Ching-Hsien Hsu
Laurence T. Yang
Jong Hyuk Park
Sang-Soo Yeo (Eds.)

LNCS 6081

Algorithms and Architectures for Parallel Processing

10th International Conference, ICA3PP 2010
Busan, Korea, May 2010
Proceedings, Part I

1 Part I

Springer

Ching-Hsien Hsu Laurence T. Yang
Jong Hyuk Park Sang-Soo Yeo (Eds.)

Algorithms and Architectures for Parallel Processing

10th International Conference, ICA3PP 2010
Busan, Korea, May 21-23, 2010
Proceedings, Part I

Springer

Volume Editors

Ching-Hsien Hsu
Chung Hua University, Department of Computer Science
and Information Engineering
Hsinchu, 300 Taiwan, China
E-mail: chh@chu.edu.tw

Laurence T. Yang
St. Francis Xavier University, Department of Computer Science,
Antigonish, NS, B2G 2W5, Canada
E-mail: ltyang@stfx.ca

Jong Hyuk Park
Seoul National University of Technology
Department of Computer Science and Engineering
Nowon-gu, Seoul, 139-742, Korea
E-mail: parkjonghyuk1@hotmail.com

Sang-Soo Yeo
Mokwon University, Division of Computer Engineering
Daejeon 302-729, Korea
E-mail: ssyeo@mokwon.ac.kr

Library of Congress Control Number: 2010926694

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

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

ISSN	0302-9743
ISBN-10	3-642-13118-2 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-13118-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper   06/3180

Preface

It is our great pleasure to welcome you to the proceedings of the 10th annual event of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP).

ICA3PP is recognized as the main regular event covering the many dimensions of parallel algorithms and architectures, encompassing fundamental theoretical approaches, practical experimental projects, and commercial components and systems. As applications of computing systems have permeated every aspect of daily life, the power of computing systems has become increasingly critical. Therefore, ICA3PP 2010 aimed to permit researchers and practitioners from industry to exchange information regarding advancements in the state of the art and practice of IT-driven services and applications, as well as to identify emerging research topics and define the future directions of parallel processing.

We received a total of 157 submissions this year, showing by both quantity and quality that ICA3PP is a premier conference on parallel processing. In the first stage, all papers submitted were screened for their relevance and general submission requirements. These manuscripts then underwent a rigorous peer-review process with at least three reviewers per paper. In the end, 47 papers were accepted for presentation and included in the main proceedings, comprising a 30% acceptance rate. To encourage and promote the work presented at ICA3PP 2010, we are delighted to inform the authors that some of the papers will be accepted in special issues of the Journal of Grid Computing, the Journal of Parallel and Distributed Computing, the International Journal of High Performance Computing and Networking, the Journal of Database Management and the Journal of Information Processing Systems. All of these journals have played a prominent role in promoting the development and use of parallel and distributed processing.

An international conference of this scale requires the support of many people. First of all, we would like to thank the Steering Committee Chairs, Andrzej Goscinski, Yi Pan and Wanlei Zhou, for nourishing the conference and guiding its course. We appreciate the participation of the keynote speakers, Wanlei Zhou and Rajkumar Buyya, whose speeches greatly benefited the audience. We are also indebted to the members of the Program Committee, who put in a lot of hard work and long hours to review each paper in a professional way. Thanks to them all for their valuable time and effort in reviewing the papers. Without their help, this program would not be possible. Special thanks go to Tony Li Xu and Sang-Soo Yeo for their help with the conference web, paper submission and reviewing system and a lot of detailed work, which facilitated the overall process. Thanks also go to the entire local arrangements committee for their help in making the conference a wonderful success. We take this opportunity to thank all the authors, participants and session chairs for their valuable efforts, many of whom traveled long distances to attend this conference and make their valuable contributions. Last but not least, we would like express our gratitude to all of the organizations that supported our efforts to bring the conference to fruition. We are grateful to Springer for publishing the proceedings.

The conference was held in the beautiful city of Busan. Besides the academic nature of the conference, the whole city is ample with beautiful scenic spots and cultural sources. We hope that all our academic entertainments and at the same time you were able to see the magnificent natural beauty.

This conference owed its success to the support of many academic and industrial organizations. Most importantly, we owe a lot to of inspiration all conference participants for their contributions to the conference the participants enjoyed

May 2010

Laurence T. Yang
Jong Hyuk Park
J. Daniel Garcia
Ching-Hsien (Robert) Hsu
Alfredo Cuzzocrea
Xiaojun Cao

Conference Committees

Honorary Chair

Doo-soon Park	SoonChunHyang University, Korea

Steering Committee Chairs

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

Advisory Committee

H.J. Siegel	Colorado State University, USA
Shi-Jinn Horng	National United University, Taiwan
Young-Sik Jeong	Wonkwang University, Korea
Kai Hwang	University of Southern California, USA

General Chairs

Laurence T. Yang	St. Francis Xavier University, Canada
Jong Hyuk Park	Seoul National University of Technology, Korea
J. Daniel Garcia	University Carlos III of Madrid, Spain

General Vice-Chairs

Sang-Soo Yeo	Mokwon University, Korea
Tony Li Xu	St Francis Xavier University, Canada
Ki-Ryong Kwon	Pukyung National University, Korea

Program Chairs

Alfredo Cuzzocrea	ICAR, National Research Council and University of Calabria, Italy
Ching-Hsien Hsu	Chung Hua University, Taiwan
Xiaojun Cao	Georgia State University, USA

Workshop Chairs

Kuo-Chan Huang	National Taichung University, Taiwan
Yu Liang	Central State University, USA

Publication Chairs

Deok Gyu Lee	ETRI, Korea
Jongsung Kim	Korea Universtiy, Korea
Soo-Kyun Kim	PaiChai University, Korea

Publicity Chairs

Roland Wagner	Univ. of Linz, Austria
Tohru Kikuno	Osaka University, Japan
Kuan-Ching Li	Providence University, Taiwan

Local Arrangements Chairs

Kyung Hyun Rhee	Pukyong National University, Korea
Changhoon Lee	Hanshin University, Korea
Howon Kim	Pusan National University, Korea

International Program Committee

Jemal Abawajy	Deakin University, Australia
Ahmad S. AI-Mogren	AI Yamamah University, Saudi Arabia
Hüseyin Akcan	Izmir University of Economics, Turkey
Giuseppe Amato	ISTI-CNR, Italy
Cosimo Anglano	Università del Piemonte Orientale, Italy
Alagan Anpalagan	Ryerson University, Canada
Amnon Barak	The Hebrew University of Jerusalem, Israel
Novella Bartolini	University of Rome "La Sapienza", Italy
Alessio Bechini	Alessio Bechini, University of Pisa, Italy
Ladjel Bellatreche	ENSMA, France
Ateet Bhalla	Technocrats Institute of Technology, India
Taisuke Boku	University of Tsukuba, Japan
Angelo Brayner	University of Fortaleza, Brazil
Massimo Cafaro	University of Salento, Lecce, Italy
Mario Cannataro	University “Magna Græcia” of Catanzaro, Italy
Jiannong Cao	Hong Kong Polytechnic University, Hong Kong
Andre C.P.L.F. de Carvalho	Universidade de Sao Paulo, Brazil
Denis Caromel	University of Nice Sophia Antipolis-INRIA-CNRS-IUF, France

Tania Cerquitelli	Politecnico di Torino, Italy
Hangbae Chang	Daejin University, Korea
Ruay-Shiung Chang	National Dong Hwa University, Taiwan
Yue-Shan Chang	National Taipei University, Taiwan
Jinjun Chen	Swinburne University of Technology, Australia
Tzung-Shi Chen	National University of Tainan, Taiwan
Zizhong Chen	Colorado School of Mines, USA
Allen C. Cheng	University of Pittsburgh, USA
Francis Chin	University of Hong Kong, Hong Kong
Michele Colajanni	Università di Modena e Reggio Emilia, Italy
Carmela Comito	University of Calabria, Italy
Raphaël Couturier	University of Franche Comte, France
Mieso Denko	University of Guelph, Canada
Bronis R. de Supinski	Lawrence Livermore National Laboratory, USA
Julius Dichter	University of Bridgeport, USA
Der-Rong Din	National Changhua University of Education, Taiwan
Susan K. 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, Berkeley, USA
Jerry Zeyu Gao	San Jose State University, USA
Jinzhu Gao	University of the Pacific, Stockton, CA, USA
Irene Garrigós	University of Alicante, Spain
Amol Ghoting	IBM T. J. Watson Research Center, USA
Harald Gjermundrod	University of Nicosia, Cyprus
Janice Gu	Auburn University, USA
Hyoil Han	Drexel University, USA
Houcine Hassan	Universidad Politecnica de Valencia, Spain
Pilar Herrero	Universidad Politécnica de Madrid, Spain
Michael Hobbs	Deakin University, Australia
JoAnne Holliday	Santa Clara University, USA
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
Young-Sik Jeong	Wonkwang University, Korea
Qun Jin	Waseda University, Japan
Xiaolong Jin	University of Bradford, UK
Soo-Kyun Kim	PaiChai University, Korea
Jongsung Kim	Kyungnam University, Korea
Dan Komosny	Brno University of Technology, Czech Republic
Gregor von Laszewski	Rochester Institute of Technology, USA
Changhoon Lee	Hanshin University, Korea
Deok Gyu Lee	ETRI, Korea
Yang Sun Lee	Chosun University, Korea
Laurent Lefevre	INRIA, University of Lyon, France

External Reviewers

Some of the names listed below came from second hand data sources, so it was not always possible to confirm correct spellings. With apologies to reviewers not accurately listed

George Pallis
Amirreza Tahamtan
Fengguang Song
Zhanpeng Jin
Joseph Oresko
Eugenio Cesario
Uwe Tangen
Todd Gamblin
Soo-Kyun Kim
Giuseppe M. Bernava
Humberto Ortiz-Zauzaga
Rodrigo Mello
Rafael Arce-Nazario
Martin Koehler
Sotirios Ziavras
Anastasios Gounaris
Bo Wang
Wuu Yang
Heng Qi
Yu Wang
Santosh Kabbur
Chris Kauffman
Le Dinh Minh
Emmanuel Udoh

Amirreza Tahamtan
Chris Taylor
Houcine Hassan
Kathryn Mohror
Joseph Oresko
Zhanpeng Jin
Fengguang Song
Hongxing Li
Ruay-Shiung Chang
Amnon Barak
Judit Bar-Ilan
Henrique Kawakami
Gianluigi Folino
Alexandro Baldassin
Diego F. Aranha
Claudio Vairo
Ahmad Al-Mogren
Casiano Rodriguez-Leon
Rubem Pereira
Dakai Zhu
Marcel Rosu
Max Walter
Paul Ruth
Wei Huang

Table of Contents – Part I

Keynote Papers

Efficient Web Browsing with Perfect Anonymity Using Page Prefetching 1
Shui Yu, Theerasak Thapngam, Su Wei, and Wanlei Zhou

InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services 13
Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros

Parallel Algorithms

Scalable Co-clustering Algorithms 32
Bongjune Kwon and Hyuk Cho

Parallel Pattern Matching with Swaps on a Linear Array 44
Fouad B. Chedid

Parallel Prefix Computation in the Recursive Dual-Net 54
Yamin Li, Shietung Peng, and Wanming Chu

A Two-Phase Differential Synchronization Algorithm for Remote Files 65
Yonghong Sheng, Dan Xu, and Dongsheng Wang

A New Parallel Method of Smith-Waterman Algorithm on a Heterogeneous Platform 79
Bo Chen, Yun Xu, Jiaoyun Yang, and Haitao Jiang

Improved Genetic Algorithm for Minimizing Periodic Preventive Maintenance Costs in Series-Parallel Systems 91
Chung-Ho Wang and Te-Wei Lin

A New Hybrid Parallel Algorithm for MrBayes 102
Jianfu Zhou, Gang Wang, and Xiaoguang Liu

Research and Design of Deployment Framework for Blade-Based Data Center 113
Haiping Qu, Xiuwen Wang, Lu Xu, Jiangang Zhang, and Xiaoming Han

Query Optimization over Parallel Relational Data Warehouses in Distributed Environments by Simultaneous Fragmentation and Allocation 124
Ladjel Bellatreche, Alfredo Cuzzocrea, and Soumia Benkrid

Parallel Architectures

Function Units Sharing between Neighbor Cores in CMP 136
Tianzhou Chen, Jianliang Ma, Hui Yuan, Jingwei Liu, and Guanjun Jiang

A High Efficient On-Chip Interconnection Network in SIMD CMPs 149
Dan Wu, Kui Dai, Xuecheng Zou, Jinli Rao, and Pan Chen

Network-on-Chip Routing Algorithms by Breaking Cycles 163
Minghua Tang and Xiaola Lin

A Fair Thread-Aware Memory Scheduling Algorithm for Chip Multiprocessor 174
Danfeng Zhu, Rui Wang, Hui Wang, Depei Qian, Zhongzhi Luan, and Tianshu Chu

Efficient Partitioning of Static Buses for Processor Arrays of Small Size 186
Susumu Matsumae

Formal Proof for a General Architecture of Hybrid Prefix/Carry-Select Adders 193
Feng Liu, Qingping Tan, Xiaoyu Song, and Gang Chen

An Efficient Non-Blocking Multithreaded Embedded System 205
Joseph M. Arul, Tsung-Yun Chen, Guan-Jie Hwang, Hua-Yuan Chung, Fu-Jiun Lin, and You-Jen Lee

A Remote Mirroring Architecture with Adaptively Cooperative Pipelining 215
Yongzhi Song, Zhenhai Zhao, Bing Liu, Tingting Qin, Gang Wang, and Xiaoguang Liu

