of the International Symposium on System-on-Chip (SoC 2003), Tampere, Finland, pp. 159–163 (November 2003)
11. Netto, E.B.W., Azevedo, R., Centoducatte, P., Araújo, G.: Multi-Profile Based Code Compression. In: Proceedings of the 41th Annual Design Automation Conference (DAC 2004), San Diego, CA, USA, pp. 244–249 (June 2004)
12. de Oliveira, A.S., de Andrade, F.S.: Embedded Systems - Hardware and Firmware in Practice, 316 p. Érica, São Paulo (2006)
13. Wolfe, A., Chanin, A.: Executing Compressed Programs on an Embedded RISC Architecture. In Proceedings of 25th Annual International Symposium on Microarchitecture (MICRO 25), Portland, Oregon, USA, pp. 81–91 (December 1992)
14. Dias, W.R.A.: Architecture PDCCM in the Hardware for Compression/Decompression of Instructions in Embedded Systems. M.Sc. Dissertation, Department of Computer Science, Federal University of Amazonas, Brazil (April 2009)
15. IDA - The Interactive Disassembler, `http://www.hex-rays.com` (accessed on May 03, 2010)

Optimized Two Party Privacy Preserving Association Rule Mining Using Fully Homomorphic Encryption

Md. Golam Kaosar, Russell Paulet, and Xun Yi

School of Engineering and Science
Victoria University, Melbourne, VIC 8001, Australia
{golam.kaosar,xun.yi}@vu.edu.au, russell.paulet@live.vu.edu.au

Abstract. In two party privacy preserving association rule mining, the issue to securely compare two integers is considered as the bottle neck to achieve maximum privacy. Recently proposed fully homomorphic encryption (FHE) scheme by Dijk et.al. can be applied in secure computation. Kaosar, Paulet and Yi have applied it in preserving privacy in two-party association rule mining, but its performance is not very practical due to its huge cyphertext, public key size and complex carry circuit. In this paper we propose some optimizations in applying Dijk et.al.'s encryption system to securely compare two numbers. We also applied this optimized solution in preserving privacy in association rule mining (ARM) in two-party settings. We have further enhanced the two party secure association rule mining technique proposed by Kaosar et.al. The performance analysis shows that this proposed solution achieves a significant improvement.

1 Introduction

1.1 Privacy Preserving Association Rule Mining

Privacy preserving ARM is one of the simplest way of mining data among multiple data sites without revealing any private information. At the end of mining, any data site can learn nothing, but its own inputs and the final outcomes. Unfortunately, so far not may solutions exist to ensure entire data privacy. Many privacy oriented applications along with association rule mining is restricted by a long time open problem of securely comparing two numbers efficiently. Most distributed data mining solutions require privacy preservation. Privacy preserving data mining encompasses two goals [20]: (1) achieving privacy requirement and (2) achieving valid mining result. Sometimes achieving both becomes difficult since increase of one may decrease the other. Privacy preserving association rule mining algorithm first was proposed by Agrawal [10] in 2000 which used data perturbation and reconstruction based solution. After that, many research works have been performed to preserve the privacy in data mining applications. Privacy preservation techniques are approached through various methods such as - data

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 360–370, 2011.

perturbation and randomization [3], [21], [19], [4], sensitive data hiding, rule hiding and sanitization [2], [6], [14], [23], [30], data sampling and cryptographic techniques [16], [25], [8], [9], [31], [28], [17]. Dijk et.al.'s [7] FHE system is based on Gentry's lattice based solution [12,13]. To ensure proper functioning their solution had to evolve from symmetric to asymmetric to somewhat homomorphic to bootstrapable to fully homomorphic solution. The cryptosystem is inefficient due to its key size and accumulative nature of noise terms within the ciphertext. Gentry proposes some solutions to alleviate these problems in [13]. [13] also proposes the technique to reduce the key size. [27] proposes some improvements in Gentry and Dijk et.al.'s solution. This fully homomorphic encryption technique is already proposed to be used in many applications within this short period of time since its publication such as - in outsourced computing [11], and secure comparison [18]. Moreover this can be applied in many applications such as private information retrieval, oblivious transfer, cloud computing.

Most privacy preserving ARM solutions are incapable to ensure full privacy. They disclose intermediate results such as total database size. Moreover, the privacy preservation in two party settings sometimes becomes infeasible for some solutions. Kaosar et.al. [18] have proposed a secure two party ARM solution using FHE of [7] which maximizes the data privacy since it does not disclose any intermediate result. In the solution of two-party distributed horizontally partitioned database settings, no party can learn anything other than its own input and the final results. But [18] is inefficient for practical applications. The identified main reasons are - inefficient nature of of the FHE scheme of [7] due to complex and inefficient mathematical techniques used in comparing two integer numbers. In this paper, we present number of optimizations compared to [18] such as - (1) In the server side (in this case site S_2), it does not need to perform encryption operation for its own inputs at all. It only performs some basic operations on the ciphertexts received from site S_1 and returns a resultant ciphertext of one bit only. (2) In [18] many multiplications and additions of ciphertexts were required, but in this paper we propose an optimized way to compare two integers which require few basic operations only. (3) We present a simplified mathematical technique to compare two integer numbers which reduces the overhead drastically.

1.2 Motivation and Problem Formulation

Consider two data sites S_1 and S_2 possess two horizontally partitioned transactional database DB_1 and DB_2 of size d_1 and d_2 respectively, where combined database $DB = \{DB_1 \bigcup DB_2\}$. Let us also assume, $I = \{i_1, i_2, ..., i_n\}$ is the set of unique items, where each transaction $T \subseteq I$. Therefore, any itemset to be frequently large, its support must be greater than or equal to the minimum support threshold denoted by s. Similarly for an association rule to be chosen, its confidence must be greater than or equal to a minimum confidence threshold denoted by c.

The association rule mining algorithm between S_1 and S_2 must be privacy preserved. This means that, neither S_1 nor S_2 should be able to learn anything other than its own input and the final result (resultant association rules). The count of each itemset is needed to compare to determine whether in unified database (DB) that particular itemset becomes globally frequent or not. Consider a simple case where S_1 and S_2 have count of a particular itemset c_1 and c_2 respectively. Their database sizes are d_1 and d_2 respectively. They want to securely compute whether that itemset would be frequently large in their combined database or not. In other words, they have to determine the following inequality securely.

$$\frac{c_1 + c_2}{d_1 + d_2} \geqslant \frac{t}{100} \tag{1}$$

Where, t is the minimum number of times the itemset has to appear in the combined database DB. Equation 1 can be further expressed as:

$$c_1 \times 100 - t \times d_1 \geqslant t \times d_2 - c_2 \times 100 \Rightarrow \alpha \geqslant \beta \tag{2}$$

The left hand side and the right hand side of the Equation 2 are to be computed by party S_1 and S_2 respectively. It should be noted that, in [18], S_1 transmits the encrypted count c_1 and database size d_1 to S_2. Therefore, S_2 performs the comparison operation based on $\{E_{pk}(c_1), E_{pk}(d_1), E_{pk}(c_2), E_{pk}(d_2), E_{pk}(t)\}$ as opposed to $\{E_{pk}(\alpha), \beta\}$ in case of this solution. In our proposed solution, we propose an optimized technique to compare α and β in Section 3.1.

As discussed above, FHE of [7] is very inefficient to be implemented in practical settings. Its operations 'AND' and 'XOR' work bit by bit; hence, to use this for integer operations involves huge amount of computation. We propose, apply this encryption system to securely compare two integers in such a way that, second party (S_2) does not perform any encryption operation on its own inputs; as opposed to [18]. S_2's inputs are represented as binary bits; based on its bit values, it performs some basic operations which essentially generates the correct outputs similar to [18]. This technique is illustrated in Figure 1 and Algorithm 1 in Section 3.1.

2 Background

2.1 Notations

Following notations will be used in rest of the paper.

$|$: Concatenation operation.
$\oplus$: Represents XOR operation between two binary bits. It works for integers too, where XOR is performed between corresponding bit by bit.
$\otimes$: Represents AND operation between two binary bits. If $a, b \in \{0, 1\}$, then ab is also equivalent to $a \otimes b$.

$\overline{X}$: Represents NOT operation of a binary bit X. It works for integers too, where values of all the bits of the integer are negated.
$\boxplus$: Represents homomorphic XOR operation between both plaintexts and ciphertexts. Operands can be either binary bits or integer numbers.
$\boxtimes$: Represents Homomorphic AND operation between both plaintexts and ciphertexts. Operands can be either binary bits or integer numbers.
$E_{pk}(X)$: Represents fully homomorphic encryption of X, using the public key pk. X can be either binary bit or integer number. If X is integer, it encrypts each bit of X separately and concatenates together.
$D_{sk}(X)$: Represents fully homomorphic decryption of X, using the private key sk where, X is a ciphertext about a binary bit.

Detail definition of $\boxplus$, $\boxtimes$, $E_{pk}(X)$, $D_{sk}(X)$ can be found in [7] and [18].

2.2 Some Binary Operations

Integer Addition. Let us consider, two n-bit integer numbers $A = \{A_n|A_{n-1}|...|A_2|A_1\}$ and $B = \{B_n|B_{n-1}|...|B_2|B_1\}$ where, $A_i, B_i \in \{0,1\}$. To add this two integers, every bit in one number is added with corresponding bit in another number along with the carry bit of previous stage. The summation result is R, where $R = \{R_n|R_{n-1}|...|R_2|R_1\}$ and carry is C_i for stage i and $C_0 = 0$. Thus expression of R_i and C_i would be as follows (Detail can be found in [26]):

$$R_i = A_i\overline{B_iC_{i-1}} + \overline{A_i}B_i\overline{C_{i-1}} + \overline{A_iB_i}C_{i-1} + A_iB_iC_{i-1} = A_i \oplus B_i \oplus C_{i-1} \quad (3)$$

$$\begin{aligned} C_i = \overline{A_i}B_iC_{i-1} + A_i\overline{B_i}Ci - 1 + A_iB_i\overline{C_{i-1}} + A_iB_iC_{i-1} \\ = C_{i-1}(A_i \oplus B_i) + A_iB_i \end{aligned} \quad (4)$$

Integer Subtraction. Consider two numbers A and B. Two's complement of $A = \overline{A} + 1$ [26], which is negative of $|A|$ in two's complement representation. Detail is discussed in [18]. Therefore $A - B = A + \overline{B} + 1$.

Comparison of Two Integers. Let us consider, two integers A and B needed to be compared to determine which one is larger. In principle, if two's compliment of B is added with A then we get $A - B$; as presented in previous paragraph. If the operation is assumed to be of n bit long, then n^{th} bit of the summation result (say R_n) determines which one is larger or equal. If $R_n = 0$, then $A \geqslant B$ otherwise $A < B$.

2.3 Fully Homomorphic Encryption (FHE)

Homomorphic encryption is a special form of encryption, where one can perform a specific algebraic operation on the plaintext by applying the same or another one operation on the ciphertext. If X and Y are two numbers, and E and D

denotes encryption and decryption function respectively, then homomorphic encryption holds following condition for algebraic operations such as '+' and '×' respectively:

$$D[E(X) + E(Y)] = D[E(X + Y)] \tag{5}$$

$$D[E(X) \times E(Y)] = D[E(X \times Y)] \tag{6}$$

Most homomorphic encryption system such as RSA [24], ElGamal [10], Benaloh [5], Paillier [22] etc are either additive or multiplicative, but fully homomorphic encryption system can be used for many operations (such as, addition, multiplication, division etc.) at the same time. Dijk et.al. [7] propose a new encryption that provides fully homomorphic encryption over integer ciphertext. This fully homomorphic scheme [7] is a simplification of an earlier work involving ideal lattices [12]. It encrypts a single bit (in the plaintext space) to an integer (in the ciphertext space). When these integers are added and multiplied, the hidden bits are added and multiplied (modulo 2). The symmetric version of the encryption function is given by $c = pq + 2r + m$, where p is the private key, q and r are chosen randomly, and m is the message $m \in \{0, 1\}$. The decryption is simply $(c \ mod \ p) \ mod \ 2$, which recovers the bit. Hence, when we add or multiply the ciphertext, the message is manipulated accordingly.

Using the symmetric version of the encryption, it is possible to construct an asymmetric version which is more useful to the association rule mining application, since another party must be able to encrypt in order to use the homomorphic property of the encryption system. Detail of the encryption system is discussed in [7] as well as in [18].

3 Proposed Solution

ARM technique consists of two major parts - (i) Global frequent itemset (say L_g) generation and (ii) Association rule generation from L_g. To determine frequent itemset within a database, it is necessary to compare counts of all possible itemsets. In many research works, some solutions are provided to reduce this number of candidate itemset, as well as number of comparisons. The Apriori algorithm [1] is one of the leading algorithms, which determines all frequent itemsets along with their support counts from a database efficiently. More detail on ARM can be found in [15,29] and [18]. This section illustrates, how some enhancements are done on two-party secure ARM proposed in [18].

3.1 ARM with Privacy Preservation

If the Equation 2 can be computed securely, two sites S_1 and S_2 can compute global frequent itemset (L_g) keeping the privacy preserved. Similarly from this L_g, association rules also can be computed securely. Detail can be found in

[18]. In Apriori based ARM frequent k-itemset (L_k) is computed from candidate $(k+1)$-itemset, where L_1 is the all frequent 1-itemset. Each iteration of our privacy preserving two party ARM algorithm can be summarized in following major steps assuming there exists L_k:

1: Generate candidate itemset: Each data site computes its local candidate itemset C_{k+1} from L_k.
2: Frequent itemset generation: Two sites exchange information securely to determine frequent itemset L_{k+1} from C_{k+1} of S_1 and S_2. Update: $L_g = L_g \bigcup L_{k+1}$.
3: Repeat Step 1 and Step 2 until there remain an itemset's count $\geqslant t$.
4: Compute association rules from L_g.

In this proposed solution we only focus on Step 2; assuming the rest of the steps are trivial. In Step 2, the itemset counts are needed to be compared securely. Therefore, the main focus of the solution is simplified into - comparing two numbers securely (as stated in Equation 2).

Secure Comparison of Two Integers. Let us assume, both S_1 and S_2 agree n be the bit length of both α and β. S_1 computes $E_{pk}(\alpha)$, form of which is a concatenation of n integers. i.e. $E_{pk}(\alpha) = \hat{\alpha} = \{a_n|a_{n-1}|...|a_1\}$, where a_i is integer and $a_i = E_{pk}(\alpha_i)$, $i = 1$ *to* n. S_1 sends $\hat{\alpha}$ to S_2. S_2 computes $B = \overline{\beta}+1 = \{B_n|B_{n-1}|...|B_1\}$, where $B_i \in \{0,1\}$, $i = 1$ *to* n. To compare the numbers, S_2's operations are illustrated in Figure 1. It should be noted that, a_i, R_i and C_i are encryption of some bits; represented as an integer. On the other hand, b_i is plain binary digits visible to S_2. Instead of encrypting its value, S_2 performs some conditional operations based on the value of b_i to deduce R_n. S_2 runs Algorithm 1 and returns R_n to S_1.

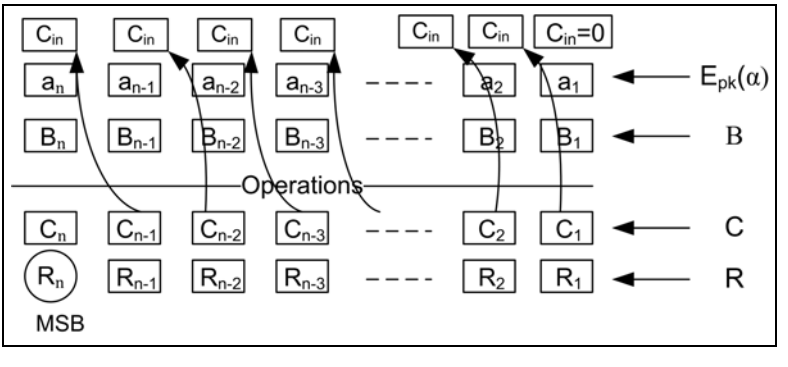


Fig. 1. S_2 computes the final bit to determine comparison result. Result (R) and Carry (C) are computed based on Equation 3 and 4 respectively.

Algorithm 1. S_2 computes the resultant bit for comparison

input : $\hat{\alpha}, B$ /* $\hat{\alpha} = \{a_n|a_{n-1}|...|a_1\}$ and $B = \overline{\beta} + 1 = \{B_n|B_{n-1}|...|B_1\}$*/
output : R_n /* R_n is the resultant bit of the comparison operation*/
$c_0 \leftarrow 0$
for $i = 1$ *to* n **do**
  if $B_i = 0$ **then**
    $R_i \leftarrow a_i \boxplus C_{i-1}$
    $C_i \leftarrow C_{i-1} \boxtimes a_i$
  else
    $R_i \leftarrow (a_i \boxplus 1) \boxplus C_{i-1}$
    $C_i \leftarrow C_{i-1} \boxtimes (a_i \boxplus 1) \boxplus a_i$
  end if
end for
return R_i

S_1 decrypts R_i as $r \leftarrow D_{sk}(R_i)$. If $r = 0$ then $\alpha \geqslant \beta$, else $\alpha < \beta$. S_1 shares r with S_2.

Correctness Proof. Let us assumed FHE holds $\boxplus$ and $\boxtimes$ for one bit ciphertext. It is also proven from Section 2.2 that, n^{th} bit of the result of the summation operation provides the clue about which number is greater or smaller. Now let us consider Algorithm 1, S_2 modifies the equation of summation and carry based on the corresponding bit. If $B_i = 0$ then Equation 3 and 4 can be simplified as:

$$R_i = a_i \oplus B_i \oplus C_{i-1} = a_i \oplus 0 \oplus C_{i-1} = a_i \oplus C_{i-1} \tag{7}$$

$$C_i = C_{i-1}(a_i \oplus B_i) + a_i B_1 = C_{i-1}(a_i \oplus 0) + a_i x 0 = C_{i-1} a_i \tag{8}$$

Similarly if $B_i = 1$ then Equation 3 and 4 can be simplified as:

$$\begin{aligned} R_i &= a_i \oplus B_i \oplus C_{i-1} = a_i \oplus E_{pk}(1) \oplus C_{i-1} \\ &= (a_i \boxplus 1) \boxplus C_{i-1} = a_i \oplus C_{i-1} + 1 \end{aligned} \tag{9}$$

At the end of the loop in the algorithm, i becomes equal to n. Therefore $R_i = R_n$. In Equation 9, both a_i and C_{i-1} are ciphertexts of the form $m + 2r + pq$ (basic ciphertext form of FHE). Therefore $a_1 \oplus C_{i-1} \oplus E_{pk}(1) = a_1 \oplus C_{i-1} + 1$. Similarly equation of carry can be simplified as:

$$\begin{aligned} C_i &= C_{i-1}(a_i \oplus B_i) + a_i B_1 = C_{i-1}(a_i \oplus E_{pk}(1)) + a_i E_{pk}(1) \\ &= C_{i-1} \boxtimes (a_i \boxplus 1) \boxplus a_i = C_{i-1}(a_i + 1) + a_i \end{aligned} \tag{10}$$

Thus, R_n is correctly formed which is a function of ciphertext and plaintext computed by basic operations supported by FHE.

4 Performance and Security

4.1 Performance Analysis

The number of operations involved in this proposed solution and [18] to perform one comparison operation securely are discussed as follows:

This solution:
S_2 does not need to perform any kind of encryption at all. According to Algorithm 1, number of multiplication and addition operations are n and n respectively. Number of encryptions are also n.

Solution of [18]:
Counts of the particular itemset and the database size is encrypted in both S_1 and S_2. Therefore, number of encryptions are $2n$. To compute total number of multiplication and addition operations, let us count them according to the solution of [18]:

- To compute $\frac{E_{pk}(c_1)+E_{pk}(c_2)}{E_{pk}(|DB_1|)+E_{pk}(|DB_2|)}$ into the form of $\frac{\alpha}{\beta}$, it requires $6n$ multiplications and $8n$ additions.
- Secure comparison of two fractional numbers in Section 3.3 involves $600n$ multiplications and $800n$ additions assuming two decimal precision of the threshold value is considered.
- Secure comparison of two encrypted integers in Section 3.2 involves $6n$ multiplications and $10n$ additions.

Table 1 summarizes the number of different operations involved in comparing two integers securely. From the performance comparison it is clearly understood that, our optimized solution demonstrates outstanding performance compared to the solution of [18]. This is due to (i) mathematical simplification in comparing two integers, (ii) simplified carry bit expression and (iii) avoidance of encryption in S_2 side. Moreover, the table is about one single comparison operation in the ARM process. If the whole mining process is considered then the performance comparison would show a dramatic difference between the two solutions.

Table 1. Performance comparison

Parameter	Proposed Solution		Solution of [18]	
	S_1	S_2	S_1	S_2
Number of multiplications	0	n	0	$\approx 600n$
Number of additions	0	n	0	$\approx 800n$
Number of encryptions	n	0	n	n
Number of decryptions	1	0	1	0

4.2 Security Analysis

The security of our protocol includes security of S_1 and security of S_2. The security of S_1 means any input from S_1 during association rule mining is kept private from S_2 and security of S_2 means any input from S_2 is kept private from S_1.

In our protocol, any input from S_1 is encrypted by the public key of S_1, and only S_1 knows the secret key for decryption. If our underlying fully homomorphic encryption scheme is secure, S_2 cannot obtain any information about the input of S_1. Our underlying fully homomorphic encryption, Dijk et.al.s scheme, uses

a bootstrapping technique to achieve fully homomorphic property, where the addition or multiplication of two ciphertexts is implemented by squashing the two ciphertexts with a very complicated decryption circuit at first and then adding or multiplying the resultant ciphertexts. Due to bootstrappability, the relationship between the encryption of the sign bit and the input from S_2 is so random that it is impossible for S_1 to guess the input of S_2 from the encryption of the sign bit. Based on this, S_2 security can be achieved. In summary, our protocol secures both S_1 and S_2, since FHE of [7] is secure, based on approximate GCD problem.

5 Conclusion

In this paper, we have investigated many ways to improve the previously proposed solution [18] towards the secure two party ARM algorithm. Some of the achievements are: (i) mathematics behind itemset count comparison is enhanced, (ii) carry bit expression is much simplified, (iii) encryption in S_2 is avoided (iv) number of multiplication and addition operations between ciphertexts are reduced dramatically. However, this two-party solution can be extended for multi-party too by repeatedly comparing itemset counts among all participants.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487–499. VLDB, Santiago (1994)
2. Atallah, M., Elmagarmid, A., Ibrahim, M., Bertino, E., Verykios, V.: Disclosure limitation of sensitive rules. In: KDEX 1999: Proceedings of the 1999 Workshop on Knowledge and Data Engineering Exchange, p. 45. IEEE Computer Society, Washington, DC, USA (1999)
3. Chen, K., Liu, L.: Privacy preserving data classification with rotation perturbation. In: ICDM 2005: Proceedings of the Fifth IEEE International Conference on Data Mining, pp. 589–592. IEEE Computer Society, Washington, DC, USA (2005)
4. Chen, K., Liu, L.: A random geometric perturbation approach to privacy-preserving data classification. Technical Report, College of Computing, Georgia Tech. (2005)
5. Clarkson, J.B.: Dense probabilistic encryption. In: Proceedings of the Workshop on Selected Areas of Cryptography, pp. 120–128 (1994)
6. Dasseni, E., Verykios, V.s., Elmagarmid, A.K., Bertino, E.: Hiding association rules by using confidence and support. In: IHW 2001: Proceedings of the 4th International Workshop on Information Hiding, London, UK, pp. 369–383. Springer, Heidelberg (2001)
7. Dijk, M.V., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010)
8. Duan, Y., Canny, J., Zhan, J.: Efficient privacy-preserving association rule mining: P4p style. In: IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2007, March 1-April 5, pp. 654–660 (2007)

9. Evfimievski, A., Srikant, R., Agrawal, R., Gehrke, J.: Privacy preserving mining of association rules. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2002, pp. 217–228 (2002)
10. El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakely, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (1984)
11. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Heidelberg (2010)
12. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC 2009: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 169–178. ACM, New York (2009)
13. Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. Cryptology ePrint Archive, Report 2010/520 (2010)
14. Gkoulalas-Divanis, A., Verykios, S.V.: Association Rule Hiding for Data Mining. Springer, Heidelberg (2010) ISBN:9781441965691
15. Han, J., Kamber, M.: Data Mining Concepts and Techniques, 2nd edn. Elsevier Inc., Amsterdam (2006)
16. Hussein, M., El-Sisi, A., Ismail, N.: Fast cryptographic privacy preserving association rules mining on distributed homogenous data base. In: Lovrek, I., Howlett, R.J., Jain, L. (eds.) KES 2008, Part II. LNCS (LNAI), vol. 5178, pp. 607–616. Springer, Heidelberg (2008)
17. Kantarcioglut, M., Clifton, C.: Privacy-preserving distributed mining of association rules on horizontally partitioned data. IEEE Trans. on Knowl. and Data Eng. 16(9), 1026–1037 (2004)
18. Kaosar, M.G., Paulet, R., Yi, X.: Secure two-party association rule mining. In: Australasian Information Security Conference (AISC 2011), pp. 17–20 (January 2011)
19. Liu, K., Kargupta, H., Ryan, J.: Random projection-based multiplicative data perturbation for privacy preserving distributed data mining. IEEE Trans. on Knowl. and Data Eng. 18(1), 92–106 (2006)
20. Oliveira, S., Oliveira, S.R.M., Zaane, O.R.: Toward standardization in privacy-preserving data mining. In: Proc. of the 3nd Workshop on Data Mining Standards (DM-SSP 2004), in conjuction with KDD 2004, pp. 7–17 (2004)
21. Oliveira, S.R.M., Zaane, O.R.: Achieving privacy preservation when sharing data for clustering. In: Proc. of the Workshop on Secure Data Management in a Connected World (SDM 2004) in conjunction with VLDB 2004, Toronto, Canada, pp. 67–82 (2004)
22. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
23. Ramaiah, B.J., Rama, A.R.M., Kumari, M.K.: Parallel privacy preserving association rule mining on pc clusters, pp. 1538–1542 (March 2009)
24. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)
25. Rizvi, S., Haritsa, J.R.: Maintaining data privacy in association rule mining. In: Proceedings of the 28th VLDB Conference, Hong Kong, pp. 682–693 (2002)
26. Saha, A., Manna, N.: Digital Principles and Logic Design. Laxmi Publications (2008)

27. Stehlé, D., Steinfeld, R.: Faster fully homomorphic encryption. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 377–394. Springer, Heidelberg (2010)
28. Su, C., Sakurai, K.: A distributed privacy-preserving association rules mining scheme using frequent-pattern tree. In: Tang, C., Ling, C., Zhou, X., Cercone, N.J., Li, X. (eds.) ADMA 2008. LNCS (LNAI), vol. 5139, pp. 170–181. Springer, Heidelberg (2008)
29. Tan, P.N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Pearson Education, Inc., London (2006)
30. Wu, C.M., Huang, Y.F., Chen, J.Y.: Privacy preserving association rules by using greedy approach. In: World Congress on Computer Science and Information Engineering, vol. 4, pp. 61–65 (2009)
31. Yi, X., Zhang, Y.: Privacy-preserving distributed association rule mining via semi-trusted mixer. Data & Knowledge Engineering 63, 550–567 (2007)

SLA-Based Resource Provisioning for Heterogeneous Workloads in a Virtualized Cloud Datacenter

Saurabh Kumar Garg, Srinivasa K. Gopalaiyengar, and Rajkumar Buyya

Cloud Computing and Distributed Systems Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{sgarg,raj}@csse.unimelb.edu.au

Abstract. Efficient provisioning of resources is a challenging problem in cloud computing environments due to its dynamic nature and the need for supporting heterogeneous applications with different performance requirements. Currently, cloud datacenter providers either do not offer any performance guarantee or prefer static VM allocation over dynamic, which lead to inefficient utilization of resources. Earlier solutions, concentrating on a single type of SLAs (Service Level Agreements) or resource usage patterns of applications, are not suitable for cloud computing environments. In this paper, we tackle the resource allocation problem within a datacenter that runs different type of application workloads, particularly non-interactive and transactional applications. We propose admission control and scheduling mechanism which not only maximizes the resource utilization and profit, but also ensures the SLA requirements of users. In our experimental study, the proposed mechanism has shown to provide substantial improvement over static server consolidation and reduces SLA Violations.

1 Introduction

With the increasing popularity of Cloud computing, research centers and enterprises have started outsourcing their IT and computational needs to on-demand cloud services [3].The clouds are typically large scale virtualized datacenters hosting thousands of servers. While there are several advantages of these virtualized infrastructures such as on-demand scalability of resources, there are still issues which prevent their widespread adoption in clouds. In particular, for a commercial success of this computing paradigm, the cloud datacenters need to provide a better and strict Quality of Service (QoS) guarantees. These guarantees which are documented in the form of Service Level Agreement (SLA) are crucial, since only then the customers can be confident in outsourcing their jobs to clouds [19]. Resource provisioning plays a key role in ensuring that the cloud providers adequately accomplish their obligations to customers while maximizing the utilization of underlying infrastructure. An efficient resource management scheme would require to automatically allocate to each service request,

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 371–384, 2011.

the minimal resources needed for acceptable fulfillment of SLAs, leaving the surplus resources free to deploy more virtual machines. The provisioning choices must adapt to changes in load as they occur, and respond gracefully to unanticipated demand surges. For these reasons, partitioning the datacenter resources among the various hosted applications automatically is a challenging task. Current cloud datacenters hosts a wider range of applications with different SLA requirements[12]. The intrinsic differences among these different workloads further make the resource provisioning a challenging task [15]. First, the SLA requirements of different applications are different. The transactional applications require response time and throughput guarantee, while non-interactive batch job concerns performance (e.g. completion times). Second, the resource demand of transactional applications such as Web application tend to be highly unpredictable and busty in nature[4], while of batch jobs can be predicted to a higher degree[14]. Hence, the satisfaction of complex and different requirements of all applications makes the goal of utilization maximization while meeting different types of SLAs far from trivial.

Traditionally, to meet SLA requirements, over-provisioning of resources to meet worst case demand are used. However, since servers operate most of the time at very low utilization level, it leads to waste of resources in non-peak times. This over-provisioning of resources results in extra maintenance costs including server cooling and administration [8]. Many researchers tried to address these issues by dynamic provisioning of resources using virtualization, but they focused mainly on scheduling based on one specific type of SLA or application type such as transactional workload. Even though computationally intensive applications are increasingly becoming the part of enterprise datacenter, still research with mixed types of applications having different SLAs is in infancy. Today, most of the datacenters run different types of applications on separate VMs without any awareness of their different SLA requirements such as deadline, which may result in resource under-utilization and management complexity.

Therefore, in this paper, we present a novel dynamic resource management strategy that not only maximizes the utilization by sharing resources among multiple concurrent applications owned by different users, but also considers SLAs of different types. We handle scheduling of two types of applications i.e., compute intensive non-interactive jobs and transactional applications, each having different types of SLA requirements and specifications. Our strategy makes dynamic placement decisions to respond to the changes in transactional workload, and also considers the SLA penalties for making future decisions. To schedule batch jobs, our proposed resource provisioning mechanism predicts the future resource availability and schedules the jobs by stealing CPU cycles, which are unutilized by the transactional applications.

2 Related Work

In this section, we compare our work with the most relevant ones. Meng et al. [9] proposed a joint-VM provisioning and borrowing approach by exploiting the statistical multiplexing among the workload patterns. Zhang et al. [20] designed

an approach to quickly reassign the resources for a virtualized utility computing platform using ghost virtual machines (VMs). These works concentrate on fixed number of VMs, while we have considered variable amount of incoming workload. Vijayaraghavan and Jennifer [16] focussed on the analysis and resource provisioning for the datacenters's management workloads which have considerable network and disk I/O requirements. Singh et al. [13] argue that the workload in internet applications is non-stationary and consider the workload mix received by a Web application for their mix-aware dynamic provisioning technique. In contrast, our work concentrates on handling multiple types of SLA both for High Performance Computing (HPC) and Web based workloads with a new admission control policy. Quiroz et al. [12] present a decentralized, robust online clustering approach for a dynamic mix of heterogeneous applications on Clouds, such as long running computationally intensive jobs, bursty and response-time sensitive WS requests, and data and IO-intensive analytics tasks. When compared to our approach, the SLA penalty is not considered and it uses a static number of VMs. Wang et al. [18] evaluated the overhead of a dynamic allocation scheme in both system capacity and application-level performance relative to static allocation. In our work, the idea of dynamic allocation is extended for multiple types of workloads including HPC and Web. Carrera et al. [4] developed a technique that enables existing middleware to fairly manage mixed workloads both in terms of batch jobs and transactional applications. The aim of this paper is towards a fairness goal while also trying to maximize individual workload performance. But, our aim is to efficiently utilize the datacenter resources while meeting the different type of SLA requirements of the applications.

The main **contributions** of this paper lies with the design of an efficient admission control and scheduling mechanism for Cloud datacenters with the following salient features: a) adaptive admission control and dynamic resource provisioning facility, b) considered multiple type of SLAs based on application requirements, c) integration of mixed/heterogeneous workloads (such as non-interactive and transactional applications) for better utilization of resources, and d) variable penalties depending on the type of SLA violation.

3 System Model

3.1 Datacenter Model

We consider a Cloud virtualized datacenter model. Each server is interconnected with a high-speed LAN network and high bandwidth link to the Internet. The key components involved in the process of scheduling an application on a VM are: admission control, VM manager, job scheduler and SLA manager. The admission control component decides whether the requested VM (for an application) can be allocated and the QoS requirements can be met with a given number of available resources. If an application is accepted for execution, SLA is signed and penalty rates are negotiated with the user. The VM manager will initiate a VM and allocate it to a server having the required capacity. Job scheduler will schedule applications on this newly initiated VM. SLA manager monitors the current

SLAs and service level for each accepted application. We consider the two types of application workloads i.e., transactional and non-interactive batch jobs. Since both applications have different QoS requirements, different charging models are used. Transactional applications are offered a set of VMs with varying capacity to allow the user to choose as per his requirements and they can be charged on hourly basis. The auto-scaling facility can also be provided if resource demand exceeds the VM size allocated. In the next section, we discuss the two types of application workloads considered in this work along with their SLA definitions.

3.2 SLA and Application Models

The transactional workloads include Web applications whose resource demand can vary with time. On the other hand, for the non-interactive workloads, we model the HPC compute intensive bag of task applications. Most of the scientific workloads include mostly independent single-machine jobs (tasks) grouped into single "bag of tasks" [6]. Thus, it is assumed that there is no data communication between each task. The SLA model, that is used as the basis for determining the VM capacity, is discussed in detail below. We consider a discrete-time scenario in which time is slotted into intervals with equal length (T).

SLA Model for Transactional Workload: A user gives QoS requirements in terms of response time t_i with each transactional application i. This response time requirement can be translated to CPU power α_i needed to achieve this response time [4]. SLA will be broken if the capacity C_{t_i} allocated to the application is less than the required capacity at time t. A penalty q will incur if number of such violation increases beyond a threshold β_i. Thus, the net amount the user needs to pay at the end of the period (T_1, T_2) would be: $r * \alpha_i * (T_2 - T_1) - q$, if r is rate charged. Here, the user can choose three types of penalties:

- **Fixed Penalty:** The fixed penalty q is charged whenever the cloud provider fails to fulfil current capacity demand.
- **Delay-dependent Penalty:** The penalty is proportional to the delay incurred by the service provider in returning the capacity. If q' is the agreed penalty rate in the SLA and T_1 and T_2 are the time instants between which the capacity for the current demand is less than the reserved capacity (according to SLA), then the service provider's penalty due to the user is calculated as: $q' * (T_2 - T_1)$.
- **Proportional Penalty:** This is also a form of delay-dependent penalty, where the penalty to be credited to a user which is proportional to the difference between the user's provisioned capacity C_1 and its current allocation C_2. If q' is the agreed penalty rate per unit capacity per unit time, T_1 and T_2 are the respective times when the capacity was requested and allocated, then the penalty is given as: $q' * (T_2 - T1) * (C_1 - C_2)$.

SLA also contains the information regarding auto-scaling option. If the user choose this facility, when demand for resources increases beyond the initial requested amount, more resources will be allocated to meet this spike and

automatically reduced when demand is decreased to a threshold. The Cloud provider can charge an additional amount for offering such flexibility.

SLA Model for Non-interactive Batch Jobs: QoS requirements for batch jobs are deadline and the amount of CPU time allocated. These jobs require performance based guarantees. Since, the performance of VMs allocated can vary with the usage of datacenter and we define SLA as the p amount of CPU Cycles to be provided by the Cloud provider before the deadline d in order to ensure successful completion of job. Let db be the delay before allocating p CPU cycles. Then, if the provider fails to complete the given job, following penalty applies: $q = y * db$, where y is the penalty rate.

4 Admission Control and Scheduling Policy

As discussed in the earlier sections, we need to consider the requirements of two different application workloads before accepting the new requests and also while serving the accepted one. The main idea is to monitor the resource demand during the current time window in order to make decisions about the server allocations and job admissions during the next time window. Datacenter resource allocation is monitored and reconfigured in regular intervals. At a given point of time, it is assumed that if a host is idle for certain amount of time or not running any applications, then it will be switched-off. In each scheduling cycle, admission control and scheduling can perform three functions:

- Admission Control: It decides to accept or reject a new application (transactional or batch) based on present and future resource availability.
- SLA Enforcement: The resource demand of each application is monitored based on agreed QoS level guaranteed in SLAs. In this step, dynamic resource reconfiguration is done to handle the demand bursts of transactional applications and to meet SLA requirements.
- Auto-Scaling: If the resource demand of any application exceeds the requested (reserved) capacity in the SLA, a new VM instance can be initiated based on the resource availability and auto-scaling thresholds.

All the above operations depend on the forecasting module which predicts the future resource availability and the expected resource demand of each application. We will discuss the above three functions and methodologies in following sections. The *batch job queue* component represents the queue of VMs (corresponding to each batch job) which are waiting for execution.

The *batch job queue* is sorted based on a *threshold time* up to which a batch job can be delayed without any penalty. Let the d be the deadline, MI be the CPU cycles (Millions of Instructions) required for completion of job, and C_{min} be the Million of Instruction Per Second(MIPS) of smallest standardized VM offered by the Cloud provider, then the threshold time $threshTime(i)$ for batch job i is calculated as : $threshTime(i) = d - \frac{MI}{C_{min}}$.

4.1 Forecasting Model

In this paper, we have used an Artificial Neural Network (ANN) based forecasting model, which is a multi-layer feedforward network. It had been shown in literature [17] [1] that ANNs perform better than other linear models (e.g., regression models), specifically, for more irregular series and for multiple-period-ahead forecasting. Such neural network techniques has also been used and well studied in previous works for distributed application scheduling in context of Grids [5]. *It is important to note that it is not our goal here to determine the best prediction technique.* However, our contribution is that if the resource requirements of an application is well predicted, we can maximize the resource utilization considering different SLA requirements of different applications. In addition, our scheduling policy is designed to be robust and tolerant towards the incorrect prediction of the forecasting model.

In this paper, the standard Back Propagation (BP) algorithm is used to predict the CPU utilization. The forecasting model predicts one day utilization values of each VMs from one week data from the dataset, with minimum Root Mean Square Error (RMSE) network. Here, the ANN is modeled with one input layer, one output layer and variable hidden layers between the inputs and outputs. As all the nodes at each layer are interconnected by weights, a training algorithm is used to attain a set of weights that minimizes the difference between the predicted value and the actual output through the network. For a given resource usage of an application, the model can be tested for accuracy of prediction by varying the number of hidden layers, training set, and the learning rate. To avoid the overhead of forecasting, this process occur in offline mode.