SV: Enhancing SIMD Architectures via Combined SIMD-Vector Approach 226
Libo Huang and Zhiying Wang

A Correlation-Aware Prefetching Strategy for Object-Based File System 236
Julei Sui, Jiancong Tong, Gang Wang, and Xiaoguang Liu

An Auxiliary Storage Subsystem to Distributed Computing Systems for External Storage Service 246
MinHwan Ok

Grid/Cluster Computing

Checkpointing and Migration of Communication Channels in Heterogeneous Grid Environments 254
John Mehnert-Spahn and Michael Schoettner

On-Line Task Granularity Adaptation for Dynamic Grid Applications 266
Nithiapidary Muthuvelu, Ian Chai, Eswaran Chikkannan, and Rajkumar Buyya

Message Clustering Technique towards Efficient Irregular Data Redistribution in Clusters and Grids 278
Shih-Chang Chen, Tai-Lung Chen, and Ching-Hsien Hsu

Multithreading of Kostka Numbers Computation for the BonjourGrid Meta-desktop Grid Middleware 287
Heithem Abbes, Franck Butelle, and Christophe Cérin

Adaptable Scheduling Algorithm for Grids with Resource Redeployment Capability 299
Cho-Chin Lin and Chih-Hsuan Hsu

Using MPI on PC Cluster to Compute Eigenvalues of Hermitian Toeplitz Matrices 313
Fazal Noor and Syed Misbahuddin

Cloud Computing/Virtualization Techniques

idsocket: API for Inter-domain Communications Base on Xen 324
Liang Zhang, Yuein Bai, and Cheng Luo

Strategy-Proof Dynamic Resource Pricing of Multiple Resource Types on Federated Clouds 337
Marian Mihailescu and Yong Meng Teo

Adapting Market-Oriented Scheduling Policies for Cloud Computing 351
Mohsen Amini Salehi and Rajkumar Buyya

A High Performance Inter-VM Network Communication Mechanism 363
Yuebin Bai, Cheng Luo, Cong Xu, Liang Zhang, and Huiyong Zhang

On the Effect of Using Third-Party Clouds for Maximizing Profit 381
Young Choon Lee, Chen Wang, Javid Taheri, Albert Y. Zomaya, and Bing Bing Zhou

A Tracing Approach to Process Migration for Virtual Machine Based on Multicore Platform 391
Liang Zhang, Yuebin Bai, and Xin Wei

GPU Computing and Applications

Accelerating Dock6's Amber Scoring with Graphic Processing Unit 404
Hailong Yang, Bo Li, Yongjian Wang, Zhongzhi Luan, Depei Qian, and Tianshu Chu

Optimizing Sweep3D for Graphic Processor Unit 416
Chunye Gong, Jie Liu, Zhenghu Gong, Jin Qin, and Jing Xie

Modular Resultant Algorithm for Graphics Processors 427
Pavel Emeliyanenko

A Novel Scheme for High Performance Finite-Difference Time-Domain (FDTD) Computations Based on GPU 441
Tianshu Chu, Jian Dai, Depei Qian, Weiwei Fang, and Yi Liu

Parallel Programming, Performance Evaluation

A Proposed Asynchronous Object Load Balancing Method for Parallel 3D Image Reconstruction Applications 454
Jose Antonio Alvarez-Bermejo and Javier Roca-Piera

A Step-by-Step Extending Parallelism Approach for Enumeration of Combinatorial Objects 463
Hien Phan, Ben Soh, and Man Nguyen

A Study of Performance Scalability by Parallelizing Loop Iterations on Multi-core SMPs 476
Prakash Raghavendra, Akshay Kumar Behki, K. Hariprasad, Madhav Mohan, Praveen Jain, Srivatsa S. Bhat, V.M. Thejus, and Vishnumurthy Prabhu

Impact of Multimedia Extensions for Different Processing Element Granularities on an Embedded Imaging System 487
Jong-Myon Kim

Fault-Tolerant/Information Security and Management

Reducing False Aborts in STM Systems 499
Daniel Nicácio and Guido Araújo

Fault-Tolerant Node-to-Set Disjoint-Path Routing in Hypercubes 511
Antoine Bossard, Keiichi Kaneko, and Shietung Peng

AirScope: A Micro-scale Urban Air Quality Management System 520
Jung-Hun Woo, HyungSeok Kim, Sang Boem Lim, Jae-Jin Kim, Jonghyun Lee, Rina Ryoo, and Hansoo Kim

Wireless Communication Network

Design of a Slot Assignment Scheme for Link Error Distribution on Wireless Grid Networks 528
Junghoon Lee, Seong Baeg Kim, and Mikyung Kang

Wireless Bluetooth Communications Combine with Secure Data Transmission Using ECDH and Conference Key Agreements 538
Hua-Yi Lin and Tzu-Chiang Chiang

Robust Multicast Scheme for Wireless Process Control on Traffic Light Networks 549
Junghoon Lee, Gyung-Leen Park, Seong-Baeg Kim, Min-Jae Kang, and Mikyung Kang

A Note-Based Randomized and Distributed Protocol for Detecting Node Replication Attacks in Wireless Sensor Networks 559
Xiangshan Meng, Kai Lin, and Keqiu Li

Author Index 571

Table of Contents – Part II

The 2010 International Symposium on Frontiers of Parallel and Distributed Computing (FPDC 2010)

Parallel Programming and Multi-core Technologies

Efficient Grid on the OTIS-Arrangment Network 1
Ahmad Awwad, Bassam Haddad, and Ahmad Kayed

Single Thread Program Parallelism with Dataflow Abstracting Thread 11
Tianzhou Chen, Xingsheng Tang, Jianliang Ma, Lihan Ju, Guanjun Jiang, and Qingsong Shi

Parallel Programming on a Soft-Core Based Multi-core System 22
Liang-Teh Lee, Shin-Tsung Lee, and Ching-Wei Chen

Dynamic Resource Tuning for Flexible Core Chip Multiprocessors 32
Yongqing Ren, Hong An, Tao Sun, Ming Cong, and Yaobin Wang

Ensuring Confidentiality and Integrity of Multimedia Data on Multi-core Platforms 42
Eunji Lee, Sungju Lee, Yongwha Chung, Hyeonjoong Cho, and Sung Bum Pan

A Paradigm for Processing Network Protocols in Parallel 52
Ralph Duncan, Peder Jungck, and Kenneth Ross

Real-Time Task Scheduling on Heterogeneous Two-Processor Systems 68
Chin-Fu Kuo and Ying-Chi Hai

Grid/Cluster Computing

A Grid Based System for Closure Computation and Online Service 79
Wing-Ning Li, Donald Hayes, Jonathan Baran, Cameron Porter, and Tom Schweiger

A Multiple Grid Resource Broker with Monitoring and Information Services 90
Chao-Tung Yang, Wen-Jen Hu, and Bo-Han Chen

Design Methodologies of Workload Management through Code Migration in Distributed Desktop Computing Grids 100
Makoto Yoshida and Kazumine Kojima

Dynamic Dependent Tasks Assignment for Grid Computing 112
Meddeber Meriem and Yagoubi Belabbas

Implementation of a Heuristic Network Bandwidth Measurement for Grid Computing Environments ... 121
Chao-Tung Yang, Chih-Hao Lin, and Wen-Jen Hu

Parallel Algorithms, Architectures and Applications

An Efficient Circuit-Switched Broadcasting in Star Graph ... 131
Cheng-Ta Lee and Yeong-Sung Lin

Parallel Domain Decomposition Methods for High-Order Finite Element Solutions of the Helmholtz Problem ... 136
Youngjoon Cha and Seongjai Kim

Self-Organizing Neural Grove and Its Distributed Performance ... 146
Hirotaka Inoue

A Massively Parallel Hardware for Modular Exponentiations Using the m-ary Method ... 156
Marcos Santana Farias, Sérgio de Souza Raposo, Nadia Nedjah, and Luiza de Macedo Mourelle

Emulation of Object-Based Storage Devices by a Virtual Machine ... 166
Yi-Chiun Fang, Chien-Kai Tseng, and Yarsun Hsu

Balanced Multi-process Parallel Algorithm for Chemical Compound Inference with Given Path Frequencies ... 178
Jiayi Zhou, Kun-Ming Yu, Chun Yuan Lin, Kuei-Chung Shih, and Chuan Yi Tang

Harnessing Clusters for High Performance Computation of Gene Expression Microarray Comparative Analysis ... 188
Philip Church, Adam Wong, Andrzej Goscinski, and Christophe Lefèvre

Mobile Computing/Web Services

Semantic Access Control for Corporate Mobile Devices ... 198
Tuncay Ercan and Mehmet Yıldız

A New Visual Simulation Tool for Performance Evaluation of MANET Routing Protocols ... 208
Md. Sabbir Rahman Sakib, Nazmus Saquib, and Al-Sakib Khan Pathan

A Web Service Composition Algorithm Based on Global QoS Optimizing with MOCACO ... 218
Wang Li and He Yan-xiang

Distributed Operating System/P2P Computing

Experiences Gained from Building a Services-Based Distributed Operating System 225
Andrzej Goscinski and Michael Hobbs

Quick Forwarding of Queries to Relevant Peers in a Hierarchical P2P File Search System 235
Tingting Qin, Qi Cao, Qiying Wei, and Satoshi Fujita

iCTPH: An Approach to Publish and Lookup CTPH Digests in Chord 244
Zhang Jianzhong, Pan Kai, Yu Yuntao, and Xu Jingdong

Fault-Tolerant and Information Security

Toward a Framework for Cloud Security 254
Michael Brock and Andrzej Goscinski

Cluster-Fault-Tolerant Routing in Burnt Pancake Graphs 264
Nagateru Iwasawa, Tatsuro Watanabe, Tatsuya Iwasaki, and Keiichi Kaneko

Edge-Bipancyclicity of All Conditionally Faulty Hypercubes 275
Chao-Ming Sun and Yue-Dar Jou

The 2010 International Workshop on High Performance Computing Technologies and Applications (HPCTA 2010)

Session I

Accelerating Euler Equations Numerical Solver on Graphics Processing Units 281
Pierre Kestener, Frédéric Château, and Romain Teyssier

An Improved Parallel MEMS Processing-Level Simulation Implementation Using Graphic Processing Unit 289
Yupeng Guo, Xiaoguang Liu, Gang Wang, Fan Zhang, and Xin Zhao

Solving Burgers' Equation Using Multithreading and GPU 297
Sheng-Hsiu Kuo, Chih-Wei Hsieh, Reui-Kuo Lin, and Wen-Hann Sheu

Support for OpenMP Tasks on Cell Architecture 308
Qian Cao, Changjun Hu, Haohu He, Xiang Huang, and Shigang Li

Session II

A Novel Algorithm for Faults Acquiring and Locating on Fiber Optic Cable Line 318
Ning Zhang, Yan Chen, Naixue Xiong, Laurence T. Yang, Dong Liu, and Yuyuan Zhang

A Parallel Distributed Algorithm for the Permutation Flow Shop Scheduling Problem 328
Samia Kouki, Talel Ladhari, and Mohamed Jemni

A Self-Adaptive Load Balancing Strategy for P2P Grids 338
Po-Jung Huang, You-Fu Yu, Quan-Jie Chen, Tian-Liang Huang, Kuan-Chou Lai, and Kuan-Ching Li

Embedding Algorithms for Star, Bubble-Sort, Rotator-Faber-Moore, and Pancake Graphs 348
Mihye Kim, Dongwan Kim, and Hyeongok Lee

Session III

Performance Estimation of Generalized Statistical Smoothing to Inverse Halftoning Based on the MTF Function of Human Eyes 358
Yohei Saika, Kouki Sugimoto, and Ken Okamoto

Power Improvement Using Block-Based Loop Buffer with Innermost Loop Control 368
Ming-Yuan Zhong and Jong-Jiann Shieh

An Efficient Pipelined Architecture for Fast Competitive Learning 381
Hui-Ya Li, Chia-Lung Hung, and Wen-Jyi Hwang

Merging Data Records on EREW PRAM 391
Hazem M. Bahig

The 2010 International Workshop on Multicore and Multithreaded Architecture and Algorithms (M2A2 2010)

Session I

Performance Modeling of Multishift QR Algorithms for the Parallel Solution of Symmetric Tridiagonal Eigenvalue Problems 401
Takafumi Miyata, Yusaku Yamamoto, and Shao-Liang Zhang

A Parallel Solution of Large-Scale Heat Equation Based on Distributed Memory Hierarchy System 413
Tangpei Cheng, Qun Wang, Xiaohui Ji, and Dandan Li

A New Metric for On-line Scheduling and Placement in Reconfigurable Computing Systems 422
Maisam Mansub Bassiri and Hadi Shahriar Shahhoseini

Session II

Test Data Compression Using Four-Coded and Sparse Storage for Testing Embedded Core 434
Zhang Ling, Kuang Ji-shun, and You zhi-qiang

Extending a Multicore Multithread Simulator to Model Power-Aware Hard Real-Time Systems 444
José Luis March, Julio Sahuquillo, Houcine Hassan, Salvador Petit, and José Duato

Real-Time Linux Framework for Designing Parallel Mobile Robotic Applications 454
Joan Aracil, Carlos Domínguez, Houcine Hassan, and Alfons Crespo

Author Index 465

Efficient Web Browsing with Perfect Anonymity Using Page Prefetching

Shui Yu, Theerasak Thapngam, Su Wei, and Wanlei Zhou

School of Information Technology
Deakin University
Burwood, VIC 3125, Australia
{syu,tthap,suwei,wanlei}@deakin.edu.au

Abstract. Anonymous web browsing is a hot topic with many potential applications for privacy reasons. The current dominant strategy to achieve anonymity is packet padding with dummy packets as cover traffic. However, this method introduces extra bandwidth cost and extra delay. Therefore, it is not practical for anonymous web browsing applications. In order to solve this problem, we propose to use the predicted web pages that users are going to access as the cover traffic rather than dummy packets. Moreover, we defined anonymity level as a metric to measure anonymity degrees, and established a mathematical model for anonymity systems, and transformed the anonymous communication problem into an optimization problem. As a result, users can find tradeoffs among anonymity level and cost. With the proposed model, we can describe and compare our proposal and the previous schemas in a theoretical style. The preliminary experiments on the real data set showed the huge potential of the proposed strategy in terms of resource saving.

Keywords: perfect anonymity, web browsing, prefetch.

1 Introduction

The purpose of this paper is to present an efficient and novel way for web browsing with perfect anonymity, moreover, we also target on solving the contradictory constraints between high anonymity and low delay in the applications of web browsing. Anonymous web browsing becomes a hot topic recently because of the booming of Internet based applications, such as information retrieval, online shopping, online voting. In order to meet the privacy needs of these kinds of activities, a number of anonymous systems have been proposed, implemented and practiced on the Internet, such as, mix and mix networks [1], crowds [2], onion routing [3], tor [4]. The best solution against attacks is to design a system with perfect anonymity which can never be breached in any condition, and it is possible according to Shannon's perfect secrecy theory (we use perfect anonymity to replace perfect secrecy in this paper)[5], however, the cost for perfect anonymity is extremely high, and it may not be achievable in practice under some constraints, for example, the delay constraint for web browsing.

The anonymity issue on web browsing has been widely explored in recent years, such as various attacks and defenses on the tor system [6],[7]. Traffic analysis is the most powerful tool against anonymous communications. The traffic analysis attack includes two categories: profiling attack [8], [9] and timing attack [10],[11],[12]. Rather than obtaining the content of communication, adversaries who use traffic analysis attacks usually target on finding whether two entities communicate with each other; which web sites that target users access, and so on. They try to derive such information as much as possible from traffic metadata, such as message lengths, number of packets, packet arrival time intervals. In terms of profiling attack, adversaries have a list of possible web sites and the profiles respectively. The task is to find which ones the target user accesses. Timing attacks [10],[11] based on the fact that low-latency anonymous systems, such as onion routing, do not introduce any delays or significant alter on the timing patterns of an anonymous connection. The time interval of arrival packets of html text and http objects usually similar for the target user and the adversary if they access the same webpage, then it is easy for the adversary to figure out which web site the target user accessed from the list.

Researchers currently employ dummy packet padding technique to fight against traffic analysis. Usually, dummy packets are injected into the intended network traffic to change the patterns or fingerprints of the actual traffic to achieve anonymity [13],[14]. In order to disguise the timing information of connections, the packet rate of a connection should be the same all the time, then we need to add dummy packets when the real traffic is low or none, on the other hand, when the actual traffic rate is high, we have to drop some packets to align with the planed packet rate. This is called link padding [15],[16],[17]. Wright, Coull and Monrose recently proposed a traffic morphing method to protect the anonymity of communication [18]. This work is quite creative, however, it also uses dummy packet padding strategy and requires extra network resources, such as bandwidth.

The strategy of dummy packet padding results in two major problems in communication: extra delay and extra bandwidth demands [19]. These disadvantages are extraordinarily challenging in wireless, ad hoc, and sensor networks [14]. Because of the strict delay constraint from web viewers, high level of anonymization on web browsing may not always be achievable using dummy packet padding.

In this paper, we are motivated by these challenges and propose a novel strategy to resolve the challenges in anonymous web browsing. Our proposal comes from the following fact: users usually access a number of web pages at one web site according to their own habits and interests, and this has been confirmed by the applications of web caching and web page prefetching technologies [20],[21],[22]. Therefore, we can use prefetched data to replace dummy packets for padding. This novel strategy fundamentally solves the problems of extra delay and extra bandwidth cost of packet padding; moreover, our proposal makes it possible to achieve perfect anonymity of web browsing.

The contributions of this paper are summarized as follows.

- We proposed a novel strategy for packet padding using prefetched data as cover traffic for anonymous web browsing. The proposed strategy makes it possible to achieve perfect anonymity in web browsing under rigorous delay constraint;

- We transformed the anonymous communication problem into an optimization problem based on our model. As the result, we can figure out the tradeoffs between the anonymity level and the cost for applications. This makes it possible for users to find the best anonymity level once the delay constraint is known.
- The proposed schema can reduce bandwidth waste and network delay significantly.
- We established a mathematical model to describe and analyze the anonymization systems.

The rest of this paper is organized as follows. Related work and background are presented in Section 2. We setup the problem formally in Section 3. In Section 4 we present the details of system modeling and analysis, followed by performance evaluation in Section 5. Finally, we summarize and paper and point out the future work in Section 6.

2 Related Work

The HTTP protocol document [23] shows clearly that when a client submits an http request to an URL, the corresponding server will deliver the html text to the client, and the html text includes the references of the related objects, e.g. images, flashes. The objects will be downloaded to the client one after the other. Therefore, each web page has its own fingerprint. Some web server may encrypt the content of packets, however, an observer can clearly see the packet head, which includes critical information of the server, such as IP address.

A number of works have been done in terms of traffic analysis. Sun et al. tried to identify encrypted network traffic using the http object number and size. Their investigation shows it is sufficient to identify a significant fraction of the World Wide Web sites quite reliably [8]. Following this direction, Wright, Monrose and Masson further confirmed that websites can be identified with high probability even it is encrypted channel [9]. Hintz [13] suggested to add noise traffic (also named as *cover traffic* in some papers) to users which will change the fingerprints of the server, and transparent pictures are employed to add extra fake connections against fingerprint attacks.

Researchers also explored profiling attacks and proposed solutions. Coull et al. evaluated the strength of the anonymization methodology in terms of preventing the assembly of behavioral profiles, and concluded that anonymization offers less privacy to web browsing traffic than what we expected [24]. Liberatore and Levine used a profiling method to infer the sources of encrypted http connections [25]. They applied packet length and direction as attributes, and established a profile database for individual encrypted network traffic. Based on these information, they can infer the source of each individual encrypted network traffic. The adversary obtained the features of the target network traffic and then compared it with the individual record in the profile database, and inferred the possible source of the target traffic. The match technique based on a similarity metric (Jaccard's coefficient) and a supervised learning technique (the naïve Bayesian classifier). The extensive experiments showed that the proposed method can identify the source with the accuracy up to 90%.

Wright, Coull and Monrose recently proposed a traffic morphing method to protect the anonymity of communication [18]. They transformed the intended website (e.g. www.webmd.com) fingerprint to the fingerprint of another website (e.g. www.espn.com). The transformation methods that they took include packet padding, packet splitting. Optimal techniques are employed to find best the cover website (in terms of minimum cost for tranformation) from a list. They tested their algorithm against the data set offered in Liberatore and Levine's work [25], and found that the proposed method can improve the anonymity of web site accessing and reduce overhead at the same time. Venkitasubramaniam, He and Tong [14] notice the delay caused by adding dummy packets into communication channel, and proposed transmission schedules relay nodes to maximize network throughput given a desired level of anonymity. Similar to the other works, this work is also based on the platform of dummy packet padding.

Web caching and prefetching are effective and efficient solutions for web browsing when bandwidth is a constraint. Award, Khan and Thuraisingham tried to predict user web surfing path using a hybrid model, which combines Markov model and Supporting Vector Machine [22]. The fusion of the two models complements each other's weakness and worked together well. Montgomery et al. found that the information of users' web browsing path offers important information for a transaction, and a successful deal usually followed by a number of same path accessing [26]. Teng, Chang and Chen argued that there must be an elaborate coordinating between client side caching and prefetching, and formulated a normalized profit function to evaluate the profit from caching an object [20]. The proposed function integrated a number of factors, such as object size, fetching cost, reference rate, invalidation cost, and invalidation frequency. Their event-driven simulations showed that the proposed method performed well.

3 Problem Setting

3.1 Background of the Problem

We suppose that an adversary (Bob) focuses on finding which website that the monitored user (Alice) accesses from a list of possible websites. We suppose Bob has the knowledge of all the websites in the list, and he also captures all the network traffic of Alice's computer.

In general, every web page is different from the others, such as length of html text, number of web objects, timing of packet transportation, number of packets for each web object. We use web *fingerprint* to represent the uniqueness of a web site. The web service may be accessed in an encrypted way for users, such as using SSL. However, encryption brings limited changes on the fingerprint.

We suppose there are *n* possible web sites that Alice accesses, $w_1, w_2, ..., w_n$, and this is known to Bob. The a priori of w_i $(1 \le i \le n)$ is denoted as $p(w_i)$ $(1 \le i \le n)$. For each website w_i $(1 \le i \le n)$, we denote its fingerprint with $p_i^1, p_i^2, ..., p_i^k$. For example, for a given web site w_i, if we count the number of packets for every web object, such as html text, different images, et al., and save them as $x_1, x_2, ..., x_k$. We unify this vector and obtain the distribution as $p_i^1, p_i^2, ..., p_i^k$, where $p_i^j = x_j \cdot \left(\sum_{m=1}^{k} x_m \right)^{-1}$, $1 \le j \le k$.

Bob monitors Alice's local network, and obtains a number of observations

$$\tau = \{\tau_1, \tau_2, \tau_3, ...\}$$

Based on these observations and Bayesian Theorem, Bob can claim that Alice accesses website w_i with the following probability.

$$p(w_i \mid \tau) = \frac{p(w_i) \cdot p(\tau \mid w_i)}{p(\tau)}$$

On the other hand, the task for defenders is to decrease $p(w_i \mid \tau)$ to the minimum by anonymization operations, such as packet padding, link padding.

According to Shannon's perfect secrecy theory [5], an adversary cannot break the anonymity if the following equation holds.

$$p(w_i \mid \tau) = p(w_i)$$

Namely, the observation τ offers no information to the adversary. However, the cost for perfect anonymity is extremely expensive by injecting dummy packets according to the perfect secrecy theory.

3.2 Anonymity Measurement

Because of the strict delay constraint of user perception, we cannot always achieve perfect anonymity in web browsing cases. Suppose that user perception threshold is Δ in terms of seconds, we can bear this constraint in mind and try our best to improve the anonymity of web browsing. In order to measure the degree of anonymity, we make the following definition.

Definition 1. *Anonymity level.* Let S be the original network traffic for a given a session of network activity, S may be encrypted or covered by the sender, therefore, the adversary can only obtain an observation τ about S. We define a measure for the degree of anonymity as follows.

$$\alpha = \frac{H(S \mid \tau)}{H(S)}$$

Where $H(S)$ is the entropy [27] of the random variable *S*, which is defined as follows.

$$H(S) = -\sum_{s \in \chi} p(s) \log p(s)$$

where $p(s)$ is the distribution of S and χ is the probability space of S. $H(S \mid \tau)$ is the conditional entropy [27] of S given τ , which is defined as follows.

$$H(S \mid \tau) = \sum_{\tau_i \in \Gamma} p(\tau_i) H(S \mid \tau = \tau_i)$$

where $p(\tau_i)$ is the distribution of τ and Γ is the probability space of τ .

Because $H(S \mid \tau) \leq H(S)$ [27], we have $0 \leq \alpha \leq 1$. Following the definition, we obtain $\alpha = 1$ when $H(S \mid \tau) = H(S)$ holds, namely, S and τ are independent from each other, the adversary can infer no information about S from τ, we therefore achieve perfect anonymity. If $\tau = S$, then we have $\alpha = 0$, namely the adversary is completely sure about session S, and there is no anonymity to him at all.

Let *S* be the intended traffic. In order to achieve a given anonymity level α, we need a cover traffic *Y*. We use function $C(X)$ to represent the cost of network traffic *X*. The cost could be bandwidth, delay or number of packets, etc. Therefore, the cost for the intended traffic is *C(S)*, and the cost of cover traffic under anonymity level α is $C(Y \mid S, \alpha)$. We define a measure for the cost as follows.

Definition 2. *Anonymity cost coefficient.* For a given network traffic S, in order to achieve an anonymity level α, we inject a cover traffic Y. The anonymity cost coefficient is defined as

$$\beta = \frac{C(Y \mid S, \alpha)}{C(S)}$$

We expect β as small as possible in practice.

4 System Modelling and Analysis

In this section, we will model the anonymization system with our definitions and compare the proposed strategy against the dummy packet padding schemas based on our model.

4.1 System Modelling

In real applications, situations and accessing patterns could be very complex. In this paper, we target on showing the novelty of our proposal on anonymization mechanism and the potentials that the proposed strategy can achieve, therefore, we bound our research space with the following conditions in order to make our mechanisms to be understood smoothly.

- We only study the cases that the adversary focuses on one target user in this paper, and the adversary knows the possible list of web sites that the target user accesses.
- We focus on anonymity of network traffic sessions, and ignore link padding issue in this paper.
- We only discuss the cases on making anonymity by packet padding, and there is no packet splitting operations in this study. In case of packet splitting, the cost is the extra delay, rather than bandwidth.
- We only analyze the attack method of packet counting in this paper, however, our model and strategy is also effective for the other attack methods, such as attacks using packet arrival time intervals.
- We suppose Alice accesses one web site for one session, and she opens the default web page (index.html) of the website first, and then follows the hyperlinks to open other web pages at the website. Our model and strategy can deal with other accessing patterns, we just need more time and computing.