The neural network considered in this paper, forecasts the CPU utilization with minimum RMSE network [10]. In the experiments[1], a time series vector of CPU utilization for one week is given as an input. The number of hidden layers is varied to tune the performance of the network and through iterations it was found to be optimum at the value of 5 hidden layers. 80% of the data is considered for training and at any point of time 24 periods of CPU utilization for every next 5 minutes is forecasted. The learning rate is found to be optimum at 0.1. The network generates a neural model with minimum MSE (Mean Square Error) and outputs the minimum of RMSE. The 8064 instances of CPU utilization (About one month data, recorded on every five minutes) taken from workload traces is given as the input to the model and the CPU utilization for next one day (288 instances) are predicted.

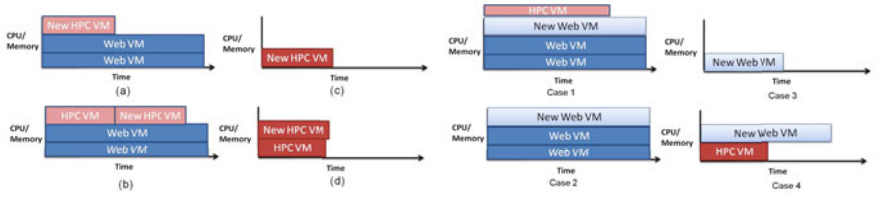
4.2 Admission Control and Scheduling

In this process, the decision on whether an application can be accepted and executed based on the available resources and QoS requirements of the application. To estimate how much resources will be available, we used ANN forecasting model (described in the previous section) which predicts future demand of all

[1] Please see section 5 for the details of web application workload.

accepted applications. If free resources on a host are sufficient to run the application, it will be accepted. The scheduling of application is based on following simple but effective principle: "*In initial allocation, for transactional applications, reserve resources are equivalent to their peak demand. For non-interactive jobs, scheduler tries to allocate slack of resources remaining on a host which is running a transactional application. At regular intervals, consolidate those VMs deployed on hosts which are not running any non-interactive jobs*".

The scheduler always tries to run non-interactive jobs with transactional applications whose resource demand is highly dynamic. This strategy aims to minimize the SLA violations and network overhead of VM migration. It is clear that, this strategy is quite different from general approaches used in previous works where, during migration, multiple types of SLAs are not considered. Each type of application has different resource demand characteristics and different QoS requirements. The details of admission control and scheduling for each of them are described below:



(a) Non-interactive (b) Transactional

Fig. 1. Job Scheduling Scenarios

Non-Interactive Job: The datacenter accepts the request for execution of a non-interactive batch job only when it can be allocated with the required amount of CPU capacity before its deadline. At any moment, when a job arrives for execution, datacenter have some servers running the workload and others which are switched-off to save the power. Thus, the scheduler first checks the resource availability of active servers where a Web application is already hosted. The resource availability on each host are calculated based on the forecasted server demand. To know whether the job can be successfully executed within its deadline, the start and completion time of each jobs is estimated based on resource availability on each host. Figure 1(a) shows the four possible ways a given job can be scheduled. If the job can be completed before the deadline on any host then, the job is accepted by the datacenter for execution, otherwise it will be rejected. SLA is signed between both the user and the provider.

For scheduling, a VM image based on QoS requirement of a given job is created. The VM is deployed on a server (S_j) where the start time (Exp_St) of job is minimum and with an already hosted Web VM. Thus, for execution the new HPC VM, it will utilize the slack resources on the server and such VM is termed as *dynamic HPC VM*. These HPC VMs are allowed to migrate and the resources allocated to them are also dynamic. In this case, if on the given host, there is already one HPC VM is running, then the new VM will be deployed after the completion of the job on the current VM (as demonstrated in Figure 1(a)).

If no such server is available and the threshold time of job is greater than the next scheduling interval, then the deployment of VM is delayed and job is queued up in the batch job Queue. This is done to exploit errors in the resource demand forecast. If actual demand of the resources is less than forecasted one, then some HPC job could finish before their estimated completion time. Thus, new VM can be deployed without switching on to the new servers. If the threshold time of the job is less than the next scheduling interval, then either it is scheduled on a server running only the HPC VMs, or a new server which is switched on. In this case, the VM deployment will be static. Therefore, these VMs are not allowed to migrate, and the resources allocated to them will not be changed till the agreed amount (as in SLA) of CPU cycles are used for its execution. These VMs are called *static HPC VM* indicated by a different color in Figure 1(a). This is done to avoid the effects of rescheduling of VMs on the network and on the other VMs running in the server.

Transactional Applications: A user having a transactional application can ask for VMs of different standard sizes offered by the Cloud provider. Let the user request for a VM with capacity C_k. A request is accepted when the datacenter can schedule the VM with capacity C_k on any server assuming all hosted Web VMs are running at 100% utilization and without considering resources used by dynamic HPC VMs. The Web VM is scheduled based on the best-fit manner. As Figure 1(b) shows, there are four possibilities to deploy the new Web VM.

Algorithm 1. SLA Enforcement and Rescheduling

Input: Current Utilization of VMs and Current Resource Demand
Output: Decision on Capacity Planning and Auto-scaling
Notations: VM_{web-i}: VM running Transactional (Web) Applications; $CurResDemand(VM_{web-i})$: Current Resource Demand; $CurAllocResVM_{web-i}$: Current Allocated Capacity; $ReservedRes(VM_{web-i})$: Reserved VMs Capacity Specified in SLA; VM_{hpc-i}: VM running HPC Application

1: **for** Each VM_{web-i} **do**
2: Calculate the current resource demand $CurResDemand(VM_{web-i})$
3: **if** $CurResDemand(VM_{web-i}) < CurAllocResVM_{web-i}$ **then**
4: Reduce the resource capacity of $VM_{web-i}$to match the demand
5: **else**
6: **if** $CurResDemand(VM_{web-i}) \leq ReservedRes(VM_{web-i})$ **then**
7: Increase the resource capacity of VM_{web-i} to match the demand
8: Reduce correspondingly the resource capacity allocated to HPC application (VM_{hpc-i}) on the same server
9: **else**
10: **if** SLA contains Auto-scaling Option **then**
11: Initiate new VMs and offload the application demand to new VMs
12: **end if**
13: **end if**
14: **end if**
15: **end for**
16: **for** Each Batch Job VM_{hpc-i} **do**
17: **if** slack resources available on the server where HPC VM is running **then**
18: Allocate the slack resources
19: **end if**
20: Recompute the estimated finish time of the job
21: Reschedule the Batch Job VM if missing the deadline.
22: **end for**

Case 1: If new Web VM is deployed on a server hosting both a dynamic HPC VM and Web VMs, then the future resources available to the dynamic HPC VM will get affected. This scarcity of resources will delay the completion time of HPC job. Thus, the HPC VM will be paused and rescheduled (migrated) to other servers if the HPC job is missing its deadline after deployment of new Web VM. The rescheduling of HPC job is done in such a way that the minimum penalty occurs due to SLA violation.

Case 2 - 4: In these cases, since, while scheduling of new Web application, the full utilization of resources by other VMs is considered. Therefore, there will not be any perceptible effect on the execution of other VMs. It can be noted that in Case 4, since static HPC Vm (denoted by red color) is hosted therefore, the available resources on the server for executing new Web application will be the amount of resources unused by HPC VM.

4.3 SLA Enforcement and Rescheduling of VMs

To fulfil SLAs, the regular SLA monitoring of each application is required, since our initial allocation and admission control is based on the forecasting. The forecasting model only gives approximate future resource demand and thus there may be a time when SLAs are violated. The steps involved during SLA enforcement process is given in Algorithm 1. In every scheduling cycle, scheduler will do following functions: a) enforce SLAs and b) schedule the jobs from batchjob Queue, and c) Consolidation. For SLA enforcement, the resource demand for each transactional application till next scheduling cycle is recalculated (Line 1-2). If any Web VM requires less than the forecasted (currently allocated capacity) then the extra resources are allocated to HPC VM running on the same host (Line 3-5 and 20-21). Otherwise, if the Web VM requires more resources than allocated ($\leq$ promised capacity in SLA), then the resources allocated to the VM are increased to match the demand (Line 6-7). Correspondingly, the resources allocated the HPC VM will be reduced (Line 8). If the Web VM requires resource capacity more than specified in SLA, then the decision is taken based on whether the user has opted for auto-scaling or not (Line 10-15). This process is repeated for each transactional application. After that, for each HPC job, their VM capacity is increased if some slack resources is available on the server where the VM is hosted (Line 17-18). Based on allocated resources to the HPC VM, the job completion time is recomputed (Line 20). If any HPC job which is currently running is expected to miss deadline, then its corresponding VM is migrated and scheduled on another server using strategies discussed in previous section (Line 21). Similar process is repeated for each HPC job (VM) in batch job queue. The scheduler consolidates Web VMs which are running on servers having no HPC VM. If due to consolidation, some Web VMs may be short of resources, in that case, SLA can be violated.

5 Performance Evaluation

We have simulated a datacenter that comprises of 1500 heterogeneous physical nodes. Simulation approach gives advantage of repeating the experiments under similar environment. Thus, it allows the comparison of different scheduling strategies. In addition, it is pretty difficult to gain workload for many real applications as they are considered in this work. Each node is modelled to have one CPU core with performance equivalent to 4000 Million Instructions Per Second (MIPS), 16 GB of RAM, 10 GB/s network bandwidth and 1 TB of storage. We have considered for different types of VMs with 1000, 2000, 2500 or 3500 MIPS. The smallest instance will be allocated 1 GB of RAM, 100 Mb/s network bandwidth and 1 GB of storage. The CPU MIPS ratings are similar to different Amazon EC2 instance sizes. The users submit requests for provisioning of 500 heterogeneous VMs. Each VM is randomly assigned a workload trace from one of the servers from the workload data as described in the following section. The pricing for each VM instance is taken same as used by Amazon EC2 for different sizes of VM. Even though only four types of VMs are considered, our model can be easily extended for other types of VM instances.

5.1 Workload Data

For our experiments, we have used two different workloads data, each for transactional and non-interactive applications. For transactional data, data is collected from CoMon [11], a monitoring infrastructure for PlanetLab (http: //comon.cs.princeton.edu). The data contains the CPU utilization, memory and bandwidth usage of more than thousand servers located at about 500 places around the world. The data has been collected for each five minutes during the period from the 22nd to 29th of July 2010. The data is interpolated to generate CPU utilization for every second. The data satisfy our requirement of transactional application and thus have some good number of peak utilization levels and very low off-peak utilization level: the average CPU utilization is below 50%. For non-interactive batch jobs, the LCG workload traces from Grid Workload Archive (GWA) [7] are used. Since this paper focuses on studying the Cloud users with non-interactive applications, the GWA meets our objective by providing workload traces that reflect the characteristics of real applications running on one VM. From this trace, we obtain the submit time, requested number of VMs, and actual runtime of applications. Since workload traces do not contain any data on deadline and penalty rates specified in SLAs, for our evaluation, these are generated using uniform distribution. The deadline of a non-interactive application i is given by: $SubTime(i) + ExeTime(i) + ExeTime(i) * \lambda$, where $SubTime$ is submit time and ExeTime is execution time. λ is urgency factor derived from uniformly distribution(0,2).

5.2 Performance Metrics

The simulations have been run for 10 hours of each workload category to determine the resource provisioning policies that delivers the best utilization, the

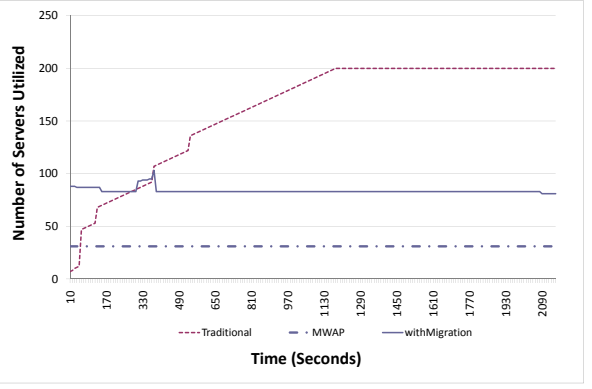


Fig. 2. Effect on Datacenter Utilization

least number of SLA violation and VM migrations, and accepted maximum user requests. Two metrics are used to compare the policies: number of hosts utilized and revenue generated. We have compared our resource provisioning policy MWAP (Mixed Workload Aware Policy) against two other following well known strategies used in current datacenters:

1. Traditional Approach (*aka* Traditional): In this approach, during the whole VMs life cycle, an application will be allocated the capacity of server as specified in SLA. Thus, VM allocation will not change with time.
2. VM Migration and Consolidation Approach (*aka* withMigration): This strategy is used in various papers to address the problem of maximizing the utilization of datacenters [2]. In this approach, many VMs are consolidated based on one server and their usage. If an application demands more server capacity, either it is migrated or capacity of server assigned to the VM running the application is increased. Since VMMigration does not consider the SLA requirements of non-interactive applications, for fair comparison the VM capacity allocated to each such applications does not change with time. This reduces the chance of batch application to miss its deadline.

6 Analysis of Results

Although several experiments are conducted by varying different parameters, due to space constraint, in this section, we discuss only the key results of our evaluation. All the results are summarized in Figure 2, Table 1 and 2.

Effect on Datacenter Utilization: Since our main objective is to maximize the utilization of datacenter, first we compare all the techniques based on their effectiveness in maximizing the datacenter utilization. The datacenter utilization is indicated by the number of host which are used for a given workload. Thus, Figure 2, shows how the number of hosts utilized is varied with time to meet the SLA requirements of applications and complete them successfully. It can be noticed that number of VMs utilized by MWAP policy remains constant with time

and utilized about 60% less number of servers on average. The reason for such a large difference is that MWAP tries to run VMs of batch jobs by using unutilized resources of VMs running Web applications. With more batch job submissions, the number of servers used by the traditional approach has increased from almost zero to 200. This is due to static allocation of server capacity to VMs based on SLA requirements. In this case, with the time, number of active servers become almost constant since enough servers are available to meet the resource demand of incoming non-transactional workload. In the case of withMigration resource provisioning policy, there is an increase in the number of servers between T=300 to T=400 due to high volume of batch job submissions. The later due to consolidation, withMigration policy reduces the number of server utilized to about 83. This clearly shows the importance of considering resource usage pattern of different type of applications, which can result in efficient datacenter utilization. Thus, datacenter can simultaneously serve more users with same server capacity.

Table 1. Effect of SLA consideration on provider and user Parameters

Policy	Revenue (Batch Jobs)	Revenue Transactional	VM Migrations	SLA Violation (Transactional)	Batch Job Completed
Traditional	481848	647700	0	0	285
withMigration	31605	623370	267	26	160
MWAP	475732.8	647700	69	0	284

Table 2. Effect of different type of SLA penalties

Penalty Rate	Fixed		Delay Dependent		Proportional	
	(Penalty$)	SLA Violation	Penalty($)	SLA Violation	Penalty($)	SLA Violation
Low	87.78063157	6	146.3010526	10	292.6021052	10
Medium	219.4515789	6	365.7526316	10	731.5052631	10
High	438.9031579	6	731.5052631	10	1463.010526	10

Effect on Revenue Generated and SLA Violation: In general, the most important thing for a Cloud provider is, the profit generated by serving the VM request of users. Secondly, the Cloud provider wants is to satisfy as many users as possible by meeting their SLA requirements. Table 1 gives the details of revenue generated for each type of applications i.e., batch jobs and transactional applications. The revenue generated from Traditional policy and the proposed policy MWAP are similar because of zero number of violations both for transactional and batch jobs. While, withMigration policy results in about 26 SLA violation due to the migration delays which results in reduction of revenue. The withMigration policy also results in very low batch job revenue. The reason behind this is migration delays which results in SLA penalty. Therefore, the consideration of SLA penalty with VM migration and consolidations plays an important role in dynamic resource provisioning, otherwise Cloud provider will incur huge revenue loss.

Migration overhead and Batch Job Completion: It is well known that VM migration is not free and it always incur some network and CPU overhead. In this section, we will show the number of migrations MWAP required to perform in order to meet the SLA requirements in comparison to withMigration approach.

It can be clearly noticed from Table 1, the huge reduction in VM migration by MWAP policy which results in almost 75% less number of migrations. The with-Migration policy tries to optimize the utilization by migrating and consolidating the underutilized VMs which results in very high number of migrations. The migration overhead causes unnecessary delays in batch job execution which results in almost 45% (Table 1 : Batch Job Completed) of successful completions before deadline. This problem can further increase if the number of VMs initiated is not constant, which is accounted in our MWAP by using intelligent admission control policy.

Effect of SLAs Types: In this section, we present the further results on the importance of considering different types of SLA penalties (requirements) along with dynamic provisioning of VMs within a Cloud datacenter. Since, there is no SLA violations noticed in the case of MWAP, we have conducted the experiments using withMigration policy to understand the role of different types of SLA penalties (fixed, delay dependent and proportional) in resource allocation. For each application, Table 2 summarizes the results with variation of penalty rate (q) from low to high. The proportional penalty incur almost 50% more in comparison to other penalties. As the penalty rate is varied, the total penalty incurred becomes more and more prominent. Since, in withMigration policy, there is no consideration of type of SLA penalties, as it results in more number of SLAs with delay-dependent and proportional penalty, and this further enhances the penalty. Thus, while doing resource allocation, the provisioning policy should take into account these penalty types and give priority to the applications with low penalty rates.

7 Conclusions and Future Directions

In this work, we have proposed a novel technique that maximizes the utilization of datacenter and allows the execution of heterogenous application workloads, particularly, transactional and non-interactive jobs, with different SLA requirements. Our approach dynamically assigns VMs in such a way that SLA signed with customer can be met without any penalty. The paper, also describes how the proposed technique can easily integrate with the admission control and facilities like auto-scaling offered by the Cloud providers. By extensive performance evaluation, it is demonstrated that, the proposed mechanism MWAP reduces number of servers utilized by 60% over other strategies like consolidation and migration with negligible SLA violation. Our proposed mechanism MWAP performs reasonably well and is easily implementable in a real Cloud Computing environment.

Therefore, we demonstrate that for designing more effective dynamic resource provisioning mechanisms, it is important to consider different types of SLAs along with their penalties and the mix of workloads for better resource provisioning and utilization of datacenters; otherwise it will not only incur unnecessary penalty to Cloud provider but can also lead to under utilization of resources. This motivates further enquiry into exploration of optimizing the resource provisioning techniques by extending our approach to other type of workloads such

as workflows and parallel applications. In future, we also plan to extend our model by considering the multi-core CPU architectures as well as network and memory conflicts.

References

1. Azoff, E.: Neural network time series forecasting of financial markets. John Wiley & Sons, Inc., New York (1994)
2. Beloglazov, et al.: A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems. In: Zelkowitz, M. (ed.) Advances in Computers. Elsevier, Amsterdam (2011) ISBN 13: 978-0-12-012141-0
3. Buyya, et al.: Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. FGCS 25(6), 599–616 (2009)
4. Carrera, D., Steinder, M., Whalley, I., Torres, J., Ayguadé, E.: Enabling resource sharing between transactional and batch workloads using dynamic application placement. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 203–222. Springer, Heidelberg (2008)
5. Dodonov, E., de Mello, R.: A novel approach for distributed application scheduling based on prediction of communication events. FGCS 26(5), 740–752 (2010)
6. Iosup, A., Epema, D.: Grid computing workloads: Bags of tasks, workflows, pilots, and others. IEEE Internet Computing 99(PrePrints) (2010)
7. Iosup, et al.: The grid workloads archive. FGCS 24(7), 672–686 (2008)
8. Kim, J.K., Siegel, H.J., Maciejewski, A.A., Eigenmann, R.: Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling. IEEE Trans. Parallel Distrib. Syst. 19(11), 1445–1457 (2008)
9. Meng, et al.: Efficient resource provisioning in compute clouds via vm multiplexing. In: Proc. of the 7th Intl. Conf. on Auton. Comp., Washington, DC, USA (2010)
10. Mohammadi, S., Abbasi-Nejad, H.: Forecasting With Matlab. 129.3.20.41/eps/prog/papers/0505/0505001.pdf (2005)
11. Park, K., Pai, V.: CoMon: a mostly-scalable monitoring system for PlanetLab. ACM SIGOPS Operating Systems Review 40(1), 65–74 (2006)
12. Quiroz, et al.: Towards autonomic workload provisioning for enterprise grids and clouds. In: Proc. of 10th IEEE/ACM Intl. Conf. on Grid Comp., USA (2009)
13. Singh, et al.: Autonomic mix-aware provisioning for non-stationary data center workloads. In: Proc. of the 7th Intl. Conf. on Autc. Comp., USA (2010)
14. Smith, et al.: Secure on-demand grid computing. FGCS 25(3), 315–325 (2009)
15. Sotomayor, et al.: Combining batch execution and leasing using virtual machines. In: Proc. of the 17th Intl. Sym. on HPDC, Boston, MA, USA (2008)
16. Soundararajan, et al.: The impact of mngt. operations on the virtualized datacenter. In: Proc. of the 37th Ann. Intl. Sym. on Comp. Arch., France (2010)
17. Srinivasa, et al.: An efficient fuzzy based neuro-genetic algorithm for stock market prediction. Intl. Jnl. of Hyb. Intelligent Sys. 3(2), 63–81 (2006)
18. Wang, et al.: Capacity and performance overhead in dynamic resource allocation to virtual containers. In: Proc. of the 10th IFIP/IEEE Intl. Symp. on Intgd. Net. Mangt., Munich, Germany (2007)
19. Yeo, C., Buyya, R.: Service Level Agreement based Alloc. of Cluster Resources: Handling Penalty to Enhance Utility. In: Proc. of the 7th IEEE Intl. Conf. on Cluster Comp., Boston, USA (2005)
20. Zhang, et al.: Agile resource management in a virtualized data center. In: Proc. of Ist Joint WOSP/SIPEW Intl. Conf. on Perf. Eng., California, USA (2010)

ΣC: A Programming Model and Language for Embedded Manycores

Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David

CEA, LIST, Embedded Real-Time Systems Lab
Mail Box 94, F-91191 Gif-sur-Yvette Cedex, France
name.surname@cea.fr

Abstract. We present ΣC, a programming model and language for high performance embedded manycores. The programming model is based on process networks with non determinism extensions and process behavior specifications. The language itself extends C, with parallelism, composition and process abstractions. It is intended to support architecture independent, high-level parallel programming on embedded manycores, and allows for both low execution overhead and strong execution guarantees. ΣC is being developed as part of an industry-grade tool chain for a high performance embedded manycore architecture.

Keywords: programming model, programming language, embedded manycores, embedded high performance computing.

1 Introduction

Programming embedded systems is a difficult task and so is parallel programming. Embedded manycores, that is systems-on-chip with over a hundred general purpose cores, are full scale parallel machines, typically employing a mix of shared and local memory, distributed global memory or multilevel cache hierarchy, and a network on chip (NoC) to enable communication between cores. Compared to their full scale brethren, they provide a limited amount of memory per core, no guarantee on memory coherency and are subject to strict dependability and performance constraints (e.g., guaranteed performance at peak utilization or close to peak).

As a consequence, developing for those targets suppose handling simultaneously the following three difficulties: meeting performance and dependability requirements subject to limited resources, running correctly large parallel programs, as well as exploiting efficiently the underlying parallel architectures. To render this task manageable and cost-effective, we have identified the following set of requirements for a programming model and language suitable for manycores:

- Ability to handle a variety of algorithms, both data flow (streaming) and control oriented at least from the fields of signal and image processing.
- Familiarity to embedded developers, that is similarity to C and ability to integrate efficiently with existing C code.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 385–394, 2011.

– A compiler able to prove that the final executable is guaranteed to execute in bounded memory and that any run is reproducible.
– A development tool chain able to support good programming practices, modularity, encapsulation and code reuse.
– A run time and support microkernel fit for embedded manycores.

The main result described in this paper is a well defined model of computation, and the ΣC programming language which implements it. The paper is organized as follows. We first position our model with respect to the embedded models of computation and parallelism. We then detail the programming model, the language and the compilation process, before a short overview of the techniques used to guarantee execution in bounded memory, as well as the points for compilation optimization as currently implemented.

2 Related Work

ΣC as a programming model takes place among the numerous works done in the field of embedded models of computation[1]. The programming model belongs to the class of Process Networks (PN), or Kahn process networks (KPN)[2]. One of the main trade-off made during the design of ΣC, in this space, is the amenability to formal analysis, the field being split between models with restricted expressive power for which interesting properties such as deadlock freeness and bounded memory are decidable (SDF or Synchronous DataFlow[3], CSDF or Cyclo-Static DataFlow[4], HDF or Heterochronous DataFlow[5]) and models with increased expressive power such as the aforementioned process networks, Boolean DataFlow (BDF) and others, which come at the cost of Turing-completeness. The ΣC programming model trade-off consist in increasing the expressive power above SDF and CSDF while being less general than non deterministic process networks or BDF; in that following a similar constructive approach as [6].

The ΣC programming model is a subset of a process network model of computation with non deterministic extensions, of sufficient expressive power for most applications, while maintaining the possibility of tractably performing a formal analysis, for example using the well-known VASS or Petri net formalisms[7].

As a programming language, ΣC relates to StreamIt[8], Brook[9], XC[10], and more recently OpenCL[11], all programming languages, either new or extensions to existing programming languages, able to describe parallel programs in a stream oriented model of computation (CSP for XC).

ΣC differs by extending C without restrictions on the type of C supported or changes in the semantics of C apart from the extensions; the ΣC compiler parses standard C and support computational kernels of any level of complexity and call depth. It is high-level: no mention of the memory hierarchy or chip layout is necessary in the source code. It supports proving guaranteed execution in bounded memory over non-deterministic process network topologies. It supports stand-alone systems, with no host and no operating system, on an embedded manycore target with tens of Kbytes of RAM per core and over a thousand cores.

The implementation as a programming language extension provides strong type checking, scoping, modularity and correctness at the source code level.

3 The ΣC Programming Model

3.1 Components

The basic unit of the programming model is called an agent. It is an independent process, with one thread of execution and its own address space. It communicates through point to point, unidirectional, typed links behaving as fifos.

Communication through links is done in blocking read, non-blocking write fashion, and the link buffers are considered large enough. Agents have an interface, that is a list of typed and oriented ports to which links may connect, and a behavior specification.

An application, for example in figure 1, is a graph of interconnected agents, with an entry point, the `root` agent. The graph is static, and does not evolve through time (no agent creation or destruction, no change to the topology, during the execution of an application).

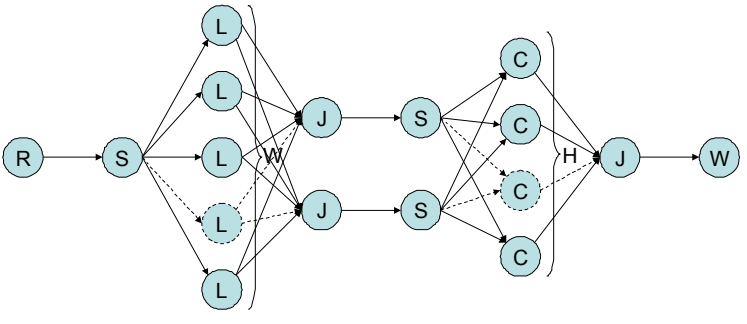


Fig. 1. Process network for a Laplacian computation (Huertas-Médioni operator), showing line (L) and column (C) filter agents

System agents ensure distribution of data and control, as well as interactions with external devices. Data distribution agents are Split, Join (distribute or merge data in round robin fashion over respectively their output ports / their input ports), Dup (duplicate input data over all output ports) and Sink (consume all data).

3.2 Behavior

The communication behavior of each agent is specified. It is a cyclic state machine with variable amounts of data. Each transition specifies a fixed or variable amount of data consumed on in ports, and a fixed or variable amount of data produced on out ports of this agent. All variable amounts are bounded ranges.

An example is a run length decoding agent with input and output ports of type unsigned char, described by : `{{input[2]}; {output[1:256]}}`. This specification means that this agent consumes two tokens on its input in one transition, and then produces 1 to 256 tokens on its output in the second transition, before looping back.

Data-dependent control in the process graph is introduced through `Select` and `Merge` agents which take, in addition to their data links, a control link selecting which input or output link is activated. `Select` and `Merge` are evaluated as simulation of execution of the branch not taken (and execution of the branch taken), so as to ensure correct execution of agents connected to the branch. In this semantic, a branch not taken is seen as being executed for its environment.

3.3 Designing for Execution Guarantees

As already emphasized, the ΣC programming model has been designed with amenability to formal analysis in mind, in particular with respect to properties such as absence of deadlock and memory bounded execution. In essence, formal analysis is performed on the basis of both the topological and behavioral specifications provided by the programmer.

In particular, being a special case of KPN, the ΣC programming model inherits their nice properties with respect to determinism and monotony. This has interesting consequences on the problem of safely dimensioning the statically dimensioned communication buffers: as long as one finds a deadlock-free buffer size solution for all links (preferably small with respect to an adequate objective function), any other solution with non-smaller buffers is also deadlock-free. This thus allows one to decouple the problems of proving deadlock-freeness and of tuning the buffer dimensioning so as to achieve a high level of performance.

At the programming language level (see below), the programming model appears hierarchical, with behavior specifications and interfaces. This aspect is designed to support hierarchical, divide-and-conquer analysis of minimum buffer bounds and deadlock-free solutions.

4 The ΣC Programming Language

The ΣC programming language is designed as an extension to C. It adds to C the ability to express and instantiate agents, links, behavior specifications, communication specifications and an API for topology building, but does not add communication primitives. It defines a component model with composition [12], and enforces strict encapsulation and type checking on components interfaces.

The ΣC language has been designed to allow two levels of execution: off-line and on-line. The static topology of a ΣC application is handled through instantiation of components and topology building at compile-time, by executing off-line parts of the application source code dedicated to that effect. The on-line level is the application execution on the target hardware.

Listing 1. A Line filter prototype

```
1 agent LineFilter(int width) {
2   interface {
3     in<int> in1;
4     out<int> out1, out2;
5     spec {in1[width]; out1[width]; out2[width]};
6   }
7 }
```

4.1 Components

The basic entity in a ΣC program is an agent. It abstracts a programming model process and so corresponds to an execution unit, with its own address space and a single thread of execution. It has an interface, describing its communication ports (direction and type) and the specification of its behavior in the model described above. It is written as a C scoping bloc with an identifier and parameters, containing C unit level terms (functions and declarations), ΣC-tagged sections: `init`, `map`, and `exchange` functions.

The `interface` section contains the communication ports description and the behavior specification. Each port is a direction (`in`, `out` or `inout`), a type (any C except pointers and functions) and an identifier. Ports may be provided with a default value (if `out` or `inout`) and a sliding window size (if `in`). Arrays of ports can be expressed. The behavior specification is a `spec` espression describing the behavior as shown on line 5 of listing 1.

The `map` section contains component instantiation, topology building and initialization code for the agent. As a ΣC agent in the language is an abstraction, it has to be instantiated to be part of an application. This is done off-line by executing the `map` section of agents. Each agent is then responsible for its initialization, the instantiation of the agents it contains (an agent is a composite) and the topology it encapsulates. Two extensions to C are introduced here: a `new` keyword for instantiating an agent, and template-like type parametric instantiation for system agents. A topology building API is used here, with two functions : `connect` to connect two ports, and `preload` to preload data in a link. For an example, see listing 2 for the map code used to build the network of figure 1. All assignments done to agent state variables in the `map` section are saved and integrated in the final executable, allowing for off-line complex initialization sequences on a per-instance basis.

The ΣC language enforces strict encapsulation: the internals of an agent, and all its contents (contained agents, links, local ports, etc...) cannot be accessed from outside that agent.

`Exchange` functions implement the communicating behavior of the agent. An `exchange` function is a C function with an additional `exchange` keyword, followed by a list of parameter declarations enclosed by parenthesis (see line 8 of listing 3). In the parameter declaration, the type is the name of a port and the

Listing 2. Topology building code

```
1  subgraph root(int width, int height)
2  {
3    interface { spec {}; }
4    map
5    {
6      ...
7      agent output = new StreamWriter<int>(ADDROUT, width*height);
8      agent sy1 = new Split<int>(width, 1);
9      agent sy2 = new Split<int>(width, 1);
10     agent jf = new Join<int>(width, 1);
11     ...
12     connect(jf.output, output.input);
13     ...
14     for(i=0; i<width; i++) {
15       agent cf = new ColumnFilter(height);
16       connect(sy1.output[i], cf.in1);
17       connect(sy2.output[i], cf.in2);
18       connect(cf.out1, jf.input[i]);
19     }
20   }
21 }
```

declarator creates an `exchange` variable of the type of that port. They can be used in the code in exactly the same way as function parameters, the direction of the port (in, out or inout) indicating whether the variable resolves to an input or an output buffer. An `exchange` function call is exactly like a standard C function call, the `exchange` parameters being hidden to the caller. `Exchange` variables can be used as parameters to C function calls without overhead or hidden data copy in most cases. Listing 3 is an example of an agent implementation.

The agent behavior is implemented as in C, as an entry function named `start()`, which is able to call other functions as it sees fit, functions which may be `exchange` functions or not. No communication primitives are available or visible at the function or `exchange` function level, and it supports `exchange` functions calling `exchange` functions with, however, possible performance effects.

Subgraphs are similar to agents, except that a `subgraph` implements only composition, without behavior. As such, a `subgraph` has only an `interface` and a `map` scope, and subgraph ports have a slightly different meaning: they are aliases for the ports of internal agents or subgraphs instances.

4.2 System Agents

System agents are special agents implementing data distribution and synchronization, and making it available to the compilation tools for transformation and optimisation purposes. They are handled through stream-like agents: `Split`, `Dup`, `Join`, `Select`, `Merge` and `Sink`. Those agents are type generic and take a C type in parameter when instantiated.

Listing 3. The Column Filter agent used in figure 1

```
1  agent ColumnFilter(int height) {
2    interface {
3      in<int> in1, in2;
4      out<int> out1;
5      spec {in1[height]; in2[height]; out1[height]};
6    }
7
8    void start() exchange (in1 a[height], in2 b[height], out1 c[height]) {
9        static const int
10           g1[11] = { -1, -6, -17, -17, 18, 46, 18, -17, -17, -6, -1},
11           g2[11] = {0, 1, 5, 17, 36, 46, 36, 17, 5, 1, 0};
12       int i, j;
13       for(i=0;i<height;i++) {
14         c[i] = 0;
15         if(i < height - 11)
16           for(j=0; j < 11; j++) {
17             c[i] += g2[j] * a[i+j];
18             c[i] += g1[j] * b[i+j];
19           }
20       }
21    }
22 }
```

4.3 Input / Output

Input / Ouput is handled with a special class of agents, identified by the keyword `ioagent`, and support type parametrization with a C++ template like syntax. They provide a way to write agents handling low level system and input / output tasks as well as drivers or protocol stacks, but interfacing with the ΣC programming model. The tool chain is then open to the possibility of targeting those agents on dedicated hardware such as DMA engines, IO Processors or different execution environments.

A sample of such ioagents are implemented to access external memory: those are the `StreamReader`, `MemReader`, `StreamWriter` and `MemWriter` ioagents, with variants allowing for synchronization of memory transfers. `Timer` ioagents produce a stream of timed events, allowing for time-based synchronization.

4.4 Software Architecture

Agents and subgraphs can represent any granularity: large processes, thread-size entities, fine grain parallelism (SIMD-like). It is designed so that the port of a sequential C program to ΣC may be done by making it a single agent, and to progressively turn it into a massively parallel implementation by repeated decomposition in smaller agents and subgraphs.

ΣC supports libraries of ΣC components and the design of carefully crafted and optimized libraries of algorithms abstracted behind a generic interface, or parallelism patterns.

Standard C code, even non-reentrant, may be reused and compiled as ΣC code. C pointer equivalence in `exchange` functions allow for passing pointers in standard function calls with no loss of performance or generality.

A standard C back-end is needed for completion of the compilation process; as such, ΣC is compatible with vector extensions, attributes, pragmas, inline asm, automatic vectorisation and other specifics of the target ISA and backend compiler.

5 A Sketch of the ΣC Compilation Process

The ΣC compiler chain is architectured around four passes. The first pass, the ΣC front-end, performs a lexical, syntactic and semantic analysis of the ΣC code, and generates preliminary C code for either off-line execution or further refinement by substitution.

The second pass, the ΣC middle-end, deals with agent instantiation and connection, by compiling and executing the codes generated to that end by the front-end. Once the application graph is complete, a number of parallelism reduction heuristics are applied so as to tailor the application to an abstract specification of the platform resource capacities. Most system agents are combined and transformed into shared memory buffers or NoC transfers so as to fit the system communication and memory architecture. The second pass subsequently computes a deadlock-free lowest bound of the buffers size for all links (see [13]).

The third pass performs resource allocation at the system level. This encompasses computing buffer sizes and construction of a folded (hence finitely representable) unbounded partial ordering of tasks occurrences (see [14]) as well as allocation of tasks to computing resources (cores, clusters, etc.) and NoC configuration. Resource allocation can be performed in a feedback-directed fashion so as to achieve an appropriate level of performance.

The last pass, called the ΣC back-end, is in charge of generating the final C code as well as the runtime tables which, based on the partial orderings built by the third pass, make the link with the target execution model. Using the C back-end tools, the ΣC back-end is also in charge of link edition as well as loadbuild.

Optimization Points

ΣC is designed with the following three goals when it comes to optimization. First, rely on a proven, portable, efficient backend compilation toolchain, in practice, a C compiler and associated tools (linker, assembler).

Secondly, at the process network level, optimize through buffer fusion, rewriting cascades of Split and Dup as patterned access to data, specifically if the

architecture has DMA engines. Another level of optimization is adjusting by reduction of the degree of parallelism of the process network graph, by detecting and replacing specific topologies.

The third design goal for optimization was that performance tuning of performance sensitive code on embedded devices is possible in a pragmatic way: ΣC is sufficiently flexible to allow developers to express parametric parallel code, where instances execution cycles, buffer sizes and degree of parallelism can be adjusted with instance parameters, allowing developers to adjust the shape of the process network to a better match for the target architecture.

6 Evaluation

Two teams, one internal to our lab, one within our industrial partner, are collaboratively stress test the language and the model on a first (but wide ranging) round of applications optimized for the target platform amongst which multi camera target tracking applications, augmented reality video processing, H264 video encoding, and 4G/LTE channel coding implementations. Our current results indicate that the level of expressiveness chosen has proven itself so far appropriate, that is target applications have been designed without encountering algorithmic constructions that are either clumsy or (worse) impossible to express.

Potential efficiency concerns are regularly expressed and handled in the course of the language evolution, without major changes so far. The informal usability testing underway has shown that the component model and process model expressed in ΣC has not been considered a barrier and that developers with a background in C or SystemC have few difficulties to adapt to it.

Furthermore, the model has proven interesting for designing parallel solution algorithms to some operational research problems, so we may have a possibility to retarget the implementation on larger scale, non embedded, parallel systems (or a large collection of high performance embedded manycores). It has also proved suitable as a back-end language for higher-level stream processing languages such as [15], and may be used as a target for source code automatic parallelisation tools such as Par4All/PIPS[16].

7 Conclusion

We have presented a well formed programming model and ΣC, a programming language implementing it. This programming model and language have so far a result on two of our criteria: ability to express a variety of algorithms, and familiarity to embedded developpers through C compatibility. It has now evolved through a few iterations, mostly removing unneeded features and adding target integration, and can now be seen as a stable foundation.

For the remaining three criteria set forth in the introduction, we have shown that the programming model support formal analysis and computation of a bounded memory schedule. Implementation details such as the ability to do

in-place data modifications and buffer sharing allow for a strict, memory constrained implementation with a dedicated micro-kernel. And the programming language support type checking, components and reusability.

This language and its compilation tool chain is being industrialized as part of the technology offering for a many-core architecture jointly developed with one of our semi-conductor partners.

References

1. Jantsch, A., Sander, I.: Models of computation and languages for embedded system design. IEE Proc.-Comput. Digit. Tech. 152(2), 114–129 (2005)
2. Kahn, G.: The Semantics of Simple Language for Parallel Programming. In: IFIP Congress, pp. 471–475 (1974)
3. Lee, E., Messerschmitt, D.: Synchronous Data Flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
4. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cycle-Static Dataflow. IEEE Trans. on Signal Processing 44(2), 397–408 (1996)
5. Girault, A., Lee, B., Lee, E.A.: Hierarchical Finite State Machines with Multiple Concurrency Models. IEEE Trans. on Computer-aided Design of IC & S 18(6), 742–760 (1999)
6. Gao, G.R., Govindarajan, R., Panangaden, P.: Well-behaved dataflow programs for DSP computation. In: IEEE ICASSP 1992, pp. 561–564 (March 1992)
7. Reutenauer, C.: Aspects Mathématiques des Réseaux de Petri, Dunod (1997)
8. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: Proceedings of CC 2002, Grenoble, France, pp. 179–196 (2002)
9. Buck, I.: Brook Specification v0.2. (2003), http://merrimac.stanford.edu/brook
10. Watt, D.: Programming XC on XMOS Devices. XMOS (2009)
11. Khronos OpenCL Working Group: The OpenCL Specification v1.1 (2011)
12. Lau, K., Wang, Z.: Software Component Models. IEEE Trans. on Software Engineering 33(10), 709–724 (2007)
13. Sirdey, R., Aubry, P.: A Linear Programming Approach to General Dataflow Process Network Verification and Dimensionning. Electr. Proceedings in Theorical Computer Science (to appear)
14. Sirdey, R., David, V.: Système d'ordonnancement de l'exécution de taches cadencé par un temps logique vectoriel. Patent pending, filling no 1003963 (2010)
15. De Oliveira Castro, P., Louise, S., Barthou, D.: A Multidimensional Array Slicing DSL for Stream Programming. In: Proceedings of CISIS 2010, pp. 913–918 (2010)
16. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In: Proceedings of ICS 1991, pp. 244–251 (1991)

Provisioning Spot Market Cloud Resources to Create Cost-Effective Virtual Clusters

William Voorsluys, Saurabh Kumar Garg, and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{williamv,sgarg,raj}@csse.unimelb.edu.au
http://www.cloudbus.org

Abstract. Infrastructure-as-a-Service providers are offering their unused resources in the form of variable-priced virtual machines (VMs), known as "spot instances", at prices significantly lower than their standard fixed-priced resources. To lease spot instances, users specify a maximum price they are willing to pay per hour and VMs will run only when the current price is lower than the user's bid. This paper proposes a resource allocation policy that addresses the problem of running deadline-constrained compute-intensive jobs on a pool of composed solely of spot instances, while exploiting variations in price and performance to run applications in a fast and economical way. Our policy relies on job runtime estimations to decide what are the best types of VMs to run each job and when jobs should run. Several estimation methods are evaluated and compared, using trace-based simulations, which take real price variation traces obtained from Amazon Web Services as input, as well as an application trace from the Parallel Workload Archive. Results demonstrate the effectiveness of running computational jobs on spot instances, at a fraction (up to 60% lower) of the price that would normally cost on fixed priced resources.

Keywords: spot market, virtual clusters, cloud computing.

1 Introduction

The introduction of affordable cloud computing infrastructure has had a major impact in the business IT community. These resources are also being explored as a means of accomplishing high performance processing tasks, often present in areas such as science and finance. However, the use of virtualization and network shaping have been cited as factors that hinder the viability of these resources for running compute intensive applications, as opposed to using a dedicated HPC cluster [1]. Nonetheless, the potential cost savings offered by clouds has led to an increased adoption of cloud-based virtual clusters, as well as to the practice of extending the capacity of local clusters using cloud resources in situations of peak demand [2].

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 395–408, 2011.

The cost to build these virtual clusters highly depends on the type of leased virtual machines, which are offered via various pricing schemes through separate "markets", most notably: the on-demand market, which offers standard VMs at a fixed cost; and the spot market, which offers unused capacity in the form of variable price VMs. For example, Amazon EC2 offers biddable virtual machines (VMs), also known as "spot instances", for as low as $\frac{1}{3}$ of the standard fixed price for a similar instance. In this fashion, users submit a request that specifies a maximum price (bid) they are willing to pay per hour and instances associated to that request will run for as long as the current spot price is lower than the specified bid. Prices vary frequently, based on supply and demand.

In spite of the apparent economical advantage, an intermittent nature is inherent to biddable resources, which may cause VM unavailability. When an out-of-bid situation occurs, i.e. the current spot price goes above the user's maximum bid, spot instances are terminated by the provider without prior notice. However, this situation can be avoided by bidding slightly higher, thus mitigating this uncertainty, or by using fault-tolerance techniques such as checkpointing [3].

Virtual clusters can be heterogeneous, when different types of VMs (e.g. with distinct number of CPU cores) are leased and added to the resource pool. In this case, the ratio between price and performance of different types of spot instances may not be constant over time, creating opportunities for optimizations. For example, one could decide how to run a certain compute intensive job by observing the performance per dollar ratio of two high-CPU EC2 spot instances. The "c1.medium" instance type has a CPU power of 5 ECUs and the "c1.xlarge" type has a power of 20 ECUs. One ECU is defined as equivalent to the power of a 1.0-1.2 GHz 2007 AMD Opteron or 2007 Intel Xeon processor. As the spot price of each instance type varies, the performance per dollar ratio offered by each instance type varies accordingly so that, at different periods of time, one instance type may offer a better ratio than the other. In these situations, if an application that would normally run using 5 ECUs could provide enough parallelism, it could run significantly faster on the relatively cheaper 20 ECU instance. This approach offers extra flexibility to users since they may choose to assemble a pool of VMs by bidding on resource types that are currently at a discounted hourly price and then adapt their jobs to run more efficiently on the new resources. Figure 1 depicts an example scenario of such a virtual cluster composed of virtual machines of different sizes.

This paper focuses on reducing the costs of building virtual clusters by leasing spot market resources. Specifically, we propose a **resource provisioning and job scheduling strategy that addresses the problem of dynamically building a virtual cluster out of low-cost VMs and utilizing them to run compute-intensive applications**. Our main objective is to exploit variations in price and performance of resources to run applications in a fast and economical way. Moreover, applications are assumed to be deadline-constrained. For this reason, our strategy is augmented by a job runtime estimation mechanism that aids in deciding how to allocate jobs in a way that they finish within their deadlines.

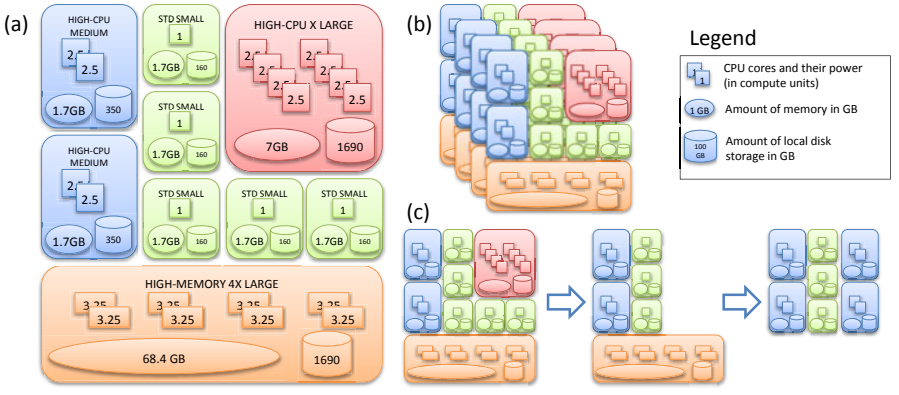


Fig. 1. A virtual cluster is dynamically assembled out of inexpensive cloud-based resources, e.g. Amazon EC2 spot instances. (a) VMs of appropriate sizes have to be chosen, depending on the immediate requirements of running applications; (b) the cluster is able to scale to much larger capacities; (c) VMs are replaced by different types as application requirements change.

Our results show that it is possible to run a stream of computational jobs by solely utilizing spot instances. Especially, we have quantified the effect of the accuracy of runtime estimation on the monetary cost. We have found out that less accurate estimations usually lead to higher costs (in the case of over estimations) or more missed deadlines (in the case of under estimations).

The Contributions of This Work Are:

- A novel system architecture that enables the creation of cloud-based virtual clusters solely by utilizing low-cost spot instances. Our system allows organizations that do not have a local cluster to run streams of computational jobs in a fast and economical way;
- A resource provisioning strategy that decides when to request spot instances to accommodate incoming jobs, as well as which instance type to request.
- An information mechanism that aids decision-making by providing runtime estimates;

The rest of this paper is organized as follows: Section 2 describes related literature; Section 3 describes the proposed architecture; Section 4 details the mechanisms that composed our resource allocation policy; Section 5 presents experimental results and their discussion; finally Section 6 concludes the paper.

2 Related Work

With the increasing popularity of cloud infrastructure, many organizations have started looking into the exploitation of cloud resources rather than maintaining

their own in-house cluster facilities, which are expensive to maintain. In this section we present the most relevant works that consider similar scenarios and compared them to our contribution.

2.1 Cloud-Based Virtual Clusters

Research on building virtual clusters using cloud resources can be generally divided into two categories: (1) techniques to extend the capacity of in-house clusters at times of the peak demand, and (2) assembling resource pools using only public cloud resources and using them to run compute intensive applications. For instance, Assuncao et al [2] have evaluated a set of well known scheduling policies, including backfilling techniques, in a system that extends the capacity of a local cluster using fixed-priced cloud resources. Similarly, Mattess et al [4] have evaluated policies that offload extra demand from a local cluster to a resource pool composed by Amazon EC2 spot instances. In contrast to these works, our system model does not consider the existence of a local cluster, instead all resources are cloud-based spot instances.

2.2 Use of Variable Pricing Resources

A few recently published works have touched the subject of leveraging variable pricing cloud resources in high-performance computing. Andrzejak et al. [5] have proposed a probabilistic decision model to help users decide how much to bid for a certain spot instance type in order to meet a given monetary budget or a deadline. The model suggests bid values based on the probability of failures calculated using a mean of past prices from Amazon EC2. It can then estimate, with a given confidence, values for a budget and a deadline that can be achieved if the given bid is used.

Yi et al. [3] proposed a method to reduce costs of computations and providing fault-tolerance when using EC2 spot instances. Based on the price history, they simulated how several checkpointing policies would perform when faced with out-of-bid situations. Their evaluation has shown that checkpointing schemes, in spite of the inherent overhead, can tolerate instance failures while reducing the price paid, as compared to normal on-demand instances.

3 System Model of a Cloud-Based Virtual Cluster

We consider a scenario where an organization is interested in building a dynamic cluster using cloud resources. We have assumed that resources are to be leased exclusively from one cloud provider, such as Amazon EC2, even though our proposed solution can be extended to support multiple providers.

Resources are virtual machines, instantiated according to a previously configured template, which defines the capacity required (or “instance type” in Amazon EC2 terminology) as well as the operating system and software stack details (i.e. Amazon Machine Image). A job scheduling and middleware system, called

Broker, is responsible for receiving job requests from users, creating a suitable VM pool, and managing the QoS of jobs, i.e. ensuring that jobs finish within the deadline.

Jobs are submitted by local users of the organization. A job request must contain information such as: the task(s) to be executed (e.g. binary files or scripts), the number of required processors, the amount of memory needed, total runtime estimation, and a deadline.

The system model we define in this work has two main components: a Broker (job scheduling and middleware); and a cloud provider-side VM management infrastructure that we term as the "cloud manager". A graphical representation of the modeled system components is presented on Figure 2.

Broker Component: Computational job execution is managed by the Broker, which takes the responsibility of receiving job submissions and assembling a pool of cloud spot instances on behalf of the organization. The broker obtains all available information about a job and uses that information to perform scheduling decisions.

More specifically, the broker manages the following activities: i) Job management: job admission, job execution, job failures, and monitoring of job QoS constraints; ii) Virtual cluster management: bidding, instance selection, instance requesting and termination,

Cloud Provider Model: Spot instance management activities are performed by the cloud provider. For this component, we assume a similar cloud model as the one described by Yi et al. [3], which reflects how the spot market currently works in the Amazon Cloud. More specifically, the model considers the following characteristics:

- Clients submit a spot instance request, specifying how many instances are needed, an instance type, and up to how much they are willing to pay per instance/hour (bid);
- The system provides spot instances whenever the client's bid is greater than the currently advertised price; on the other hand, it terminates instances that do not meet the current spot price, immediately without any notice, when a client's bid is less than or equal to the current price.
- The system does not charge the last partial hour when it stops an instance, but it charges the last partial hour when the termination is initiated by the

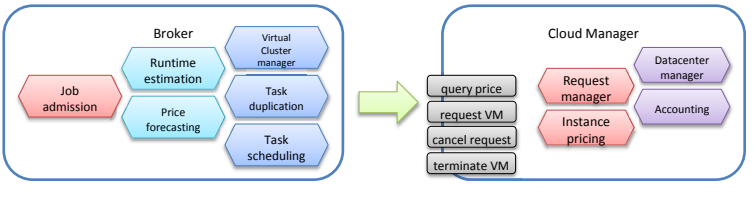


Fig. 2. Modeled architecture: Client (broker) and server (cloud) side components

client (the price of a partial hour is considered the same as a full hour). The price of each instance/hour is the spot price at the beginning of the hour.

In summary, the cloud manager is responsible for the following tasks: i) Request management: admission control based on bids, and serving valid requests; ii) Spot instance management: new instance requests, terminations due to out-of-bid situations, and billing of hourly charges.

4 Cost-Effective Resource Provisioning and Scheduling Policy

The Broker is equipped with a VM provisioning and job scheduling policy, which composes the core contribution of this work. The following steps are involved in allocating each incoming job to a virtual machine. These activities are described in Algorithm 1.

- When a job is submitted, it is inserted into a list of unscheduled jobs [Line 3].
- At each scheduling interval, the algorithm uses a runtime estimation method to predict the runtime of the job on each available instance type [Line 6].
- The broker then attempts to allocate the job to an idle, already initiated, VM with enough time before a whole hour finishes [Lines 7-10].
- If unsuccessful, it decides whether the allocation decision can be postponed, based the maximum time the job can wait so that the chance of meeting the deadline is increased [Lines 11-16].
- If the job cannot be postponed, it attempts to allocate it to a VM that is is expected to become idle soon. Runtime estimates of all jobs running on the VM are required at this step [Lines 18-24];
- If the job still cannot be allocated, the algorithm will decide whether to extend a current lease or to start a new one. This decision is made based on information about the estimated runtime of the incoming job on each VM type [Lines 18-24].
- The algorithm then checks if there are still idle VMs, which are then matched to non-urgent jobs that have been postponed in previous iterations [Lines 26-32]. Idle VMs are the ones that have been flagged to be terminated when the next whole hours finishes.
- Finally, for each job that was allocated to a VM in the current iteration, a correction event is scheduled to trigger at the time the job is expected to finish [Line 33-35]. This event, if necessary, will then activate the estimation correction and rescheduling mechanism, which corrects the estimation and reinserts all jobs allocated to the affected instance into the list of unscheduled jobs.

These steps are conducted in regular intervals (time t, which can be defined based on the arrival rate of jobs). In this work, we have set t to 10 seconds, so that our algorithm operates in on-line fashion, especially to guarantee that jobs with a tight

```
   input : available instance types types
 1 while true do
 2     while currentTime < NEXT_SCHEDULE_TIME do
 3         Receives incoming jobs and inserts in the list LJ
 4     vms ← all VMs currently in the pool;
 5     foreach Job j ∈ LJ do
 6         erts ← compute estimated runtime of j on each type ∈ types;
 7         decision ← FindFreeSpace(j, vms);
 8         if decision.allocated = true then
 9             AllocateJobToVM(j, decision.vm);
10             continue;
11         mwt ← compute maximum wait time for j;
12         if CanPostpone(j) then
13             Delay allocation of j by mwt;
14             Add j to list LNU of non urgent jobs;
15             Remove j from LJ;
16             continue;
17         extendDecision = ComputeLeaseExtensions(j, vms);
18         newDecision = ComputeCostForANew(j);
19         if extendDecision.cost <= newDecision.cost then
20             trigger lease extension;
21             AllocateJobToVM(j, extendDecision.vm);
22         else
23             newVM ← LeaseNewVM();
24             AllocateJobToVM(j, newVM);
25     update state of VMs;
26     if there are idle VMs and LNU is not empty then
27         foreach Job j ∈ LNU do
28             decision ← FindFreeSpace(j, idlevms);
29             if decision.allocate = true then
30                 AllocateJobToVM(j, decision.vm);
31                 remove j from LNU;
32                 add j to LJ
33     foreach Job j ∈ LJ do
34         if j.isAllocate = true then
35             dispatch correction and rescheduling event at time now + j.ert
36     clear LJ
37     NEXT_SCHEDULE_TIME = NEXT_SCHEDULE_TIME + t;
```

Algorithm 1. Resource provisioning and job scheduling algorithm

deadline are given to opportunity to start as soon as possible. Still, the use of techniques such as job postponing, runtime estimation and delayed termination of idle instances ensures that the policy keeps a holistic view of the workload.

Details of the various steps performed by our algorithm, such as runtime estimation and correction, rescheduling, and job speedup characteristics are given in following sections.

4.1 Estimating Job Runtimes

In order to circumvent inaccurate user-supplied estimations, especially harmful in backfilling schedulers, several works have proposed methods to predict job runtimes, where the system computes the estimated runtime of a job and uses it in place of a user-supplied estimate. Although complex methods have been proposed, Tsafir [6] has observed that "even extremely trivial algorithms (e.g. using the average runtime of two preceding jobs by the same user) result in significant improvement".

Although we do not make use of backfilling techniques in this work, our motivation to employ runtime estimates is similar to those schedulers: improve system utilization, especially important because there is a minimum cost involved for each new VM added to the pool. Due to an hourly billing fashion, assumed in this work to reflect practices of cloud computing providers, every VM runs for a minimum of one hour. Therefore, relying solely on user provided runtimes (mostly over-estimated) might lead to unnecessary requests.

In our scenarios, runtime predictions aid the decision-making process in the following ways: i) the broker can decide whether to add new resources to the pool based on the expected time that currently busy VMs would be free; ii) along with information about current instance prices, it is possible to estimate the cost to run a job on a given instance type, thus increasing the chances of meeting monetary constraints.

Previous studies have shown that the "one size fits all" notion does not apply to runtime estimation of job runtimes. For this reason, we have implemented 5 different methods, especially because no method has been shown to work well in all scenarios [6]. A detailed discussion of each of these methods is presented in the evaluation section of this paper.

4.2 Estimation Correction and Rescheduling

We have equipped the broker with a correction and rescheduling mechanism, which activates whenever a job is detected to have been running longer than expected, regardless of which runtime estimation method has provided the estimate. Whenever a job starts running, a correction event is scheduled to trigger immediately after the moment the job should have finished. If, at that moment, the job is still running, a correction operation is performed. The broker simply assumes a new estimate that is equal to double the old estimate, and the job remains allocated to the current VM. All other affected jobs, i.e. the ones scheduled to the same instance, which might be delayed, are resubmitted to the scheduler for rescheduling.

4.3 Job Moldability and Speedup Considerations

We assume jobs to be moldable, i.e. they can execute on any number of processing units, but restricted to a single VM. We model instance types as to contain one or more processing units, assumed to be equal to the amount of EC2 compute units of each available instance type. Each job runs on a single instance, and each instance can only run one job at a time.

To determine the runtime of a job in a particular number of compute units, we use Downey's analytical model for job speedup [7]. Downey's model requires two parameters: the average parallelism A and an approximation to the coefficient of variance of parallelism σ. To generate values for A and σ, we have used the model of Cirne & Berman [8], which has been shown to capture the behavior of a range of user jobs. Generated values were added as parameters to each job originally present in the LCG workload trace. We assume that these values are known by the users who submit the jobs, thus they can be used by the resource provisioning strategies.

5 Performance Evaluation

In this section, we evaluate the proposed resource allocation and scheduling policy using trace-driven discrete event simulation which is implemented using the CloudSim framework [9]. The overall objective of our experiments is to quantify the performance of our proposed policy based on three metrics: monetary cost, system utilization, and deadline misses. Since our proposed policy aims at minimizing the cost of building virtual clusters, the monetary cost of such activity is considered as the main metric. System utilization indicates how long instances remain idle before they are terminated. Deadline misses refers to the number of submitted jobs which did not finish within the specified deadline; this is a metric directly related to user satisfaction.

In a first scenario, the policy is compared with two base provisioning policies: worst-case and best-case resource provisioning. In the worst-case provisioning, the broker provisions only on-demand fixed-price instances but schedules jobs on the most efficient machines types for each job. This provisioning is very similar to the current solutions for building virtual cluster using cloud resources [4]. The best-case resource provisioning is a hypothetical lower bound devised to evaluate the cost-effectiveness of the proposed policy.

In a second experimental scenario, the effects of various runtime estimation on the proposed policy are evaluated to understand which runtime estimation method should be used for a given workload.

Virtual Machine Types: As stated earlier, our resource provisioning strategy aims at choosing the most efficient instance type to run a job. Five instance types were used in our experiments. They were modeled directly after the characteristics of available standard and high-CPU Amazon EC2 types. The types available to be used are M1.SMALL (1 ECU), M1.LARGE (5 ECUs), M1.XLARGE (8 ECUs), C1.MEDIUM (5 ECUs), C1.XLARGE (20 ECUs).

Workload: The chosen job stream was obtained from the LHC Grid at CERN [10], and is composed of grid-like embarrassingly parallel tasks. A total of 100,000 jobs were submitted over a period of seven days of simulation time. This workload is suitable to our experiments to due to its bursty nature and for being composed of highly variable job lengths. These features require a highly dynamic computation platform that must serve variable loads while maintaining cost efficiency. Originally, this workload trace did not contain information about user-supplied job runtime estimates and deadlines. User runtime estimates were generated according to the model of Tzafrir et al. [11]. A job's maximum allowed runtime corresponds to the runtime estimate multiplied by a random multiplier, uniformly generated between 1.5 and 4. Consequently, the job deadline corresponds to its submission time plus its maximum allowed runtime.

5.1 Comparison with Best-Case and Worst-Case Scenarios

In this section, we compare our proposed scheduling policy with other base policies. Based on information from the workload trace (actual job length and parallelism parameters A and σ), we have computed how much would be the best possible cost that could be achieved to run all 100,000 jobs using the most efficient instance type for each job, considering multiple pricing schemes. The most efficient match for a job depends on its maximum speedup, the job's length and the instance's cost per hour. As a general rule, short jobs or jobs that provide little or no parallelism run more efficiently on less powerful, cheaper instances; whereas longer jobs (execution time in the order of hours) that provide good parallelism are more suitable for high-CPU instances, which provide a lower cost per ECU.

Table 1 lists the costs that would be obtained in both best-case and worst-case resource provisioning scenarios. Particularly, the cost of $2790.28 corresponds to the best possible. Therefore, the aim of any resource provisioning strategy is to obtain a cost as close as possible to this value.

Our proposed policy, when running with the *"Recent Average"* estimation method was able to obtain an improvement of about 60% over the worst case provisioning policy and just 23% worse than the best case.

Table 1. Total cost compared with two base policies

Instance type	Percentage of jobs	Worst-case	Best-case	Proposed Policy
M1.SMALL (1 ECU)	6.646%	$1114.62	$371.54	NA[1]
C1.MEDIUM (5 ECUs)	84.564%	$6942.38	$2314.13	NA
C1.XLARGE (20 ECUs)	8.790%	$313.84	$104.61	NA
	Total:	$8370.84	$2790.28	$3628.25

[1] We report individual percentage of jobs that ran on each instance type only for the deterministic scenarios (worst-case and best-case) as an indication of the bias towards high-cpu instances. These values are not necessarily meaningful of how the policy allocates jobs in practice, where the total cost is the metric that really matters.

5.2 Effect of Different Runtime Estimation Methods

We now describe in detail the 5 runtime estimation schemes and compare their effects on the following metrics: monetary cost, number of deadline misses, and system utilization. All values presented correspond to an average of 30 simulation runs. Each run is set to start at a random point in time, uniformly chosen between March 1st, 2010 and February 1st, 2011. These dates correspond to the available price traces obtained from Amazon EC2.

In the *"Actual runtime"* approach, the actual job length, as per the workload trace, is supplied to the allocation algorithm; while the *"Actual runtime with error"* approach consists of using the actual length slightly modified by a random percentage between 0 and 10%. Naturally, these two approaches are not real as they are based on information that would normally not be available in practice. They are included here for comparison purposes. However, should a nearly perfect estimation method be available, say in a highly controlled environment where detailed information about the workload characteristics is known, we can then foresee that our proposed allocation algorithm would perform well, as these two strategies yield the best results.

The *"User Supplied"* approach assumes the job length to be the value informed by the user at job submission. Based on previous observations that user-supplied estimated runtimes are mostly over estimated, we have also devised the *"Fraction of User Supplied"* approach, that uses a value equal to $\frac{1}{3}$ of original value as the job length.

The *"Recent Average"* approach consists of using the average runtime of two jobs completed prior to the submission of an incoming job by the same user. If not enough information is available to compute the estimated length of an incoming job, i.e. less than two jobs have completed at the time of decision-making, the estimation is assumed to be given by the *"User Supplied"* method.

We conducted two sets of experiments. In first set, our strategy was not equipped with the correction and rescheduling mechanism, described on section 4.2. In these experiments, once the strategy made a decision based on a runtime estimate, jobs would remain allocated to the instance chosen on the first decision. This resulted in an excessive amount of deadline misses, especially when using the *"Recent Average"* estimation method, as can be see on Figure 3. This fact is due to under estimations, that caused many jobs to be allocated to the same instances. Once a certain job that was expected to finish at a certain time did not finish, all other jobs would be delayed. By adding correction and rescheduling, the strategy was able to virtually eliminate the occurrence of deadline misses.

Figures 4(a), and 4(b) show the effect of changing the runtime estimation component on the total monetary cost, and system utilization respectively. These results correspond our second set of experiments, which were collected after the correction and rescheduling mechanism was implemented.

Results demonstrate that, contrary to our early belief, precise information does not necessarily translates into more efficient allocation, especially in terms of cost.

This fact can be observed in the comparison between the *"Actual runtime"* and *"Actual runtime with error"* methods, where the latter performs better. Based on observations of the simulation logs, we can attribute this difference to moderate over-estimations, that cause more instances to be requested, which are reused often by short jobs. On the other hand, the exact estimates provided by the optimal method cause the allocation system to request fewer instances and also to terminate instances more often. Incoming jobs then cause more new instances to be requested, which are then more likely to remain idle if the job that triggered the request was a short one. This fact can be confirmed by observing the system utilization under the effects of the *"Actual runtime"* method (90%) and *"Actual runtime with error"* (96%).

In term of deadline misses, runtime estimations methods that tend to overestimate provide better results. However, excessively over-estimated runtimes were shown to increase costs significantly, as they cause a much higher number of instances to be requested, especially more expensive instances. Although these instances are sometimes reused by other jobs, in most cases the allocation

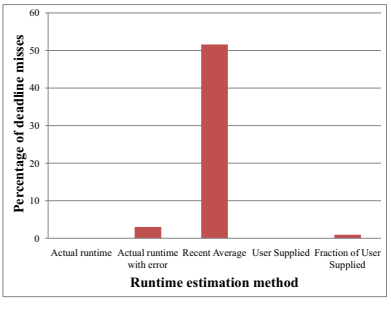
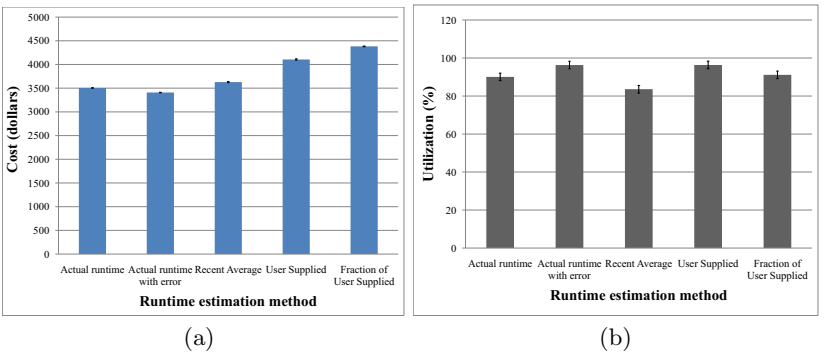


Fig. 3. Deadline misses before the correction and rescheduling mechanism was introduced



(a) (b)

Fig. 4. Effect of different runtime estimation methods; (a) on monetary cost, and (b) on system utilization

algorithm will choose to create new instances, by mistakenly considering that incoming jobs are long, thus requiring more powerful instances to complete their execution within their deadlines. This observation can be inferred from the results obtained by the *"User Supplied"* and *"Fraction of User Supplied"* estimation methods, also presented on 4(a); both methods tend to provide excessively over-estimated runtimes.

We have also observed that good system utilization alone does not necessarily translates into lower costs. For example, by using the *"Recent Average"* estimation method, our strategy could achieve a better cost than *"User Supplied"*, but the utilization was significantly lower. The key aspect to observe in this scenario is the importance of choosing the correct type of instances for each job. Smaller instances, even when not utilized efficiently, have less impact on the final cost, than larger and more costly instance, which must be used efficiently to compensate for their cost.

In conclusion, in our scenarios, slightly over-estimated runtimes have shown to be beneficial. On the other hand, excessively over-estimations are ineffective because of the bias towards larger and costly instances. Under-estimations have shown to increase the chance of deadline misses, but they are not as ineffective, as they can be corrected and affected jobs can be rescheduled. In any case, we can conclude that accurate estimations help in achieving lower costs.

6 Conclusions and Future Work

Building dynamic virtual clusters using cloud resources is an effective way of saving in monetary cost. This paper has provided a cost-effective solution to provision a virtual cluster using spot market cloud resources to run computational jobs having a deadline as a QoS constraint. To address this challenge, a resource allocation and job scheduling policy, which takes into account variations in price and performance of cloud resources and aims at choosing the most efficient virtual machines for deadline-constrained jobs. We have evaluated the performance of the proposed policy, by comparing it to worst-case and best-case scenarios. An improvement of up to 60% in cost has been obtained in comparison with the worst-case, and the policy has performed close to the best-case.

We have also evaluated 5 runtime estimation methods and their effects on the policy performance. For the given workload, we have concluded that more accurate estimation methods provide significantly superior results, when compared to methods that excessively over-estimate runtimes.

As future work, we will integrate our proposed solution into real cloud schedulers, and evaluate further performance benefits of our approach. We will also tackle the problem of jobs failures and delays due to the intermittent nature of spot instances, by applying fault tolerance techniques such as workload migration, checkpointing and task duplication.

References

1. Hill, Z., Humphrey, M.: A Quantitative Analysis of High Performance Computing with Amazon's EC2 Infrastructure: The Death of the Local Cluster? In: Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (October 2009)
2. de Assuncao, M.D., di Costanzo, A., Buyya, R.: Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters. In: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing. HPDC 2009. ACM, New York (2009)
3. Yi, S., Kondo, D., Andrzejak, A.: Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In: 2010 IEEE 3rd International Conference on Cloud Computing, pp. 236–243. IEEE, Los Alamitos (2010)
4. Mattess, M., Vecchiola, C., Buyya, R.: Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market. In: Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications, pp. 180–188. IEEE Computer Society, Los Alamitos (2010)
5. Andrzejak, A., Kondo, D., Yi, S.: Decision Model for Cloud Computing under SLA Constraints. Technical report, INRIA (2010)
6. Tsafrir, D.: Using inaccurate estimates accurately. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 208–221. Springer, Heidelberg (2010)
7. Downey, A.B.: A Model For Speedup of Parallel Programs. Technical report, Berkeley, CA, USA (1997)
8. Cirne, W., Berman, F.: A Model for Moldable Supercomputer Jobs. In: Proceedings of the 15th International Parallel and Distributed Processing Symposium. IEEE Computer Society, Los Alamitos (2001)
9. Buyya, R., Ranjan, R., Calheiros, R.: Modeling and Simulation of Scalable Cloud Computing Environments and the Cloudsim Toolkit: Challenges and Opportunities. In: Proceeding of the International Conference on High Performance Computing & Simulation, HPCS 2009, pp. 1–11. IEEE, Los Alamitos (2009)
10. Feitelson, D.: Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload
11. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Modeling User Runtime Estimates. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 1–35. Springer, Heidelberg (2005)

A Principled Approach to Grid Middleware
Status Report on the Minimum Intrusion Grid

Jost Berthold[1], Jonas Bardino[2], and Brian Vinter[2]

[1] Department of Computer Science, University of Copenhagen, Denmark
berthold@diku.dk
[2] eScience Center, University of Copenhagen, Denmark
{bardino,vinter}@nbi.dk

Abstract. This paper provides an overview of MiG, a Grid middleware for advanced job execution, data storage and group collaboration in an integrated, yet lightweight solution using standard software.

In contrast to most other Grid middlewares, MiG is developed with a particular focus on usability and minimal system requirements, applying strict principles to keep the middleware free of legacy burdens and overly complicated design. We provide an overview of MiG and describe its features in view of the Grid vision and its relation to more recent cloud computing trends.

1 Introduction

The Grid computing vision of Foster and Kesselman [9] in the late '90s promised to substantially facilitate access to remote computing and storage resources. However, today's Grid middlewares in practical use only provide a somewhat reduced model. "The Grid" falls short on expectations. Today's large-scale Grid systems demand considerable expertise from the user, maintenance is staff-intensive, and they tend to seclude their user base to a few privileged scientists. As a consequence, potential users turn away from Grid solutions today, and increasingly spend subsidies on dedicated hardware (for instance, GPGPUs). We argue that the reason for this disappointing reality is excessive middleware complexity and inapt prioritisation in its development. The Minimum intrusion Grid (MiG) started in 2004 [18] to address a number of shortcomings in existing Grid middlewares, and follows a principled approach free of inherited legacy burdens. First, MiG's principle is to minimise requirements for both users and resource providers in a Grid. Second, MiG offers more than a simple "job-shop" system: it provides a complete working environment with not just computational power but also storage, collaboration software (Wiki, forum and version control repository) and an integrated web portal that can even encapsulate whole workflows. In this paper, we give an overview of MiG, discuss its design principles and realisation, and present its advanced features for group collaboration and resource sharing. We argue for a revival of the Grid vision in view of the current *cloud computing* trend. Presenting the advanced key features of MiG, we point out that the Grid vision goes far beyond cloud ideas, and how a principled Grid middleware can be realised in a flexible and user-friendly manner.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 409–418, 2011.

2 Background and Motivation

2.1 Grid Vision and Grid Practice

When the term of Grid computing was coined in the late '90s by Foster and Kesselman [9], Grid was envisioned as much easier to use than conventional large-scale computing. The Grid promised transparency and single entry points to powerful parallel computing resources (computational Grids) and storage (data Grids), organised in easily managed abstract entities (Virtual Organisations). However, implemented Grid systems exposed a continuously decreasing ambition, towards a substantially reduced model for the sake of reliability and system management. And yet, using one of the existing Grid middlewares (for instance, NorduGrid ARC [6], gLite [10] or EGI [5]) which inherit code base and paradigms from the Globus Toolkit [8] still has a number of problems:

- Even simple Grid system services are very complex to administrate. As a survey of Grid middleware support mailing lists shows, middleware configuration is extremely intricate and requires considerable internal knowledge.
- A common practical problem is to reliably provide special software packages for particular users. The usual solution, manual installation and maintenance of all software on all computing resources, is error-prone, cumbersome, and totally unsatisfactory against the original Grid ideas.
- Despite it being the key feature of Grid systems, granting and obtaining access to Grid resources usually involves manual work and human factors. As a result, Grid users are more often than not privileged scientists who rely on good administrator contacts and support to suit their computing needs.
- Virtual Organisations should improve this but often create middleware incompatibilities by proprietary X.509 certificate extensions. More important, implementations of virtual organisations merely provide administrative grouping, not working environments and collaboration support.

Reasons for this unfortunate development include, among others, the inherent danger of large-scale multinational projects to produce overly complex architectures and to focus on aspects of minor interest to users. For instance, large parts of the ARC [6] middleware deal with monitoring, accounting and user roles for access control. In contrast, ARC's solution for providing software on computing resources has remained strikingly ad-hoc after years in production. The primary focus of a useful Grid middleware should be ease of use for resource users and resource providers, and maximised resource usage (including harvesting spare cycles). Besides, Grid should go beyond a mere "job shop". Users should profit from advanced middleware features tailored to better collaboration.

2.2 The Cloud Vision – A Better Grid?

Far more recent, yet substantially related to Grid, is the fashion of cloud computing – alas, a cloudy subject with a range of definitions. A restrictive one comes

from Berkeley, limiting the term to flexible virtualised infrastructure provisioning (IaaS) [2]. In line with this is the widespread understanding of "cloud" as providing full user control over an entire virtual machine (as opposed to running jobs in a pre-existing setup), fostered by Amazon's "Elastic Cloud Computing" (EC2) platform (the breakthrough in cloud computing of modern imprint in late 2006). We favour a slightly broader definition by the National Institute of Standards and Technology (NIST), which also subsumes software and platform services, and emphasises that "configurable computing resources [..] can be rapidly provisioned and released with minimal management effort or service provider interaction." [14]

Be it virtual hardware or a software setup, cloud computing is characterised by minimal setup and maintenance effort, and user self-service. In essence, the goals of convenience and resource usage are the same as in Grid, and when realised merely through virtual machines, one can indeed argue that the cloud ideas are a commercialised and reduced variant of the earlier academic Grid vision. The main reduction lies in virtual organisations and group collaboration, an essential goal of Grid but completely missing from cloud computing as of today.

3 Overview of the Minimum Intrusion Grid

3.1 System Architecture

A bird's eye perspective on the MiG system (in Fig. 1) shows its central motivation: *to virtualise user access to resources*. The Grid system acts as a gateway between users and resources that provide storage or computing capabilities.

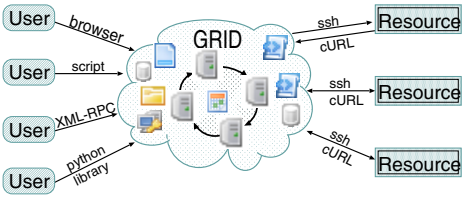


Fig. 1. Abstracted MiG Architecture

This gateway itself can be composed of replicated server instances that coordinate job execution and introduce failover safety. Users access the system via different methods, all based on secured HTTP. Communication from resources to a server uses secure HTTP as well, while a server uses SSH to address resources. The HTTP interface provides job submission and management, server file space, virtual organisations (VGrids) with shared file and web space, and means to provide and manage resources and software.

3.2 MiG Design Principles and Rationale

MiG was developed with a principled approach to follow the original Grid vision, and to address the issues observed in other Grid middlewares. This led to a number of principles and goals, ranging from technical implementation principles over architectural requirements to usability.

Non-intrusive Software Installation. The first and foremost design goal of MiG is to reduce software complexity and to avoid legacy burdens of any kind. Resources and users must be able to join and use the Grid with *minimal software*

requirements. Experiences with other Grid middlewares show that large installations on the resource side create compatibility problems and – contrary to the Grid philosophy – require more centralised administration and maintenance. Likewise, new users may be unwilling or unable to install a large software.

Minimal Dependencies. Typical production Grid systems carry heavy burdens of legacy code and suffer from incompatibilities. Use of many different languages and tools complicates maintenance and hampers porting software to new platforms. MiG was designed to reduce maintenance requirements, and to automate maintenance operations whenever possible. This relates to the choice of implementation language as well as the tool requirements.