A typical anonymous communication with packet padding is shown in Figure. 1. As a user, Alice sends http request to a web server w_i via an anonymous channel. The web server returns the intended traffic $\mathrm{P} = \{\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_k\}$, where P_i represents the

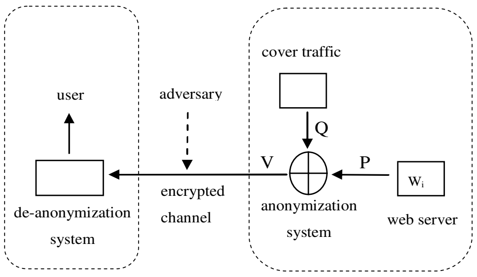


Fig. 1. A packet padding system for anonymous communication

number of packets of web object i. Let $\|P\|=\sum_{i=1}^{k} P_i$ denote the total number of packets of the intended traffic P, and we extract the fingerprint of this session as $p=\{p_1,p_2,...,p_k\}$, where $p_i = P_i / \|P\|, 1 \le i \le k$, and $\sum_{i=1}^{k} p_i = 1$. In order to make it anonymous to adversaries, we create the cover traffic $Q=\{Q_1,Q_2,\ldots,Q_k\}$ at the server side, where Q_i denotes the number of packets that is assigned to cover P_i, and let $\|Q\|=\sum_{i=1}^{k} Q_i$. Similar to the intended traffic P, the fingerprint of Q is $q=\{q_1,q_2,...,q_k\}$. We use $\oplus$ to represent the anonymization operations, and $V = P \oplus Q$ is the output of the anonymization operation on P and Q. The fingerprint of V is $v=\{v_1,v_2,...,v_k\}$, $\sum_{i=1}^{k} v_i = 1$, and $\|V\|$ is the total packet number of *V*. The adversary's observation τ is the mix of *V* and other network background traffic. Once *V* arrives the user side, there is a de-anonymization operation to recover *P* from *V*.

In previous works, dummy packets are employed to work as the cover traffic Q; In this paper, we propose to use prefetching web pages as the cover traffic, rather than dummy packets. Let W be the set of all possible content of web site w_i, and S be a traffic sessions of accessing w_i, Δ be the threshold of maximum cost that users can tolerate. Then the anonymization problem can be transformed to an optimization issue as follows.

$$\text{Maximize } \alpha$$
$$s.\ t.$$
$$C(S \oplus Y) \le \Delta$$
$$S \in W$$
$$Y \in W$$
$$\|Y\| \le C_b$$

Where C_b is the storage capacity of the client, and we suppose $C_b = \infty$ in this paper.

In the previous dummy packet padding solutions, $Y \notin W$ and $S \cap Y = \phi$.

4.2 System Analysis

Following the definitions and expressions in the previous section, in order to achieve perfect anonymity, the following equations must hold.

$$v_1 = v_2 = \ldots = v_k$$

Suppose $p_m = \max\{p_1, p_2, \ldots, p_k\}$, then the minimum cover traffic to achieve perfect anonymity is given as follows.

$$Q = \{(p_m - p_1)\cdot \| P \|, (p_m - p_2)\cdot \| P \|, \ldots, (p_m - p_k)\cdot \| P \|\}$$

We have fingerprint of Q as

$$q = \{q_1, q_2, \ldots, q_k\} = \{(p_m - p_1), (p_m - p_2), \ldots, (p_m - p_k)\}$$

Then the anonymity cost coefficient in terms of number of packet is

$$\beta = \frac{\sum_{i=1}^{k} Q_i}{\sum_{i=1}^{k} P_i} = \frac{\| Q \|}{\| P \|}$$

In general, given an intended traffic P and an anonymity level α ($0 \le \alpha \le 1$), the cost of the cover traffic could be expressed as

$$C(Q \mid P, \alpha)$$

For dummy packet padding strategies, the anonymity cost coefficient β_d could be denoted as follows

$$\beta_d = \frac{C(Q \mid P, \alpha)}{C(P)}$$

In our proposed strategy, the cover traffic is part of P in long term viewpoint. The extra cost is caused by the data that we prefetched but not used. We suppose the prefetch accuracy is $\eta (0 \le \eta \le 1)$, then the extra cost from the cover traffic is as follows.

$$Q^{'} = P \cdot (1 - \eta)$$

Then β_p, the anonymity cost coefficient of our strategy is

$$\beta_p = \frac{C(Q^{'} \mid P, \alpha)}{C(P)} = \frac{C(P \cdot (1-\eta) \mid P, \alpha)}{C(P)}$$

If $\eta = 0$, then $\beta_p = \beta_d$, namely the anonymity cost coefficient of our strategy is the same as the others; on the other hand, if $\eta = 1$, then $\| Q \| = 0$, and we have $\beta_p = 0$, in other words, there is no waste of resources at all. In general, for $0 \le \eta \le 1$, if the cost function $C(\cdot)$ is a linear function, then we can simplify β_p as follows.

$$\beta_p = (1-\eta)\cdot\alpha$$

5 Performance Evaluation

In order to show the potentials of the proposed strategy, we extract the relationship between anonymity level and cost against the real data set used in [25], the data set was also used in [18] in order to indicate the cost efficiency of their morphing strategy.

The data set includes the tcpdump files of 2000 web sites from February 10, 2006 to April 28, 2006, which are sorted by popularity from site0 to site1999 from the most to the least. There are a few different sessions everyday for each web site respectively, for each session users may access different web pages in the same day. We take site0, site500, and site1999 as three representatives from the data set. For a given network session, we use the number of packets of web objects as fingerprints (For the same website, user may access different web pages, therefore, the fingerprint for each session is only a part of the entire fingerprint). The fingerprints of the aforementioned 6 sessions are shown in Fig. 2.

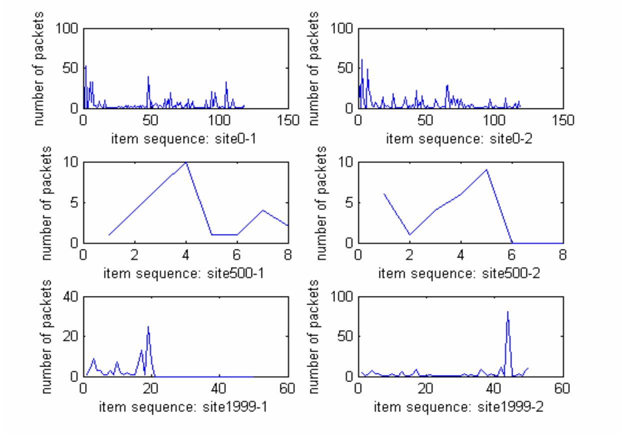


Fig. 2. Fingerprints of network sessions

In the case of perfect anonymity, the number of packets for every web object is the same since we added cover traffic to the intended traffic. In order to obtain different level of anonymity, we start the perfect anonymity operation partially, and pad the related cover traffic to intended traffic one web object after the other, and start from the first web object to the last. This is a simple way for anonymization with $\alpha \prec 1$. The result of the anonymity level against the progress of anonymization is shown in Figure 3. The results show that α approaches 1 when we process more and more web objects. This indicates that higher anonymity level demands more cover traffic, this matches our analysis in the previous sections.

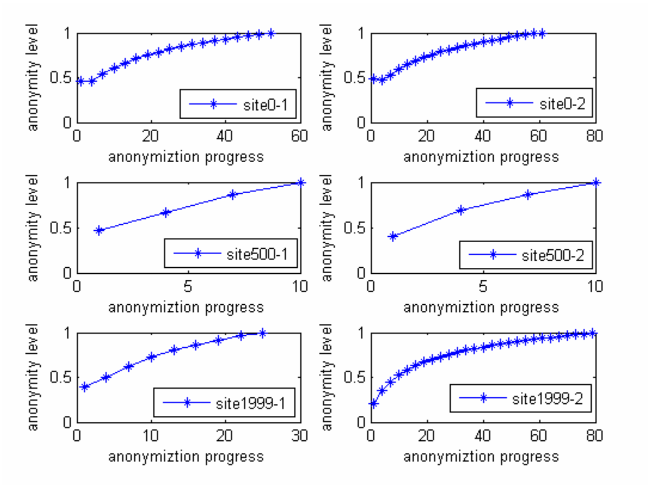


Fig. 3. Anonymity level against anonymization progress on web objects

In order to directly show the relationship between anonymity level and anonymity cost coefficient, we extract this information from the 6 network sessions carefully, and the results are presented in Figure 4.

From Figure 4 we find that the cost for perfect anonymity is extremely expensive. For example, the volume of cover traffic for site1999-2 is more than 120 times of that of the intended traffic; it is more than 10 times for simple web site, such as site500, which has small number of packets as indicated in Figure 2. These preliminary experiments showed the huge potential of resource saving of our proposed strategy.

Moreover, the experiments on site0 in Figure 4 show an anomaly at the beginning: more cover traffic results lower anonymity level. We are very interested to investigate this as a future work.

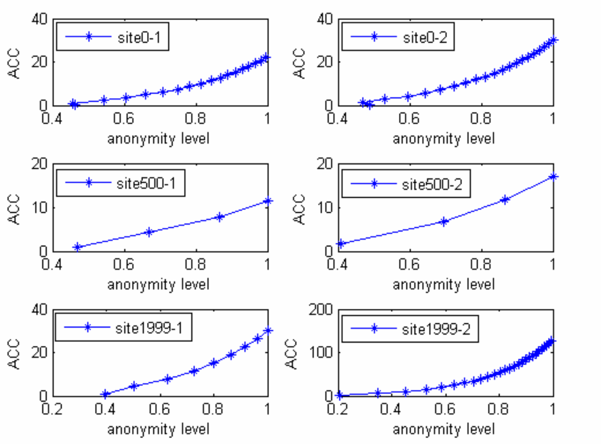


Fig. 4. Anonymity cost coefficient (ACC) against anonymity level

6 Summary and Future Work

We noticed that the cost for dummy packet padding is very expensive for anonymous web browsing. Therefore, a novel strategy is proposed to reduce the cost significantly for web browsing – taking web prefetched data as cover traffic rather than dummy packets. Moreover, it is hard to achieve perfect anonymous in some web browsing cases, we therefore defined a measure, anonymity level, to be a metric to measure user requirements on anonymity. We modeled the anonymous system in a theoretical way, and described the relationship between anonymity level and the cost in theory. Furthermore, we transformed the anonymity problem into an optimization problem, and this transform brings numerous possible solutions from the optimization field. We believe that this model offers a solid platform for the further researches. Some preliminary experiments have been done to show the great potential of the proposed strategy.

As the future work, we expect to extend the model to multiple user scenario, and we will also include link padding with the multiple user cases. Web prefetching algorithms will be integrated into the model, and extensive tests against the data set are expected in the near future. More effort is desired on the process of de-anonymization at the client side efficiently and accurately.

References

1. Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM 4 (1981)
2. Reiter, M., Rubin, A.: Crowds: Anonymity for web transactions. ACM Transaction on Information and System Security 1 (1998)
3. Reed, M., Syverson, P., Goldschlag, D.: Anonymous connections and onion routing. IEEE Journal on Selected Areas in Communications 16, 482–494 (1998)
4. `http://www.torproject.org`
5. Shannon, C.E.: Communication Theory of Secrecy Systems. Journal of Bell System Technology 28, 656–715 (1949)
6. Yu, W., Fu, X., Graham, S., Xuan, D., Zhao, W.: DSSS-Based Flow Marking Technique for Invisible Traceback. In: Proceedings of IEEE Symposium on Security and Privacy (S&P), Oakland, California, USA (2007)
7. Jia, W., Tso, F.P., Ling, Z., Fu, X., Xuan, D., Yu, W.: Blind Detection of Spread Spectrum Flow Watermarks. In: IEEE INFOCOM 2009 (2009)
8. Sun, Q., Simon, D.R., Wang, Y.-M., Russell, W., Padmanabhan, V.N., Qiu, L.: Statistical Identification of Encrypted Web Browsing Traffic. In: The 2002 IEEE Symposium on Security and Privacy. IEEE, Berkeley (2002)
9. Wright, C., Monrose, F., Masson, G.: On Inferring Application Protocol Behaviors in Encrypted Network Traffic. Journal of Machine Learning Research 2006, 2745–2769 (2006)
10. Shmatikov, V., Wang, M.: Timing analysis in low-latency mix networks: Attacks and defenses. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 18–33. Springer, Heidelberg (2006)
11. Murdoch, S., Zielinski, P.: Smapled traffic analysis by internet-exchange-level adversaries. In: Borisov, N., Golle, P. (eds.) PET 2007. LNCS, vol. 4776, pp. 167–183. Springer, Heidelberg (2007)
12. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: Proceedings of ACM Conference on Computer and Communication Security, Athens, Greece, pp. 25–32 (2000)