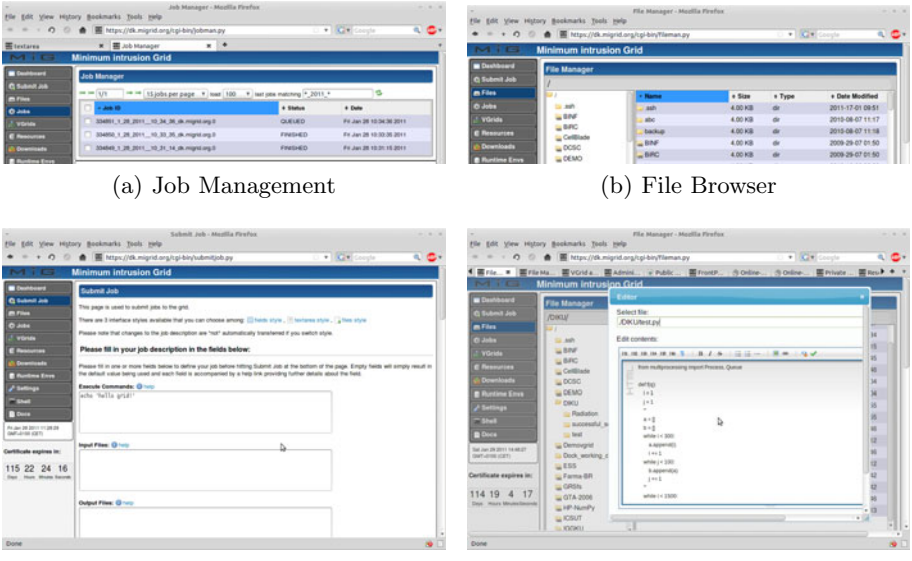
The distribution aspects of the entire system are based on secure HTTP communication. For the user, MiG only requires an X.509 user certificate from a trusted CA and a standards-compliant web browser (or other HTTP software). MiG resources require secure HTTP (cURL [4]), SSH, and a standard shell – system interfaces that can safely be assumed stable. Basing all functionality on standard software and few commonly used protocols/ports maximises MiG firewall compliance, and enables the server to update all software (scripts) on the resource without manual intervention. The MiG server software is based on the widespread Apache HTTP server and written in widely portable Python.

Fault Tolerance. Failing machines or processes within the Grid should not stop users or resources from using the Grid. If a resource crashes, the middleware should be responsible for recovering and repeating any lost jobs. The MiG design includes a concept of distributed communicating servers, where all data are replicated several times for failover safety, while keeping clear responsibilities for consistency. Jobs are replicated and quickly rescheduled by servers upon failure.

Anonymity of users and resources. Job submission in most Grid systems involves direct communication between the submitting client and potential execution resources. In such a design, a resource provider can theoretically monitor Grid usage of all users by their incoming execution requests. In the MiG design, the mediating servers can completely shield users and resources from one another if desired (but with server audit logs to trace any abuse). Assuming sufficient job throughput, it is not possible for a resource owner to identify the user who submitted a particular job. Likewise, resources can be provided in an anonymised fashion. This is particularly desirable in industrial Grid usage where the same resources might provide services for competing companies.

Straightforward Comfortable Usage. In analogy to the system related principle of minimal intrusion, a Grid system should support the user in a straightforward manner. The MiG middleware helps beginners by hiding complex features and providing reasonable defaults for any optional feature.

Advanced Support for Group Collaboration. It has been stated [7] that the essence of Grid is collaboration in virtual organisations, so a Grid middleware should provide appropriate tools especially for this task. MiG provides a wide range of user-controlled collaborative structures and tools, to truly enable collaborating virtual organisations and improve the overall Grid usability.



(a) Job Management (b) File Browser

(c) Job Submission Interface (d) Simple Built-in Editor

Fig. 2. MiG Browser Interface Screenshots

3.3 Browser-Based Interface

The central user interface to a MiG system is the web browser. Users are authenticated and authorised by an X.509 certificate installed in their browser, and can then access all functionality through a single portal. The MiG design carefully avoids transferring any temporary "proxy" credentials to servers or resources.

As Figure 2 illustrates, users should experience MiG as a multiuser workstation with a system account and pre-installed software. The interface to submitting and managing jobs is shown in Figures 2(a) and 2(c). Figures 2(b) and 2(d) demonstrate another essential component of the "workstation" view: the user's home directory on the server and how to manipulate files from within the browser. Some folders are shared with other users (based on virtual organisations), and are seamlessly integrated in the home directory; a custom icon indicates the sharing. We will explain some key features shown in the menu on the left in detail subsequently (VGrids, Resources, and Runtime Env.s).

3.4 How MiG Executes Grid Jobs

Efficient job execution by resources is the primary Grid functionality, realised in MiG in a uniform, straightforward and server-centric manner. Contrary to Globus based Grids, MiG job requests are submitted to a Grid job queue handled by one or more MiG servers from start to finish. There is no direct communication between the user and the resource where the job ends up executing and the user can safely go offline as soon as the job is queued.

Simple Job Specification. Figure 2(c) shows the simple graphical interface with individual fields to specify requirements and actions for a Grid job. While this HTML form is the default interface, users can also directly use the internal job specification language *mRSL*, a simple line-delimited text format. Compared with richer standards such as JSDL or RSL [1,8], mRSL is simplistic but sufficient. A job specification includes a list of commands to execute, input and output files, and a range of other (optional) properties: hard resource requirements like node and CPU count, soft requirements like memory and disk consumption and wall-clock time limits, and a retry counter for failover scheduling.

Grid Job Scheduling in MiG. Production Grids typically have a long queue of waiting jobs, as there are often more jobs than execution slots. In MiG, pending jobs are stored on servers until a resource becomes available. Resources actively request jobs from a server (*pull* scheduling), and the server selects the best-fitting job according to a configurable scheduling policy. This allows for scheduling jobs across multiple physical resources to optimise throughput.

Scheduling in MiG is guided by the job's software requirements, user-provided memory and disk limits, node and CPU count, and by the execution history of the resource. A number of scheduling algorithms are implemented and deliver good throughput on average while preventing starvation. Apart from the classical variants FIFO, Random and First-Fit, MiG implements a Best-Fit scheduler that avoids occupying oversized resources, a Fair-Fit scheduler which additionally prioritises jobs that have waited longer, and a scheduler based on a Vickrey auction [3], which lays grounds for a pay-per-use "Grid economy" of resources.

3.5 Software Deployment

In order to run anything but trivial jobs on a Grid resource, one typically needs special software. Requirements beyond the basic Grid system have to be be negotiated between users and resources, typically realised in Grid systems by the concept of *Runtime Environments* [12]. A runtime environment in MiG is a data structure that describes system properties – for instance, a special numeric software library, or special hardware – and specifies environment variables to be used in job scripts that require them – for instance, a variable containing necessary compiler flags, the install location, or the path to an executable tool. When a resource provides this library, the owner adds the respective runtime environment to the resource's specification and assigns values to these environment variables. Any MiG user can define runtime environments, but they cannot be modified (only deleted) after creation – redefinitions would cause inconsistencies. And, well-understood, there is no guarantee that any resource in the system implements a particular runtime environment.

Dedicated computing resources usually have their own native software package management system, which only privileged users can use. Users often have to manually intervene and ask a resource owner to install the necessary software, a typical productivity bottleneck in today's production Grids. To address this issue, a mechanism was developed to automatically install software *on-demand*

from a software catalogue hosted on the MiG servers. The software catalogue in MiG uses the Zero Install [16] packaging system to support installation and automatic maintenance of pre-packaged software on resources. When a resource provides the ZeroInstall runtime environment, a job can install and use packaged software in a secure and controlled way, instead of using a native installation. Runtime environment specifications are automatically generated from Zero Install package descriptions, making the entire process transparent to the user.

4 MiG Features beyond the "Job-Shop"

4.1 VGrids: Virtual Organisations in MiG

In early 2000, *Virtual Organisations* (VOs) were stated by Foster [7] as the original "Grid problem", motivation for developing the Grid concept altogether. The term *Virtual Organisation* describes dynamically evolving groups of users that belong to distinct administrative domains, but want to share resources (compute power, storage, software etc.) for a specific purpose. Flexible and dynamic resource sharing across administrative domains is the core of Grid technology.

The MiG system models Virtual Organisations as *VGrids* [11]. As the MiG E/R model in Figure 3 shows, VGrids are hierarchical and act as an organisational entity to define the relationship between resources and users, and among users. Since the model relies on VGrids, it includes a default VGrid which includes all users and where any resource can (but does not have to) participate by default.

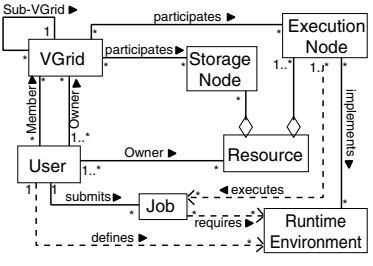


Fig. 3. MiG E/R Model

All services are managed via VGrids: Jobs execute inside a VGrid where the submitter is a member, execution nodes of a resource contribute compute power to members of one or more particular VGrids, and storage nodes expose a specified directory on the resource to members of the VGrids that they have signed up to. Each VGrid also provides a shared folder for members on the server. VGrids are easy to create and manage using the MiG web interface. Furthermore, VGrid operations (adding members or owners, creating sub-structures) are entirely user controlled, enabling ad-hoc VGrid formation and efficient collaboration without administrative hurdles. As we are going to see next, this VGrid concept allows MiG to lift *resource provisioning to user level*, and VGrids can also provide *cloud storage services* and advanced *workflow integration*.

4.2 Resource Management

Any MiG user can add a resource to the Grid through a simple HTML form, by specifying its properties: hardware and software specifications (CPUs, memory, disk size) and login details for the user account that executes the Grid jobs. After submitting the form, the resource is added to the default VGrid.

Different levels of resource trust can thus be modelled with different VGrids. The default VGrid is open to anyone and thus has the lowest level of trust. Resource owners configure which VGrids their resources should participate in as an execution or/and storage resource. Respectively, VGrid owners need to accept the resources into their VGrids before jobs can be executed (to prevent job hijacking). The entire process requires no Grid administrator intervention.

4.3 Storage in MiG

One of the distinctive features of MiG in comparison to other Grid middlewares is the concept of a central user home directory. This is part of the MiG philosophy of the Grid being a virtual workstation: users store their files in their home directory and reference them with relative file names in jobs, and each VGrid provides shared file space which is only visible to VGrid members. The central home directory can be accessed in the MiG web interface through a graphical file manager with context menu and a simple text editor, as shown in Figure 2(d). Optionally, it can even be securely mounted into a user's local file system through an SSHFS interface, to allow transparently working with MiG home files locally.

As useful and intuitive it may be to have files in a centralised MiG home directory, space limitations and privacy policies may prevent users from storing all data there. Jobs can specify external locations for input and output files (via common protocols like HTTP(S), SCP, or (S)FTP), and classified data can be stored in MiG in special VGrid-restricted storage nodes. For each VGrid in which a storage node participates, a directory on the node is securely mounted into the shared VGrid folder in the MiG servers' file system via SSHFS.

4.4 Advanced VGrid Features for Group Collaboration

Apart from being a fundamental concept to structure access levels and entities in MiG, VGrids also provide advanced services for group collaboration and resource sharing to their users. MiG servers host shared private folders and classical collaboration software for every VGrid, including public and private web pages, a wiki, a web forum, and a version control system. All this is integrated into the server middleware and browser interface, again in view of the browser being the primary desktop for all activities.

The VGrid web pages and shared folders can be used to realise specific workflows where conceptually similar Grid jobs are submitted frequently and require a custom setup carried out by VGrid owners. As an example, consider a workflow where scientific applications are implemented in a special-purpose language that requires a custom compiler, but the compiled executables can be run on various resources by means of a pre-packaged runtime library. In such a case, a VGrid can be used to provide a dedicated compilation resource, and other resources can use a runtime environment for execution – the whole compilation/execution workflow can be encapsulated in custom HTML forms in the VGrid web space.

We have successfully implemented this workflow encapsulation in two prototypes: one for the McStas neutron raytracing simulator [13] and one for a general

Matlab setup using a license-limited Matlab compiler [17]. Both prototypes expose the described workflow of compilation and iterated execution, followed by a post-processing step in case of the McStas software.

To encapsulate the job workflow logics in the VGrid web pages using HTML forms and javascript has clear advantages in usability: no additional authentication is required, and the easy interface for non-experts hides all uninteresting boilerplate code for Grid operation. For the implementor, two other advantages exist: Scientific expert software like the McStas compiler and post-processor is sometimes complicated to set up. In our setup, maintenance can be reduced to only a few dedicated resources that provide the special parts, and the compiled McStas simulation code is ISO C99. The Matlab software setup is easy, but a license is required; in our case only for the dedicated compilation resource.

5 Current Status and Future Directions

MiG started as a proof-of-concept implementation, but became an ongoing success over the last years, thanks to its sage architecture. At the time of writing, our group is operating a MiG installation connecting several compute clusters and special resources based on Cell-based game console and Screen saver software. The system is used for scientific projects in combinatorial genome research, bibliometric analysis, medical imaging, and for teaching purposes. MiG is actively maintained, the latest additions include an improved browser interface, a remote memory library to enable computing resources with limited memory and disk [15], and running virtual machines as interactive Grid jobs.

Development is underway for improving the virtual machine support. One essential ingredient is a light-weight VNC client that runs inside a browser (based on javascript and websockets), to realise the MiG goal of minimal installation requirements. Together with MiG's browser-based interface and the storage resource concept realising cloud storage, the virtual machine support is the final step towards cloud-style IaaS – virtualised resources – embedded in Grid.

6 Conclusions

Grid computing has come of age in the past years, yet its practical incarnations somewhat fall short of the initial vision. We have presented MiG, a Grid middleware that follows rigid principles derived from the original Grid literature and shortcomings observed in existing systems. The stated MiG principles prove useful as a general touchstone for any Grid middleware, and the presentation of MiG demonstrates that such a design is feasible and useful in practice. Especially MiG's advanced features for group collaboration, and in general its VGrid-centric design, are key features for more flexibility and user self-service than usual job-shop Grid systems can provide. The more recent trend of cloud computing has given important impulses to the Grid community, as it advances modern virtualisation techniques and the idea of a consequently consumption-based Grid economy. A principled Grid approach should incorporate modern virtualisation

techniques in its services, to realise a strict superset of the cloud vision in terms of stability, user-friendliness, and self-managed virtual collaboration.

Availability. MiG is open source software released under GNU GPL-2. More information and MiG code can be found at http://www.migrid.org.

References

1. Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., Pulsipher, D., Savva, A.: Job Submission Description Language (JSDL) Specification, V.1.0. Tech. Rep. GFD.136, Grid Forum (2008)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. CACM 53, 50–58 (2010)
3. Brandt, F., Wei, G.: Vicious strategies for vickrey auctions. In: Müller, J.P., Andre, E., Sen, S., Frasson, C. (eds.) Proceedings of Autonomous Agents 2001. ACM, New York (2001)
4. cURL. Open Source Software, http://curl.haxx.se/
5. European Grid Infrastructure, http://www.egi.eu
6. Ellert, M., Grønager, M., Konstantinov, A., Kónya, B., Lindemann, J., Livenson, I., Nielsen, J.L., Niinimäki, M., Smirnova, O., Wäänänen, A.: Advanced resource connector middleware for lightweight computational Grids. Future Generation Computer Systems 23, 219–240 (2007)
7. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: Enabling scalable virtual organizations. Intl. J. of High Perf. Computing Appl. 15(3), 200–222 (2001)
8. Foster, I.: Globus toolkit version 4: Software for service-oriented systems. J. of Computer Science and Technology 21(4), 513–520 (2006)
9. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
10. gLite Grid Computing Middleware, http://glite.web.cern.ch/glite/
11. Karlsen, H.H., Vinter, B.: VGrids as an Implementation of Virtual Organizations in Grid Computing. In: Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006). IEEE Press, New York (2006)
12. Keahey, K., Doering, K., Foster, I.T.: From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In: Buyya, R. (ed.) Proceedings of Grid Computing (GRID 2004). IEEE Press, New York (2004)
13. Lefmann, K., Willendrup, P.: McStas, a Neutron Ray-trace Simulation Package, http://neutron.risoe.dk
14. Mell, P., Grance, T.: The NIST Definition of Cloud Computing (Draft). Tech. Rep. 800-145, National Institute of Standards and Technology (NIST) (2011)
15. Rehr, M., Vinter, B.: The User-Level Remote Swap Library. In: High Performance Computing and Communications (HPCC 2010). IEEE Press, New York (2010)
16. Leonard, T., et al.: Zero Install, a decentralised cross-distribution software installation system. Software under LGPL (2003-2011), http://www.0install.net
17. MathWorks: The Matlab Compiler^{TM}, http://www.mathworks.com/products/compiler
18. Vinter, B.: The Architecture of the Minimum intrusion Grid (MiG). In: Broenink, J.F., Roebbers, H.W., Sunter, J.P.E., Welch, P.H., Wood, D.C. (eds.) Communicating Process Architectures (CPA 2005). IOS Press, Amsterdam (2005)

Performance Analysis of Preemption-Aware Scheduling in Multi-cluster Grid Environments

Mohsen Amini Salehi, Bahman Javadi, and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
{mohsena,bahmanj,raj}@csse.unimelb.edu.au

Abstract. In multi-cluster Grids each cluster serves requests from external (Grid) users along with their own local users. The problem arises when there is not sufficient resources for local users (which have high priority) to be served urgently. This problem could be solved by *preempting* resources from Grid users and allocating them to the local users. However, resource preemption entails decreasing resource utilization and increasing Grid users' response time. The question is that how we can minimize the number of preemptions taking place in a resource sharing environment. In this paper, we propose a preemption-aware scheduling policy based on the queuing theory for a virtualized multi-cluster Grid where the number of preemptions is minimized. Simulation results indicate that the proposed scheduling policy significantly decreases the number of virtual machine (VM) preemptions (up to 22.5%).

1 Introduction

Resources provisioning for user applications is one of the main challenges in the Resource sharing environments. Resource sharing environments enable sharing, selection, and aggregation of resources across several Resource Providers (also called clusters in this paper), which are connected through high bandwidth network connections. Nowadays, heavy computational requirements, mostly from scientific communities, are supplied by these resource providers. Examples of production-level resource providers include DAS-2 [5].

Virtual Machine (VM) technology has emerged to enable another style of resource management based on the *lease* abstraction. Due to advantages of this form of management for resource sharing environments [8], we consider a virtualized multi-cluster environment in this paper. Typically, in large-scale resource sharing environments (e.g. InterGrid [3]) computational resources in each cluster are shared between external (Grid) users and local users. Hence, resource provisioning in resource sharing environments is done for two different types of users, namely: local users and Grid users. Local users (hereafter termed local requests), refer to users who ask their local cluster for resources. Grid users (hereafter termed Grid requests) are those users who send their requests to a gateway to get access to larger amount of resources. Typically, local requests have priority over Grid requests in each cluster [3]. In other words, the organization that owns the resources would like to ensure that its community has priority

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 419–432, 2011.

access to the resources. In this circumstance, Grid requests are welcome to use resources if they are available. Nonetheless, Grid requests should not delay the execution of local requests.

In our previous research [2], we proposed preemption of Grid requests in favor of local requests to remove this contention. We demonstrated that preemption decreases waiting time for local requests. However, the side-effects of preemption is twofold:

- From the system owner perspective, preemption imposes a considerable overhead to the underlying system and degrades resource utilization. This overhead is more notable in circumstances that VMs are used for resource provisioning [8].
- From the Grid user perspective, preemption increases the response time of the Grid requests.

Therefore, the main problem we are dealing with in this research is how to decrease the number of preemptions that take place in a multi-cluster Grid environment.

In this paper, we propose a preemption-aware scheduling policy for a virtualized multi-cluster Grid that distributes Grid requests amongst different clusters in a way that the number of preemptions minimizes. The proposed policy is based on the stochastic analysis of routing in parallel non-observable queues. This policy is not dependent to the availability information of the clusters and does not impose any overhead to the system. In summary our paper makes the following contributions:

- Proposing analytical queuing model based on the routing in parallel non-observable queues.
- Adapting the proposed analytical model to a preemption-aware scheduling policy.
- Evaluating the proposed scheduling policy under realistic workload models.

We consider this problem in the context of InterGrid. In the InterGrid each request has a type, number of VMs, duration, and the deadline (optional). We consider several types of Grid requests in InterGrid. These Grid requests can broadly be classified as Best-Effort (BE) and Deadline-Constraint (DC) requests. BE Grid requests can be preempted in favor of local requests. If there is not enough resources to start BE requests, they are scheduled in the first available time-slot. DC Grid requests cannot be preempted if the deadline is tight. Additionally, DC requests are rejected if there is not enough resources for them to start. BE Grid requests can be either *Cancelable:* which can be started at any time and is terminated in the case of preemption; or *Suspendable:* which can be started at any time and is rescheduled in later time-slot in the case of preemption. DC Grid requests can be *Migratable:* which are sent to another cluster inside the same Grid in the case of preemption; or *Non-preemptive:* which cannot be preempted at all. We also consider local requests of a cluster as Non-preemptive requests [2].

The rest of this paper is organized as follows: Proposed analytical queuing model is described in Section 2 which is followed by the preemption-aware

scheduling policy in Section 3. Performance of the proposed policy is reported in Section 4. Then, in Section 5 related research work are introduced. Finally, conclusion and future works are provided in Section 6.

2 Analytical Queuing Model

In this section we describe the analytical modeling of preemption in a multi-cluster Grid environment based on routing in parallel queues. This section is followed by proposing a scheduling policy in IGG (gateway) built upon the analytical model provided in this part.

The queuing model that represents a gateway along with several non-dedicated clusters (i.e. clusters with shared resources between local and Grid requests) is depicted in Figure. 1. According to this figure, there are N clusters in a Grid where each cluster j receives requests from two independent sources. One source is a stream of local requests with arrival rate λ_j and the other source is a stream of Grid requests which are sent by the gateway with arrival rate $\hat{\Lambda}_j$. The gateway receives Grid requests from other peer gateways [3] (G_1,..,G_g in Figure 1). Therefore, Grid request arrival rate to the gateway is $\Lambda = \bar{\Lambda}_1 + \bar{\Lambda}_2 + ... + \bar{\Lambda}_g$ where g indicates the number of gateways that potentially can send Grid requests to the gateway. Submitted local requests to cluster j must be executed on cluster j unless the requested resources is occupied by another local request or a non-preemptable Grid request. The first and second moment of service time of local requests in cluster j are τ_j and μ_j, respectively. On the other hand, a Grid request can be allocated to any cluster but it might get preempted later on. We consider θ_j and ω_j as the first and second moment of service time of Grid requests on cluster j, respectively. For the sake of clarity, Table 1 gives the list of symbols we use in this paper along with their meaning. Indeed, the analytical model aims at distributing the total original arrival rate of Grid requests (Λ) amongst clusters. In this situation if we consider each cluster as a single queue and the gateway as a meta-scheduler that redirects each incoming Grid request to one of the clusters, then the problem of scheduling Grid requests in the gateway can be considered as a routing problem in distributed parallel queues.

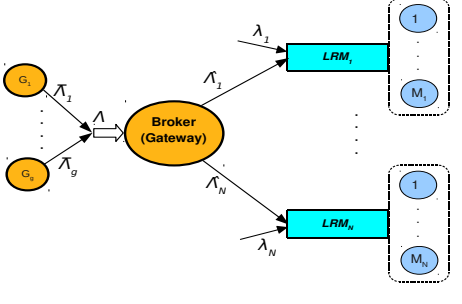


Fig. 1. Queuing model for resource provisioning in a Grid with N clusters

Table 1. Description of symbols used in the queueing model

Symbol	Description
N	Number of clusters
M_j	Number of computing elements in cluster j where $1 \leq j \leq N$
$\bar{\Lambda}_j$	Original arrival rate of Grid requests to cluster j
$\hat{\Lambda}_j$	Arrival rate of Grid requests to cluster j after load distribution
Λ	$= \sum_{i=1}^{g} \bar{\Lambda}_i = \sum_{j=1}^{N} \hat{\Lambda}_j$
θ_j	Average service time of a Grid request on cluster j
ω_j	Second moment of Grid requests service time on cluster j
γ_j	$= \theta_j \cdot \hat{\Lambda}_j$
R_j	Average response time of Grid requests on cluster j
λ_j	Arrival rate of local requests to cluster j
κ_j	Arrival rate of local requests plus Grid requests to cluster j
τ_j	Average service time of local requests on cluster j
μ_j	Second moment of local requests service time on cluster j
ρ_j	$= \tau_j \cdot \lambda_j$
m_j	$= \frac{\hat{\Lambda}_j}{\kappa_j}\omega_j + \frac{\lambda_j}{\kappa_j}\mu_j$
u_j	Utilization of cluster j $(= \gamma_j + \rho_j)$
r_j	Average response time of local requests on cluster j
η_j	Number of VM preemptions that happen in cluster j
T	Average response time of all Grid requests
T_j	Average response time of Grid requests on cluster j

Considering the mentioned situation, the goal of the scheduling in the gateway is to schedule the Grid requests amongst the clusters in a way that minimizes the overall number of VM preemptions in a Grid. Therefore, our primary objective function can be expressed as follows:

$$\min \sum_{j=1}^{N} \eta_j \tag{1}$$

However, minimizing the response time of requests is easier than the number of preemptions in such a system. Furthermore, more researches have been undertaken in similar circumstances to minimize the response time.

The most related research has been carried out by Li [6]. He has analyzed the load distribution problem in a cluster in the presence of two types of requests namely, local (dedicated) and Grid (generic) requests. Nonetheless, Li's goal of optimization is minimizing the response time of Grid (generic) requests whereas our goal is minimizing the overall number of preemptions. The other significant difference is that Li has solved the problem for a single cluster whereas our problem is in the context of a multi-cluster Grid. Li has mentioned the analysis of a multi-cluster system as a future direction of his work. From this perspective, our research can be considered as the future work of Li's research.

Although there are even more differences between our problem and the problem investigated by Li, we believe that this analysis can still be modified and applied to solve our problem. More specifically, from the results of some initial experiments as well as results of our previous research [2] we noticed an association between response time and number of VM preemptions in the system. To assess the strength of association between response time and number of VM preemptions we performed regression analysis between the two factors. Result of the regression analysis shows a positive correlation between number of VM preemptions in a cluster and response time of Grid requests (regression equation:

$R = 3.09 + 0.012\eta$ where R and η indicate the response time of Grid requests and number of VM preemptions). The regression analysis acknowledges that decreasing response time can be also applied for the purpose of minimizing the number of VM preemptions. Details of the modified analysis is discussed over the next paragraphs.

To minimize the average response time of Grid requests we should minimize:

$$T = \frac{1}{\Lambda} \sum_{j=1}^{N} \hat{\Lambda}_j \cdot T_j \tag{2}$$

where the constraint is: $\hat{\Lambda}_1 + \hat{\Lambda}_2 + ... + \hat{\Lambda}_N - \Lambda = 0$. The response time of Grid requests for each cluster j (T_j) is worked out based on Equation 3 by assuming each cluster j as an M/G/1 queue [6]:

$$T_j = \frac{1}{1-\rho_j} \left(\theta_j + \frac{\kappa_j m_j}{2(1-u_j)} \right) \tag{3}$$

Lagrange multiplier method is used to minimize Equation 2. By solving the above minimization problem, input arrival rate of each cluster is calculated based on the Equation 4:

$$\hat{\Lambda}_j = \frac{(1-\rho_j)}{\theta_j} - \frac{1}{\theta_j} \sqrt{\frac{(1-\rho_j)(\omega_j(1-\rho_j)) + \theta_j \lambda_j \mu_j}{2\theta_j(1-\rho_j)z + (\omega_j - 2\theta_j^2)}} \tag{4}$$

where z is the Lagrange multiplier.

Considering that $\Lambda = \hat{\Lambda}_1 + \hat{\Lambda}_1 + ... + \hat{\Lambda}_N$, then z can be calculated using the following Equation:

$$\sum_{j=1}^{N} \frac{1}{\theta_j} \sqrt{\frac{(1-\rho_j)(\omega_j(1-\rho_j)) + \theta_j \lambda_j \mu_j}{2\theta_j(1-\rho_j)z + (\omega_j - 2\theta_j^2)}} = \left(\sum_{j=1}^{N} \frac{(1-\rho_j)}{\theta_j} \right) - \Lambda \tag{5}$$

In fact, Equation 5 expresses the relation between different parameters of the system in which z is unknown. By solving Equation 5 for all clusters and working out z, Equation 4 can be solved. However, finding a generic closed form solution in Equation 5 for finding z is impossible. Nonetheless, a numerical solution can be found by searching z in range of $[lb,ub]$ using a bisection algorithm [6]. For this purpose, considering that $\hat{\Lambda}_j \geq 0$ and from Equation 4 we can infer that:

$$z \geq \frac{\lambda_j \mu_j}{2(1-\rho_j)^2} + \frac{\theta_j}{(1-\rho_j)} \tag{6}$$

Therefore, for all $1 \leq j \leq N$ the lower bound (lb) of the interval is:

$$lb = \max_{j=1}^{N} \left(\frac{\lambda_j \mu_j}{2(1-\rho_j)^2} + \frac{\theta_j}{(1-\rho_j)} \right) \tag{7}$$

If we define $\phi_j(z)$ according to Equation 8:

$$\phi_j(z) = \frac{1}{\theta_j}\sqrt{\frac{(1-\rho_j)(\omega_j(1-\rho_j)) + \theta_j\lambda_j\mu_j}{2\theta_j(1-\rho_j)z + (\omega_j - 2\theta_j^2)}} \tag{8}$$

and considering Equation 5, then we have:

$$\sum_{j=1}^{N}\phi_j(lb) \geq \left(\sum_{j=1}^{N}\frac{(1-\rho_j)}{\theta_j}\right) - \Lambda \tag{9}$$

The upper bound also can be worked out based on Equation 10. ub can be reached by doubling lb up until the condition is met.

$$\sum_{j=1}^{N}\phi_j(ub) \leq \left(\sum_{j=1}^{N}\frac{(1-\rho_j)}{\theta_j}\right) - \Lambda \tag{10}$$

If condition in Equation 9 is not met, then we have to decrease lb by removing clusters which are heavily loaded. Load of a cluster j is comprised of local requests that have been arrived and Grid requests which are already assigned to the cluster. The load can be calculated as follows.

$$\psi_j = \frac{\lambda_j\mu_j}{2(1-\rho_j)^2} + \frac{\theta_j}{(1-\rho_j)} \tag{11}$$

For the sake of simplicity, in Equation 12 we have assumed that $\psi_1 \leq \psi_2 ... \leq \psi_N$.

$$\sum_{j=1}^{k}\phi_j(\psi_k) \geq \left(\sum_{j=1}^{k}\frac{(1-\rho_j)}{\theta_j}\right) - \Lambda \tag{12}$$

It is worth mentioning that values bigger than k would not receive any Grid request from the gateway (i.e. $\hat{\Lambda}_{k+1} = \hat{\Lambda}_{k+2} = ... = \hat{\Lambda}_N = 0$).

3 Preemption-Aware Scheduling Policy

In this section we discuss how the analysis mentioned in previous section can be adapted as the scheduling policy for Grid requests inside IGG.

In fact, the analysis provided in Section 2 was based on some widely used assumptions. However, some of these assumptions do not hold for case of the multi-cluster that we are investigating. In the analysis we assumed that:

- each cluster was an M/G/1 queue. However, in InterGrid we are investigating each cluster as a G/G/M_j queue.
- all requests needed one VM. However, in InterGrid we consider requests that need several VMs for a certain amount of time.
- local requests could preempt Grid requests. However, in InterGrid not all Grid requests are preemptable. In fact, if the Grid request is *Non-Preemptable*, it cannot be preempted by local requests.

– each queue is run in FCFS fashion. However, in order to improve the resource utilization we consider conservative backfilling method in the local schedulers.

Considering the above differences, we do not expect that the preemption-aware scheduling policy performs optimally. In fact, we are trying to examine how efficient the above analysis would be by substituting the above assumptions with some approximations.

To adapt the analysis in a way that covers requests that need several VMs we modify the service time of Grid requests on cluster j (θ_j) and local requests on cluster j (τ_j) in the following way:

$$\theta_j = \frac{\bar{v}_j \cdot \bar{d}_j}{M_j s_j} \tag{13}$$

$$\tau_j = \frac{\bar{\zeta}_j \cdot \bar{\varepsilon}_j}{M_j s_j} \tag{14}$$

where $\bar{v}_j$ and $\bar{d}_j$ show the average number of VMs needed and average duration of Grid requests. $\bar{\zeta}_j$ and $\bar{\varepsilon}_j$ show the average number of VMs needed and average duration of local requests. Finally, s_j shows the processing speed in cluster j. This change also affects second moment of service time for both local and Grid requests. We can use coefficient of variance ($CV = StDev/Mean$) to obtain the modified second moment. Assuming that CV is given, the second moment of service time for Grid and local requests on cluster j is calculated according to Equation 15 and 16, respectively.

$$\omega_j = (\alpha_j \cdot \theta_j)^2 + \theta_j^2 \tag{15}$$

$$\mu_j = (\beta_j \cdot \tau_j)^2 + \tau_j^2 \tag{16}$$

where α_j and β_j show the CV of Grid requests and local requests service time on cluster j respectively. The preemption-aware scheduling policy (PAP), which is built upon analysis of Section 2, is shown in the form of pseudo-code in Algorithm 1. According to Algorithm 1, at first ψ is calculated for all clusters. Then, in steps 3 to 9, to exclude the heavily loaded clusters, clusters are sorted based on the ψ value in the ascending order. Then, the value of k is increased up until condition defined in Equation 12 (step 6) is met. ub is found by starting from $2 \cdot lb$ and is doubled up until condition in step 12 is met. Steps 14-19 show the bisection algorithm mentioned in Section 2 for finding proper value for z. Finally, in steps 20 and 21 the arrival rate to each cluster is determined. Steps 22 and 23 guarantee that clusters $k+1$ to N, which are heavily loaded, do not receive any Grid request.

4 Performance Evaluation

4.1 Experimental Setup

We use GridSim a discrete event simulator, to evaluate performance of the scheduling policies. We consider a Grid with 3 clusters with 32, 64, and 128

Algorithm 1. Preemption-Aware Scheduling Policy (PAP)

Input: $\bar{\Lambda}_j$,θ_j,ω_j,λ_j,τ_j,μ_j, for all $1 \leq j \leq N$.
Output: $(\hat{\Lambda}_j)$ load distribution of Grid requests to different clusters, for all $1 \leq j \leq N$.

for $j \leftarrow 1$ **to** N **do**
  $\psi_j = \frac{\lambda_j \mu_j}{2(1-\rho_j)^2} + \frac{\theta_j}{(1-\rho_j)}$;
Sort (ψ);
$k \leftarrow 1$;
while $k < N$ **do**
  if $\sum_{j=1}^{k} \phi_j(\psi_k) \geq \left(\sum_{j=1}^{k} \frac{(1-\rho_j)}{\theta_j} \right) - \Lambda$ **then**
    break;
  else
    $k \leftarrow k + 1$;
$lb \leftarrow \psi_k$;
$ub = 2 * lb$;
while $\sum_{j=1}^{k} \phi_j(ub) > \left(\sum_{j=1}^{k} \frac{(1-\rho_j)}{\theta_j} \right) - \Lambda$ **do**
  $ub = 2 * ub$;
while $ub - lb > \epsilon$ **do**
  $z \leftarrow (lb + ub)/2$;
  if $\sum_{j=1}^{k} \phi_j(z) \geq \left(\sum_{j=1}^{k} \frac{(1-\rho_j)}{\theta_j} \right) - \Lambda$ **then**
    $lb \leftarrow z$;
  else
    $ub \leftarrow z$;
for $j \leftarrow 1$ **to** k **do**
  $\hat{\Lambda}_j = \frac{(1-\rho_j)}{\theta_j} - \frac{1}{\theta_j} \sqrt{\frac{(1-\rho_j)(\omega_j(1-\rho_j)) + \theta_j \lambda_j \mu_j}{2\theta_j(1-\rho_j)z + (\omega_j - 2\theta_j^2)}}$;
for $j \leftarrow k + 1$ **to** N **do**
  $\hat{\Lambda}_j = 0$;

nodes with homogeneous computing speed $s_j = 1000$ MIPS for all clusters. Each cluster is managed by an LRM and a conservative backfilling scheduler. Clusters are interconnected using a 1000 Mbps network bandwidth. We assume all nodes of each cluster as a single core with one VM. The maximum number of VMs in the generated requests of each cluster does not exceed the number of nodes in that cluster. We consider size of each VM, 1024 MB [10]. The overhead time imposed by preempting VMs varies based on the type of Grid leases involved in preemption [8]. For *Cancelable* leases the overhead is the time needed to terminate the lease and shutdown its VMs. This time is usually much lower than the time needed for suspending or migrating leases [8]. In our experiments, suspension time (t_s) and resumption time (t_r) are 160 and 126 seconds, respectively [8]. The time overhead for transferring (migrating) a VM with similar configuration is 165 seconds [10].

Baseline Policies. For the sake of comparison, we evaluate the proposed scheduling policy (PAP) against other two policies which are described below:

- Round Robin Policy (RRP): In this policy IGG distributes Grid requests between different clusters of a Grid in a round-robin fashion with a deterministic sequence. Formally, this policy is demonstrated as $\hat{\Lambda}_j = \Lambda/N$
- Least Rate Policy (LRP): In this policy the rate of Grid requests submitted to each cluster has inverse relation with arrival rate of local requests to that cluster. In other words, clusters that have larger rate of incoming local requests would be assigned less number of Grid requests by IGG. Formal presentation of the policy is as $\hat{\Lambda}_j = (1 - \frac{\lambda_j}{\sum_{j=1}^{N} \lambda_j}) \cdot \Lambda$

We have also implemented PAP with the following details:

- We assumed that in step 14 of Algorithm 1 the precision is 1 ($\epsilon = 1$).
- In Equations 15 and 16, to work out the second moment of service time for local and Grid requests, we assumed that in all clusters $\alpha_j = \beta_j = 1$ (i.e. CV of service time for both Grid and local requests is 1).
- We believe that users mostly request for Suspendable and Nonpreemptable types. Therefore, in the experiments we consider: BE-Suspendable:40%; BE-Cancelable:10%; DC-Nonpreemptable:40%; and DC-Migratable:10%. These request types are uniformly distributed in Grid requests.

Workload Model. In the experiments conducted, DAS-2 workload model [5] has been configured to generate two-day-long workload of parallel requests. This workload model is based on the DAS-2 multi-cluster in the Netherlands.

We intend to study the behavior of different policies when they face workloads with different characteristics. More specifically, we study situations where Grid requests have:

- different number of requested VMs: In this case for Grid requests, we keep average *duration=30* minutes and average *arrival rate=1.0*.
- different request duration: In this case for Grid requests, we keep average *number of VMs=3.0* and average *arrival rate=1.0*.
- different arrival rate: In this case for Grid requests, we keep average *number of VMs=3.0* and average *duration=30* minutes.

Each experiment is performed on each of these workloads separately for 30 times and the average of the results is reported. To generate these workloads, we modify parameters of DAS-2 model. Local and Grid requests have different distributions in each cluster. Based on the workload characterization [5], the inter-arrival time, request size, and request duration follow Weibull, two-stage Loguniform, and Lognormal distributions, respectively. These distributions with their parameters are listed in Table 2.

4.2 Experimental Results

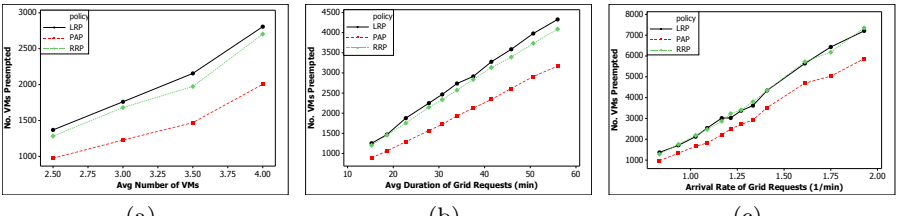
Number of VM Preemptions. As mentioned earlier, both resource owners and users benefit from fewer VM preemptions. From the resource owner

Table 2. Input parameters for the workload model

Input Parameter	Distribution	Values Grid Requests	Values Local Requests
No. of VMs	Loguniform	$(l = 0.8, 1.5 \leq m \leq 3, h = 5, q = 0.9)$	$(l = 0.8, m = 3, h = 5, q = 0.9)$
Request Duration	Lognormal	$(1.5 \leq a \leq 2.6, b = 1.5)$	$(a = 1.5, b = 1.0)$
Inter-arrival Time	Weibull	$(0.7 \leq \alpha \leq 3, \beta = 0.5)$	$(\alpha = 0.7, \beta = 0.4)$
P_{one}	N/A	0.2	0.3
P_{pow2}	N/A	0.5	0.6

perspective, fewer preemption leads to less overhead for the underlying system and improves the utilization of resources. From the user perspective, however, preempting Grid leases has different impacts based on the lease types. For Suspendable and Migratable leases, preemption leads to increasing completion time. For Cancelable leases preemption results in terminating that lease. Since users of different lease types have distinct expectation from the system, it is not easy to propose a common criterion to measure user satisfaction. Nonetheless, all types of leases Grid users suffer from lease preemption. Therefore, we believe that the number of VM preemptions in a Grid is a generic enough metric to express Grid users' satisfaction. In this experiment we report the number of VMs getting preempted by applying different scheduling policies. As we can see in all sub-figures of Figure 2, the number of VMs preempted almost linearly increases by increasing the average number of VMs (Figure 2(a)), duration (Figure 2(b)), and arrival rate of Grid requests (Figure 2(c)).

In all cases PAP outperforms other policies specially when the average number of VMs increases or when duration of Grid requests increases. Nonetheless, we observe less difference between the PAP and two other policies when the inter-arrival time of Grid requests increases (Figure 2(c)). In all cases the difference between PAP and other policies become more significant when there is more load in the system which shows the efficiency of PAP when the system is heavily loaded. In the best situation (in Figure 2(b) where the average duration of Grid requests is 55 minutes) we observe that PAP results in around 1000 (22.5%) less VM preemptions.



(a) (b) (c)

Fig. 2. Number of VMs preempted by applying different policies. By modifying (a) the average number of VMs, (b) the average duration, and (c) the arrival rate of Grid requests.

Resource Utilization. Time overhead due to VM preemptions leads to resource under-utilization. Therefore, we are interested to see how different scheduling policies affect the resource utilization. Resource utilization is defined as follows:

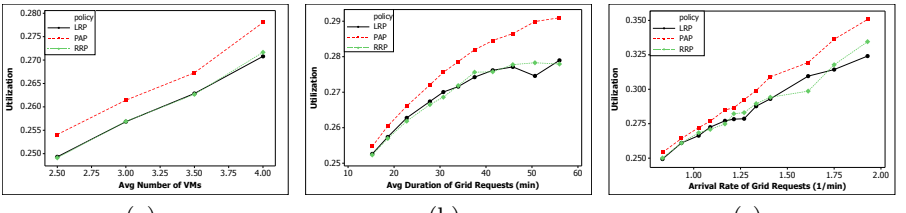
$$Utilization = \frac{computationTime}{totalTime} \tag{17}$$

where:

$$computationTime = \sum_{i=1}^{|L|} v(l_i) \cdot d(l_i) \tag{18}$$

where $|L|$ is the total number of leases allocated, $v(l_i)$ is the number of VMs in lease l_i, $d(l_i)$ is the duration of lease l_i.

In this experiment we explore the impact of preempting VMs on the resource utilization as a system centric metric. In general, resource utilization resulted from applying PAP is better than other policies as depicted in Figure 3. However, the difference is more remarkable when the average number of VMs or arrival rate of Grid requests increases (Figures 3(b) and 3(c)). We observe that PAP, which causes fewer preemptions, results in better resource utilization. In Figure 3(b), we can see that in all policies resource utilization becomes almost flat when Grid requests become long (more than 40 minutes). The reason is that when requests become long, the useful computation time dominates the overhead of VM preemptions. We can infer that VM preemption does not significantly affect resource utilization when requests are long (more than 40 minutes).



(a) (b) (c)

Fig. 3. Resource utilization resulted from different policies. By modifying (a) the average number of VMs, (b) the average duration, and (c) the arrival rate of Grid requests.

Average Response Time (ART). We are interested in ART metric to see how the investigated scheduling policies affect response time of Best-Effort Grid requests. In fact, this metric measures the amount of time on average a Best-Effort lease should wait beyond its ready time to get completed. ART in each cluster is calculated based on the Equation 19.

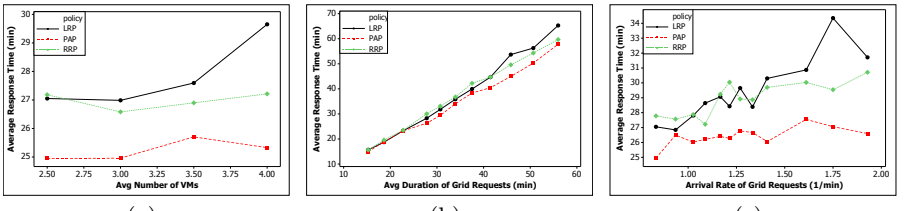
$$ART_j = \frac{\sum_{l \in \Delta}(c_l - b_l)}{|\Delta|} \tag{19}$$

where Δ is the set of Best-Effort leases. c_l and b_l show completion time and ready time for lease l, respectively. Then, ART over all clusters is the weighted average ART in each cluster.

According to the results in Figure 4, we conclude that PAP results in better ART for Grid requests. However, unlike the previous experiments, the response

time does not decrease significantly when the duration of the Grid requests increased (Figure 4(b)). The reason is that when the requests become longer, the duration and waiting times of requests normally become more dominant factor in response time comparing with the waiting times imposed because of preemption. Therefore, the number of VM preemptions is not significantly effective on average response time of the leases, particularly, when the average duration of leases is long.

We also conclude that ART does not change significantly by increasing the average number of VMs in the Grid requests (Figure 4(a) after 3.5) or their inter-arrival time (Figure 4(c) after 1.6). In fact in both cases by increasing average number of VMs of the Grid requests or their inter-arrival, more Deadline-Constraint Grid requests and even more local requests get rejected. This makes more places for other requests to fit in. Therefore, ART does not increase or even slightly decrease. For instance, in Figure 4(c), where the arrival rate for Grid requests is more than 1.6, we experience 13.5% improvement in ART.



(a) (b) (c)

Fig. 4. Average response time resulted from different policies. By modifying (a) the average number of VMs, (b) the average duration, and (c) the arrival rate of Grid requests.

5 Related Work

Assuncao et al. [3] have proposed adaptive partitioning of the availability times between local and Grid requests in each cluster. Each cluster submits its availability information to the IGG periodically. Therefore, there is a communication overhead between IGG and clusters for submitting availability information. Hence, there is a possibility that the availability information be imprecise.

Huedo et al. [9] have investigated the usage of multiple meta-schedulers to make loosely coupled connection between Grids. They use Gridway to migrate jobs from a remote cluster when the job does not get the expected processing power. However, they do not discuss how we can prioritize organization level requests versus requests coming from other Grids.

Haizea [8] is a lease scheduler which schedules a combination of advanced reservation and best effort leases. Haizea preempts best effort leases in favor of advance reservation requests. Sotomayor et al. [8], have also investigated the overhead time imposed by preempting a lease in Haizea. By contrast, we propose a scheduling policy to decrease the number of preemptions in the system.

Scojo-PECT [7] is a preemptive scheduler that aims at making a fair share scheduling between different job classes of a Grid. The approach is applying coarse-grain time sharing and suspending VMs on disk. However, the authors do not consider the overhead of suspending VMs on disk in their evaluations. The main difference with our work is the goal of scheduling. We minimize the number of VM preemptions whereas Sodan et.al's goal is fair share scheduling.

Amar et al. [1] have added preemption to cope with the non-optimality in on-line scheduling policies. The preemption policy prioritize jobs based on their remaining time as well as the job's weight. Our research is different with this work in the sense that they do not consider the lease based resource provisioning. Moreover, we try to minimize the number of preemption in a Grid where several types of Grid requests coexist.

Kettimuthu et al. [4] proposed a preemption policy, which is called Selective Suspension, where an idle job can preempt a running job if the suspension factor is adequately more than running job. The authors do not specify how to minimize the number of preemptions, instead, they decide when to do the preemption.

6 Conclusions and Future Work

In this research we proposed a preemption-aware scheduling policy (PAP) in IGG, as a virtualized multi-cluster resource sharing environment, that minimizes the side-effects of VM preemptions. Experimental results indicate that PAP resulted in up to 1000 less VM preemptions (22.5% improvement) comparing with other policies in a two-day-long workload. This decrease in number of VM preemptions improves the utilization of the resources and decreases average response time of the Grid requests (up to 13.5%). We believe that our policy is extensively applicable in lease-based Grid/Cloud resource providers where requests with higher priority coexist with other requests. A nice application is in Cloud (IaaS) providers where there is certain priorities between different users; and resource owners tend to minimize the number of VM preemptions. In future we plan to investigate how IGG can consider deadline and other QoS issues in its scheduling. Another extension would be considering co-allocation of the incoming Grid requests on different clusters to further decrease the number of preemptions.

References

1. Amar, L., Mu'alem, A., Stößer, J.: The power of preemption in economic online markets. In: Altmann, J., Neumann, D., Fahringer, T. (eds.) GECON 2008. LNCS, vol. 5206, pp. 41–57. Springer, Heidelberg (2008)
2. Amini Salehi, M., Javadi, B., Buyya, R.: Resource provisioning based on leases preemption in intergrid. In: Proceeding of the 34th Australasian Computer Science Conference (ACSC 2011), Perth, Australia, pp. 25–34 (2011)
3. de Assunção, M.D., Buyya, R.: Performance analysis of multiple site resource provisioning: Effects of the precision of availability information. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 157–168. Springer, Heidelberg (2008)

4. Kettimuthu, R., Subramani, V., Srinivasan, S., Gopalsamy, T., Panda, D.K., Sadayappan, P.: Selective preemption strategies for parallel job scheduling. Intl. Journal of High Performance Computing and Networking 3(2/3), 122–152 (2005)
5. Li, H., Groep, D.L., Wolters, L.: Workload characteristics of a multi-cluster supercomputer. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 176–193. Springer, Heidelberg (2005)
6. Li, K.: Optimal load distribution in nondedicated heterogeneous cluster and grid computing environments. J. System Architecture 54, 111–123 (2008)
7. Sodan, A.: Service control with the preemptive parallel job scheduler scojo-pect. Journal of Cluster Computing, 1–18 (2010)
8. Sotomayor, B., Keahey, K., Foster, I.: Combining batch execution and leasing using virtual machines. In: Proceedings of the 17th International Symposium on High Performance Distributed Computing, New York, NY, USA, pp. 87–96 (2008)
9. Vázquez-Poletti, J.L., Huedo, E., Montero, R.S., Llorente, I.M.: A comparison between two grid scheduling philosophies: Egee wms and grid way. Multiagent Grid Syst. 3, 429–439 (2007)
10. Zhao, M., Figueiredo, R.: Experimental study of virtual machine migration in support of reservation of cluster resources. In: Proceedings of the 3rd International Workshop on Virtualization Technology in Distributed Computing, pp. 5–11. ACM, New York (2007)

Performance Evaluation of Open Source Seismic Data Processing Packages

Izzatdin A. Aziz, Andrzej M. Goscinski, and Michael M. Hobbs

School of Information Technology, Faculty of Science and Technology,
Deakin University, Australia
{ia,amg,mick}@deakin.edu.au

Abstract. In many businesses, including hydrocarbon industries, reducing cost is of high priority. Although hydrocarbon industries appear able to afford the expensive computing infrastructure and software packages used to process seismic data in the search for hydrocarbon traps, it is always imperative to find ways to minimize cost. Seismic processing costs can be significantly reduced by using inexpensive, open source seismic data processing packages. However, hydrocarbon industries question the processing performance capability of open source packages, claiming that their seismic functions are less integrated and provide almost no technical guarantees for one to use. The objective of this paper is to demonstrate, through a comparative analysis, that open source seismic data processing packages are capable of executing the required seismic functions on an actual industrial workload. To achieve this objective we investigate whether or not open source seismic data processing packages can be executed using the same set of seismic data through data format conversions, and whether or not they can achieve reasonable performance and speedup when executing parallel seismic functions on a HPC cluster. Among the few open source packages available on the Internet, the subjects of our study are two popular packages: Seismic UNIX (SU) and Madagascar.

Keywords: High performance computing, open source seismic data processing packages, Seismic data processing, Seismic UNIX, Madagascar.

1 Introduction

The overall process of discovering hydrocarbon traps, starting with geological exploration through to seismic data processing, is very expensive and time consuming [1]. Seismic data consists of signal reflection points which are acquired during a typical geological mapping operation of the Earth Subsurface. The processing of seismic data is computationally exhaustive and includes activities such as: signal amplitude adjustment, noise filtering and geometrical correction of the multitude of seismic signal reflection points. The computational time required to process a large real-world seismic dataset may take up to 6 months to complete. In this scenario the shortage of oil and gas production relies on how soon seismic data could be computationally processed. The ability for an oil and gas company to perform seismic computation at higher speeds would provide a considerable advantage over competitors, in the race to discover new hydrocarbon reservoirs.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 433–442, 2011.

Currently, hydrocarbon industries have been using specialized commercial software packages to perform seismic processing, running on high performance computer (HPC) clusters. Such specialized commercial software packages are very expensive [2]. According to [3], commercial seismic data processing software packages are priced at 3.15 Million USD for 5 licenses for a 5 year term. An alternative approach to commercial software packages is through the use of open source seismic data processing packages.

Sequences of seismic functions are used by geophysicists on seismic datasets for analysis and interpretation. Core seismic functions are available in both commercial and open source packages. However, commercial software packages contain seismic functions arranged in an integrated form featuring enhanced graphical layout. In this paper, we performed seismic data processing based on the sequences of seismic functions as recommended in [4]. Although a number of guidelines to process seismic data have been written by geophysicists, the sequences of seismic functions recommended in [4] have served as a credible reference for the Society of Exploration Geophysics (SEG).

Open source seismic data processing packages provide a vast collection of seismic computational functions, which are consistently being enhanced and improved by the geophysical community [5]. The constant improvement made by researchers from the geophysical community allows updated research outcomes to be embedded into the packages. SU and Madagascar are among the most popular open source seismic data processing packages available on the Internet [5],[6]. Both packages have been widely used by academics and researchers as learning aids and analysis tools [7],[8].

To date there is no indication that a comparative study to evaluate the performance of SU and Madagascar executing on a HPC cluster has been conducted. The goal of this paper is to report on the outcomes of a comparative study and performance evaluation of how the open source seismic data processing packages, SU and Madagascar, execute a sequence of seismic functions on a HPC cluster. The significance for carrying out a comparative and performance evaluation study on open source seismic data packages executing on a HPC cluster are as follows. First, we would like to investigate whether the open source seismic data processing packages are able to execute a representative dataset using a complete sequence of seismic functions on a HPC cluster. Second, we would like to demonstrate the ability of SU and Madagascar to be executed on a HPC cluster.

The rest of the paper is organized as follows. Section 2 reviews related work in this area. Section 3 introduces the sequence of seismic functions that are part of both packages, SU and Madagascar, and also a basic required set of functions in commercial packages. Section 4 describes the performance evaluation which includes data format conversions, execution and results' analysis and discussion. In section 5, we conclude the paper by providing the research outcomes and some insight into our future work.

2 Related Work

The hydrocarbon industry has little confidence in using open source seismic data processing packages as a cheaper substitute for industrial scale seismic processing [10]. This is due to the perception that open source packages are just a collection of seismic functions without appropriate the function integration and support as expected

in commercial software packages [6]. Although minor, this perception diminishes the confidence level of the majority of hydrocarbon industries towards open source seismic data processing packages, without knowing the full potential of the powerful seismic functions possess [11]. Therefore it is important to demonstrate the ability of open source seismic data processing packages to execute industrial scale seismic computational processes using functions provided by the open source packages.

Performance evaluation of SU has been carried out on Beowulf clusters [12]. However, the evaluation only included a single seismic function; the noise filtering function. Furthermore, the work did not demonstrate a complete set of seismic data functions which limits to a judgement on the capability and performance of open source seismic data processing packages.

Functionality wise, both SU and Madagascar possess similar sets of seismic processing functions [8],[13], which can easily be compared through their user manuals and source code. Having similar sets of seismic functions, SU and Madagascar, will then allow us to perform a comparable performance study. One disadvantage is that each package only accepts its own specific data format. Although efforts have been made by the Society of Exploration Geophysics (SEG) to standardize the seismic data format by using the SEGY format (which is to be used for all seismic data processing packages) [15], yet producers of seismic packages tend to design their own specific formats that better suit their applications. However, format conversions from SEGY, into SU and Madagascar formats is possible by executing specific format conversion functions provided by both packages when dealing with each datasets, respectively.

Seismic datasets vary in sizes depending on the granularity of details embedded in the dataset [14]. For instance, a seismic dataset with higher array dimensions increases the data size significantly. A full scale seismic dataset used by the hydrocarbon industry ranges from 3 TBytes up to 1 PBytes in size [16]. The ability of seismic dataset to be decomposed into smaller sets, with fewer dependencies on each other, allows data parallel computation to be carried out on a HPC cluster.

3 Sequence of Seismic Functions

The comparative study of both packages' execution was carried out using the seismic functions shown in Table 1. Among many geophysical functions available in open source seismic data processing packages, we have chosen the functions represented in table 1 because they are the common actions required for industrial grade processing of seismic data as recommended in [4]. Table 1 shows 8 seismic computational functions labelled F_1 up to F_8 and their corresponding descriptions. Both SU and Madagascar are equipped with these computational functions. These functions have to be executed in the specific order as shown in table 1.

The most computationally exhaustive seismic function is the Post-Stack Time Migration, F_8. Each signal reflection[1] points needs to be geometrically corrected to provide an accurate signal reading.

[1] Signal reflection is used during seismic data acquisition operation to geologically map the Earth subsurface [8].

Table 1. Sequence of seismic functions

Function (F)	Function Name	Description of Each Function
F_1	AGC	Adjust signal's amplitude and power
F_2	Muting	Eliminate extraordinary signal that do not match with primary signal reflections
F_3	Noise Filtering	Removing low pass frequency noise due to signal's refraction and reverberations
F_4	Static Correction	Correction of signal reflection arrival time due to transmitter and receiver displacements.
F_5	Velocity Filter	Removal of signal's near surface noise due to air coupling effects
F_6	NMO	Removal of offset[2] dependency for each signal travel time
F_7	Seismic Trace Stacking	Increases Signal to Noise Ratio by stacking of all signal travel time recording with zero offset values
F_8	Post-Stack Time Migration	Geometrical correction on signal reflectivity to give its true signal reflection point

According to [6], 1 PBytes of industrial scale seismic data took at least a month to compute using F_8 on a large HPC cluster of 128 nodes, with 1024 cores processors.

4 Performance Evaluation

In order to achieve our goal to perform a comparative and performance evaluation study on open source seismic data packages, SU and Madagascar, we have set two tasks. The two tasks replicate the steps of real world industrial work to process seismic datasets.

The first task is to perform seismic data format conversions. The original fragment of seismic data that we have obtained is in the SEGY[3] format. Each seismic data processing package, such as SU and Madagascar, uses a specific data format with different structures and memory arrangement. SU data is arranged in binary sequences, whereby Madagascar data is structured in an array of 1, 2 or higher dimensions. Format conversion from SEGY into SU and Madagascar seismic data formats is necessary prior to performing any seismic data computation for both packages. In this first task, we perform a study on the conversion between seismic data formats, SEGY into SU and Madagascar, and measure the processing time taken to accomplish each format conversion and dataset sizes of converted inputs.

The second task is to perform a comparison of the sequential and parallel execution on both packages, SU and Madagascar, on a common dataset. The performance study was carried out on a HPC cluster running both a sequential and parallel version of each package using the same seismic dataset following the same sequence of seismic functions. The parallel CPU execution time of both packages, SU and Madagascar, are benchmarked with the sequential execution time. Both packages' performance and speedup were then compared.

---

[2] Offset in seismology refers to the displacement of the signal source and receivers. The constant displacement of both source and receiver causes a signal reflection point reading to overlap with its previous recording [17].

[3] SEGY format is the most dominant seismic data format [9]. Released by the Society of Exploration Geophysicist (SEG) in 1975, hence the name SEGY; it is an open format controlled by a technical committee.

The outcome of performing these two tasks is to show that open source packages, such as SU and Madagascar, are capable of providing high performance computation of seismic data processing on HPC clusters. Additionally, it will show that the performance of different open source packages, such as SU and Madagascar, vary even though both are executed using the same dataset, infrastructure and functions.

4.1 SU and Madagascar Seismic Data Format

Table 2 shows the comparative analysis of both packages data formats, SU and Madagascar.

Table 2. Comparative study of Madagascar and SU file formats [7],[8],[13]

Data Format	Madagascar	Seismic UNIX
Header	Called the meta-information, describes the whole dataset in an ASCII character. Header may reside in a separated file from the actual seismic segment or in one file in a stream.	Called the header, it consists of 240 bytes of binary header describing the seismic segment following it. This header appears before each binary data segment.
Data Sequence	Contains seismic traces in binary value arranged in an array hypercube dimension. Data can be of any type such as integer, real or complex, based on the machine the application is executed.	Contains series of seismic traces written in one of the 4 possible types of 32 binary float formats. The type binary float data is created based on the machine that SU is executed on.
Generic example of data sequence	Meta-information→data→data…	Header→data→Header→data→...

The Madagascar data format consists of two segments, which are the meta-information and data segment. The meta-information describes the basic information about the dataset, and the data segment contains the actual seismic data in the form of a binary sequence. SU data formats are structured similarly to the Madagascar format. The header file in SU contains basic descriptions of the data, similar to the meta-information segment in Madagascar. The header file in SU and meta-information in Madagascar contain the followings elements; the name of the program and the source from where the data was originally created; the size of data in bytes and the data type.

The Madagascar file format consists of an extra parameter which is the number of array elements and its dimension. This extra information is used by Madagascar memory management to support arrays with 2, 3 or higher dimensions, where as SU data is arranged in a series of seismic traces. Madagascar's format imposes less restriction on the arrangement of the header and data segment.

4.2 Seismic Data Conversion

The seismic data used in this experiment is a representative[4] portion of a real world readings obtained from an oil and gas company[5]. The size of the dataset is 122

[4] According to the company, a representative data size is such that can be used for processing yielding significant geological result.

[5] The name of this company is not disclosed due to privacy concern and because the hydrocarbon reservoir is still subject to a revisit in the future.

GBytes in size. The initial data format is in SEGY. Format conversions were carried out to obtain SU and Madagascar data formats prior to executing the seismic functions in a sequential and a parallel computing environment. The data format conversion process was carried out sequentially on a single node of 8 cores processor. However, the sequential algorithm used only runs on a single core. Table 3 shows that the initial seismic data took 72 minutes to convert to the SU format. The data size after conversion into SU format is 122 GBytes, which is comparable to the initial data size. Note however, that there is a small difference in data size between SU and SEGY formats.

Table 3. Execution of seismic data conversions for format of packages; SU and Madagascar

Seismic Data Format Conversion	Conversion Execution Time (minutes)	Data Size after conversion (GBytes)
SEGY ⟶ SU	72	122
SEGY ⟶ Madagascar	63	115

The SEGY data is 3.6 Mbytes more than the SU data size. This is due to the fact that the SEGY data structure consists of 2 segments, the header segment and the seismic trace segment. The header segment is approximately 3600 bytes in size containing the description[6] of the SEGY data. The SU data structure does not have a separate header segment as such in SEGY and Madagascar. In fact the header segments in SU are joined together before each data sequence segment. This explains the minor difference of 3.6 Mbytes between both SU and SEGY formats. Table 3 shows that the initial seismic data took 63 minutes to convert to the Madagascar format. The data size after conversion from SEGY format into the Madagascar format is reduced to 115 GBytes, which is due to the fact that the Madagascar format is structured with less complexity and uses a contemporary memory arrangement approach [5].

4.3 Tests and Results

Both sequential and parallel executions were performed on the Deakin University Computer Cluster. The cluster consists of 20 physical nodes with each consisting of an Intel based dual 1.6 Gigahertz CPU. Each CPU in a node is made of quad core processors, which makes a total of 8 cores of processor per node. Each node is allocated 8 Gigabytes of memory. These nodes are interconnected by a 10 Gigabit Infiniband network. The computer cluster runs on Centos Linux operating systems and uses SUN Grid Engine version 6.1 to perform job queuing and management. The computer cluster is deployed as such that 10 physical nodes are used to support 20 virtual nodes and the remaining 10 act as physical nodes. The data storage system uses the RAID 5 technology through a 3Gbps data access[7] with a rotational speed of 1500 rpm.

[6] Description of SEGY data was discussed thoroughly in [8].

[7] Data is stored in a compressed form and real time decompression took place when data is accessed for processing. Therefore accessing data in the storage for read has been reduced to only small blocks.

The tests were designed such that each function F_1 up to F_8 in the specified sequence was executed for both open seismic data processing packages, SU and Madagascar. Both packages execute firstly on a single node to provide the sequential performance. The sequential execution only uses a single core processor of a node. Subsequently, both packages were executed in parallel on 2, 4, 6, 8, and 10 nodes using all 8 CPU cores from each node.

The execution of the seismic functions by each package uses the previously converted respective dataset (from section 3.3). Since the seismic dataset that we used can be decomposed[8] into smaller sets, this allows us to perform data parallelization. To obtain better performance, jobs were created to match the maximum number of CPU cores on each node, for each execution. For instance, 2 nodes consist of 16 CPU cores, therefore 16 threads were created when executing in parallel. A maximum of 80 threads were created when executing on 10 nodes with 80 available CPU cores. A one to one computational mapping of each thread with each CPU core was applied. Table 4 summarizes the test result of both sequential and parallel executions of the SU and Madagascar packages on the same dataset in their respective data format.

Table 4. Sequential and Parallel executions of seismic functions for SU and Madagascar

Function (F)	**Seismic Data Processing CPU Execution Time (in Minutes)**											
	SU (Number of Nodes)						**Madagascar (Number of Nodes)**					
	Seq.	**2**	**4**	**6**	**8**	**10**	**Seq.**	**2**	**4**	**6**	**8**	**10**
F_1	21	16	10	6	8	8	25	12	6	4	6	5
F_2	32	21	13	8	7	9	24	15	7	3	4	6
F_3	179	96	54	29	22	18	182	87	31	12	12	10
F_4	62	38	26	17	13	15	95	42	18	7	7	8
F_5	481	312	179	84	67	59	378	196	93	32	18	21
F_6	541	341	210	92	76	81	219	184	83	39	21	18
F_7	16	7	5	4	3	3	28	7	5	5	6	7
F_8	9312	6215	4126	2319	1852	1603	7825	4652	2073	1291	1025	983
Total	**10644**	**7046**	**4623**	**2559**	**2048**	**1796**	**8776**	**5195**	**2316**	**1393**	**1099**	**1058**

The total sequential execution time for SU is 10644 minutes, which is approximately 7 days, while Madagascar consumed 8776 minutes, approximately 6 days, to complete the seismic functions F_1 up to F_8. The obvious similarity is that both took longer execution time to complete function F_8. As expected, based on [17] and [18], Post-Stack Time Migration is a computationally exhaustive process. Each binary sequence in the dataset needs to be computed, and therefore took a long time to complete. Figure 1 shows the comparative CPU execution performance for both SU and Madagascar.

[8] A master-slave approach was use to perform data parallelization on the cluster, in which each slave node is allocated equal portion of seismic data to be processed and return to the master node when processing is completed.

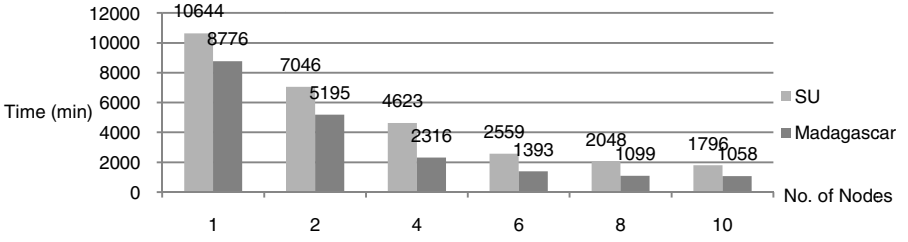


Fig. 1. Number of nodes against CPU execution time for SU and Madagascar

Figure 1 shows that both packages demonstrated significant performance improvement when more nodes were added. However, in all runs, Madagascar shows faster processing time as compared to SU. It is also observed that the best performance for the test is when executing on 10 nodes. CPU executionspeedup [12] was measured for both packages, SU and Madagascar, to execute the seismic functions.

Figure 2 shows the speedup of SU and Madagascar when executing the sequence of seismic functions from 1 up to 10 nodes. It shows that executing on more than 10 nodes could give better results in terms of speedup and CPU execution time. Madagascar shows better speedup of 8.29, as compared to SU's speedup of 5.93 when executing at 10 nodes.

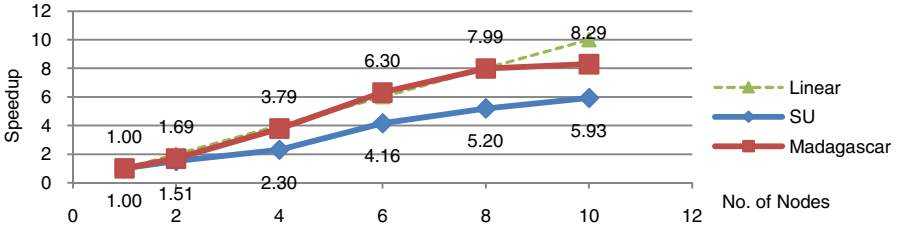


Fig. 2. Nodes execution speedup for both SU and Madagascar

A near-linear speedup result was achieved when executed with 2 nodes up to 8 nodes for both packages, SU and Madagascar, while a super linear[9] speedup reading of 6.30 was achieved when executing with 6 nodes by Madagascar. From our observation, the job dissemination for each node was equally distributed; therefore each node was processing the same portion of data size and completed processing at approximately the same time. With the sequential experiment, it appears that the test was experiencing a considerable I/O load. This overhead may be the reason why near-linear speedup was achieved by the Madagascar experiment between 2 nodes up to 8 nodes (shown in Figure 2), since little overhead in terms of I/O communication between nodes exists in this computation.

[9] Each node consists of 8 CPU cores and a total of 8 GBytes of available RAM. The collection of 8 GBytes of cache memory may have caused locality of reference, hence reduces the memory access time to the primary storage disk significantly.

It is also observed that in the seismic functions' computation, all segments of the threads were gathered from each node and consolidated at the end of the parallelization process. Therefore, the more nodes were added, the more the threads consolidation process needs to done at the end of the whole processing period. This explains the trivial additional improvement of speedup reading for both packages when reaching the end of the whole processing period.

5 Conclusion

Hydrocarbon industries appear to be capable of affording the expensive software packages and even more costly computing infrastructures required to process seismic datasets. However, the reduction of costs is the aim of many hydrocarbon industries. One of the approaches to decrease cost is through reduction of seismic data processing and computational overheads. Reduction of costs in seismic data processing can be achieved by using either open source seismic data processing packages SU and Madagascar, because they incur practically no cost to use, and exploit the power of parallel processing.

The goal in this paper was to demonstrate the capability of open source packages, SU and Madagascar, to execute a sequence of seismic functions representing the actual industrial work process. We succeeded in this by conducting two sets of tasks. First, the investigation of the problem, whether or not open source seismic data processing packages can be executed using the same set of seismic data through data format conversions. Second, whether or not they can achieve reasonable performance and speedup when execute parallel seismic functions on a HPC cluster.

The first task, to convert the original seismic data format into the respective format of SU and Madagascar, was successfully conducted and measured. The conversion was done using the seismic format conversion function provided by both packages. We now know that data format conversion for both packages, SU and Madagascar is possible and took at least 63 minutes for a 122 GBytes of seismic dataset. We successfully completed the second task, which was to conduct a performance evaluation on both packages by deploying them on a HPC cluster and executing a sequence of seismic functions in a sequential and parallel environment, using the same dataset with the respective package data formats.

To our knowledge, this paper is the first attempt to compare and contrast between open source seismic data processing packages, SU and Madagascar, by replicating industrial grade processing on a representative real world seismic dataset. The focus of this paper is on the computational analysis of processing seismic data on a HPC cluster. Therefore, the geophysical outcome of processing such as the accuracy of identifying hydrocarbon traps in the Earth subsurface is outside the scope of this paper.

The performance results show that good speedup can be achieved by using open source seismic data processing packages when executing on a HPC cluster in a parallel system. With achieving such a result it is hoped that the hesitation held by the hydrocarbon industries towards the capabilities of open source seismic data processing packages could be alleviated.

It has been demonstrated in this paper that open source seismic data processing packages are capable of being executed on a HPC cluster. In future work we would like to explore an approach where seismic data processing can be tested and deployed

on a more scalable and larger distributed computing environment, which is not viable through HPC clusters. Executing on a more scalable distributed computing environment has the potential to reduce the computational cost of seismic data processing even further.

References

1. Argonne National Laboratory: Laser Oil & Gas Well Drilling: Using high-power lasers to drill for gas & oil (2010)
2. Jianwei, M., Plonka, G., Chauris, H.: A New Sparse Representation of Seismic Data Using Adaptive Easy-Path Wavelet Transform. IEEE Journal of Geosciences and Remote Sensing 7(3) (2010)
3. Sesimic Micro Technology Inc.: Cost, Saving and Analysis Results (2010), http://www.seismicmicro.com/roi/roi_sample.htm
4. Yilmaz, O.: Seismic Data Analysis. Society of Exploration Geophysicists 1 and 2 (2001) ISBN 1560800941
5. Sergey, F., Felix, H., Paul, S.: Reproducible Research in Computational Geophysics. RSF School and Workshop, Vancouver (2006)
6. Ibrahim, N.A.: Expert Interview Session. Senior Geophysicist PETRONAS Research Sdn Bhd. (2010)
7. Cohen, J.K.: The New SU User's Manual. Colorado School of Mines Center for Wave Phenomena. The Society of Exploration Geophysicists 3(107) (2002)
8. Izzatdin A.A., Goscinski, A.: The Study of Seismic UNIX in Relation to Reflection Seismology Models. School of Information Technology Technical Report TR C10/2. Deakin University Australia (2010)
9. Barry, K.M., Cavers, D.A.: Recommended standards for digital tape formats. Journal of Geophysics 40(2), 344–352 (1975)
10. Glenn, C., Igor, M., Shannon, B.: Towards a Comprehensive Open-source System for Geophysical Data Processing and Interpretation. Canadian Society of Exploration Geophysicists (CSEG) Recorder (2007)
11. Glenn, C., Igor, M.: Integrated software framework for processing of geophysical data. Journal of Computer and Geosciences 32(6), 767–775 (2006)
12. Abdul, A.I., Thayalan, S., Nazleeni, H., Hasan, M.H., Mazlina, M.: Parallelization of Noise Reduction Algorithm for Seismic Data on a Beowulf Cluster. IJCSNS International Journal of Computer Science and Network Security 10(1) (2010)
13. Izzatdin A.A., Goscinski, A.: The Study of Madagascar Seismic Data Processing Package in Relation to Reflection Seismology Models. School of Information Technology Technical Report TR C10/5. Deakin University Australia (2010)
14. Izzatdin, A.A., Goscinski, A., Hobbs, M.: A Comparative Study of Open Seismic Data Processing Packages. School of Information Technology Technical Report TR C11/2. Deakin University Australia (2011)
15. Michael, W.N., Alan, K.F.: SEGY Rev 1 Data Exchange Format. Society of Exploration Geophysicists Technical Committee. SEG Technical Document Release 1.0 (2002)
16. Lin, D.: Optimizing Data Storage and Management for Petrel Seismic Interpretation and Reservoir Modeling: A White Paper for Upstream Oil and Gas Data Managers. Storage Systems Schlumberger Information Solutions (2009)
17. Telford, W.M., Gelbert, L.P., Sherff, R.E., Keys, D.A.: Applied Geophysics. Cambridge University Press, Cambridge (1990)
18. Scales, J.A.: Theory of Seismic Imaging. Samizdat Press, Golden Colorado (1997)

Reputation-Based Resource Allocation in Market-Oriented Distributed Systems

Masnida Hussin, Young Choon Lee, and Albert Y. Zomaya

Center for Distributed and High Performance Computing,
School of Information Technologies
The University of Sydney
NSW 2006, Australia
mhus9339@uni.sydney.edu.au,
{young.lee,albert.zomaya}@sydney.edu.au

Abstract. The scale of the parallel and distributed systems (PDSs), such as grids and clouds, and the diversity of applications running on them put reliability a high priority performance metric. This paper presents a reputation-based resource allocation strategy for PDSs with a market model. Resource reputation is determined by availability and reliable execution. The market model helps in defining a trust interaction between provider and consumer that leverages dependable computing. We also have explicitly taken into account data staging and its delay when making the decisions. Results demonstrate that our approach significantly increases successful execution, while exploiting diversity in tasks and resources.

Keywords: Dynamic resource allocation, resource reputation, market model, reliable performance.

1 Introduction

Parallel and distributed computing systems (PDSs) provide efficient resource sharing in tackling large-scale problems [1, 2]. However, their heterogeneity in resource and administration hinders their effective use. Due to the fact that each administrative domain has its own resource usage pattern [3, 4] an evaluation of the resource behavior varies. Also, the information of resource availability and performance in the scheduler is time delayed and potentially inaccurate [5]. Such unpredictable and imprecise resource behavior inherently results in unreliable task execution. The resource allocation problem is further complicated when each task needs input data for execution. Thus, resource allocation requires dependable communication links for transferring the data. In order to select trustworthy resources, our resource allocation approach incorporates a market model. The model for PDSs [6, 7] involves the definition of a computational market in which resource providers and consumers interact without a prior knowledge of underlying schedules. We focus on designing a reputation-based resource allocation strategy with a market model for which the decision is made taking into account conditions of scalability, heterogeneity and dynamism.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 443–452, 2011.

Specifically, our approach is capable of dealing with a wide variety of resource behaviors and performance fluctuations for improving system reliability. The trustworthiness of resource primarily relies on the resource's reputation where the degree of reputation significantly helps in allocation decisions in terms of reliable performance. We do not explicitly consider monetary factor to assign the task into the suitable resource. It is because the amount of 'money' could not be sufficient for evaluating the reliability of services [5]. With the market model it helps define a trust interaction between providers and consumers that creates benefit (incentives) for encouraging both parties to remain interacting. Our task scheduling approach also considers link capacity during data transmission for making optimal decisions. The proposed approach has been evaluated in a simulated large-scale computing environment. It significantly contributes to providing good-quality allocation decisions.

The reminder of this paper is organized as follows. A review of related work is presented in Section 2. In Section 3 we describe the models used in the paper. Section 4 details our reputation-based resource allocation strategy. Experimental settings and performance metrics are presented in Section 5. In Section 6 we present the experimental results. Finally, conclusions are made in Section 7.

2 Related Work

In large-scale distributed systems with heterogeneous and dynamic resources, performance modeling/prediction techniques are often adopted for various reasons including reliability. Reputation is a good way of motivating compositions in such systems to interact and collaborate with previously unknown and potentially malicious entities for providing reliable performance. The concept of reputation in task scheduling and resource allocation is popular in P2P networks [8] and also has been applied to computational grids and clouds for reliable services (e.g., [9-11]). The concept of trust in resource behavior for dealing with variability and instability of compute nodes was proposed in [8-10], where reputation is defined based on various factors, such as prior performance, network capacity and resource availability. These approaches have demonstrated the effectiveness in maximizing successful execution; however, the efficacy of the approaches in dealing with system dynamicity is limited to a certain level.