13. Hintz, A.: Fingerprinting websites using traffic analysis. In: Dingledine, R., Syverson, P.F. (eds.) PET 2002. LNCS, vol. 2482, pp. 171–178. Springer, Heidelberg (2003)
14. Venkitasubramaniam, P., He, T., Tong, L.: Anonymous Networking Amidst Eavesdroppers. IEEE Transactions on Information Theory 54, 2770–2784 (2008)
15. Venkitasubramaniam, P., He, T., Tong, L.: Relay secrecy in wireless networks with eavesdroppers. In: Proceedings of Allerton Conference on Communication, Control and Computing (2006)
16. Venkitasubramaniam, P., He, T., Tong, L.: Anonymous networking for minimum latency in multihop networks. In: IEEE Symposium on Security and Privacy (2008)
17. Wang, W., Motani, M., Srinivasan, V.: Dependent link padding algorithms for low latency anonymity systems. In: Proceedings of ACM Conference on Computer and Communication Security, pp. 323–332 (2008)
18. Wright, C., Coull, S., Monrose, F.: Traffic Morphing: An efficient defense against statistical traffic analysis. In: The 16th Annual Network and Distributed Security Symposium (2009)
19. Edman, M., Yener, B.: On Anonymity in an Electronic Society: A Survey of Anonymous Communication Systems. ACM Computer Survey (to appear)
20. Teng, W.-G., Chang, C.-Y., Chen, M.-S.: Integrating web caching and web prefetching in client-side proxies. IEEE Transactions on Parallel and Distributed Systems 16, 444–454 (2005)
21. Zeng, Z., Veeravalli, B.: H^k/T: A novel server-side web caching strategy for multimedia applications. In: Proceedings of IEEE International Conference on Communications, pp. 1782–1786 (2008)
22. Award, M., Khan, L., Thuraisingham, B.: Predecting WWW surfing using multiple evidence conbination. The VLDB Journal 17, 401–417 (2008)
23. RFC2616, http://www.w3.org/Protocols/rfc2616/rfc2616.html
24. Coull, S.E., Collins, M.P., Wright, C.V., Monrose, F., Reiter, M.K.: On Web Browsing Privacy in Anonymized NetFlows. In: The 16th USENIX Security Symposium, Boston, USA, pp. 339–352 (2007)
25. Liberatore, M., Levine, B.N.: Inferring the Source of Encrypted HTTP Connections. In: ACM conference on Computer and Communications Security (CCS), pp. 255–263 (2006)
26. Montgomery, A.L., Li, S., Srinivasan, K., Liechty, J.C.: Modeling Online Browsing and Path Analysis Using Clickstream Data. Marketing Science 23, 579–595 (2004)
27. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley Interscience, Hoboken (2007)

InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services

Rajkumar Buyya[1,2], Rajiv Ranjan[3], and Rodrigo N. Calheiros[1]

[1] **Clou**d Computing and **D**istributed **S**ystems (CLOUDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
[2] Manjrasoft Pty Ltd, Australia
[3] School of Computer Science and Engineering
The University of New South Wales, Sydney, Australia

Abstract. Cloud computing providers have setup several data centers at different geographical locations over the Internet in order to optimally serve needs of their customers around the world. However, existing systems do not support mechanisms and policies for dynamically coordinating load distribution among different Cloud-based data centers in order to determine optimal location for hosting application services to achieve reasonable QoS levels. Further, the Cloud computing providers are unable to predict geographic distribution of users consuming their services, hence the load coordination must happen automatically, and distribution of services must change in response to changes in the load. To counter this problem, we advocate creation of federated Cloud computing environment (InterCloud) that facilitates just-in-time, opportunistic, and scalable provisioning of application services, consistently achieving QoS targets under variable workload, resource and network conditions. The overall goal is to create a computing environment that supports dynamic expansion or contraction of capabilities (VMs, services, storage, and database) for handling sudden variations in service demands.

This paper presents vision, challenges, and architectural elements of InterCloud for utility-oriented federation of Cloud computing environments. The proposed InterCloud environment supports scaling of applications across multiple vendor clouds. We have validated our approach by conducting a set of rigorous performance evaluation study using the CloudSim toolkit. The results demonstrate that federated Cloud computing model has immense potential as it offers significant performance gains as regards to response time and cost saving under dynamic workload scenarios.

1 Introduction

In 1969, Leonard Kleinrock [1], one of the chief scientists of the original Advanced Research Projects Agency Network (ARPANET) project which seeded the Internet, said: *"As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of 'computer utilities'* *which, like present electric and telephone utilities, will service individual homes and offices across the country."* This vision of computing utilities based on a service provisioning model anticipated the massive transformation of the entire computing industry in the 21st century whereby computing services will be readily available on demand, like other utility services available in today's society. Similarly, computing service users (consumers) need to pay providers only when they access computing services. In addition, consumers no longer need to invest heavily or encounter difficulties in building and maintaining complex IT infrastructure.

In such a model, users access services based on their requirements without regard to where the services are hosted. This model has been referred to as *utility computing*, or recently as *Cloud computing* [3]. The latter term denotes the infrastructure as a "Cloud" from which businesses and users are able to access application services from anywhere in the world on demand. Hence, Cloud computing can be classified as a new paradigm for the dynamic provisioning of computing services, typically supported by state-of-the-art data centers containing ensembles of networked Virtual Machines.

Cloud computing delivers infrastructure, platform, and software (application) as services, which are made available as subscription-based services in a pay-as-you-go model to consumers. These services in industry are respectively referred to as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). A Berkeley Report in Feb 2009 states "Cloud computing, the long-held dream of computing as a utility, has the potential to transform a large part of the IT industry, making software even more attractive as a service" [2].

Clouds aim to power the next generation data centers by architecting them as a network of virtual services (hardware, database, user-interface, application logic) so that users are able to access and deploy applications from anywhere in the world on demand at competitive costs depending on users QoS (Quality of Service) requirements [3]. Developers with innovative ideas for new Internet services no longer require large capital outlays in hardware to deploy their service or human expense to operate it [2]. It offers significant benefit to IT companies by freeing them from the low level task of setting up basic hardware (servers) and software infrastructures and thus enabling more focus on innovation and creating business value for their services.

The business potential of Cloud computing is recognised by several market research firms including IDC, which reports that worldwide spending on Cloud services will grow from $16 billion by 2008 to $42 billion in 2012. Furthermore, many applications making use of these utility-oriented computing systems such as clouds emerge simply as catalysts or market makers that bring buyers and sellers together. This creates several trillion dollars of worth to the utility/pervasive computing industry as noted by Sun Microsystems co-founder Bill Joy [4]. He also indicated "It would take time until these markets to mature to generate this kind of value. Predicting now which companies will capture the value is impossible. Many of them have not even been created yet."

1.1 Application Scaling and Cloud Infrastructure: Challenges and Requirements

Providers such as Amazon [15], Google [16], Salesforce [21], IBM, Microsoft [17], and Sun Microsystems have begun to establish new data centers for hosting Cloud computing application services such as social networking and gaming portals, business applications (e.g., SalesForce.com), media content delivery, and scientific workflows. Actual usage patterns of many real-world application services vary with time, most of the time in unpredictable ways. To illustrate this, let us consider an "elastic" application in the business/social networking domain that needs to scale up and down over the course of its deployment.

Social Networking Web Applications
Social networks such as Facebook and MySpace are popular Web 2.0 based applications. They serve dynamic content to millions of users, whose access and interaction patterns are hard to predict. In addition, their features are very dynamic in the sense that new plug-ins can be created by independent developers, added to the main system and used by other users. In several situations load spikes can take place, for instance, whenever new system features become popular or a new plug-in application is deployed. As these social networks are organized in communities of highly interacting users distributed all over the world, load spikes can take place at different locations at any time. In order to handle unpredictable seasonal and geographical changes in system workload, an automatic scaling scheme is paramount to keep QoS and resource consumption at suitable levels.

Social networking websites are built using multi-tiered web technologies, which consist of application servers such as IBM WebSphere and persistency layers such as the MySQL relational database. Usually, each component runs in a separate virtual machine, which can be hosted in data centers that are owned by different cloud computing providers. Additionally, each plug-in developer has the freedom to choose which Cloud computing provider offers the services that are more suitable to run his/her plug-in. As a consequence, a typical social networking web application is formed by hundreds of different services, which may be hosted by dozens of Cloud data centers around the world. Whenever there is a variation in temporal and spatial locality of workload, each application component must dynamically scale to offer good quality of experience to users.

1.2 Federated Cloud Infrastructures for Elastic Applications

In order to support a large number of application service consumers from around the world, Cloud infrastructure providers (i.e., IaaS providers) have established data centers in multiple geographical locations to provide redundancy and ensure reliability in case of site failures. For example, Amazon has data centers in the US (e.g., one in the East Coast and another in the West Coast) and Europe. However, currently they (1) expect their Cloud customers (i.e., SaaS providers) to express a preference about the location where they want their application services to be hosted and (2) don't provide seamless/automatic mechanisms for scaling their hosted services across multiple, geographically distributed data centers. This approach has many shortcomings, which include (1) it is difficult for Cloud customers to determine in advance the best location for hosting their services as they may not know origin of consumers of their services and (2) Cloud SaaS providers may not be able to meet QoS expectations of their service consumers originating from multiple geographical locations. This necessitates building mechanisms for seamless federation of data centers of a Cloud provider or providers supporting dynamic scaling of applications across multiple domains in order to meet QoS targets of Cloud customers.

In addition, no single Cloud infrastructure provider will be able to establish their data centers at all possible locations throughout the world. As a result Cloud application service (SaaS) providers will have difficulty in meeting QoS expectations for all their consumers. Hence, they would like to make use of services of multiple Cloud infrastructure service providers who can provide better support for their specific consumer needs. This kind of requirements often arises in enterprises with global operations and applications such as Internet service, media hosting, and Web 2.0 applications. This necessitates building mechanisms for federation of Cloud infrastructure service providers for seamless provisioning of services across different Cloud providers. There are many challenges involved in creating such Cloud interconnections through federation.

To meet these requirements, next generation Cloud service providers **should be able** to: (i) dynamically expand or resize their provisioning capability based on sudden spikes in workload demands by leasing available computational and storage capabilities from other Cloud service providers; (ii) operate as parts of a market driven resource leasing federation, where application service providers such as Salesforce.com host their services based on negotiated Service Level Agreement (SLA) contracts driven by competitive market prices; and (iii) deliver on demand, reliable, cost-effective, and QoS aware services based on virtualization technologies while ensuring high QoS standards and minimizing service costs. They need to be able to utilize market-based utility models as the basis for provisioning of virtualized software services and federated hardware infrastructure among users with heterogeneous applications and QoS targets.

1.3 Research Issues

The diversity and flexibility of the functionalities (dynamically shrinking and growing computing systems) envisioned by federated Cloud computing model, combined with the magnitudes and uncertainties of its components (workload, compute servers, services, workload), pose difficult problems in effective provisioning and delivery of application services. Provisioning means "high-level management of computing, network, and storage resources that allow them to effectively provide and deliver services to customers". In particular, finding efficient solutions for following challenges is critical to exploiting the potential of federated Cloud infrastructures:

Application Service Behavior Prediction: It is critical that the system is able to predict the demands and behaviors of the hosted services, so that it intelligently undertake decisions related to dynamic scaling or de-scaling of services over federated Cloud infrastructures. Concrete prediction or forecasting models must be built before the behavior of a service, in terms of computing, storage, and network bandwidth requirements, can be predicted accurately. *The real challenge in devising such models is accurately learning and fitting statistical functions [31] to the observed distributions of service behaviors such as request arrival pattern, service time distributions, I/O system behaviors, and network usage.* This challenge is further aggravated by the existence of statistical correlation (such as stationary, short- and long-range dependence, and pseudo-periodicity) between different behaviors of a service.

Flexible Mapping of Services to Resources: With increased operating costs and energy requirements of composite systems, it becomes critical to maximize their efficiency, cost-effectiveness, and utilization [30] . The process of mapping services to resources is a complex undertaking, as it requires the system to *compute the best software and hardware configuration (system size and mix of resources) to ensure that QoS targets of services are achieved, while maximizing system efficiency and utilization. This process is further complicated by the uncertain behavior of resources and services.* Consequently, there is an immediate need to devise performance modeling and market-based service mapping techniques that ensure efficient system utilization without having an unacceptable impact on QoS targets.

Economic Models Driven Optimization Techniques: The market-driven decision making problem [6] is a combinatorial optimization problem that searches the optimal combinations of services and their deployment plans. Unlike many existing multi-objective optimization solutions, the optimization models that ultimately aim to optimize both resource-centric (utilization, availability, reliability, incentive) and user-centric (response time, budget spent, fairness) QoS targets need to be developed.

Integration and Interoperability: For many SMEs, there is a large amount of IT assets in house, in the form of line of business applications that are unlikely to ever be migrated to the cloud. Further, there is huge amount of sensitive data in an enterprise, which is unlikely to migrate to the cloud due to privacy and security issues. As a result, there is a need to look into issues related to integration and interoperability between the software on premises and the services in the cloud. In particular [28]: (i) Identity management: authentication and authorization of service users; provisioning user access; federated security model; (ii) Data Management: not all data will be stored in a relational database in the cloud, eventual consistency (BASE) is taking over from the traditional ACID transaction guarantees, in order to ensure sharable data structures that achieve high scalability. (iii) Business process orchestration: how does integration at a business process level happen across the software on premises and service in the Cloud boundary? Where do we store business rules that govern the business process orchestration?

Scalable Monitoring of System Components: Although the components that contribute to a federated system may be distributed, existing techniques usually employ centralized approaches to overall system monitoring and management. We claim that centralized approaches are not an appropriate solution for this purpose, due to concerns of scalability, performance, and reliability arising from the management of multiple service queues and the expected large volume of service requests. *Monitoring of system components is required for effecting on-line control through a collection of system performance characteristics.* Therefore, we advocate architecting service monitoring and management services based on decentralized messaging and indexing models [27].

1.4 Overall Vision

To meet aforementioned requirements of auto-scaling Cloud applications, future efforts should focus on design, development, and implementation of software systems and policies for federation of Clouds across network and administrative boundaries. The key elements for enabling federation of Clouds and auto-scaling application services are: Cloud Coordinators, Brokers, and an Exchange. The resource provisioning within these federated clouds will be driven by market-oriented principles for efficient resource allocation depending on user QoS targets and workload demand patterns. To reduce power consumption cost and improve service localization while complying with the Service Level Agreement (SLA) contracts, new on-line algorithms for energy-aware placement and live migration of virtual machines between Clouds would need to be developed. The approach for realisation of this research vision consists of investigation, design, and development of the following:

- Architectural framework and principles for the development of utility-oriented clouds and their federation
- A Cloud Coordinator for exporting Cloud services and their management driven by market-based trading and negotiation protocols for optimal QoS delivery at minimal cost and energy.
- A Cloud Broker responsible for mediating between service consumers and Cloud coordinators.
- A Cloud Exchange acts as a market maker enabling capability sharing across multiple Cloud domains through its match making services.
- A software platform implementing Cloud Coordinator, Broker, and Exchange for federation.