Inspired by the stability of interaction in real markets, market model have been studied as a metaphor for resource allocation in distributed systems [6, 7, 12, 13]. It is a promising platform to accomplish efficient resource sharing and allows an organization of limited resources to overcome resource scarcity. The work in [6] proposed incentive-based resource scheduling that optimizes incentives for providers and consumers; and this scheduling leads to a sustainable market. It uses a price-adjusting algorithm in order to gain fair allocation profit in all active providers. However, the communication overhead is ignored. Another approach to realize the optimal resource allocation is using a utility-based optimization scheme [7]. The model uses a two-level market solution with three types of agent: user, resource and service. Agents perceive supply and demand in the market through a price-directed algorithm. The utility of resource allocation is used to solve the scheduling problem where user preferences and benefits of service are summarized by the utility. While most resource

allocation approaches with a market model deal with pricing and payment issues, our approach performs resource allocation solely focusing on suitability between resource reputation and processing requirement while considering network capacity for enhancing dependable computing.

3 Models

3.1 System Model

The target system used in this work consists of a set S of *r* resource sites (Fig. 1) that are loosely connected by a communication network, and each has a data repository and several compute nodes given as n_j where $j = \{1, 2\ldots, x\}$. The bandwidth between any two individual sites varies corresponding to realistic network; thus the inter-site communications are heterogeneous. Nodes within the same site are fully interconnected through a high-bandwidth network. It is also assumed that the nodes are able to access the local data repository within a negligible amount of time.

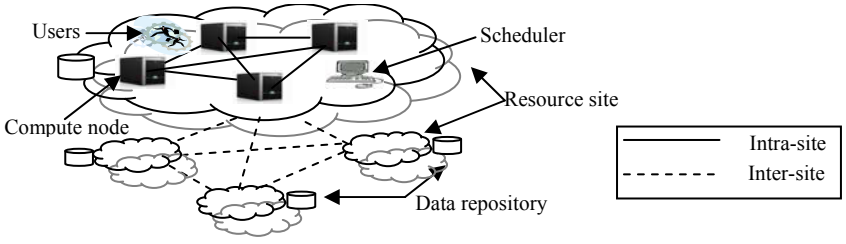


Fig. 1. System Model

Each node contains multiple processor cores with a shared cache module. The speed of cores in node n_j is homogenous and expressed in terms of million instructions per second (MIPS). Thus, the processing capacity of each node PC_j is defined as total number of tasks completed within some observation period, *L* divided by total speed of cores. It is assumed that the performance of a node in terms of processing capacity and communication capacity fluctuates. Therefore, the accurate (actual) execution time of a task *ExeT* on a particular node is difficult, if not impossible, to determine a priori. We assume that the number of tasks to be scheduled at a given time is more than the number of available nodes. Hereafter, the terms node and resource are used interchangeably.

3.2 Application Model

We model tasks that are assumed independent from each other. Since workload traces obtained from the actual systems that are publicly available in the Parallel Workload Archive [14], tasks in these traces are characterized as follows. (i) Submission time, *arv*: the timestamp at which a task arrives to the system queue, (ii) Requested time, *rt*: the estimated time of task execution that is expected by a user and (iii) Wall clock time, *wct*: the amount of time that the task has run in the system.

Each task t_i requires a different processing capacity for its completion; and this determines the priority of that task. We synthetically assign the priority to a given task that determines based on *rt* and *wct*. The priority of a task t_i is set to high if its *wct* is at least 70% of *rt*. If *wct* is at most 20% of *rt*, the priority is considered as low. Otherwise, the task is set to medium priority. The task requires input data (file) before the execution that is located in either local or remote data repository. Once a task assignment is made, its file will be transferred to the node on which that task is assigned to. Therefore, a task is associated with two types of constraint in its processing, i.e., processing requirement and expected data transfer time f_{tr}; that is defined as budget bg_i and simply given by $(wct/rt) + f_{tr}$. Note that input data and file in our work are interchangeable since these resources do not incur any database details.

3.3 Market Model

Our market model (Fig. 2) does not consider the pricing scheme, hence providers and consumers (trust) interaction concerns resource dedication and performance stability for successful execution. The market interaction establishes suitability and matching between consumer's requirement and provider's service. Consumers acquire resources according to a computing competence (reputation), and providers recommend computing services that suit to consumers' requirement (budget). The model also involves issues in optimizing benefit (incentive) for providers and consumers. We identify a successful execution as an incentive for a consumer and *market-strength* as an incentive for providers. Clearly, the consumer may want to strive for maximum satisfaction (total number of successful executions or meet deadlines). More formally,

$$satisfaction = \sum_{i=1} \omega_i; \quad where \quad \omega \begin{cases} 1 & if \quad ExeT_i \le rt_i \\ 0 & otherwise \end{cases} \tag{1}$$

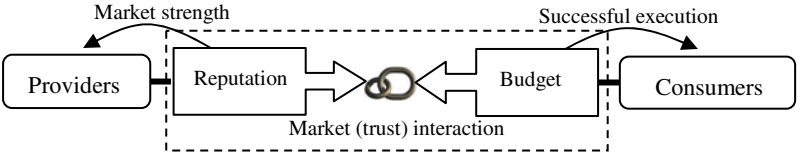


Fig. 2. Market Model

We introduce *market-strength* that denotes the capability of computing services of provider relative to competitive offerings. The provider has better *market-strength* relative to other providers if most of its nodes demonstrate good reputation. It is important for provider to highlight its reliability in services due to the consumer may not want their tasks mapped onto resources that are owned by unreliable entity. In this work, there is no extra cost or penalty applied to the provider if performance goal considering deadline constraint is not met. It is because the information of reputation is sufficient to measure and value the resource for future relationship.

4 Reputation-Based Resource Allocation

4.1 Formation of Resource Reputation

For the sake of reliability, the resource expresses its valuation of processing as a function of a computing competence (resource reputation). The reputation helps identify the suitability for a given task that concerns with its processing capacity and prior performance. The prior performance PP_j of a node j is defined as summation of success rate (*satisfaction*/*L*) and the resource availability rate *av_t*. Then, the resource reputation $rrep_j$ is computed by the prior performance of n_j divided by its processing capacity (i.e., PP_j / PC_j). For each provider, the average reputation value over all its nodes indicates its market strength.

4.2 The Suitability between Resource and Task

The consumer-provider relationship is established once a task is assigned to a resource. In our approach, the reliability of task execution primarily is the key determinant in the value of that relationship. To maximize the incentives (i.e., success rate, market strength), the fitness value between resource n_j and task t_i is analysed. That is:

$$fitval = \left| bg_i + rrep_j \right| \;/\; \left| bg_i - rrep_j \right| \tag{2}$$

A task is assigned to a resource that gives the highest *fitval* that is guaranteed to accomplish the execution to limit of its reputation. If there are two or more resources suitable for a task, the resource with the highest reputation value is chosen. The *fitval* contributes to the improvement of reliable execution if the processing capacity is actually realized.

4.3 Incorporation of Data Staging

Once an allocation decision has been made, the input data required by a task is staged (duplicated) into the node on which the task executes. The input data can be exists at a remote data repository, hence, its staging affects task execution to a certain degree due to heterogeneous communication links. Since a task only starts its execution when the required file is successfully transmitted to the node, the scheduling takes into account computing priority and link capacity. We incorporate an insertion scheme into the task scheduling in order for a task to be scheduled into the time slot between two consecutively scheduled tasks. For a given task, its start time of processing might be affected by the fluctuation in link capacity; more specifically, delay in data transmission (i.e., t_c = latency/bandwidth). Hence, we adopt a two-step queuing model (Fig. 3). In the priority-based scheduling, tasks are primarily scheduled and grouped according to their computing priority *pr*; i.e., high (*h*), medium (*m*) or low (*l*). The tasks in each priority are queued by their arrival time (in sequence, *ars*). File-based (re)scheduling is then carried out in the way that scheduled tasks are further inspected with respect to their t_c for possible rescheduling. This rescheduling is merely for the task in each pr without affecting tasks in other priorities. The task with minimum t_c is scheduled first. This repeats until no further improvements in the schedule is possible.

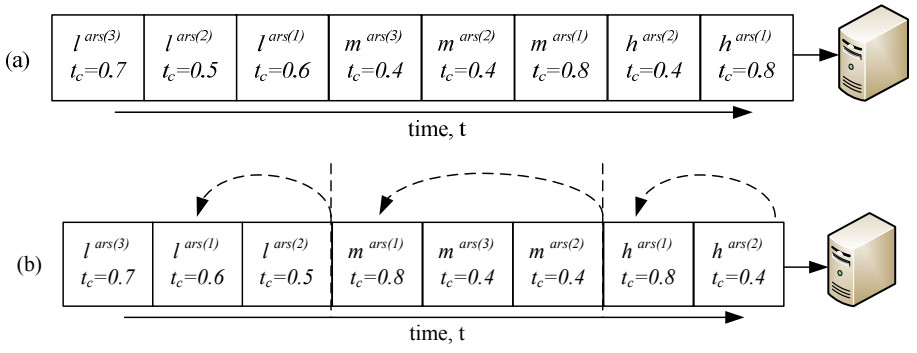


Fig. 3. The Two-step Queuing (a) Priority-based scheduling (b) File-based scheduling

5 Experimental Methodology

5.1 Experimental Settings

Log workloads extracted from the Parallel Workload Archive (PWA) [14] were used in our simulations. We have selected two trace sets: SDSC Blue and HPC2N. Our simulation used 5000 jobs from both workload traces after excluding trace job entries that have a negative run time. In our simulation system, there are 5 to 10 providers and each contains a varying number of compute nodes ranging from 4 to 10. The number of processor cores ranges from 2 to 8 with an interval of 2. The number of cores in a node is might vary from that in other nodes. Its speed is selected in the range of 500 and 1000MIPS. Estimated file transfer time is selected randomly from the following set: {0.01, 0.5, 1.0, 3.5, 5.0, 7.5, 10.0, 12.5}. Resource availability *av_t* is random and uniformly distributed within the range of 20% to 80%. We induce two network settings for inter-site communication. Specifically, the bandwidth and latency are randomly generated from a uniform distribution ranging from the following sets; Setting 1: {50 to 100} and {0.1 to 1}, and Setting 2: {10 to 50} and {0.01 to 1}, respectively. We varied the (average) inter-arrival times of jobs by multiplying the job submission times included in the trace files by the inverse of a load factor. This factor is increased by 0.1 increments from a low to a maximum value for 5000 jobs.

5.2 Performance Metrics

- **Successful execution rate:** It is defined as *satisfaction* /*L*, particularly to measure the degree of reliable execution for dealing with various priority tasks.
- **Utilisation rate:** We define the utilization rate of a resource as $RU = busy_j / (busy_j + idle_j)$ where $busy_j$ is the total time when the node n_j is busy for servicing tasks and $idle_j$ is total idle time of n_j, respectively.
- **Relative match:** This is used to measure the degree of suitability between task and resource for each allocation decision. It varies between 0 and 1 where 0 means conditional match and 1 means perfect match. It is given in Eq. 3. The decision is identified as α if *fitval* is more than average *fitval* (fit_{ave}); otherwise it is β.

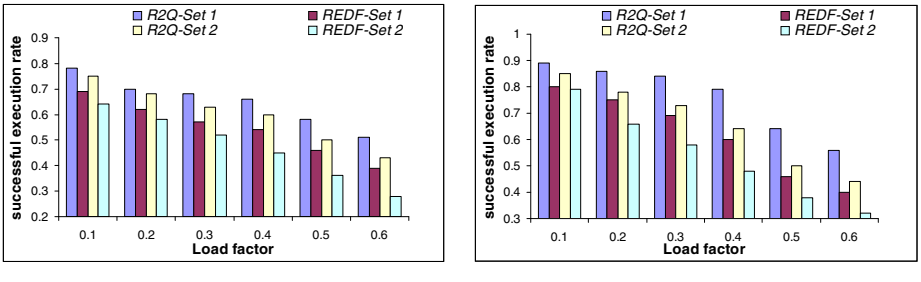
$$relMatch = \left(\sum_{i=1}^{J} |\alpha - \beta| \right) \Big/ \left(\sum_{i=1}^{J} |\alpha + \beta| \right) \quad (3)$$

6 Experimental Results

We first study how the performance of our algorithm (Reputation-RA) is influenced by network performance and job characteristics. The experiments have been conducted with two schemes: reputation with insertion (R2Q) and reputation without insertion (REDF). With REDF, tasks are sorted by their deadline (i.e., earliest deadline first) without considering the rescheduling. Those algorithms are evaluated with each of Setting1 and Setting2. Then, Reputation-RA is compared with extended versions of Utility-RA [6] and Incentive-RA [7]. Since Reputation-RA concerns input data, we have revised both algorithms to fit into our model. In the comparison, their equilibrium prices are obtained only when the communication link between providers for transferring the input-data is established. The Setting1 is used for this comparison.

6.1 Impact of Scheduling with Variability of Network Capacity

The pattern of performance fluctuation in Fig. 4 does not significantly differ as observed to be about 5%. There is a tendency of reliability growth towards variability on the communication links in HPC2N and SDSC Blue workload traces. However, the system performance advantage of R2Q-Setting1 over other policies is huge particularly in Fig. 4(b). The effectiveness of R2Q and REDF in gaining better performance is presented in Fig. 5. Specifically, R2Q-Setting1 still outperforms others. Interestingly, the performance shows different patterns as the load increases. The traces from HPC2N illustrate that relative match reduces linearly while the SDSC Blue is shown an exponential reduction curve, and relatively high in *relMatch* reaching nearly 90%. It is demonstrated that reputation-based allocation is able to work in such workload scenarios; this is particularly true in heavy loads. Results in Fig. 5(b) clearly show that REDF-Setting1 and R2Q-Setting2 have comparable performance with the difference being small (about 2% on average). It is because the larger bandwidth in REDF-Setting1 enables more input loads to be accurately matched. But, the rescheduling scheme in R2Q-Setting2 has given much more benefit for the market strength (Fig. 6).



(a) (b)

Fig. 4. Successful execution rate (a) HPC2N workload. (b) SDSC Blue workload.

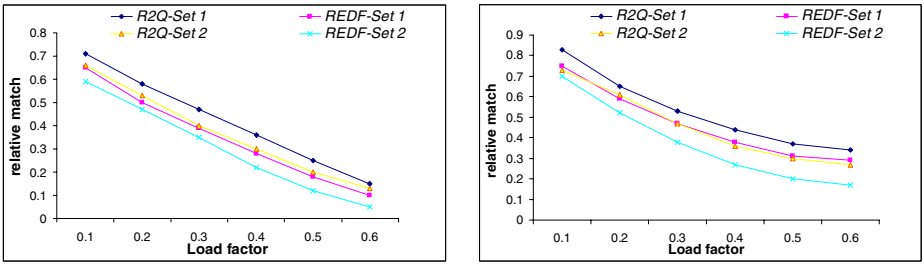


Fig. 5. Relative match (a) HPC2N workload. (b) SDSC Blue workload.

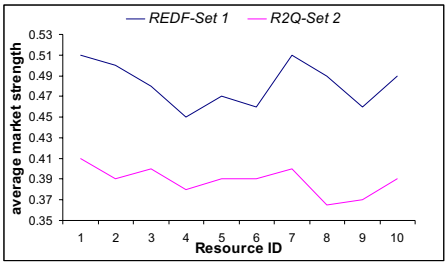
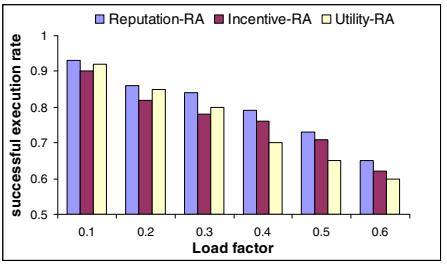


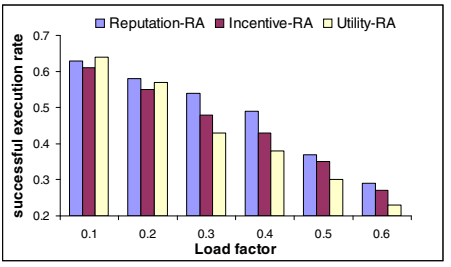
Fig. 6. Market strength of providers for SDSC Blue workload (Load factor =0.3)

6.2 Comparing Market-Based Resource Allocation Approaches

We enhance the analysis of the SDSC Blue Horizon as it meets our objective in that it contains the variability in task traces. Fig. 7(a) shows that all approaches have reached their success rate more than 50%. Notwithstanding this observation, it can be seen that results of Reputation-RA is very close to Utility-RA under least loads, and comparable with Incentive-RA under heaviest loads. It is because the pricing strategy in both approaches tends to deliver better performance at a particular load factor. Apparently, when we reduce *av_t* (approximately equal to 30%) there is significant degradation in success rate as shown in Fig. 7(b). Although the experiment ran on the high-bandwidth (Setting1), the degree of resource availability does influence those approaches to maintain reliable execution. Although payment agreement was not considered, Reputation-RA still showed good results that turned out to be the perfect allocation more than 52% on average, as shown in Fig. 8(a). Fig. 8(b) clearly illustrates the utilization rate do not significantly differ and the discrepancy of RU among the algorithms is small about 10% on average. In addition, the benefit of using a market-oriented distributed system for maximizing the resource utilization was not apparent – less than 80%. This is mainly because a resource with a good offer (i.e., less price or high availability) is more popular to be chosen to accommodate the input load that affects the results.

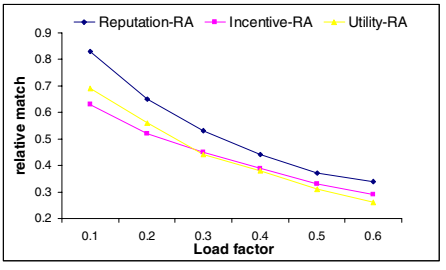


(a) av_t = 20% - 80%

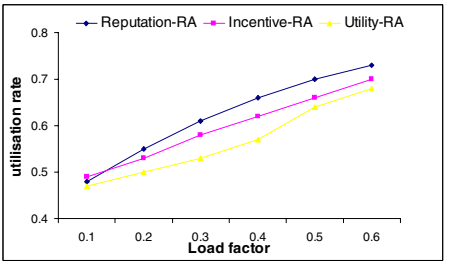


(b) av_t = 30%

Fig. 7. Successful execution rate under different resource allocation approaches



(a) Relative-match



(b) Utilization rate

Fig. 8. Performance comparisons with different resource allocation approaches

7 Conclusion

The emergence of distributed computing infrastructures with heterogeneous resources and diverse processing requirements has much increased the importance of reliable performance/execution. In this paper, we have presented a reputation-based resource allocation approach in a market-oriented distributed system for reliable execution. Our approach is devised in a way that actively adapts and deals with performance fluctuations and variability in resource capacity. Based on results from our extensive experiments, the approach has demonstrated a substantial positive effect on reliable performance under different workload conditions. We have confirmed that the incorporation of reputation factor into resource allocation is an effective means to ensure reliable performance in dynamic heterogeneous computing environments.

References

1. Lee, Y.C., Zomaya, A.Y.: Scheduling in grid environments. In: Rajasekaran, S., Reif, J. (eds.) Handbook of Parallel Computing: Models, Algorithms and Applications, pp. 21.1–21.19. CRC Press, Boca Raton (2008)
2. Czajkowski, K., Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (2003)

3. Eymann, T., Konig, S., Matros, R.: A Framework for Trust and Reputation in Grid Environments. Journal Grid Computing 6(3), 225–237 (2008)
4. Hussin, M., Lee, Y.C., Zomaya, A.Y.: ADREA: A Framework for Adaptive Resource Allocation in Distributed Computing Systems. In: 11th Int'l Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 50–57 (2010)
5. Dabrowski, C.: Reliability in Grid Computing System. Concurrency and Computation: Practice and Experience 21(8), 927–959 (2009)
6. Xiao, L., Zhu, Y., Ni, L.M., et al.: Incentive-Based Scheduling for Market-Like Computational Grids. IEEE Transaction on Parallel and Distributed Systems 19(7), 903–913 (2008)
7. Chunlin, L., Layuan, L.: A Utility-based Two Level Market Solution For Optimal Resource Allocation In Computational Grid. In: Proc. of the 34th Int'l Conf. on Parallel Processing (ICPP), Washington (2005)
8. Damiani, E., Vimercati, S.D.C.d., Paraboschi, S., et al.: A Reputation-based Approach for Choosing Reliable Resources in Peer-to-Peer Networks. In: Proc. of the 9th ACM Conf. on Computer and Communications Security, Washington, DC, USA, pp. 207–216 (2002)
9. Sonnek, J., Chandra, A., Weissman, J.B.: Adaptive Reputation-based Scheduling on Unreliable Distributed Infrastructures. IEEE Transaction on Parallel and Distributed Systems 18(11), 1551–1564 (2007)
10. Liang, Z., Shi, W.: A reputation-driven scheduler for autonomic and sustainable resource sharing in Grid computing. Journal Parallel and Distributed Computing 70(2), 111–125 (2010)
11. Hwang, K., Kulkareni, S., Hu, Y.: Cloud Security with Virtualized Defense and Reputation-based Trust Management. In: 8th IEEE Int'l Conf. on Dependable, Autonomic and Secure Computing, Chengdu, China, pp. 717–722 (2009)
12. Casavant, T.L., Kuhl, J.G.: A Taxonomy of Scheduling in general-purpose Distributed Computing Systems. IEEE Transaction on Software Engineering 14(2), 141–154 (1988)
13. Shetty, S., Padala, P., Frank, M.P.: A Survey of Market-based Approaches to Distributed Computing. University of Florida, Florida (2003)
14. PWA: Parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

Cooperation-Based Trust Model and Its Application in Network Security Management

Wu Liu[1], Hai-xin Duan[1], and Ping Ren[2]

[1] Network Research Center of Tsinghua University Beijing, P.R. China
[2] School of Economics & Management, Chongqing Normal University, P.R. China
liuwu@ccert.edu.cn

Abstract. This paper presents a User Cooperation Trust Model (UCTM) which not only encourages the good behavior liberally, but also punishes those minatory behaviors decisively. In this model, a malicious user will be kicked out from the network system and refused from accessing resources of the network automatically when the user's reputation is lower than some threshold.

Keywords: Trust, Computer Network, Security Management, User Behavior.

1 Introduction

Reputation system [1][2] emerges as a viable solution for evaluating the trustworthiness of users in some environments such as online e-commerce markets and communities. The basic idea of reputation system is to have a mechanism for rating the users' behaviors on various aspects, a way for collecting these ratings from other users, and a method for computing the reputation values of the target users. The reputation system can achieve two benefits: first, to stimulate the users to provide good quality of services [10]; Second, to restrain bad behaviors which would blemish the profit of sincere users [11].

In our opinion, a reputation system is mainly composed of two components: one is the system architecture, which could be centralized or distributed; the other one is the reputation model which is considered as the core component. In this paper, we propose a User Cooperation Trust Model (UCTM), which not only encourages the sincere users to behave well continuously, but also punishes the adversaries immediately.

2 Related Work

Reputation system has become a hot research issue in the past few years, and there is a rapidly growing literature on the theory and application of reputation systems. Josang[2] and Marti[3] have made concrete surveys of reputation systems in the field of online services and P2P systems. In this section, we introduce some typical reputation systems.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 453–459, 2011.

eBay[4] is a popular auction site that allows sellers to list items for sale, and buyer to bid for these items. The so-called Feedback Forum on eBay gives buyers and seller the opportunity to rate each other as negative, neutral or positive (i.e. -1,0,1) after completion of transactions. eBay collects all the ratings and calculates the reputation values. The total reputation value of each user is the percentage of positive ratings compared with the total ratings[2]. Josang[5] developed and evaluated the beta reputation system for e-commerce markets based on β distribution by modeling reputation as posterior probability given a sequence of experiences. Xiong[6] presented the PeerTrust reputaton system for peer-to-peer electronic communities. It includes a coherent adaptive trust model for quantifying and comparing the trustworthiness of peers based on the feedback of transactions. EigenTrust[7] is a global history-based reputation system, which uses a distributed algorithm similar to PageRank[8] to compute a global reputation value for every user using individual transaction rating weighted by the reputation value of the recommend users[3].

The limitations of these reputation models include:

(1) They deal with users' behaviors separately, and do not take the continuance of their behaviros into consideration. Take the eBay system's reputation model as example, the reputation value of a user who provides 100 good services and that of a user who provides only 1 good service are both 100%, which is unfair to these users who have continuously provided good services for a long time.
(2) These reputation models do not punish the bad behaviors decisively, which leaves the opportunities for the adversaries to gain unwarrantable profit by taking bad behaviors. For example, a user could provide 99 good services at the beginning and then provide a bad service, which computes his reputation value as 99%, but other users might not be able to distinguish the difference between 100% and 99%, and accept them both as dependable.

3 User Cooperation Trust Model (UCTM)

In this section, we introduce the proposed reputation model: User Cooperation Trust Model (UCTM). We would like to give three principles that guide the design of this model firstly, and then we describe the reputation model in detail.

3.1 Design Principle

- ***Encouragement Principle:*** The first goal of UCTM is to encourage the users to provide good services all the time. If a user provides good services continuously, his reputation should be higher than that of who provides good services brokenly.
- ***Punishment Principle:*** The second goal of UCTM is to reduce the minatory behaviors such as providing bad services, in order to protect the profit of sincere users. If a user performs some minatory behavior, he should be punished immediately by reducing his reputation rapidly.
- ***Restriction Principle:*** combining the above principles with authentication and access control technology to restrict users action

3.2 The Reputation Model

In UCTM, reputations are represented in two levels: local and global reputation. Local reputation represents each individual user's direct experience of transactions with other users and the global reputation is the result by aggregating multiple local reputations from different recommenders.

Let $R_{i,j}$ denotes the local reputation resulting from direct experiences of user *i* on user *j*. For the *k*th transaction, we define the transaction rating $r_{i,j}^k$ as follows: $r_{i,j}^k \in (0,1]$, if the result of the transaction is good in some extent; And $r^k_{i,j}$ =0, if the result is bad.

$$r_{i,j}^k = \begin{cases} (0,1] & Success \\ 0 & Fail \end{cases} \tag{1}$$

The local reputation is updated based on the rating of the last transaction as follows.

$$R_{i,j} = \begin{cases} 0 & r_{i,j}^k = 0 \\ \lambda r_{i,j}^{k-1} + (1-\lambda) r_{i,j}^k & r_{i,j}^k \in (0,1] \end{cases} \tag{2}$$

Where λ, a value between 0 and 1, is called the history factor, which reflects the weight of history transactions in relation with the last one. $\lambda \approx 1$ means the history experiences have very high importance and the last transaction has little influence in the reputation evaluation; $\lambda \approx 0$ means the history experiences are merely forgotten. $r_{i,j}^k$ is the rating of the *k*th transaction.

As the proposed reputation model intends to encourage the users to continuously provide good services, we introduce the concept of *Latest Continuous Times of Good Service (LCTGS)* and *Permanence Function (PF)*.

3.2.1 User Behavior Inspiritment Factor in the Reputation Model

Definition 1. Latest Continuous Times of Good Service (LCTGS) is an integer variable which records the times of good service that a user continuously provides recently.

$$LCTGS(n) = \begin{cases} 0 & Failed\ at\ n^{th}\ transaction \\ LCTGS(n-1)+1 & Success\ at\ n^{th}\ transaction \end{cases} \tag{3}$$

For example, assuming the rating set of A to B is {1, 0.8, 0, 0.9, 0.4, 1},

LCTGS(1)=1, LCTGS(2)=2, LCTGS(3)=0,
LCTGS(4)=1, LCTGS(5)=2, LCTGS(6)=3
If the 7th rating is 0, then LCTGS(7)=0
Else if the 7th rating is greater than 0, then LCTGS(7)=LCTGS(6)+1=3+1=4

Definition 2. Permanence Function(PF) is a monotonic increase function which maps the user's *Latest Continuous Times of Good Service (LCTGS)* to a real value between [0, 1].

$$PF = f(LCTGS) \tag{4}$$

The *Permanence Function* should have the following mathematical properties:

a. $f(a)<f(b)$, whenever $0<a<b$.
b. $f(0)=0$, and $\lim_{x\to\infty} f(x)=1$

In this paper, we choose the normalized inverse tangent function as the *Permanence Function*:

$$PF = \frac{atan(LCTGS - a) + atan(a)}{\pi/2 + atan(a)} \tag{5}$$

3.2.2 The Reputation Model

From the view of user *i*, the global reputation of user *j* can be calculated as follows.

$$T_{i,j} = (1-\omega)R_{i,j} \cdot PF + \omega \frac{\sum_{l=1}^{N} R_{i,l}R_{l,j}}{N} \tag{6}$$

Where N is the number of recommenders. ω is called the recommendation factor. If $\omega \approx 1$, then the direct experience between user *i* and user *j* is not taken into consideration, while $\omega \approx 0$ means that user *i* just believes his own experience. And RF is the noise restriction factor which will discussed in § **4.2.3**

After the Global Reputation $T_{i,j}$ is calculated, user i will decide whether to transaction with user j based on the trust policy:

$$\begin{cases} Trust & if\ T_{i,j} \geq T \\ Distrust & otherwise \end{cases} \tag{7}$$

Where T is a predefined threshold or the value of the current transaction.

4 Simulation Detail

We implement a simulation framework to emulate a P2P file sharing system. This simulation framework is developed on the basis of RePast[9], which is a multi-agent simulation toolkit. The simulation is based on discrete time ticks. At each tick, every user is supposed to participate in a transaction with another user and rate each other after the transaction is completed. Although the proposed reputation model is simulated in a distributed manner in this paper, it could also be implemented in centralized systems.

In the simulation, each user can act as one of the three roles in a transaction: *Requester, Provider* and *Recommender*. The *Requester* asks for services, the *Provider* responds to these requests, and the *Recommender* is responsible for providing reputation values about both *Requester* and *Provider.*

The process of a transaction is depicted in Fig.1.

Step1. The *Requester* firstly sends QueryService(SERVICE) messages to other users in the system.

Step2. The *Providers* that has the requested service respond the request by replying QueryResult(SERVICE) messages.

Step3. The *Requester* chooses one of the provides and send QueryReputation(PROVIDER) messages to the recommenders.

Step4. The *Recommenders* reply the *Provider's* reputation value by sending ReputationValue(PROVIDER) messages.

Step5. The *Requester* aggregates the collected reputation values and valuates the trustworthiness of the *Provider*.

Step6. If the *Provider* is considered to be creditable, the *Requester* sends a RequestService(SERVICE) message to the *Provider*. Otherwise, the *Requester* chooses another *Provider* and repeats this process from Step3.

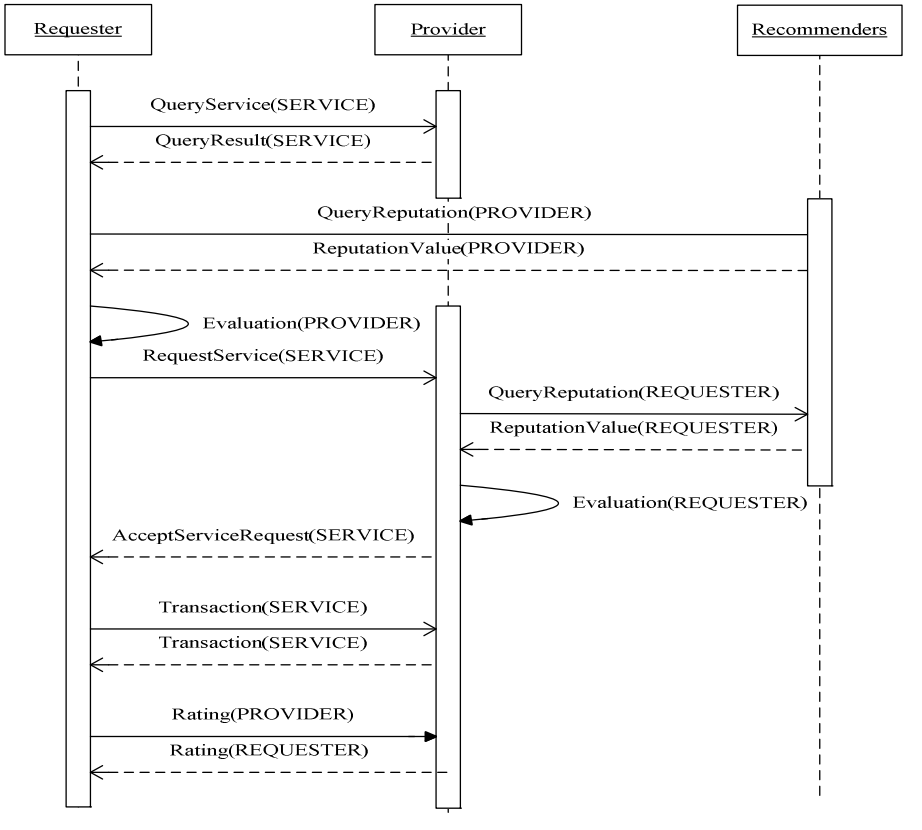


Fig. 1. Transaction Process

Step7, 8, 9. The *Provider* also estimates the *Requester's* trustworthiness by the same operation as described in Step3, 4, 5.

Step10. If the *Requester* is considered creditable, the *Provider* sends a AcceptServiceRequest(SERVICE) message to the *Requester*. Otherwise, he rejects the service request.

Step11, 12. The *Requester* and *Provider* complete the transaction with each other.

Step13, 14. After completion of the transaction, the *Requester* and *Provider* rate each other according to the result of the transaction.

5 Simulation Results

The objective of this simulation is to evaluate the effectiveness of the UCTM against the three bad behaviors (traitorous, malicious and collusive). In this scenario, the percentages of traitorous, malicious and collusive users vary from 5% to 10%.

Fig. 2 shows the Successful Transaction Percentage and Average Reputation Value of each type of users with 85% sincere users, 5% traitorous users, 5% malicious users and 5% collusive users, and Fig. 9 shows the Successful Transaction Percentage and Average Reputation Value of each type of users with 70% sincere users, 10% traitorous users, 10% malicious users and 10% collusive users. We could observe from these figures that the sincere users could participate in more transactions, get more good services and accumulate higher reputations by providing good services than the adversary users.

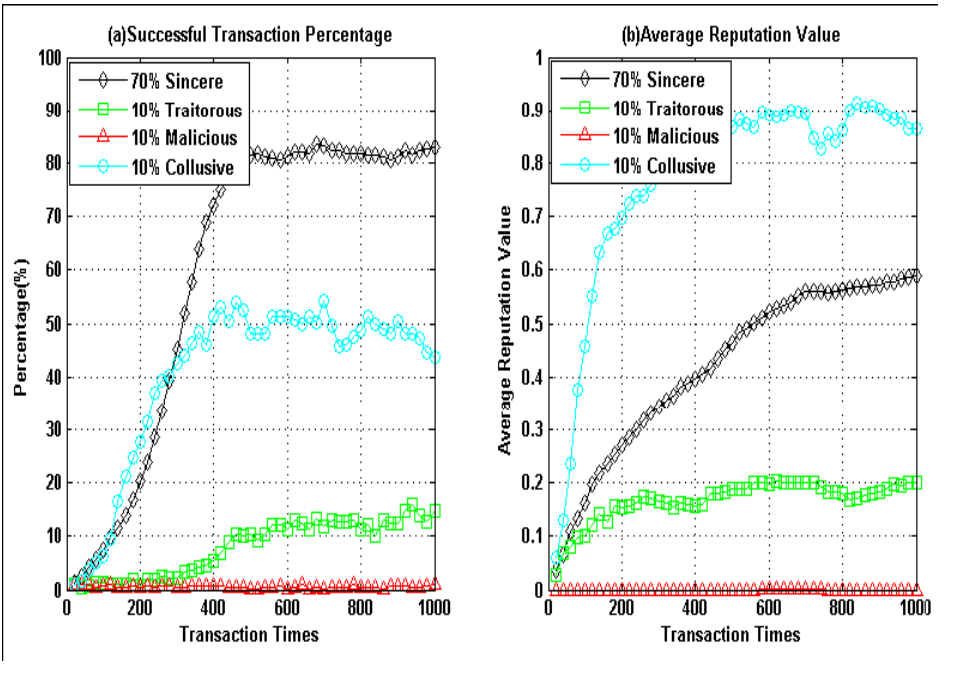


Fig. 2. *Successful Transaction Percentage* and *Average Reputation Value* with 10% malicious, 10% traitorous and 10% collusive users

6 Conclusion

In this paper, we first propose a new reputation model: User Cooperation Trust Model (UCTM), which not only encourages the user to provide good services continuously, but also punishes the minatory behaviors immediately. Then we develop a simulation framework to measure the effectiveness of our model, and the results show that the proposed reputation model can effectively resist against the common minatory behaviors.

Acknowledgments. This work is supported by grants from the National Natural Science Foundation of China (Grant No. 60203044, 90412010) , China 863 Project #2008BAH37B04 and Chongqing Science & technology Commission Program (CSTC2011AC2143).

References

1. Resnick, P., Kuwabara, K., Zeckhauser, R., Friedman, E.: Reputation systems. Communications of the ACM 43(12), 45–48 (2000)
2. Josang, A., Ismail, R., Boyd, C.: A survey of trust and reputation systems for online service provision. Decision SupportSystem 43(2), 618–644 (2007)
3. Marti, S., Garcia-Molina, H.: Taxonomy of trust: categorizing p2p reputation systems. Computer Networks 50(4), 472–484 (2006)
4. eBay. The world's online marketplace, `http://www.ebay.com/`
5. Josang, A., Ismail, R.: The beta reputation system. In: Proceedings of the 15th Bled Conference on Electronic Commerce, pp. 324-337 (2002)
6. Xiong, L., Liu, L.: PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities. IEEE Transactions on Knowledge and Data Engineering 16(7), 843–857 (2004)
7. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The EigenTrust algorithm for reputation management in p2p networks. In: Proceedings of the 12th International Conference on World Wide Web, pp. 640–651. ACM, New York (2003)
8. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web, Technical Report, Database Group (1998)
9. Repast Homepage, `http://repast.sourceforge.net/`
10. Rasmusson, L., Jansson, S.: Simulated social control for secure internet commerce. In: New Security Paradigms Workshop, pp. 18–26. ACM, New York (1996)
11. Adar, E., Huberman, B.A.: Free riding on gnutella, Technical Report (2000)

Performance Evaluation of the Three-Dimensional Finite-Difference Time-Domain(FDTD) Method on Fermi Architecture GPUs

Kaixi Hou, Ying Zhao, Jiumei Huang, and Lingjie Zhang

Information Centre, Beijing University of Chemical Technology, Beijing, China
{houkx,zhanglj}@grad.buct.edu.cn,
{zhaoy,huangjm}@mail.buct.edu.cn

Abstract. GPUs excel at solving many parallel problems and hence dramatically increase the computation performance. In electrodynamics and many other fields, FDTD method is widely used due to its simplicity, accuracy, and practicability. In this paper, we applied the FDTD method on the Fermi Architecture GPUs, the latest product of NVidia, for a better understanding of Fermi's new features, such as the double precision support and improved memory hierarchy. Then we make a comparison between the strategies using the shared memory, the traditional optimization method on GPUs, and using L1 cache. Next, the paper provides insights into the disparity of these two strategies. We demonstrate that parallel computations only using L1 cache can reach the similar or even better performance as the traditional optimization method using the shared memory does when the dataset is not too large or the frequency of repeated use of the related data is low.

Keywords: Fermi, finite-difference time-domain, shared memory, l1 cache.

1 Introduction

The finite-difference time-domain (FDTD) method, which was introduced by Kane Yee in his seminal 1966 paper [1], has already become one of the most important methods to obtain numerical solutions of Maxwell's equations in electromagnetics. Since about 1990, FDTD computational electromagnetics modeling has gained rapid popularization and been used successfully in many related fields, such as wireless communication simulation, radar signature technology, mobile phone safety studies, etc. [2] Despite many merits of FDTD formulation -not least its simplicity and accuracy- it is unbearable for most of us that significant computational resources including both of the CPUs and memory would be tied up with the tedious routine computation for an undetermined length of time.

Because of the fine discretization of the spatial and temporal domain in FDTD simulation, a whopping amount of memory resources and time would be consumed for the iterative computation after that. On the other hand, with the development of GPU and the introduction of CUDA [4], developers now can enjoy their novel scientific fruits and the benefits from the massively parallel architectural features of GPU co-processors [3]. In this context, the parallelization and acceleration of FDTD using GPU has become a hotspot in recent years.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 460–469, 2011.

Developers barely had time to calm down from the excitement of the promotion of their studies and scientific researches with GPU computing than NVidia Corporation released its next-generation GPU products, which use a creative new GPU computing architecture called "Fermi" [5]. Fermi changes the number of Stream Processors (SP, they are now called CUDA cores) per Stream Multiprocessors (SM), extends the cache size and completes the memory hierarchy. Also coupled with the continuous improvement of the underlying computing units, such as ECC support, more shared memory, faster context switching, and many more [8], Fermi has great difference compared with the NVidia's previous G80 and GT200 GPUs.

When optimized FDTD processes are needed on traditional GPUs, there are two main impediments: (a) the explicit use of shared memory and (b) the additional effort involved in design of data arrangement and efficient memory configuration. In this article, it is assumed that the readers have previously studied FDTD method. Then, the paper shifts the emphasis to the FDTD parallelization on Fermi and the performance analysis and comparison.

The rest of this paper is organized as follows. Frist, we briefly introduce the background of the FDTD method and the corresponding parallelization in section 2. Section 3 presents the simulation model. Performance analysis and comparison between two optimization strategies are described in detail in section 4. The last section is the conclusion of the paper and expectation.

2 Background

2.1 A Brief Overview of the FDTD Method

In this section we will make a brief introduction of the FDTD method based on 3-dimension Maxwell electromagnetic equations to help readers understand the parallel mechanism to be used in our experiments. For detailed discussions about the FDTD method, readers should refer to [2] to get detailed information.

Maxwell's curl equations in linear, isotropic, non-dispersive and lossy materials are given by

$$\frac{\partial \vec{H}}{\partial t} = -\frac{1}{\mu}\nabla \times \vec{E} - \frac{1}{\mu}\left(\vec{M}_{source} + \sigma^* \vec{H}\right) \tag{1}$$

$$\frac{\partial \vec{E}}{\partial t} = \frac{1}{\varepsilon}\nabla \times \vec{H} - \frac{1}{\varepsilon}\left(\vec{J}_{source} + \sigma \vec{E}\right) \tag{2}$$

where $\vec{E}$ is known as the electric field, $\vec{H}$ is the magnetic field, μ is the magnetic permeability, ε is the electrical permittivity.

Due to the independency of $\vec{J}$ and $\vec{M}$ sources -- electric current density and equivalent magnetic current density -- and the influence from the media, $\vec{J}_{source}$ and $\vec{M}_{source}$ are used to substitute original $\vec{J}$ and $\vec{M}$ with the help of the material parameters σ and σ^*, which represent the electric conductivity and equivalent magnetic loss respectively. Then six coupled scalar equations in Cartesian coordinates can be written out, in which the values of $\vec{E}$ and $\vec{H}$ are interdependent causing great

trouble for scientists. Yee algorithm and the famous Yee Cell [1] giving a second-order both in time and space FDTD method to Maxwell's curl equations, according to this, the algorithm interleaves the values of $\vec{E}$ and $\vec{H}$ at a discrete point in the grid known as Yee Cell and at a discrete point in time. The spatial structure of the electrical and magnetic fields can be compared to an intertwined double mesh [6], where each $\vec{E}$ component of x-y-z directions is circulated by four different $\vec{H}$ components and vice versa. So together with the introduction of the $\frac{1}{2}\Delta t$ time difference, the $\vec{E}$ and $\vec{H}$ components can be calculated at staggered time intervals respectively, for example, we first compute the $\vec{E}$ values at α time and then the $\vec{H}$ values at a half time interval shift $\alpha + \frac{1}{2}\Delta t$ can be computed consequently.

Along with the notation of $u_{i,j,k}^{n} = u(i\Delta x, j\Delta y, k\Delta z, n\Delta t)$ and semi-implicit approximation, we then have, taking the x-component of the electrical field for example,

$$E_x\big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}} = \left[\frac{1-\frac{\sigma_{i,j+\frac{1}{2},k+\frac{1}{2}}\Delta t}{2\varepsilon_{i,j+\frac{1}{2},k+\frac{1}{2}}}}{1+\frac{\sigma_{i,j+\frac{1}{2},k+\frac{1}{2}}\Delta t}{2\varepsilon_{i,j+\frac{1}{2},k+\frac{1}{2}}}}\right] E_x\big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n-\frac{1}{2}} + \left[\frac{\frac{\Delta t}{\varepsilon_{i,j+\frac{1}{2},k+\frac{1}{2}}}}{1+\frac{\sigma_{i,j+\frac{1}{2},k+\frac{1}{2}}\Delta t}{2\varepsilon_{i,j+\frac{1}{2},k+\frac{1}{2}}}}\right] \cdot$$

$$\left[\frac{H_z\big|_{i,j+1,k+\frac{1}{2}}^{n} - H_z\big|_{i,j,k+\frac{1}{2}}^{n}}{\Delta y} - \frac{H_y\big|_{i,j+\frac{1}{2},k+1}^{n} - H_y\big|_{i,j+\frac{1}{2},k}^{n}}{\Delta z} - J_{source_x}\big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n}\right] \tag{3}$$

Similar equations about the $\vec{E}$ and $\vec{H}$ components of other directions can be derived in the same way. The key point, however, is that we can compute E_x at any point of the space using the known values of E_x and H_x from previous known time period, which ensures that each value of $\vec{E}$ and $\vec{H}$ does not rely on its neighbors in the same period.

2.2 Parallelization of FDTD Method on Fermi Architecture GPUs

From the discussion above, we will naturally improve our FDTD programs to the direction of parallelization and thus the computation efficiency would be accelerated, though not to the same extent. In order to achieve reasonable parallelization and good performance, one should have a good understanding of Fermi architecture GPUs and CUDA (Compute Unified Device Architecture) programming model. Due to the fact that CUDA has been widely used in the parallel computing domain [14], the extent we introduce this part is limited to enabling readers to gain a better understanding of some new features that related to our experiment of Fermi architecture.

Despite many improvements of the novel Fermi architecture GPUs, such as ECC support, more shared memory, faster context switching, and many more [8], we will focus on the new cache hierarchy. As NVidia realized the significance of both shared memory and cache, Fermi provides for programmers a new memory hierarchy with configurable 64kB L1 cache / shared memory, which can be configured as either larger 48kB shared memory or opposite larger 48kB L1 cache. The following hierarchical graph explains this architecture.

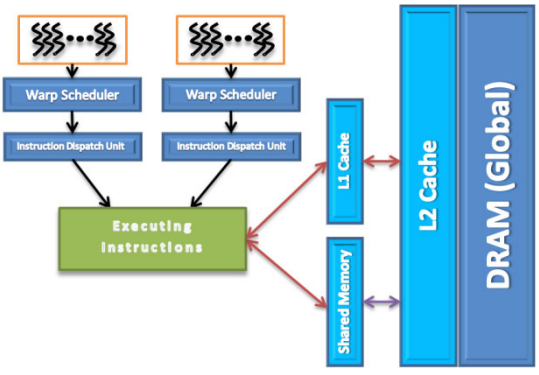


Fig. 1. Fermi's dual warp scheduler and memory hierarchy

As above *Figure 1* shows, Fermi adopts dual warp scheduler to enable diverse warps to be served concurrently. The waiting time when one warp needs to read memory can be utilized rationally using this mechanism of the model of dual-issue. By doing so, Fermi can take sufficiently advantages of the hardware.

When memory access happens during the instruction execution, threads have two ways to offset the memory access latency. The first is the traditional but widespread method having the data, to be used in the block, stored in the shared memory. The number of memory access --one of the bottlenecks of the GPU's computing performance-- can be reduced efficiently along with the introduction of the intermediate shared memory. But everything has gains and losses; the complexity of data organization rises correspondingly. On the other hand, according to [4], both of the L1 cache and shared memory are from the same on-chip memory leading to same access speed. Furthermore, the adoption of L1 cache rather than shared memory will not enhance the complexity of the program. In our experiment, two programs based on shared memory or L1 cache are developed for testing their performance.

3 Method

This section will focus on setting up a simulation in preparation for the following performance analysis.

3.1 Simulation Model

This experiment will set up a three-dimensional field ($128 \times 128 \times 128 = 2^{21}$) where the inside electrical fields and the magnetic fields are changed with time. Ideal electric conductors are assumed to exist at the boundaries of the region. In order to stimulate the outgoing radio waves, suppose the electric component E_z is at the center of the grid changing as differential Gaussian function with time going by. Electric fields curled around magnetic fields at varying points in time can be obtained as indicated in *Figure 2* of the simulation field.

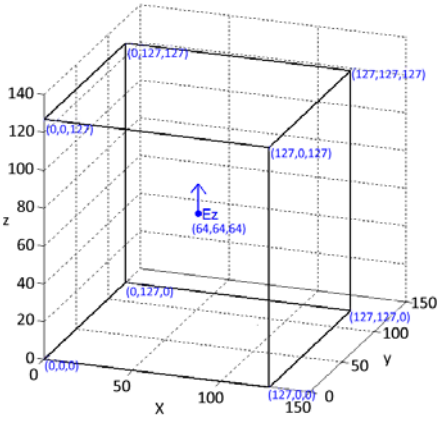


Fig. 2. The simulation field and the excitation source E_z

As the excitation source plays a very important role in effective FDTD simulation [10], choosing the right source seems significant. The pure Gaussian, the differential Gaussian and the modulated Gaussian are in common use as the excitation pulses in simulation experiments [11]. Here, the differential Gaussian pulse was adopted along the z-axis based on its having no zero-frequency component. The differential Gaussian pulse function is given by,

$$E_z(t) = (t - t_0) \exp\left[-\frac{4\pi(t - t_0)^2}{\tau^2}\right] \tag{4}$$

where we take value 5.0×10^{-10}s of t_0 to ensure that all of the positive pulses are received. In order to generate one pulse as soon as possible, τ related to the wave length is set to 4.0×10^{-8}s.

According to some scientific experiment results, numerical errors might be detected due to the inappropriate choice of Δx, Δy, Δz, and Δt. So here is Courant stability limit as shown in the following *inequality (5)*.

$$c\Delta t \leq \frac{1}{\sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}}} \tag{5}$$

In the inequality, c is the light speed and we take $0.005m$ as the value of step length of the differential process ($\Delta x = \Delta y = \Delta z$). Thus the value $9.6289 \times 10^{-12}s$ of Δt can be computed. For more detailed information about Courant stability limit, please refer to [2].

3.2 Implementation in CUDA

In the following section, we will extend our simulation model to the CUDA programming model.

Block Partition of the Model. According to the product instruction, the maximum number of threads per block in M2050 is 1024. However, we adopt 512 ($= \mathbf{8^3}$) threads per block in the experiment based on the consideration of compatibility and divisibility.

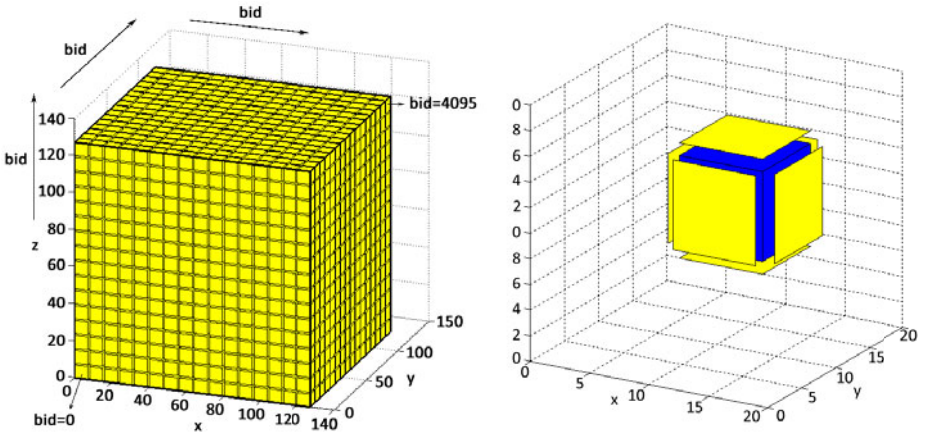


Fig. 3. (a) The block partition of the simulation model and (b) Loading areas for the shared memory

As *Figure 3(a)* indicates, the three-dimensional space is partitioned to 4096 blocks. Each block has 512 threads to calculate the corresponding values of the electrical field or the magnetic field. So there are 4096×512×2 threads, half of which compute the electrical field and half the magnetic field, doing their own job in parallel.

Using Shared Memory. After all, the use of shared memory might increase the complexity of the code. Even though we are reluctant to use the shared memory to optimize our programs, here we first adopt the shared memory method and in the following sections, we will compare this mechanism with that using L1 cache. From *Formula (3)*, we can find out that each component value, whether of the electrical field or of the magnetic field, would be used 4 times during the calculation. So in order to reduce memory access, each thread is first supposed to load its own corresponding data from the global memory to the shared memory. It should be noted that threads distributed on the surfaces also need to load data from neighboring blocks, as shown in *Figure 3(b).*

In the figure shown, the dark cube indicates current executing block (the block's id is 272) and it needs to pre-fetch all the related data to the shared memory including data from its own location and the light-colored areas. Then *__syncthread()* function is used to make sure all the needed data have been loaded to the shared memory. The next step is to compute the all the new values regarding the electrical field and the magnetic field.

As said above, we can see clearly that the data partitioning introduced by use of shared memory will increase the difficulty of design and the complexity of codes. However, if only non-Fermi architecture GPU cards are available, the use of shared memory becomes the only solution for the optimization of the parallel performance.

4 Performance Analysis and Comparison between Using Shared Memory and L1 Cache

First, we have compared the CPU and GPU implementations of the FDTD with different iteration times. The running time we measured includes the memory allocation, CPU computing or kernel execution in GPU, and results saving. We have conducted the experiments for 10 trials and the average performance is reported in the following *Table 1*. The experiment test bed was the HP ProLiant SL390s G7 Server [9] including 2.4-GHz Intel Xeon Processors E5620 with 24 GB of memory and Tesla M2050 GPUs [15]. Red Hat Enterprise Linux 6 and NVidia CUDA version 3.2 are used in the development environment.

Table 1. The average execution time (in seconds) of GPU and CPU approach for different iterations times

		Iteration times				
		100	500	1000	5000	10000
Test bed	CPU	186.482	924.762	1750.825	8896.725	17852.855
	GPU	16.936	84.501	170.545	846.036	1698.05

We can see from the table that the GPU approach using the shared memory can help developers obtain about 10 times speedup.

As mentioned in the previous sections, in order to reduce the total number of the global memory access, the shared memory is widely used on GPU computation as a method of traditional optimization strategy for the sake of GPU's parallel performance and efficiency. However, in such a case, developers would have some special problems to address requiring the low-level background knowledge of the hardware and also some careful consideration of coalesced memory accesses [13].

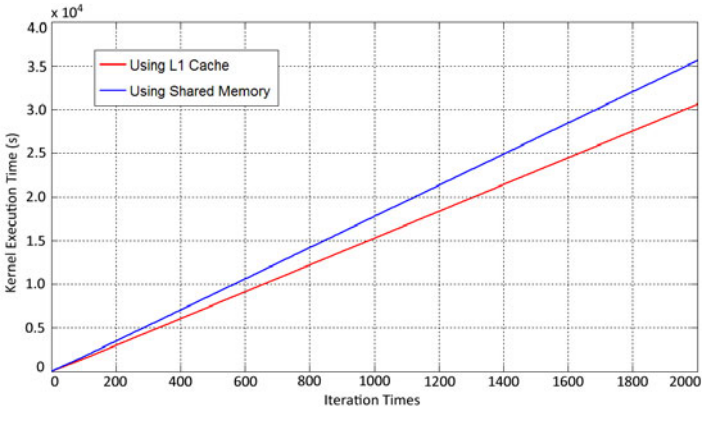


Fig. 4. Comparison of kernel execution time within 2000 iterations between strategies using shared memory and L1 cache respectively

As can be seen from *Figure 4* above, the running time of GPU program adopting L1 cache strategy rather than the shared memory is slightly faster than its counterpart in the front part. However, we should not belittle the difference, because it would generate cumulative effect when the iteration times are large enough, which happens a lot in FDTD simulation. For example, the disparity of 3s per iteration from this figure will be accumulated to nearly 2 hours after 2000 iterations.

When the shared memory is introduced into the implementation, we tend to add *__syncthread()* method after all the data required are loaded into the shared memory. This results in the ***synchronization delay*** [7]. On the other hand, the use of L1 cache is not associated with the synchronization delay. In Fermi, the prefetching of "soon-to-be-used" data can be achieved on hardware. That is what the last FDTD example proves. However, we still have to pay attention to the property of generality using L1 cache.

Here are some other examples for observing this difference. The first example is about the sum of squares which can be easily implemented and highly parallelized. When using shared memory strategy, we adopt addition tree to avoid the bank conflicts in shared memory. The second one is a parallelized Levenberg-Marquart algorithm for curve-fitting, including lots of matrix related operation. As the following figures show, we can clearly see that using L1 cache can exhibit its advantages when the data size is not too large.

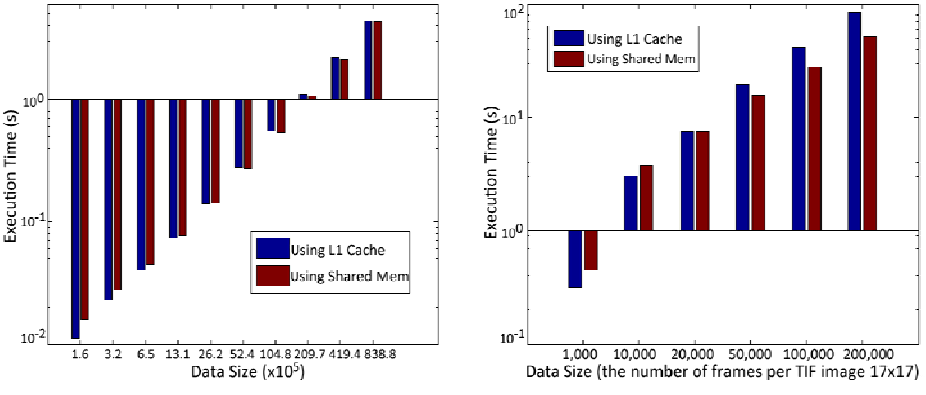


Fig. 5. The comparison between the two strategies of the algorithms of sum of squares and Levenberg-Marquardt

Taking the matrix multiplication [4] as an example, when the matrices are tiled with the help of the shared memory, the parallel computation can dramatically lower the execution time and at the same time increase the performance. Traditional parallel algorithm for matrix multiplication requires so many repeated global memory accesses retrieving and outputting the necessary data that only use of L1 cache is not suitable under this scenario. However, from the following *Figure 6(a)*, we can learn that when the dataset is not too large, making use of L1 cache rather than the shared memory may not come with low performance. As the data volume grows, the benefits for using the shared memory can manifest themselves. Note that the y-axis scale has been adjusted to the logarithmic.

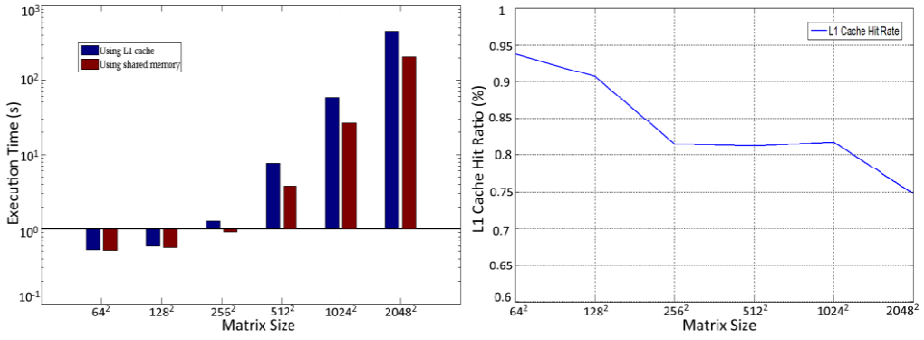


Fig. 6. (a) Using shared memory Vs. L1 cache and (b) L1 cache hit rate

The reason why the L1 cache strategy was defeated is probably because the gradually decreasing cache load hit ratio. The L1 global load hit and L1 global load miss values can be obtained from the NVidia's compute visual profiler [12].

Then we can get the curve of L1 cache hit ratio from *Figure 6(b)*, which indicates that the L1 cache hit ratio keeps decreasing with the matrix size rising. That means when the L1 cache misses happen in the tendency of rapidly increasing, the system has to reload the needed data to the cache pipeline which make an impact on the performance of the GPU computing.

5 Conclusions and Future Work

In this paper, we first analyzed the precision and accuracy of double-precision calculation on Fermi architecture GPUs when applying FDTD method. Then we evaluated the performance of FDTD method on the new Fermi architecture GPU. This work has shown that the parallel computation only using L1 cache can reach the similar or even better performance as the traditional optimization method using the shared memory does when the dataset is not too large or the frequency of repeated use of the related data is low. In the design and implementation of parallel algorithms, which will be optimized by using shared memory, people have to take efforts to make clear the hierarchy of the memory architecture and rearrange the way the data is stored on the shared memory to avoid the bank conflicts. Using L1 cache help developers focus on the logic of the algorithm rather than the manner of the data storage on the low-level devices (shared memory, global memory, etc.).

In condition of smaller scale of input data sets, the Fermi architecture can pre-fetch the data to be used into L1 cache which can obtain similar access speed with the shared memory. With the growth of the data sets, the pre-fetched data are distributed more scatteredly in different cache lines and hence the probability of cache miss, when the data is needed for computing, would increase leading to some loss of performance. However, in the strategy of using shared memory optimization, the data assignment and the size of the loaded data in shared memory will have already been determined by programmers. In this situation, computing programs can still

effectively load the data into the high-speed shared memory without worry about the increase of data sets.

In future research, we are expected to find the specific relation of the scale of dataset between using L1 cache and the shared memory to reach the possible peak performance. And then more different algorithms will be introduced to compare the optimization method between L1 cache and shared memory. After all, we want to maximize the efficiency of the L1 cache and decrease the complexity of the code.

This paper was supported by the National Basic Research program of China (973 program) (Grant no.2011CB706900).

References

1. Kane, S.Y.: Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media. IEEE Transactions on Antennas and Propagation (1966)
2. Allen, T., Susan, C.H.: Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd edn. Artech House Inc., MA (2005)
3. John N., Ian B., Michael G., Kevin S.: Scalable Parallel Programming with CUDA. Queue, 40–53 (2008)
4. NVIDIA Corporation: NVIDIA CUDA C Programming Guide: Version 4.0 (2011)
5. Next Generation CUDA Architecture, Code Named Fermi, http://www.nvi-dia.com/object/fermi_architecture.html
6. Mehmet, F.S., Ihab, E-K., David, A.B., Shawn-Yu, L.: A Novel FDTD Application Featuring Open MP-MPI Hybrid Parallelization. In: Proceedings of International Conference on Parallel Processing, Montreal, Quebec, Canada, pp. 373–379 (2004)
7. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: Proc. ISCA, pp. 152–163 (2009)
8. NVIDIA Corporation: NVidia Fermi Compute Architecture Whitepaper Version 1.1
9. Hewlett-Packard Development Company: HP ProLiant SL390s G7 2U half width Server Maintenance and Service Guide
10. Jun, L., Tian, Y., Tong, L.: Analysis of the Electromagnetic Characteristics of Coplanar Waveguide by FDTD Method. Testing and Diagnosis (2009)
11. Wenhua, Y.: Electromagnetic Simulation Techniques Based on the FDTD Method, pp. 84–85. John Wiley and Sons Inc., Chichester (2009)
12. NVIDIA Corporation: Compute Visual Profiler User Guide (2010)
13. Phuong Hoai, H., Tsigas, P., Anshus, O.J.: The Synchronization Power of Coalesced Memory Accesses. IEEE Transactions on Parallel and Distributed System, 939–953 (2010)
14. CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html
15. Tesla GPU Computing Solutions for Data Centers, http://www.nvidia.com/object/preconfigured-clusters.html

The Probability Model of Peer-to-Peer Botnet Propagation

Yini Wang, Sheng Wen, Wei Zhou, Wanlei Zhou, and Yang Xiang

Deakin University, Victoria 3125, Australia
{yiniwang,wsheng,weiz,wanlei.zhou,yang}@deakin.edu.au

Abstract. Active Peer-to-Peer worms are great threat to the network security since they can propagate in automated ways and flood the Internet within a very short duration. Modeling a propagation process can help us to devise effective strategies against a worm's spread. This paper presents a study on modeling a worm's propagation probability in a P2P overlay network and proposes an optimized patch strategy for defenders. Firstly, we present a probability matrix model to construct the propagation of P2P worms. Our model involves three indispensible aspects for propagation: infected state, vulnerability distribution and patch strategy. Based on a fully connected graph, our comprehensive model is highly suited for real world cases like Code Red II. Finally, by inspecting the propagation procedure, we propose four basic tactics for defense of P2P botnets. The rationale is exposed by our simulated experiments and the results show these tactics are of effective and have considerable worth in being applied in real-world networks

Keywords: Botnet, Worms, Peer-to-Peer, Propagation probability.

1 Introduction

Nowadays, P2P botnets are widely believed to be one of the most serious dangers in the Internet since it is not easy for them to be detected and taken down. Botnets have evolved from the Slapper Worm in 2003 which was the first P2P worm, to the Storm Worm [2] which is the most wide-spread P2P bot currently. Today, P2P worms are becoming more complicated and sophisticated.

In order to take an effective countermeasure to prevent the propagation of P2P worms to the greatest extent, we must understand the propagation mechanism. In recent years, some papers [3-5] have discussed the spreading model and approaches used by P2P worms. Fan and Xiang [3] used a logic matrix approach to model the spreading of P2P worms. It presented two different topologies: a simple random graph topology and a pseudo power law topology. The research studied their impacts on a P2P worm's attack performance and analyzed related quarantine strategies for these two topologies. This paper adopts two constants of logic type (True or 1, False or 0) as the value of matrix variables. This 0-1 matrix stands for the propagation ability of nodes, that is whether they can allow the virus to spread or not. In the real word,

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 470–480, 2011.

however, according to the definition of Peer-to-Peer networks, two nodes in a P2P network are absolutely connected even if the probability of the connection's existence is very small. Hence, a node always can propagate the virus to other node with a certain probability in a P2P fashion. Taking Code Red II [6] as an example, the probability of the virus propagating to the same class A IP address is 3/8; to the same class A and B IP address is 1/2; and to the random IP address is 1/8. Therefore, the model in [3] has great limitations and is not in accordance with real virus instances. Furthermore, in the classical simple epidemic model [7-10], the authors considered only two statuses of all hosts: susceptible and infectious. However, in the propagation of P2P worms, the paramount objective is to find an optimized patch strategy to minimize the scale of a P2P botnet.

Thus, we are motivated to model a probability prototype for P2P worm propagation. It concerns three key factors: infected state, vulnerability distribution and patch tactic. To the best of our knowledge, there are few papers refer to the propagation probabilities of each node in the network. The goal of this research is to find the most effective defense against P2P botnet. The major contributions of this paper are as follows. 1) A probability matrix model is proposed to construct a propagation model of P2P worms; 2) Three key factors common to all worms are introduced to describe their propagation in this comprehensive model; 3) Based on the concept of a fully connected graph, our model adopts real propagation probabilities from Code Red II, which suits real world cases better; 4) The most significant contribution is that we successfully summarize four basic tactics in response to a worm outbreak, which are each very promising for the future defense of P2P botnets.

The rest of the paper is organized as follows. In Section 2, we survey related work. In Section 3, we describe the proposed probability propagation model. Next, we conduct an analysis and deduce the result for obtaining an optimized patch strategy in Section 4. Section 5 performs an evaluation. Finally, the conclusion and future work are present in Section 6.

2 Related Work

In the area of epidemiology, models [5-15] like deterministic epidemic models and stochastic propagation models have been used to study the propagation process of worms.

A. Deterministic Epidemic Model

The traditional deterministic epidemic models [7, 8] are Susceptible-Infectious (SI) models, in the sense that all hosts can have only one of two states: susceptible or infectious. These classical simple models are based on an assumption: an infected host can infect any other susceptible nodes with an equal possibility. However, it is no longer suitable for worm modeling since most worms propagate through the internet and have different propagating probabilities.

Staniford et al. [9] presented a random constant spread model (RCS) for the Code-Red I v2 worm. It is essentially the above classical simple epidemic model allowing for the infection rate to be constant, and without considering the patching cases. On the basis of the simple epidemic model, Zou et al. [5] proposed a two-factor model

which improves the classical simple models. It introduced human countermeasures in patching, the removal of hosts from both infectious and susceptible population, and considered the infectious rate as a variable but not a constant. Additionally, models from Z. Chen *et al.* [11] and Y. Wang *et al.* [12] took into account the time taken to cause an infection from spreading the virus from one infected host to other hosts.

B. Stochastic Epidemic Model

The stochastic epidemic model is based on the theory of stochastic processes. K.R. Rohloff *et al.* [13] presented a stochastic density-dependent Markov jump process propagation model for RCS (Random constant Scanning) worms, drawn from the field of epidemiology [14, 15]. Sellke *et al.* [16] built up a stochastic branching process model to characterize the propagation of worms using a random scanning approach. It developed an automatic worm containment tactic for preventing the worm propagation beyond its early states.

Nevertheless, all existing models are based on a linear structure or a one-to-many hierarchy. Thus, these modes are not applicable to topology-aware worms and cannot describe the spreading of P2P worms.

3 Theoretical Propagation Model

In this section we present a probability propagation model used to estimate the optimized patch strategy.

A. Topology Propagation Matrix (TPM)

The traditional representation of a P2P network employs a directed graph to model the topology. We propose an alternate representation using an n by n square matrix P with elements t_{ij} to indicate a P2P overlay network consisting of n peers. We consider that two peers in a P2P network are connected even if the probability of the connection's existence is very small, thereby making node i and j immediate neighbors. In this matrix, each element t_{ij} represents a propagation probability of spreading worms from node i to node j under the condition of node i being infected. We term such kind of matrix the topology propagation probability matrix (TPM) of the P2P overlay network, as shown in (1).

$$P = \begin{bmatrix} t_{11} & \cdots & \cdots \\ \cdots & t_{ij} & \cdots \\ \cdots & \cdots & t_{nn} \end{bmatrix}_{n\times n} \qquad t_{ij} = p(N_j | N_i) \quad t_{ij} = 0 \ (i = j), \ \sum_{j=1}^{n} t_{ij} \in [0,1] \tag{1}$$

Each row of the TPM represents a propagation probability from one infected peer to other peers. Each column of the TPM represents a propagation probability from infected peers to a target peer. We assume one peer cannot propagate the worm to itself, so the probability of self-propagation is zero.

B. Propagation Probability

Considering the propagation process between two peers under the real condition, worms could be spread from node i to node j via one or more intermediate nodes. We assume that worm's propagation from node i (N_i) to node j (N_j) via and only via k

intermediate nodes in a network consisting of n peers, which is denoted by $t_{ij}^{(k)}$. It is defined in (2). N_i bar represents the nodes excluding node N_i. It means we will not consider the propagation cycles in the spreading path.

$$t_{ij}^{(k)} = p(N_j^{(k)} \Big| N_i \cup \overline{N_i}^{(k-1)}) \quad = \sum_{m=1}^{m=n, m\neq i} t_{im}^{(k-1)} t_{mj} \quad k \in [1, n-2], \quad i = 1,..., n, \quad j = 1,..., n \tag{2}$$

Since N_i self-propagation via k nodes is meaningless in the real world, we let the value of propagation probability be zero; namely $t_{ij}^{(k)} = 0$ when $i=j$. We introduce a function γ to conduct the iterated procedure. It is defined in (3):

$$\gamma^k(P) = \underbrace{P \bullet P \bullet \cdots \bullet P}_{k+1} \gamma^0(P) = P, \qquad \gamma^1(P) = \gamma(P) = P \bullet P \tag{3}$$

Operation $\bullet$ is the traditional matrix multiplication. Subsequently, the TPM can be represented by the following equation when worm's propagation is via and only via k intermediate nodes, as in (4).

$$P^{(k)} = \begin{bmatrix} t_{11}^{(k)} & \cdots & \cdots \\ \cdots & t_{ij}^{(k)} & \cdots \\ \cdots & \cdots & t_{nn}^{(k)} \end{bmatrix}_{n\times n} = \gamma^k(P) \tag{4}$$

It is possible that there may be more than one path for worms to spread from one peer to another peer. So we assume that worm's propagation from node i (N_i) to node j (N_j) is at most via k intermediate nodes and is denoted by $t_{ij}^{(k)'}$. It is defined in (5):

$$t_{ij}^{(k)'} = t_{ij}^{(k-1)'} + t_{ij}^{(k)} = t_{ij}^{(1)} + \sum_{m=2}^{k} t_{ij}^{(m)} \qquad k \in [1, n-2], \quad i = 1,..., n, \qquad j = 1,..., n \tag{5}$$

C. Three Key Factors

In a P2P overlay network, there are three significant factors for a worm's propagation: infected state, vulnerability distribution and patch strategy. For the infected state, this represents if the peer has been infected or not. Vulnerability distribution reflects the situation of vulnerable peers in the topology. Patch strategy provides a cure approach for infected peers. After being patched, infected peers cannot be infected again.

C.1 Infected State Probability Vector (S)

An initial infected state probability vector (S) can be defined as in (6). Here, one represents an infectious peer that can propagate worms with a probability of one. Zero means a peer is healthy without the ability to propagate the worms.

$$S = \left[s_1, s_2, \ldots s_i, \ldots s_n\right]^T, \quad s_i = 0 \ \ or \ \ 1, \quad i = 1 \ldots . n \tag{6}$$

We firstly assume every peer in the TPM is vulnerable. Therefore, in the propagation process, each intermediate node can be infected and become infectious so that the infected state in the TPM is variable. After the worm propagates via k nodes, an update S can be defined as shown in (7):

$$S^{(k)} = \left[\left[S^{(k-1)}\right]^T \bullet \gamma(S^{(k-1)} \&_L P^{(k-1)})\right]^T, s_i \geq 0 \ \ (i = 1,..., n) \tag{7}$$

$\&_L$ indicates a new logic AND operation of a column vector A and a matrix B, called *Left Logic AND*. The result of $A \ \&_L \ B$ is a new logic matrix of the same dimension as B. Each element in the new matrix is the result of the product of the corresponding elements a_i and b_{ij} from each column of matrix B. It is defined in (8):

$$A \ \&_L \ B = \begin{bmatrix} a_1 \\ \dots \\ a_n \end{bmatrix} \begin{bmatrix} b_{11} & \dots & \dots \\ \dots & b_{ij} & \dots \\ \dots & \dots & b_{nn} \end{bmatrix}_{n \times n} = \begin{bmatrix} a_1 \cdot b_{11} & \dots & a_1 \cdot b_{1n} \\ \dots & a_i \cdot b_{ij} & \dots \\ a_n \cdot b_{n1} & \dots & a_n \cdot b_{nn} \end{bmatrix}_{n \times n} \tag{8}$$

In real-world worms it is observed that an infected peer can propagate worms and a vulnerable peer can also be infected to become a new infectious node for future propagation with a certain probability. Therefore, except for the initial state of S, each element of any updated S is the probability denoted by a real value, which is not simply zero or one. Consequently, the TPM can be represented by the following equation when worm's propagation is via and only via k intermediate nodes, as in (9). $P_s^{(k)}$ represents the infected scale of the network under the infected state (S) after the worm spread via k intermediate nodes.

$$P_s^{(k)} = \gamma(S^{(k-1)} \ \&_L \ P_s^{(k-1)}) \tag{9}$$

C.2 Vulnerable Distribution Vector (V)

In real-world conditions, the vulnerability of a peer is an objective fact. Therefore, a healthy peer without any vulnerability cannot become infectious in the worm's propagation process. On the basis of this fact, we need to consider the vulnerability distribution in the TPM. A vulnerable distribution vector (V) is defined in (10). For an element in V, the value of one represents that a peer is vulnerable. Zero means that the peer is healthy without any vulnerability.

$$V = [v_1, v_2, \dots v_i, \dots v_n]^T, \ v_i = 0 \ or \ 1, \quad i = 1 \dots . n \tag{10}$$

After considering the vulnerability distribution vector, the TPM can be represented by the following equation when the worm propagates via and only via k intermediate nodes, as in (11). $P_{sv}^{(k)}$ represents the infected scale of the network under the vulnerable distribution (V) after the worm spread via k intermediate nodes.

$$P_{sv}^{(k)} = \gamma(P_s^{(k-1)} \ \&_R \ V^T) \tag{11}$$

We define $\&_R$ to indicate a new logic AND operation of a column vector A and a matrix B, called *Right Logic AND*, which is different from *Left Logic AND*. The result of $A \ \&_R \ B$ is a new logic matrix of the same dimension as B. Each element in the new matrix is the result of the product of the corresponding elements a_j and b_{ij} from each row of matrix B. It is defined in (12):

$$B \ \&_R \ A = \begin{bmatrix} b_{11} & \dots & \dots \\ \dots & b_{ij} & \dots \\ \dots & \dots & b_{nn} \end{bmatrix}_{n \times n} \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} = \begin{bmatrix} b_{11} \cdot a_1 & \dots & b_{1n} \cdot a_n \\ \dots & b_{ij} \cdot a_j & \dots \\ b_{n1} \cdot a_1 & \dots & b_{nn} \cdot a_n \end{bmatrix}_{n \times n} \tag{12}$$

C.3 Patch Strategy Vector (Q)

An infected peer can be cured to become a healthy node, which cannot spread worms to other peers again. Therefore, we need to remove these nodes from the propagation process in time. We define a patch vector Q in (13). For each element in Q, the value of one represents that a peer has been patched and become to a healthy node. A value of zero indicates that a peer is still vulnerable.

$$Q = [q_1, q_2, \ldots q_i, \ldots q_n]^T, \quad q_i = 0 \ or \ 1, \quad i = 1 \ldots . n \tag{13}$$

After considering the patch strategy vector, the TPM can be represented by the following equation when the worm propagates via and only via k intermediate nodes, as in (14). We define $\underline{\&}$ to indicate a new logic AND operation between two elements. The definition for $\underline{\&}$ operation is shown in Table 1.

$$P_{svq}^{(k)} = \gamma(P_{sv}^{(k-1)} \&_R (V^T \underline{\&} Q^T)) \tag{14}$$

Table 1. Truth table for new logic and operation

V^T	Q^T	$V^T \underline{\&} Q^T$
1	1	0
0	1	0
1	0	1
0	0	0

4 Propagation Ability and Quarantine Ability

In real world scenarios, attackers expect to control a significant proportion of the P2P overlay network to enable the worm's propagation. The topology's propagation ability (*PA*) is related to the number of peers that the worm can propagate to with high probability. In consideration of more than one path for the propagating worm, we adopt a modified P'_{svq} to represent a sum of probabilities for the worm's propagation between two peers with at most k intermediate nodes. It is defined in (15):

$$P'_{svq} = \sum_{i=1}^{k-2} P_{svq}^{(i)} \tag{15}$$

In order to evaluate the *PA* of a topology, we assume an ability threshold δ to estimate each peer's *PA*. If an element t'_{ij} of P'_{svq} is greater than or equal to δ, then the value of $PA(t'_{ij})$ is equal to one, or else it is equal to zero. The number of times *PA* is equal to one is defined as x in (16). Consequently, a large value of x indicates a strong *PA* of the topology.

$$x = \sum_{i=1}^{n|} \sum_{j=1}^{|n|} t'_{ij} \tag{16}$$

Table 2. PA and QA in 90% vulnerability distribution (V=90%)

	$Q_1(10\%)$	$Q_2(20\%)$	$Q_3(10\%)$	$Q_4(20\%)$
$S_1(10\%)$	(12840,147160)	(11560,148440)	(12880,147120)	(11640,148360)
$S_2(20\%)$	(25840,134160)	(22690,137040)	(25920,134080)	(22960,137040)
$S_3(10\%)$	(12920,147080)	(11440,148560)	(12960,147040)	(11520,148480)
$S_4(20\%)$	(26080,133920)	(23120,136880)	(26080,133920)	(22880,137120)

Table 3. PA and QA in 90% vulnerability distribution (V=70%)

	$Q_1(10\%)$	$Q_2(20\%)$	$Q_3(10\%)$	$Q_4(20\%)$
$S_1(10\%)$	(10080,149920)	(9010,150990)	(10160,149840)	(9040,150960)
$S_2(20\%)$	(20080,139920)	(17520,142480)	(19680,140320)	(18640,141360)
$S_3(10\%)$	(10120,149880)	(9170,150830)	(10160,149840)	(8800,151200)
$S_4(20\%)$	(20140,139860)	(17680,142320)	(20240,139760)	(17760,142240)

On the contrary, defenders focus on the quarantine ability (*QA*) of a P2P overlay network. The *QA* is related to the number of peers with low infected probability in the topology. Likewise to the definition of the TPM, we define R to represent an infected probability matrix, as in (17).

$$R^{(k)} = \begin{bmatrix} r_{11}^{(k)} & \cdots & \cdots \\ \cdots & r_{ij}^{(k)} & \cdots \\ \cdots & \cdots & r_{nn}^{(k)} \end{bmatrix} r_{ij}^{(k)} = p(N_i^{(k)} \mid N_j^{(k)}) = \frac{p(N_j^{(k)} \mid N_i^{(k)}) p(N_i^{(k)})}{p(N_j^{(k)})} = \frac{t_{ij}^{(k)} \sum_{k=1}^{n} t_{ik}^{(k)}}{\sum_{k=1}^{n} t_{kj}^{(k)}} \tag{17}$$

$k \in [1, n-2], \quad i = 1,...,n, \qquad j = 1,...,n$

When considering more than one path for a single peer being infected, we adopt a modified infected probability matrix R' to represent the sum of infected probabilities. It is defined in (18):

$$R' = \sum_{i=1}^{k-2} R^{(i)} \tag{18}$$

In order to evaluate the *QA* of a topology, we assume an ability threshold θ to distinguish each peer's *QA*. If an element r_{ij} of R' is greater than or equal to θ, then the value of $QA(r_{ij})$ is equal to one, or else it is equal to zero. The number of times *QA* is equal to zero is defined as y in (19). Therefore, a large value of y indicates a strong *QA* of a topology.

$$y = n^2 - \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij} \tag{19}$$

For attackers, a reasonable distribution strategy for infectious peers results in a high *PA* of a P2P network. Similarly, an effective patch strategy for defenders will lead to a high *QA* of a P2P network. Therefore, the paramount objective of this paper is to

discover patch strategies that can significantly suppress the propagation function of a P2P botnet.