The rest of this paper is organized as follows: First, a concise survey on the existing state-of-the-art in Cloud provisioning is presented. Next, the comprehensive description related to overall system architecture and its elements that forms the basis for constructing federated Cloud infrastructures is given. This is followed by some initial experiments and results, which quantifies the performance gains delivered by the proposed approach. Finally, the paper ends with brief conclusive remarks and discussion on perspective future research directions.

2 State-of-the-Art in Cloud Provisioning

The key Cloud platforms in Cloud computing domain including Amazon Web Services [15], Microsoft Azure [17], Google AppEngine [16], Manjrasoft Aneka [32], Eucalyptus [22], and GoGrid [23] offer a variety of pre-packaged services for monitoring, managing and provisioning resources and application services. However, the techniques implemented in each of these Cloud platforms vary (refer to Table 1).

The three Amazon Web Services (AWS), Elastic Load Balancer [25], Auto Scaling and CloudWatch [24] together expose functionalities which are required for undertaking provisioning of application services on Amazon EC2. Elastic Load Balancer service automatically provisions incoming application workload across available Amazon EC2 instances. Auto-Scaling service can be used for dynamically scaling-in or scaling-out the number of Amazon EC2 instances for handling changes in service demand patterns. And finally the CloudWatch service can be integrated with above services for strategic decision making based on real-time aggregated resource and service performance information.

Table 1. Summary of provisioning capabilities exposed by public Cloud platforms

Cloud Platforms	Load Balancing	Provisioning	Auto Scaling
Amazon Elastic Compute Cloud	√	√	√
Eucalyptus	√	√	×
Microsoft Windows Azure	√	√ (fixed templates so far)	√ (Manual)
Google App Engine	√	√	√
Manjrasoft Aneka	√	√	√
GoGrid Cloud Hosting	√	√	√ (Programmatic way only)

Manjrasoft Aneka is a platform for building and deploying distributed applications on Clouds. It provides a rich set of APIs for transparently exploiting distributed resources and expressing the business logic of applications by using the preferred programming abstractions. Aneka is also a market-oriented Cloud platform since it allows users to build and schedule applications, provision resources and monitor results using pricing, accounting, and QoS/SLA services in private and/or public (leased) Cloud environments. Aneka also allows users to build different run-time environments such as enterprise/private Cloud by harness computing resources in network or enterprise data centers, public Clouds such as Amazon EC2, and hybrid clouds by combining enterprise private Clouds managed by Aneka with resources from Amazon EC2 or other enterprise Clouds build and managed using technologies such as XenServer.

Eucalyptus [22] is an open source Cloud computing platform. It is composed of three controllers. Among the controllers, the Cluster Controller is a key component to application service provisioning and load balancing. Each Cluster Controller is hosted on the head node of a cluster to interconnect outer public networks and inner private networks together. By monitoring the state information of instances in the pool of server controllers, the Cluster Controller can select the available service/server for provisioning incoming requests. However, as compared to AWS, Eucalyptus still lacks some of the critical functionalities, such as auto scaling for built-in provisioner.

Fundamentally, Windows Azure Fabric [17] has a weave-like structure, which is composed of node (servers and load balancers), and edges (power, Ethernet and serial communications). The Fabric Controller manages a service node through a built-in service, named Azure Fabric Controller Agent, which runs in the background, tracking the state of the server, and reporting these metrics to the Controller. If a fault state is reported, the Controller can manage a reboot of the server or a migration of services from the current server to other healthy servers. Moreover, the Controller also supports service provisioning by matching the services against the VMs that meet required demands.

GoGrid Cloud Hosting offers developers the F5 Load Balancers [23] for distributing application service traffic across servers, as long as IPs and specific ports of these servers are attached. The load balancer allows Round Robin algorithm and Least Connect algorithm for routing application service requests. Also, the load balancer is able to sense a crash of the server, redirecting further requests to other available servers. But currently, GoGrid Cloud Hosting only gives developers programmatic APIs to implement their custom auto-scaling service.

Unlike other Cloud platforms, Google App Engine offers developers a scalable platform in which applications can run, rather than providing access directly to a customized virtual machine. Therefore, access to the underlying operating system is restricted in App Engine. And load-balancing strategies, service provisioning and auto scaling are all automatically managed by the system behind the scenes. However, at this time Google App Engine can only support provisioning of web hosting type of applications.

However, no single Cloud infrastructure providers have their data centers at all possible locations throughout the world. As a result Cloud application service (SaaS) providers will have difficulty in meeting QoS expectations for all their users. Hence, they would prefer to logically construct federated Cloud infrastructures (mixing multiple public and private clouds) to provide better support for their specific user needs. This kind of requirements often arises in enterprises with global operations and applications such as Internet service, media hosting, and Web 2.0 applications. This necessitates building technologies and algorithms for seamless federation of Cloud infrastructure service providers for autonomic provisioning of services across different Cloud providers.

3 System Architecture and Elements of InterCloud

Figure 1 shows the high level components of the service-oriented architectural framework consisting of client's brokering and coordinator services that support utility-driven federation of clouds: application scheduling, resource allocation and

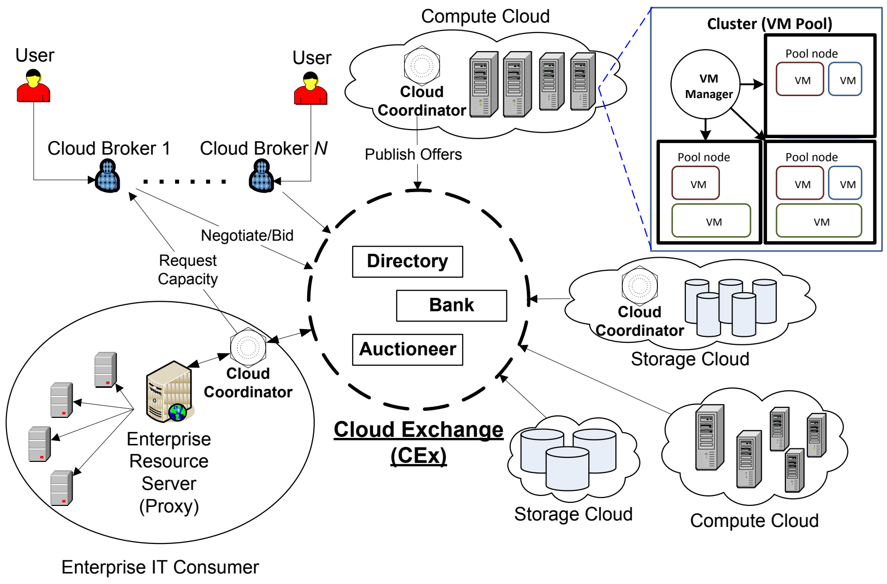


Fig. 1. Federated network of clouds mediated by a Cloud exchange

migration of workloads. The architecture cohesively couples the administratively and topologically distributed storage and computes capabilities of Clouds as parts of single resource leasing abstraction. The system will ease the cross-domain capabilities integration for on demand, flexible, energy-efficient, and reliable access to the infrastructure based on emerging virtualization technologies [8][9].

The Cloud Exchange (CEx) acts as a market maker for bringing together service producers and consumers. It aggregates the infrastructure demands from the application brokers and evaluates them against the available supply currently published by the Cloud Coordinators. It supports trading of Cloud services based on competitive economic models [6] such as commodity markets and auctions. CEx allows the participants (Cloud Coordinators and Cloud Brokers) to locate providers and consumers with fitting offers. Such markets enable services to be commoditized and thus, would pave the way for creation of dynamic market infrastructure for trading based on SLAs. An SLA specifies the details of the service to be provided in terms of metrics agreed upon by all parties, and incentives and penalties for meeting and violating the expectations, respectively. The availability of a banking system within the market ensures that financial transactions pertaining to SLAs between participants are carried out in a secure and dependable environment. Every client in the federated platform needs to instantiate a Cloud Brokering service that can dynamically establish service contracts with Cloud Coordinators via the trading functions exposed by the Cloud Exchange.

3.1 Cloud Coordinator (CC)

The Cloud Coordinator service is responsible for the management of domain specific enterprise Clouds and their membership to the overall federation driven by market-based

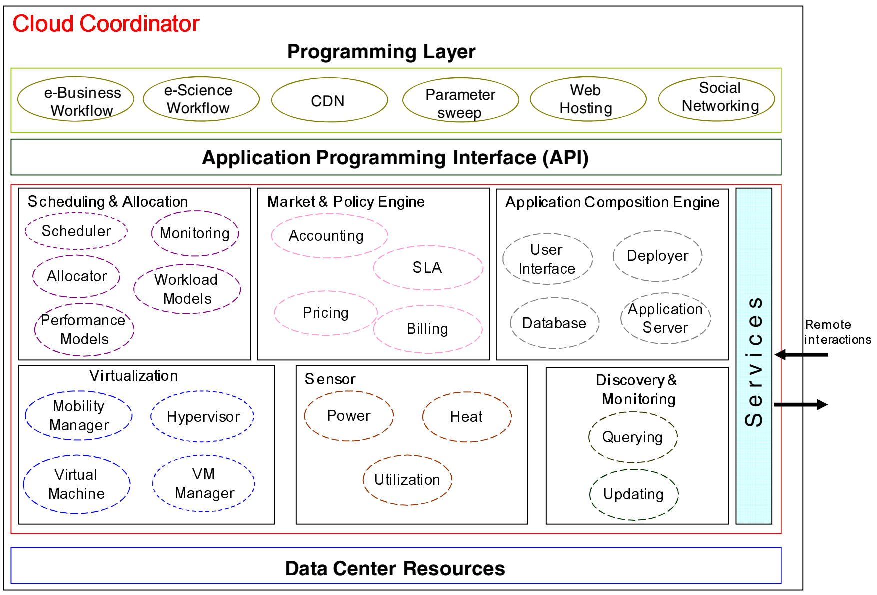


Fig. 2. Cloud Coordinator software architecture

trading and negotiation protocols. It provides a programming, management, and deployment environment for applications in a federation of Clouds. Figure 2 shows a detailed depiction of resource management components in the Cloud Coordinator service.

The Cloud Coordinator exports the services of a cloud to the federation by implementing basic functionalities for resource management such as scheduling, allocation, (workload and performance) models, market enabling, virtualization, dynamic sensing/monitoring, discovery, and application composition as discussed below:

Scheduling and Allocation: This component allocates virtual machines to the Cloud nodes based on user's QoS targets and the Clouds energy management goals. On receiving a user application, the scheduler does the following: (i) consults the Application Composition Engine about availability of software and hardware infrastructure services that are required to satisfy the request locally, (ii) asks the Sensor component to submit feedback on the local Cloud nodes' energy consumption and utilization status; and (iii) enquires the Market and Policy Engine about accountability of the submitted request. A request is termed as accountable if the concerning user has available credits in the Cloud bank and based on the specified QoS constraints the establishment of SLA is feasible. In case all three components reply favorably, the application is hosted locally and is periodically monitored until it finishes execution.

Data center resources may deliver different levels of performance to their clients; hence, QoS-aware resource selection plays an important role in Cloud computing. Additionally, Cloud applications can present varying workloads. It is therefore essential to carry out a study of Cloud services and their workloads in order to identify common behaviors, patterns, and explore load forecasting approaches that can potentially lead to more efficient scheduling and allocation. In this context, there is need to analyse sample applications and correlations between workloads, and attempt to build performance models that can help explore trade-offs between QoS targets.

Market and Policy Engine: The SLA module stores the service terms and conditions that are being supported by the Cloud to each respective Cloud Broker on a per user basis. Based on these terms and conditions, the Pricing module can determine how service requests are charged based on the available supply and required demand of computing resources within the Cloud. The Accounting module stores the actual usage information of resources by requests so that the total usage cost of each user can be calculated. The Billing module then charges the usage costs to users accordingly.

Cloud customers can normally associate two or more conflicting QoS targets with their application services. In such cases, it is necessary to trade off one or more QoS targets to find a superior solution. Due to such diverse QoS targets and varying optimization objectives, we end up with a Multi-dimensional Optimization Problem (MOP). For solving the MOP, one can explore multiple heterogeneous optimization algorithms, such as dynamic programming, hill climbing, parallel swarm optimization, and multi-objective genetic algorithm.

Application Composition Engine: This component of the Cloud Coordinator encompasses a set of features intended to help application developers create and deploy [18] applications, including the ability for on demand interaction with a database backend such as SQL Data services provided by Microsoft Azure, an application server such as

Internet Information Server (IIS) enabled with secure ASP.Net scripting engine to host web applications, and a SOAP driven Web services API for programmatic access along with combination and integration with other applications and data.

Virtualization: VMs support flexible and utility driven configurations that control the share of processing power they can consume based on the time criticality of the underlying application. However, the current approaches to VM-based Cloud computing are limited to rather inflexible configurations within a Cloud. This limitation can be solved by developing mechanisms for transparent migration of VMs across service boundaries with the aim of minimizing cost of service delivery (e.g., by migrating to a Cloud located in a region where the energy cost is low) and while still meeting the SLAs. The Mobility Manager is responsible for dynamic migration of VMs based on the real-time feedback given by the Sensor service. Currently, hypervisors such as VMware [8] and Xen [9] have a limitation that VMs can only be migrated between hypervisors that are within the same subnet and share common storage. Clearly, this is a serious bottleneck to achieve adaptive migration of VMs in federated Cloud environments. This limitation has to be addressed in order to support utility driven, power-aware migration of VMs across service domains.

Sensor: Sensor infrastructure will monitor the power consumption, heat dissipation, and utilization of computing nodes in a virtualized Cloud environment. To this end, we will extend our Service Oriented Sensor Web [14] software system. Sensor Web provides a middleware infrastructure and programming model for creating, accessing, and utilizing tiny sensor devices that are deployed within a Cloud. The Cloud Coordinator service makes use of Sensor Web services for dynamic sensing of Cloud nodes and surrounding temperature. The output data reported by sensors are feedback to the Coordinator's Virtualization and Scheduling components, to optimize the placement, migration, and allocation of VMs in the Cloud. Such sensor-based real time monitoring of the Cloud operating environment aids in avoiding server breakdown and achieving optimal throughput out of the available computing and storage nodes.

Discovery and Monitoring: In order to dynamically perform scheduling, resource allocation, and VM migration to meet SLAs in a federated network, it is mandatory that up-to-date information related to Cloud's availability, pricing and SLA rules are made available to the outside domains via the Cloud Exchange. This component of Cloud Coordinator is solely responsible for interacting with the Cloud Exchange through remote messaging. The Discovery and Monitoring component undertakes the following activities: (i) updates the resource status metrics including utilization, heat dissipation, power consumption based on feedback given by the Sensor component; (ii) facilitates the Market and Policy Engine in periodically publishing the pricing policies, SLA rules to the Cloud Exchange; (iii) aids the Scheduling and Allocation component in dynamically discovering the Clouds that offer better optimization for SLA constraints such as deadline and budget limits; and (iv) helps the Virtualization component in determining load and power consumption; such information aids the Virtualization component in performing load-balancing through dynamic VM migration.

Further, system components will need to share scalable methods for collecting and representing monitored data. This leads us to believe that we should interconnect and monitor system components based on decentralized messaging and information indexing infrastructure, called Distributed Hash Tables (DHTs) [26]. However, implementing scalable techniques that monitor the dynamic behaviors related to services and resources is non-trivial. In order to support a scalable service monitoring algorithm over a DHT infrastructure, additional data distribution indexing techniques such as logical multi-dimensional or spatial indices [27] (MX-CIF Quad tree, Hilbert Curves, Z Curves) should be implemented.

3.2 Cloud Broker (CB)

The Cloud Broker acting on behalf of users identifies suitable Cloud service providers through the Cloud Exchange and negotiates with Cloud Coordinators for an allocation of resources that meets QoS needs of users. The architecture of Cloud Broker is shown in Figure 3 and its components are discussed below:

User Interface: This provides the access linkage between a user application interface and the broker. The Application Interpreter translates the execution requirements of a user application which include what is to be executed, the description of task inputs including remote data files (if required), the information about task outputs (if present), and the desired QoS. The Service Interpreter understands the service requirements needed for the execution which comprise service location, service type, and specific details such as remote batch job submission systems for computational services. The Credential Interpreter reads the credentials for accessing necessary services.

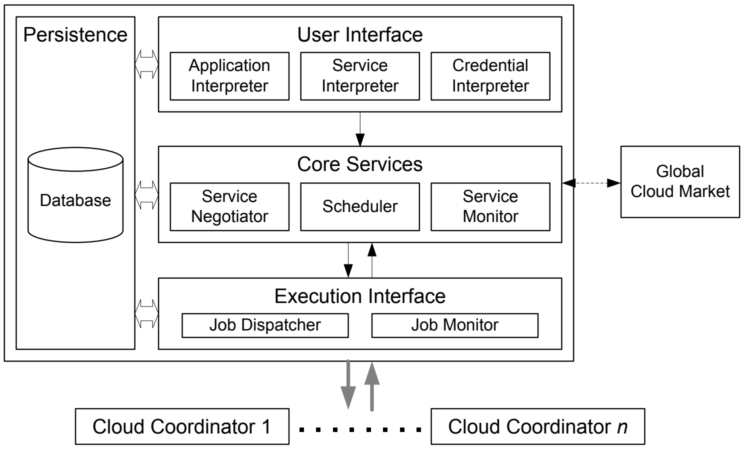


Fig. 3. High level architecture of Cloud Broker service

Core Services: They enable the main functionality of the broker. The Service Negotiator bargains for Cloud services from the Cloud Exchange. The Scheduler determines the most appropriate Cloud services for the user application based on its application and service requirements. The Service Monitor maintains the status of Cloud services by periodically checking the availability of known Cloud services and discovering new services that are available. If the local Cloud is unable to satisfy application requirements, a Cloud Broker lookup request that encapsulates the user's QoS parameter is submitted to the Cloud Exchange, which matches the lookup request against the available offers. The matching procedure considers two main system performance metrics: first, the user specified QoS targets must be satisfied within acceptable bounds and, second, the allocation should not lead to overloading (in terms of utilization, power consumption) of the nodes. In case the match occurs the quote is forwarded to the requester (Scheduler). Following that, the Scheduling and Allocation component deploys the application with the Cloud that was suggested by Cloud market.

Execution Interface: This provides execution support for the user application. The Job Dispatcher creates the necessary broker agent and requests data files (if any) to be dispatched with the user application to the remote Cloud resources for execution. The Job Monitor observes the execution status of the job so that the results of the job are returned to the user upon job completion.

Persistence: This maintains the state of the User Interface, Core Services, and Execution Interface in a database. This facilitates recovery when the broker fails and assists in user-level accounting.

3.3 Cloud Exchange (CEx)

As a market maker, the CEx acts as an information registry that stores the Cloud's current usage costs and demand patterns. Cloud Coordinators periodically update their availability, pricing, and SLA policies with the CEx. Cloud Brokers query the registry to learn information about existing SLA offers and resource availability of member Clouds in the federation. Furthermore, it provides match-making services that map user requests to suitable service providers. Mapping functions will be implemented by leveraging various economic models such as Continuous Double Auction (CDA) as proposed in earlier works [6].

As a market maker, the Cloud Exchange provides directory, dynamic bidding based service clearance, and payment management services as discussed below.

- **Directory:** The market directory allows the global CEx participants to locate providers or consumers with the appropriate bids/offers. Cloud providers can publish the available supply of resources and their offered prices. Cloud consumers can then search for suitable providers and submit their bids for required resources. Standard interfaces need to be provided so that both providers and consumers can access resource information from one another readily and seamlessly.
- **Auctioneer:** Auctioneers periodically clear bids and asks received from the global CEx participants. Auctioneers are third party controllers that do not represent any providers or consumers. Since the auctioneers are in total control of the entire trading process, they need to be trusted by participants.
- **Bank:** The banking system enforces the financial transactions pertaining to agreements between the global CEx participants. The banks are also independent and not controlled by any providers and consumers; thus facilitating impartiality and trust among all Cloud market participants that the financial transactions are conducted correctly without any bias. This should be realized by integrating with online payment management services, such as PayPal, with Clouds providing accounting services.

4 Early Experiments and Preliminary Results

Although we have been working towards the implementation of a software system for federation of cloud computing environments, it is still a work-in-progress. Hence, in this section, we present our experiments and evaluation that we undertook using CloudSim [29] framework for studying the feasibility of the proposed research vision. The experiments were conducted on a Celeron machine having the following configuration: 1.86GHz with 1MB of L2 cache and 1 GB of RAM running a standard Ubuntu Linux version 8.04 and JDK 1.6.

4.1 Evaluating Performance of Federated Cloud Computing Environments

The first experiment aims at proving that federated infrastructure of clouds has potential to deliver better performance and service quality as compared to existing non-federated approaches. To this end, a simulation environment that models federation of three Cloud providers and a user (Cloud Broker) is modeled. Every provider instantiates a Sensor component, which is responsible for dynamically sensing the availability information related to the local hosts. Next, the sensed statistics are reported to the Cloud Coordinator that utilizes the information in undertaking load-migration decisions. We evaluate a straightforward load-migration policy that performs online migration of VMs among federated Cloud providers only if the origin provider does not have the requested number of free VM slots available. The migration process involves the following steps: (i) creating a virtual machine instance that has the same configuration, which is supported at the destination provider; and (ii) migrating the applications assigned to the original virtual machine to the newly instantiated virtual machine at the destination provider. The federated network of Cloud providers is created based on the topology shown in Figure 4.

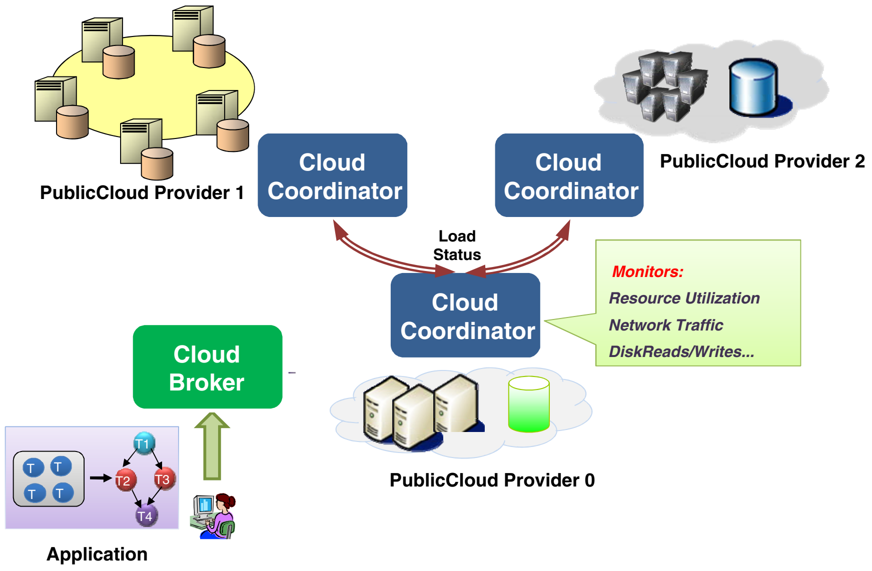


Fig. 4. A network topology of federated Data Centers

Every Public Cloud provider in the system is modeled to have 50 computing hosts, 10GB of memory, 2TB of storage, 1 processor with 1000 MIPS of capacity, and a time-shared VM scheduler. Cloud Broker on behalf of the user requests instantiation of a VM that requires 256MB of memory, 1GB of storage, 1 CPU, and time-shared Cloudlet scheduler. The broker requests instantiation of 25 VMs and associates one Cloudlet (Cloud application abstraction) to each VM to be executed. These requests are originally submitted with the Cloud Provider 0. Each Cloudlet is modeled to have 1800000 MIs (Million Instrictions). The simulation experiments were run under the following system configurations: (i) a federated network of clouds is available, hence data centers are able to cope with peak demands by migrating the excess of load to the least loaded ones; and (ii) the data centers are modeled as independent entities (without federation). All the workload submitted to a Cloud provider must be processed and executed locally.

Table 2 shows the average turn-around time for each Cloudlet and the overall makespan of the user application for both cases. A user application consists of one or more Cloudlets with sequential dependencies. The simulation results reveal that the availability of federated infrastructure of clouds reduces the average turn-around time by more than 50%, while improving the makespan by 20%. It shows that, even for a very simple load-migration policy, availability of federation brings significant benefits to user's application performance.

Table 2. Performance Results

Performance Metrics	**With Federation**	**Without Federation**	**% Improvement**
Average Turn Around Time (Secs)	2221.13	4700.1	> 50%
Makespan (Secs)	6613.1	8405	20%

4.2 Evaluating a Cloud Provisioning Strategy in a Federated Environment

In previous subsection, we focused on evaluation of the federated service and resource provisioning scenarios. In this section, a more complete experiment that also models the inter-connection network between federated clouds, is presented. This example shows how the adoption of federated clouds can improve productivity of a company with expansion of private cloud capacity by dynamically leasing resources from public clouds at a reasonably low cost.

The simulation scenario is based on federating a private cloud with the Amazon EC2 cloud. The public and the private clouds are represented as two data centers in the simulation. A Cloud Coordinator in the private data center receives the user's applications and processes them in a FCFS basis, queuing the tasks when there is available capacity for them in the infrastructure. To evaluate the effectiveness of a hybrid cloud in speeding up tasks execution, two scenarios are simulated. In the first scenario, tasks are kept in the waiting queue until active tasks finish (currently executing) in the private cloud. All the workload is processed locally within the private cloud. In the second scenario, the waiting tasks are directly sent to available public cloud. In other words, second scenario simulates a Cloud Bursts case for integrating local private cloud with public cloud form handing peak in service demands. Before submitting tasks to the Amazon cloud, the VM images (AMI) are loaded and instantiated. The number of images instantiated in the Cloud is varied in the experiment, from 10% to 100% of the number of machines available in the private cloud. Once images are created, tasks in the waiting queues are submitted to them, in such a way that only one task run on each VM at a given instance of time. Every time a task finishes, the next task in the waiting queue is submitted to the available VM host. When there were no tasks to be submitted to the VM, it is destroyed in the cloud.

The local private data center hosted 100 machines. Each machine has 2GB of RAM, 10TB of storage and one CPU run 1000 MIPS. The virtual machines created in the public cloud were based in an Amazon's small instance (1.7 GB of memory, 1 virtual core, and 160 GB of instance storage). We consider in this example that the virtual core of a small instance has the same processing power as the local machine.