Table 4. PA and QA in different percentage of patching nodes (V=90%)

	Q_1(10%)	Q_2(20%)	Q_3(30%)	Q_4(40%)	Q_5(50%)	Q_6(60%)
S_1 (10%)	(12840, 147160)	(11560, 148440)	(7760, 152240)	(6840, 153160)	(5640, 154360)	(4640, 155300)
S_2 (20%)	(25840, 134160)	(22690, 137040)	(15680, 144320)	(13680, 146320)	(11520, 148480)	(9040, 150960)

5 Simulation Experiments

Our implementation is in MATLAB. It assumes there are a total of 10,000 peers (computers) belonging to a P2P overlay network under consideration. Therefore, the TPM is represented by a 10,000 by 10,000 square matrix P and its initial state is defined according to the propagation probability of ***Code Red II***. We divide matrix P into 10,000 partitioned matrixes $A_{ij}B_{xy}$ and each of them is a 100 by 100 square matrix. Matrix P and $A_{ij}B_{xy}$ are shown as follows.

$$P = \begin{bmatrix} A_{11}B_{xy} & \cdots & A_{1,100}B_{xy} \\ \cdots & A_{ij}B_{xy} & \cdots \\ A_{100,1}B_{xy} & \cdots & A_{100,100}B_{xy} \end{bmatrix}_{100\times100} \qquad A_{ij}B_{xy} = \begin{bmatrix} A_{ij}B_{11} & \cdots & A_{ij}B_{1,100} \\ \cdots & A_{ij} & \cdots \\ A_{ij}B_{100,1} & \cdots & A_{ij}B_{100,100} \end{bmatrix}_{100\times100}$$

We define $p(A_{ij}B_{xy})$ to represent the propagation probability from one peer with A class IP i and B class IP x to another peer with A class IP j and B class IP y. Thus, if $i=j$ and $x=y$, then $p(A_{ij}B_{xy})=1/2$; if $i=j$ *and* $x\neq y$, then $p(A_{ij}B_{xy})=3/8$; or else, $p(A_{ij}B_{xy})=1/8$. We assume the P2P overlay network consists of n peers, and the connection probability of two random nodes is $1/n$. Therefore, according to the Multiplication Rule, the TPM's propagation probability is $n^{-k}\cdot\gamma^k(P)$ when the worm's propagation is at most via k intermediate nodes.

In our simulation experiment, we analyze the impact of changing the matrix dimensionality used in the experiments and find that a larger dimension will not produce significantly different results. In order to show these results clearly, we choose reasonable network sizes (5000 nodes) and examine them under different scenarios. There are two scenarios for the vulnerability distribution: 90% and 70% of the total peers respectively. We also arrange two scenarios for in infectious nodes (10% and 20%), which follow a Uniform or Gaussian distribution. Additionally, patched nodes are also grouped in 10% and 20%, which similarly follow a Uniform or Gaussian distribution.

There are 5 iterations of the TPM. We show the results from these experiments in Table 2, Table 3 and Table 4. Each item in the tables represents the pair of PA and QA under a S_x and Q_x (x$\in$ [1, 4]).

1) Infectious Node Rate VS. Patching Node Rate

Based on the results from Table 2 and Table 3, firstly we focus on S and Q under the same distribution. The blocks include (S_1, S_2; Q_1, Q_2), (S_1, S_2; Q_3, Q_4), (S_3, S_4; Q_1, Q_2) and (S_3, S_4; Q_3, Q_4). We find that the best strategy for both S and Q in each block are in S_x 20% (x=2,4) and Q_y 10% (y=1,3). Even though Q targets a large rate of patching, this strategy is helpless to improve QA. This indicates that the attacking effect is more sensitive to the percentage of infectious nodes than defending effect.

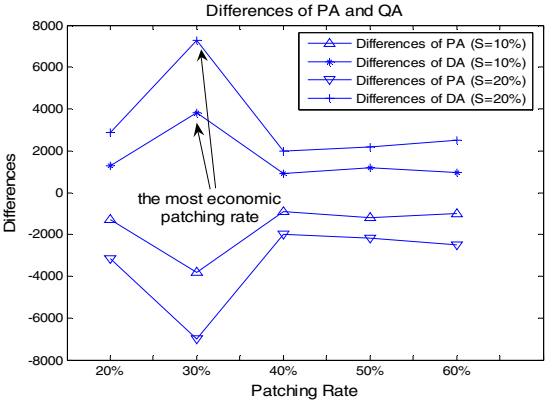


Fig. 1. Differences of PA and QA

2) Uniform Distribution VS. Gaussian Distribution

Based on the results from Table 2 and Table 3, secondly we focus on S and Q under the same infectious nodes rate and patching nodes rate. The blocks include (S_1, S_3; Q_1, Q_3), (S_2, S_4; Q_2, Q_4), (S_1, S_3; Q_2, Q_4) and (S_2, S_4; Q_1, Q_3). We find that all th strategies are similar, which indicates that the distribution of nodes has less impact on PA and QA.

3) Vulnerability Rate in the topology

Based on the results from Table 2 and Table 3, thirdly we focus on the impact on PA and QA with an independent V. Each item in Table 2 outperforms the ones in Table 3, which leads us to a conclusion that a greater number of vulnerable nodes in the topology will benefit attackers (PA_{xy} $(V=90\%) > PA_{xy}$ $(V=70\%)$ & $QA_{xy}(V=90\%) > QA_{xy}$ $(V=70\%)$, $x,y \in [1, 4]$).

4) Patching Rate

Fourthly, based on the results from Table 4 we focus on the most economic patching rate. As long as the patching rate increases, PA increases monotonously while QA decreases gradually. However, the larger the patching rate, the greater the economic impact will be in real world. We compare each pair of patching rates and identify their differences in Fig.1. We clearly see that the most economic patching rate is 30%, because any larger rate can only bring a limited increase to QA. We believe this conclusion is valuable, particularly with respect to economical and industrial benefits.

6 Conclusion and Future Work

This paper presents a probability model of Peer-to-Peer botnet propagation for finding an optimized patch strategy so that defenders can prevent the bots spreading in a reasonable and economic approach. Firstly, we present a probability matrix model to construct the propagation model of P2P worms. This comprehensive model involves three indispensible aspects for propagation. Based on a fully connected graph, our model suits real world cases like Code Red II. The most significant contribution is that we successfully summarize four basic strategies in response to a worm outbreak, which are each very promising for the future defense of P2P botnet.

There are some limitations in our paper. Firstly, we did not use a real data set to model the propagation procedure and provide a more accurate patching rate for defense attackers effectively. Secondly, there are some other parameters that should be involved such as the impact of the number of nodes in a topology. In the future, we plan to model our propagation probability theory by using a real data set and provide a more comprehensive proof for our patching strategy.

References

1. Arce, I., Levy, E.: An analysis of the slapper worm. IEEE Security & Privacy Magazine 1, 82–87 (2003)
2. Holz, T., Steiner, M., Dahl, F., Biersack, E., Freiling, F.: Measurement and mitigation of peer-to-peer-based botnets: a case study on storm worm. In: The 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats, San Francisco, USA, pp. 1–9 (April 2008)
3. Fan, X., Xiang, Y.: Modeling the propagation of Peer-to-Peer worms. Future Gener. Comp. Sy. 26, 1433 (2010)
4. Yu, W.: Analyze the worm-based attack in large scale P2P networks. In: 8th IEEE International Symposium on High Assurance Systems Engineering, pp. 308–309. IEEE Press, Tampa (2004)
5. Zou, C.C., Gong, W., Towsley, D.: Code Red worm propagation modeling and analysis. In: 9th ACM Conference on Computer and Communications Security, Washington, pp. 138–147 (2002)
6. CAIDA Analysis of Code-Red, `http://www.caida.org/research/security/code-red/`
7. Bailey, N.T.: The Mathematical Theory of Infectious Diseases and its Applications. Hafner Press, New York (1975)
8. Frauenthal, J.C.: Mathematical Modeling in Epidemiology. Springer, New York (1980)
9. Staniford, S., Paxson, V., Weaver, N.: How to own the internet in your spare time. In: The 11th USENIX Security Symposium, San Francisco, pp. 149–167. ACM, CA (2002)
10. Andersson, H., Britton, T.: Stochastic Epidemic Models and their Statistical Analysis. Springer, New York (2000)
11. Chen, Z., Gao, L., Kwiat, K.: Modeling the spread of active worms. In: IEEE INFOCOM, pp. 1890–1900 (2003)
12. Wang, Y., Wang, C.: Modeling the effects of timing parameters on virus propagation. In: WORM 2003, Washington, DC, USA, pp. 61–66 (2003)

13. Rohloff, K., Basar, T.: Stochastic behavior of random constant scanning worms. In: The 14th ICCCN, San Diego, CA, USA, pp. 339–344 (2005)
14. Daley, D.J., Gani, J.: Epidemic Modelling: An Introduction. Cambridge University Press, Cambridge (1999)
15. Andersson, H., Britton, T.: Stochastic Epidemic Models and their Statistical Analysis. Springer, New York (2000)
16. Sellke, S., Shroff, N.B., Bagchi, S.: Modeling and automated containment of worms. In: DSN 2005, pp. 528–537 (2005)

A Parallelism Extended Approach for the Enumeration of Orthogonal Arrays

Hien Phan[1], Ben Soh[1], and Man Nguyen[2]

[1] Department of Computer Science and Computer Engineering, LaTrobe University, Australia
[2] Faculty of Computer Science and Engineering, University of Technology, Ho Chi Minh City, Vietnam

Abstract. Orthogonal Array plays an important role in Design of Experiment and software testing. The most challenging aspect of Orthogonal array is the enumeration problem, that is for given parameter sets, we want to count exactly how many isomorphism classes exist. Enumeration of orthogonal array usually requires huge effort needed to complete its computation. There are several algorithms that have been proposed for enumeration of orthogonal array, however, there exists ineffective parallelism approach for handling this problem.

In this paper, we present a step-by-step parallelism extending approach for enumeration of orthogonal array. The experiments show that this proposed approach could get an relative speedup efficiency of up to 128 processes and a great dynamic load balancing between computing processes.

1 Introduction

Orthogonal array (OA) plays an important role in the field of Design of statistic experiment. An OA of strength t, $OA(N; r_1 \cdot r_2 \cdots r_n; t)$, is an $N \times n$ array whose ith column contains r_i different factor-levels so that, for any t columns, every t-tuple of levels appear equally often in the array. Suppose with all n r_i factors, there are m s_i distinct factor sizes $s_1 > s_2 > \cdots > s_m$ and exactly a_i factors with s_i levels, we usually use a reduced symbol $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}; t)$ for representing an OA. The collection of the number of rows N, factor-levels $s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}$ and strength t is the parameter set or design type for an OA.

Two orthogonal arrays are said to be *isomorphic* if one can be obtained from the other by a sequence of row permutations, column permutations and permutations of symbols in each column. We call the collection of all arrays that can be obtained from a parent array by permuting rows, columns, or factor-levels is an *isomorphism class*.

The most challenging aspect of Orthogonal array is the enumeration problem, that is for given parameter sets, we want to count exactly how many isomorphism classes exist. In other words, enumeration of orthogonal arrays (*EOA*) of a parameter set is to generate exactly an array from every equivalent class defined by the equivalence relation isomorphism.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 481–493, 2011.

One of the major characteristics of enumeration of orthogonal array is the huge effort needed to complete its computation. Therefore, a parallel method for solving *EOA* could allow us to reduce the execution time significantly and generate new results using the power of high performance computing system. Several serial algorithms have been proposed for enumeration of orthogonal array of strength t, however there exists ineffective parallelism approach to this problem.

In this paper we propose a step-by-step parallelism approach for enumeration of orthogonal array based on an enumeration algorithm named *Minimum Complete Set (MCS)* proposed on [SEN10]. Among several serial algorithms for EOA, we choose *MSC* algorithm as a foundation algorithm to compute for *EOA* in parallel as it is the first proposed algorithm could address general OAs of given strength, run-size, and level-numbers of the factors so far.

1.1 Major Contribution

The parallel algorithm presented in this paper efficiently handles the computing-intensive nature of the EOA problem and has an efficient speedup even when hundreds of processes are used to run the algorithm. The proposed parallel algorithm is a parallelization of the efficient and general method of enumerating of orthogonal array of Eric, Pieter and Man Nguyen [SEN10]. We proposed an idea for reusing all lex-least orthogonal arrays with size of k columns to extend and enumerate concurrently all lex-least orthogonal arrays with size of $k + 1$ columns. In other words, all arrays of a new column will be enumerated from all generated arrays of the previous columns, not from scratch. This saves the cost for regenerating old arrays of previous size, especially when the number of previous columns is huge. Moreover, with using previous arrays as inputs, some efficient data parallelism strategy for each extending step could be applied and get significant speedup results.

To enable efficient and scalable parallelization, a dynamic load balancing method is proposed. In particular, a dynamic or runtime work-sharing method [DOS+07] in combination with a stack splitting procedure [FM87] guarantees a minimal idle time of each process. As a result, we achieve a significant dynamic load balancing where execution time is always nearly identical between processes take part in computation.

With some specific parameter sets, enumeration of OA could generate a lot of outputs. It is highly desirable to have an efficient way for parallelism writing those outputs to storage systems. A minor contribution of this paper is that we propose to use *MPI IO* for allowing many processes writing concurrently output matrices to a single file.

The rest of this article is organized as follows: Section 2 presents an overview about the *MCS* algorithm for enumeration of orthogonal array. In Section 3, a step-by-step extending parallelism approach for enumeration of orthogonal array will be proposed. Detail of this implementation will be given on Section 4. Results of the experiment will be presented on section 5. And finally, some conclusion will be discussed in Section 6.

2 Background

There are some serial algorithms for *EOA* but most of them are just applied for some special parameter sets, such as OAs of strength 2 [TGM00]. In 2010 *MSC* algorithm which is proposed by Eric, Pieter and Man [SEN10] is the first *EOA* algorithm that could handle general OAs of given strength, run-size, and level-numbers of the factors. The use of generalized parameter is the key reason why we use *MSC* algorithm as a foundation algorithm to parallel the computing for *EOA*. A detailed description of the *MSC* algorithm can be found on [SEN10]. For completeness, we briefly describe the *MSC* algorithm in some detail below before specifying how it can be turned into an efficient parallelism approach.

2.1 Serial MCS Algorithm

The *MSC* algorithm generates *LMC* matrices as representatives for all nonisomorphism class. We first present the definition of *LMC* matric:

LMC Matrix. *Lexicographically less comparison*, a comparison metric of two arbitrary orthogonal arrays with the same specific design type, has been firstly proposed in [Ngu05].

For two vectors u and v of length L, we say u is lexicographically less than v, written $u < v$, if there exists an index $j = 1, 2, ..., L-1$ such that $u[i] = v[i]$ for all $1 \leq i \leq j$ and $u[j+1] < v[j+1]$.

Let $F = [c_1, ..., c_d]$, $F^{'} = [c_1^{'}, ..., c_d^{'}]$ be any pair of fractions where c_i, $c_i^{'}$ are columns. We say F is column-lexicographically less than $F^{'}$, written $F < F^{'}$, if and only if there exists an index $j \in \{1, ..., d-1\}$ such that $c_i = c_i^{'}$ for all $1 \leq i \leq j$ and $c_{j+1} < c_{j+1}^{'}$ lexicographically.

The smallest matrix of an isomorphic class which corespondents to a specific design type will be called *lexicographically minimum in column (LMC)matrix* and it is the only representative of this isomorphic class.

Backtracking for Finding LMC Matrix. The serial *MCS* algorithm uses backtrack search strategy to construct new orthogonal arrays and check whether every new generated orthogonal array is *LMC*. In particular, it will generate and extend column by column until it reaches the target column size S. We summarize the outline of the *MCS* algorithm as below:

On the outline above, $X = [x_1, x_2, ..., x_n]$ is an orthogonal array with n columns. The *MCS* algorithm uses the backtracking approach to put new values for cells on the appended column. After appending completely a new column to create a new orthogonal array, it will check whether the new one is *LMC* matrix. If not, it will backtrack to search for another new orthogonal array. If yes, it will call *MCS* algorithm recursively to continue appending new columns until it reaches the target column size S.

```
Input: An orthogonal array X = [x_1, x_2, ..., x_n], n
  if IsComplete(X) then
    process X
  end if
  if IsExtendible(X) then
    for all extension X' = [x_1, x_2, ..., x_n, x'] of X do
      if IsNewOA(X') then
        if IsLexLeast(X') then
          MCS(X', n + 1)
        end if
      end if
    end for
  end if
```

Algorithm 1. MCS algorithm

2.2 Two Key Properties of Serial MCS Algorithm

We are interested in two important properties of *MCS* algorithm that can help us to turn the algorithm to a parallelism approach. Firstly, the key idea of *MCS* algorithm is it just remains arrays only if they are of a *LMC* form that can be tested locally and comprehensively. The isomorphism testing performed on single arrays has the advantage that it permits a distribution of the calculation over several processors. Secondly, an important property of the *MCS* algorithm is that an *LMC* array of column size k is guaranteed to be an extension of exactly one preceding *LMC* array of column size $k-1$. The outcome of this characteristic is all *LMC* arrays of column size k can be generated by extending *LMC* arrays of column size $k-1$.

3 A Step by Step Extending Parallelism Approach for EOA

Note that the *MCS* algorithm allows for generation from scratch when called with the root column parameter $n = t$, extending column by column until reaching the target column of size S. Such characteristic is an issue that need to be taken into account. Suppose that we want to generate for the next level $S+1$ after finishing generation at level S, in this case we must restart the procedure again from scratch and regenerate temporary levels. Obviously, this is a waste of time and cost since we do not reuse any result in the previous step. Looking back two important properties of *MCS* algorithm above, we could see that all *LMC* arrays of column size $S+1$ can be generated by extending *LMC* arrays of column size S.

That is a great motivation for us to propose a novel step-by-step parallelism approach for*EOA* based on the above important characteristic of *MCS* algorithm. The main idea is that we divide the generation of LMC arrays in separated small steps. At each step, we just extend concurrently all *LMC* matrices

of column size k from all LMC matrices of column size $k-1$. All LMC matrices of column size k will be stored and reused for the next step.

Assume that we want to enumerate all LMC matrices of $OA(N; s_1^{a_{1_s}} \cdot s_2^{a_{2_s}} \cdots s_m^{a_{m_s}}; t)$ with having S columns: $S = a_{1_s} + a_{2_s} + ... + a_{m_s}$. In particular, our proposed method for enumeration of orthogonal array of strength t could be described as follows:

1. At the initial stage, using the original serial algorithm MCS to generate all LMC matrices $OA(N; s_1^{a_{1_0}} \cdot s_2^{a_{2_0}} \cdots s_m^{a_{m_0}}; t)$ of an initial column size $k_0 = a_{1_0} + a_{2_0} + ... + a_{m_0}$
2. The data parallelism strategy is applied to generate all LMC matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m+1}; t)$ with column size $k = a_1 + a_2 + ... + a_m + 1$ from all LMC matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}; t)$ with column size $k - 1 = a_1 + a_2 + ... + a_m$. All results will be stored and reused in the next step.
3. The step-by-step extending phases continue until the all the LMC matrices of the target column size S are reached or there are no LMC matrix generated at an arbitrary column size k_{end} ($k_{end} < S$).

With the proposed approach, we have some advantages. Most importantly, with the reusing of LMC matrices, all LMC arrays of each level will be generated precisely one time and all new LMC arrays of a new size will be enumerated from all LMC generated arrays of the previous column size, not from scratch. It saves the cost for regenerating old objects of previous size, especially when number of column size S is huge. Moreover, reusing all LMC arrays of previous column size as inputs gives us a great chance to apply proper data parallelism strategies and could get an efficient speedup in each extending step (see more detail in Section 5).

4 Parallel Computing for Each Small Extending Step

In each step of our proposed approach, we generate concurrently all LMC arrays of column size k from all LMC arrays of size $k-1$. The parallelism computing will be applied in such every small step. We discuss about the parallel implementation of each step in this section.

Our previous parallel implementation have used a simple and inefficient method for dynamic load balancing and hence, the algorithm only achieves speedup with 8 processes in the best case [PSN10]. However, in this paper we propose a better dynamic load balancing scheme.

4.1 Drawback of Previous Work with Master-Slave Load Balancing Method

The input of every extending phase is all LMC arrays of size k - *1*. We use a process rank 0 as a *work manager* or a *master* for reading these inputs from

a single file and after that work manager will deliver matrix inputs to other processes for computing outputs. The unbalanced workload of individual processes will often lead to the unbalanced termination times of the processes. In our previous work [PSN10] we use a simple dynamic load balancing method named master-slave to balance the workload between computing processes. In particular, *master* sends an input matrix to every other process named *worker* each time. Worker process after finishing of extending for an input will request the master process for a new input. If *master* still has available inputs, it will send a new input for request *worker*. This request-respond process continues until master has no input and hence, the parallelism computing will terminate. However since time consuming to extend completely each input is highly different, we just could get a balanced load balancing when using up to 8 processes. To improve the balance between computing worker, we need to break the computation for an input matrix into smaller pieces and use an proper scheme to deliver these smaller pieces to other processes. Hence, a dynamic work-sharing method [DOS+07] in combination with a stack splitting procedure [FM87] has been implemented to guarantee a balanced workload between processes.

4.2 Convert MCS Algorithm by Using Stack Data Structure

In our new approach, each process executes the computation for an input matrix at a time by calling *MCS* algorithm for that input. To break the computation for an input matrix into smaller pieces, we have to convert the *MCS* algorithm to a new form that can easily break the computation for an input matrix into smaller tasks. Since *MCS* algorithm is a backtracking search algorithm, the conventional approach is converting the *MCS* algorithm into backtracking search form with stack data structure. With input in a form of matrix, the *MCS* algorithm appends a new column for that input. It puts in succession new values for each cell, from the first cell to the last cell of the new column. Since each cell could get more than one possible value, the *MCS* algorithm continues searching with the first possible value and the rest of possible values of current cell is stored properly by using the stack data structure. In particular, the constituent values from the first cell to that possible value of current cell construct a new *candidate part* of column. We push this new *candidate part* into the local stack of the process. When backtracking, the candidate path on top of the stack is popped and it becomes data for the search for the next cell. This work continues until the stack is empty.

By using a local stack for storing the temporal candidate parts of the new column on each process, we could break the computing into smaller tasks by splitting the current stack into two different smaller stacks. In particular, after a certain of time, for instance d seconds, we might cut off c candidate paths of the stack, group them as a work chunk. We send this work chunk to another idle process and after that, continue the computing with the rest of the stack. This idea will be explained clearly in next subsection below.

```
Input: An orthogonal array X = [x_1, x_2, ..., x_n], n
  repeat
    if Have been computing over d seconds AND size of stack ≥ 2*c then
        {c is chunk size}
        Cut off c candidate paths at the bottom of the current stack and group them as
        a work chunk
        Send the work chunk to the work manager process
    else
        Pop a candidate path on the top of the stack
        Use this candidate path as input to generate possible values for next cell
        if there are several possible values for next cell then
          for all possible value of the next cell - except the first value do
            Append every possible value of the next cell to the current candidate path
            to form a new candidate path
            Push the new candidate path to the stack
          end for
          Continue process with the first value of next cell
          if reaching the last cell to form a new OA X' then
            if IsLexLeast(X') then
              write X' to output file
            end if
          end if
        end if
    end if
  until the stack is empty ;
```

Algorithm 2. MCS algorithm with stack data structure

4.3 Work Sharing Method for Dynamic Load Balancing

To get a better dynamic load balancing between computing processes, we need a proper method for redistributing the workload. In this paper we use a dynamic load balancing named *work-sharing* method [DOS+07] in combination with a stack splitting procedure [FM87] to guarantee a minimal idle time of each processor.

Work sharing method is similar to master-slave method in that they use a work manager process to service work releases and work requests, and to detect termination. However, in contrast to master-slave method in which manager just sends every input matrix at the time to balance the workload, work sharing method balances the workload using a globally shared work queue, which is contained and managed by the work manager. This shared work queue contains unexplored candidate paths. Work in the shared queue is grouped into units of transferable work called chunks and the chunk size, c, parameter defines the number of candidate paths contained within a chunk.

In our implementation of work sharing method, for each process, after every fixed d duration time, if the local stack grows to be larger than two chunks, it will send a chunk of its local work to the work manager, allowing the surplus work to be performed by other processes that have become idle. Worker manager receives that chunk and store it into the shared queue. When a process has exhausted the work on its local stack, it sends a request to the work manager and gets another chunk of unexplored candidate paths from the shared queue of

```
if process_id = 0 then
  {being a work manager process}
  CREATE a work-queue to store work chunks sending from workers
  CREATE an idle-queue for workers waiting chunks
  READ all LMC matrices of from input file
  repeat
  WAIT for receiving a work request or a chunk of work from an arbitrary
  process
  if receive a work request then
        if still having some input matrix available then
          SEND a new input matrix for the request process
        else
          {have no input matrix left}
          if work-queue is not empty then
            POP a chunk on the top of work-queue out and send it to the request
            process
          else
            {No input matrix and no work chunk to send, puts the request worker to
            idle-queue}
            ENQUEUE the request process into idle worker queue
          end if
        end if
  else
        {receive a work chunk}
        if the idle-queue is not empty then
          DEQUEUE the idle-queue to get an idle process, send a received chunk to
          that idle process
        else
          PUT the received chunk into work-queue
        end if
  end if
  until there are no more input AND work-queue is empty AND idle-queue is full ;
  SEND FINISH signal to all other process
else
  {being a worker process}
  CREATE a local stack
  repeat
  SEND a request for a new input to master
  RECEIVE a work-return from work manager
  if work-return is just a new input matrix then
        PUSH the initial value 0 to the local stack
        CALL MCS algorithm to extend a new column to that input with current
        local stack
  else
        {receive a work chunk together with a new matrix input}
        PUSH c candidate paths of the received chunk to the local stack
        CALL MCS algorithm to extend a new column to that input with current
        local stack
  end if
  until receive FINISH signal from master ;
end if
```

Algorithm 3. Small-step extending parallelism algorithm

the master manager. In case of no work is immediately available in the shared queue, the process has to wait either for more work to become available or for all other processes to reach an agreement that the computation has ended.

Note that both chunk size c and duration time d are system parameters and could be modified according to different hardware and parameter sets of OA. Since the parallel performance of our work sharing implementation depends heavily on the speed with which the manager is able to service requests, the manager does not participate in the computation. We specify the parallel computing algorithm for each small extending step in Algorithm 3 and *MCS* algorithm with stack structure in Algorithm 2.

4.4 MPI Implementation

In our implementation, we use *MPI* as a tool to implement work sharing method. The work manager process, a process with rank 0, posts a nonblocking $MPI\_Irecv()$ function for each worker in the computation. A response from an worker could be either a work release or a work request. When receiving a response from an arbitrary worker, work manager will separate them by message tag. When a worker sends a work chunk to the work manager, the work manager push the chunk into the work queue in case of no idle worker, otherwise it sends immediately the work chunk to the first idle worker in the idle queue. In case of a worker requesting a work chunk it sends a work request message to the manager and waits for a response. If the work manager has no input matrix and its work queue is empty, it adds the process to the idle queue and services the request once more work chunk becomes available. When the manager detects that all processes have become idle and no work is available in the work queue, it knows that the computation has terminated and it sends all processes a FINISH message.

All worker processes write output matrices concurrently to a single file. In writing output matrices, the order of generated output is not a concern in our algorithm. After generating a new output, worker just writes new result to the end of the output file. This process could be done with the support of the shared pointer subroutine $MPI\_File\_iwrite\_shared()$ of *MPI IO* [Mes97].

5 Experimental Evaluation

Our work has been executed on the *Hercules* cluster of La Trobe university, Australia. The cluster has 24 node AMD four-core/dual cpu servers, including 1 server as a control node (resulting in 177 cores for computation) with 32 GB RAM per CPU. There is a high speed infiniband interconnect with network switch and Gigabit Ethernet connection to the network, in addition to 1 TB disk space per node.

5.1 Experiments for Small Extending Step

To evaluate the efficiency of the approach in parallel computing for each small extending step, we first try to do an experiment having the same parameter set

with previous work [PSN10]. For more detail, at the initial stage, the *Extend-column* algorithm is used to generate all 89 lex-least-matrices of $OA(72; 3 \cdot 2^4; 3)$. After that, those 89 lex-least matrices of $OA(72; 3 \cdot 2^4; 3)$ have been chosen as inputs for enumerating all non-isomorphism class of $OA(72; 3 \cdot 2^5; 3)$. On each run, we collect the maximal execution time. We choose $c = 4$ and $d = 7$ (seconds) in these experiments. To easily compare, we show new results along with previous results [PSN10] in Table 1.

Table 1. Execution time

Number of processes	Execution time	Previous work
2 (1 + 1)	476.9'	465.9'
4 (1 + 3)	160.5'	156.23'
8 (1 + 7)	68.9'	79.6'
16 (1 + 15)	32.7'	57.45'
32 (1 + 31)	17.6'	none
64 (1 + 63)	10.7'	none
128 (1 + 127)	5.9'	none

Since in our experiments, we always need at least two processes (one as a work manager and at least one as a worker) for every parallelism experiment, we are more concerned with the relative speedup factor of the algorithm. We initial the experiments with number of processes is two. The formula for the relative speedup is given as follows: $Speedup(p) = \frac{T(p)}{T(2p)}$ The result is given in Table 2.

Table 2. Relative speedup

Number of processes	Relative speedup	Previous work
4 (1 + 3)	2.97	2.98
8 (1 + 7)	2.33	1.97
16 (1 + 15)	2.11	1.38
32 (1 + 31)	1.85	none
64 (1 + 63)	1.64	none
128 (1 + 127)	1.80	none

In previous work [PSN10], because of inefficiency of dynamic load balancing scheme, the number of processes could be used was just 16 processes. However, in our experiments with new approach, the number of processes have been doubled up to 128. Moreover, the relative speedup is really high which nearly equal to 2 even if we double the number of processes up to 128. This outcome shows the significant efficiency of the proposed parallelism approach.

5.2 Dynamic Load Balancing

Using stack splitting procedure, our method has gained a great deal of load balancing, in which the execution times are nearly identical between all processes. This could be seen in the Figure 1.

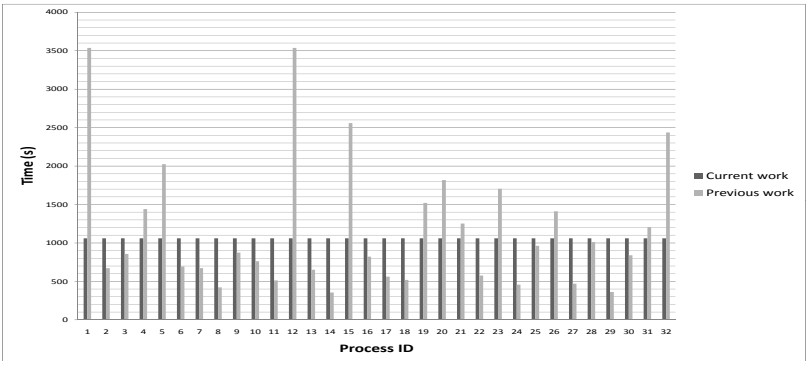


Fig. 1. Comparison execution time of 32-processes experiment of previous work and current work

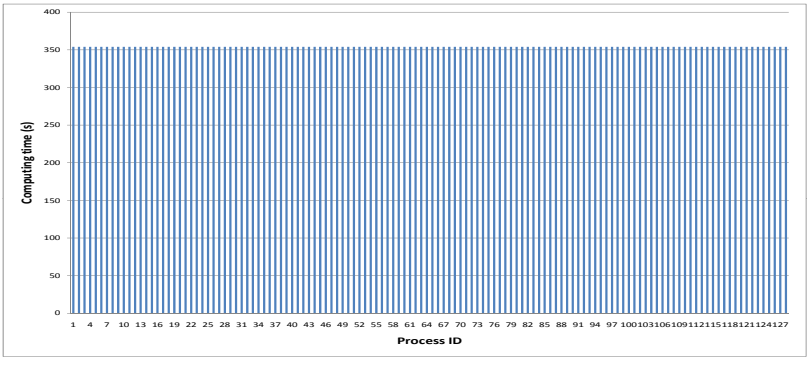


Fig. 2. Execution time of 128 processes from computing $OA(72; 3 \cdot 2^5; 3)$

Another significant finding is that when using 128 processes for computing, the dynamic load balancing works extremely well. This could be seen in this diagram Figure 2

Looking at the two diagrams and the speedup results, we could infer that the work sharing load balancing method in combination with stack splitting method could get a significant speedup when we double up the number of processes up to 128, whilst maintaining a good dynamic load balancing between computing processes.

5.3 Impact of Duration Time *d* and Chunk Size *c*

Table 3 and table 4 show the execution time of the algorithm when we use 32 process with vary of duration time d and chunk size c. To identify the impact of chunk size in the execution time, we have fixed the value of duration time ($d = 7$) and varied the chunk size c. Similarly, by fixing the chunk size $c = 4$, we are able to focus on the impact of duration time d in the execution time. Table 3 shows that when we increase duration time from $d = 7$ to $d = 23$, the execution time doesn't change a lot. It shows that duration time d might not significantly affect execution time. However, when we increase the value of chunk size c, the execution time increases dramatically. This fact shows that very large chunk sizes reduce the number of chances for cutting of the local stack, and hence decrease the number of chances for performing load balancing, leads to poor work distribution.

In fact, the choice of duration time d and chunk size c could vary and depend on the parameter sets of OA and the number of input matrices. We should choose system parameter d and c carefully to be able to get an optimization performance.

Table 3. Execution time using 32 processes (fix c = 4) vs duration time

Duration time d	Execution time
7	18.7'
11	17.5'
15	18.6'
19	18.95'
23	21.3'

Table 4. Execution time using 32 processes (fix d = 7) vs chunk size

Chunk size c	Execution time
2	16.8'
4	17.6'
6	21.3'
8	27.75'
10	54.9'

6 Conclusion and Future Work

In this paper, we have proposed a new efficient dynamic load balancing method for parallel computing EOA based on a step-by-step extending approach. There are some important online repositories of OA that are published so far such as a number of OA strength 2, 3 and 4 have been discovered by Eric, Pieter and Man on 2010 [SEN10]. However, there are still rooms for discovering. For instance,

the existence of $OA(72; 3^2 \cdot 2^t; 3)$ with $t \geq 13$ is still an open case for us so far [Ngu08]. In fact, constructing such the $OA(72; 3^2 \cdot 2^t; 3)$ with $t \geq 13$ is a big challenge since it requires a very huge computing work. Therefore, a parallelism method for enumeration of such orthogonal array could be a good solution since it allows us to reduce the execution time significantly and generate new results using the power of high performance computing system. This will be the main theme in our next article *Enumeration of* $OA(72; 3^2 \cdot 2^t; 3)$ *with* $t \geq 13$.

References

[DOS+07] Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.-W.: Dynamic load balancing of unbalanced computations using message passing. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8 (March 2007)

[FM87] Finkel, R., Manber, U.: Diba distributed implementation of backtracking. ACM Trans. Program. Lang. Syst. 9, 235–256 (1987)

[Mes97] Message-Passing Interface Forum. MPI-2.0: Extensions to the Message-Passing Interface, ch 9. MPI Forum (June 1997)

[Ngu05] Nguyen, M.: Computer-algebraic methods for the construction of designs of experiments. Ph.D. Thesis, Technische Universiteit Eindhoven (2005)

[Ngu08] Nguyen, M.V.M.: Some new constructions of strength 3 mixed orthogonal arrays. Journal of Statistical Planning and Inference 138(1), 220–233 (2008)

[PSN10] Phan, H., Soh, B., Nguyen, M.: A step-by-step extending parallelism approach for enumeration of combinatorial objects. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010. LNCS, vol. 6081, pp. 463–475. Springer, Heidelberg (2010)

[SEN10] Schoen, E.D., Eendebak, P.T., Nguyen, M.V.M.: Complete enumeration of pure-level and mixed-level orthogonal array. Journal of Combinatorial Designs 18(2), 123–140 (2010)

[TGM00] Tsai, P.-W., Gilmour, S.G., Mead, R.: Projective three-level main effects designs robust to model uncertainty. Biometrika 87(2), 467–475 (2000)

Author Index