The workload sent to the private cloud is composed of 10000 tasks. Each task takes between 20 and 22 minutes to run in one CPU. The exact amount of time was randomly generated based on the normal distribution. Each of the 10000 tasks is submitted at the same time to the scheduler queue.

Table 3 shows the makespan of the tasks running only in the private cloud and with extra allocation of resources from the public cloud. In the third column, we quantify the overall cost of the services. The pricing policy was designed based on the Amazon's small instances (U$ 0.10 per instance per hour) pricing model. It means that the cost per instance is charged hourly in the beginning of execution. And, if an instance runs during 1 hour and 1 minute, the amount for 2 hours (U$ 0.20) will be charged.

Table 3. Cost and performance of several public/private cloud strategies

	Makespan (s)	**Cloud Cost (U$)**
Private only	127155.77	0.00
Public 10%	115902.34	32.60
Public 20%	106222.71	60.00
Public 30%	98195.57	83.30
Public 40%	91088.37	103.30
Public 50%	85136.78	120.00
Public 60%	79776.93	134.60
Public 70%	75195.84	147.00
Public 80%	70967.24	160.00
Public 90%	67238.07	171.00
Public 100%	64192.89	180.00

Increasing the number of resources by a rate reduces the job makespan at the same rate, which is an expected observation or outcome. However, the cost associated with the processing increases significantly at higher rates. Nevertheless, the cost is still acceptable, considering that peak demands happen only occasionally and that most part of time this extra public cloud capacity is not required. So, leasing public cloud resources is cheapest than buying and maintaining extra resources that will spend most part of time idle.

5 Conclusions and Future Directions

Development of fundamental techniques and software systems that integrate distributed clouds in a federated fashion is critical to enabling composition and deployment of elastic application services. We believe that outcomes of this research vision will make significant scientific advancement in understanding the theoretical and practical problems of engineering services for federated environments. The resulting framework facilitates the federated management of system components and protects customers with guaranteed quality of services in large, federated and highly dynamic environments. The different components of the proposed framework offer powerful capabilities to address both services and resources management, but their end-to-end combination aims to dramatically improve the effective usage, management, and administration of Cloud systems. This will provide enhanced degrees of scalability, flexibility, and simplicity for management and delivery of services in federation of clouds.

In our future work, we will focus on developing comprehensive model driven approach to provisioning and delivering services in federated environments. These models will be important because they allow adaptive system management by establishing useful relationships between high-level performance targets (specified by operators) and low-level control parameters and observables that system components can control or monitor. We will model the behaviour and performance of different types of services and resources to adaptively transform service requests. We will use a broad range of analytical models and statistical curve-fitting techniques such as multi-class queuing models and linear regression time series. These models will drive and possibly transform the input to a service provisioner, which improves the efficiency of the system. Such improvements will better ensure the achievement of performance targets, while reducing costs due to improved utilization of resources. It will be a major advancement in the field to develop a robust and scalable system monitoring infrastructure to collect real-time data and re-adjust these models dynamically with a minimum of data and training time. We believe that these models and techniques are critical for the design of stochastic provisioning algorithms across large federated Cloud systems where resource availability is uncertain.

Lowering the energy usage of data centers is a challenging and complex issue because computing applications and data are growing so quickly that increasingly larger servers and disks are needed to process them fast enough within the required time period. Green Cloud computing is envisioned to achieve not only efficient processing and utilization of computing infrastructure, but also minimization of energy consumption. This is essential for ensuring that the future growth of Cloud Computing is sustainable. Otherwise, Cloud computing with increasingly pervasive front-end client devices interacting with back-end data centers will cause an enormous escalation of energy usage. To address this problem, data center resources need to be managed in an energy-efficient manner to drive Green Cloud computing. In particular, Cloud resources need to be allocated not only to satisfy QoS targets specified by users via Service Level Agreements (SLAs), but also to reduce energy usage. This can be achieved by applying market-based utility models to accept requests that can be fulfilled out of the many competing requests so that revenue can be optimized along with energy-efficient utilization of Cloud infrastructure.

Acknowledgements. We acknowledge all members of Melbourne CLOUDS Lab (especially William Voorsluys) for their contributions to InterCloud investigation.

References

[1] Kleinrock, L.: A Vision for the Internet. ST Journal of Research 2(1), 4–5 (2005)

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. University of California at Berkley, USA. Technical Rep UCB/EECS-2009-28 (2009)

[3] Buyya, R., Yeo, C., Venugopal, S., Broberg, J., Brandic, I.: Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. Future Generation Computer Systems 25(6), 599–616 (2009)

[4] London, S.: Inside Track: The high-tech rebels. Financial Times (September 6, 2002)

[5] The Reservoir Seed Team. Reservoir – An ICT Infrastructure for Reliable and Effective Delivery of Services as Utilities. IBM Research Report, H-0262 (Febuary 2008)

[6] Buyya, R., Abramson, D., Giddy, J., Stockinger, H.: Economic Models for Resource Management and Scheduling in Grid Computing. Concurrency and Computation: Practice and Experience 14(13-15), 1507–1542 (2002)

[7] Weiss, A.: Computing in the Clouds. NetWorker 11(4), 16–25 (2007)

[8] VMware: Migrate Virtual Machines with Zero Downtime, `http://www.vmware.com/`

[9] Barham, P., et al.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. ACM Press, New York (2003)

[10] Buyya, R., Abramson, D., Venugopal, S.: The Grid Economy. Special Issue on Grid Computing. In: Parashar, M., Lee, C. (eds.) Proceedings of the IEEE, vol. 93(3), pp. 698–714. IEEE Press, Los Alamitos (2005)

[11] Yeo, C., Buyya, R.: Managing Risk of Inaccurate Runtime Estimates for Deadline Constrained Job Admission Control in Clusters. In: Proc. of the 35th Intl. Conference on Parallel Processing, Columbus, Ohio, USA (August 2006)

[12] Yeo, C., Buyya, R.: Integrated Risk Analysis for a Commercial Computing Service. In: Proc. of the 21st IEEE International Parallel and Distributed Processing Symposium, Long Beach, California, USA (March 2007)

[13] Sulistio, A., Kim, K., Buyya, R.: Managing Cancellations and No-shows of Reservations with Overbooking to Increase Resource Revenue. In: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid, Lyon, France (May 2008)

[14] Chu, X., Buyya, R.: Service Oriented Sensor Web. In: Mahalik, N.P. (ed.) Sensor Network and Configuration: Fundamentals, Standards, Platforms, and Applications, January 2007. Springer, Berlin (2007)

[15] Amazon Elastic Compute Cloud (EC2), `http://www.amazon.com/ec2/` (March 17, 2010)
[16] Google App Engine, `http://appengine.google.com` (March 17, 2010)
[17] Windows Azure Platform, `http://www.microsoft.com/azure/` (March 17, 2010)
[18] Spring.NET, `http://www.springframework.net` (March 17, 2010)
[19] Chappell, D.: Introducing the Azure Services Platform. White Paper, `http://www.microsoft.com/azure` (January 2009)
[20] Venugopal, S., Chu, X., Buyya, R.: A Negotiation Mechanism for Advance Resource Reservation using the Alternate Offers Protocol. In: Proceedings of the 16th International Workshop on Quality of Service (IWQoS 2008), Twente, The Netherlands (June 2008)
[21] Salesforce.com, Application Development with Force.com's Cloud Computing Platform (2009), `http://www.salesforce.com/platform/` (Accessed December 16, 2009)
[22] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus Open-Source Cloud-Computing System. In: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2009), Shanghai, China, May 18-May 21 (2010)
[23] GoGrid Cloud Hosting, F5 Load Balancer. GoGrid Wiki (2009), `http://wiki.gogrid.com/wiki/index.php/F5_Load_Balancer` (Accessed September 21, 2009)
[24] Amazon CloudWatch Service, `http://aws.amazon.com/cloudwatch/`
[25] Amazon Load Balancer Service, `http://aws.amazon.com/elasticloadbalancing/`
[26] Lua, K., Crowcroft, J., Pias, M., Sharma, R., Lim, S.: A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. In: Communications Surveys and Tutorials, Washington, DC, USA, vol. 7(2) (2005)
[27] Ranjan, R.: Coordinated Resource Provisioning in Federated Grids. Ph.D. Thesis, The University of Melbourne, Australia (March 2009)
[28] Ranjan, R., Liu, A.: Autonomic Cloud Services Aggregation. CRC Smart Services Report (July 15, 2009)
[29] Buyya, R., Ranjan, R., Calheiros, R.N.: Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities. In: Proceedings of the 7th High Performance Computing and Simulation Conference (HPCS 2009), Leipzig, Germany, June 21-24. IEEE Press, New York (2009)
[30] Quiroz, A., Kim, H., Parashar, M., Gnanasambandam, N., Sharma, N.: Towards Autonomic Workload Provisioning for Enterprise Grids and Clouds. In: Proceedings of the 10th IEEE International Conference on Grid Computing (Grid 2009), Banff, Alberta, Canada, October 13-15 (2009)
[31] Feitelson, D.G.: Workload Modelling for Computer Systems Performance Evaluation (in preparation), `http://www.cs.huji.ac.il/~feit/wlmod/` (Accessed March 19, 2010)
[32] Vecchiola, C., Chu, X., Buyya, R.: Aneka: A Software Platform for .NET-based Cloud Computing. In: Gentzsch, W., Grandinetti, L., Joubert, G. (eds.) High Speed and Large Scale Scientific Computing, pp. 267–295. IOS Press, Amsterdam (2009), ISBN: 978-1-60750-073-5

Scalable Co-clustering Algorithms

Bongjune Kwon[1] and Hyuk Cho[2]

[1] Biomedical Engineering, The University of Texas at Austin,
Austin, TX 78712, USA
junenim@cs.utexas.edu
http://www.cs.utexas.edu/~junenim/

[2] Computer Science, Sam Houston State University,
Huntsville, TX 77341-2090, USA
hyukcho@shsu.edu
http://www.cs.shsu.edu/~hxc005/

Abstract. Co-clustering has been extensively used in varied applications because of its potential to discover latent local patterns that are otherwise unapparent by usual unsupervised algorithms such as k-means. Recently, a unified view of co-clustering algorithms, called Bregman co-clustering (BCC), provides a general framework that even contains several existing co-clustering algorithms, thus we expect to have more applications of this framework to varied data types. However, the amount of data collected from real-life application domains easily grows too big to fit in the main memory of a single processor machine. Accordingly, enhancing the scalability of BCC can be a critical challenge in practice. To address this and eventually enhance its potential for rapid deployment to wider applications with larger data, we parallelize all the twelve co-clustering algorithms in the BCC framework using message passing interface (MPI). In addition, we validate their scalability on eleven synthetic datasets as well as one real-life dataset, where we demonstrate their speedup performance in terms of varied parameter settings.

Keywords: Co-clustering, Message Passing Interface, Distributed Co-clustering, Scalable Co-clustering, Parallel Co-clustering.

1 Introduction

Clustering has played a significant role in mining hidden patterns in a data. Usual k-means clustering assumes rows (or columns) of similar/relevant function in a data matrix share similar profiles across all the given columns (or rows). Therefore, it may fail to identify local patterns, each of which consists of subsets of rows (or columns) co-regulated only under specific columns (or rows). Co-clustering, on the other hand, aims to find such latent local patterns and thus may provide the key to uncovering latent local patterns that are otherwise unapparent. Because of its potential to discover latent local patterns, co-clustering has been proven to be very useful for an extensive set of applications such as text mining [8], microarray analysis [5][4], recommender system [9], and so on. Co-clustering involves optimizing over a smaller number of parameters and thus could lead to several orders of magnitude reduction in the running time [8]. Recently, Banerjee et al. [2] formulated a unified view of co-clustering, called "Bregman co-clustering" (BCC), which includes six Euclidean distance and six I-divergence co-clustering algorithms. We expect to have more applications of this general framework to varied types of data being collected at an ever-increasing pace from observations and experiments in business and scientific domains. However, in case that the data is too big to fit in the main memory of a single processor, its scalability can become an inevitable challenge to overcome.

We have witnessed much research on parallelization of machine learning algorithms (e.g., [6] and [13]). In particular, the following approaches directly motivate this research and also provide technical inspiration. Dhillon and Modha [7] designed a parallel k-means algorithm, based on the message-passing model of parallel computing. Their implementation provides a general framework for parallelizing k-means-like clustering algorithms. This parallel k-means algorithm has nearly linear speedup. Nagesh et al. [11] presented a density and grid based clustering algorithm, pMAFIA, which introduces an adaptive grid framework to reduce computation overload and improve the quality of clusters. In addition, pMAFIA explores data as well as task parallelism to scale up to massive data sets and large set of dimensions. Recently, Pizzuti and Talia [12] presented a parallel clustering algorithm, p-AutoClass, to distributed memory minicomputers for Bayesian clustering. They defined a theoretical performance model of p-AutoClass to predict its execution time, speedup, and efficiency. Ahmad et al. [1] developed a parallel biclustering algorithm, called SPHier, based on bigraph crossing minimization. George and Merugu [9] proposed incremental and parallel co-clustering algorithm for collaborative filtering-based recommender system, based on the combination of Euclidean co-clustering algorithms with basis 5. Zhou and Khokar [14] parallelized the co-clustering algorithm in Cho et al. [5] (i.e., Euclidean co-clustering with basis 2) and showed scalable performance with near linear speedup.

We implement the parallel versions of all the twelve co-clustering algorithms in the unified BCC framwork using MPICH and C++ and measure their performance on the Sun Linux Cluster with 16 nodes. Each computing node has 8GB (2GB per core) of memory and two dual-core AMD OPTERON(TM) processors, connected with INFINIBAND interconnect. For validation purpose, we use the same synthetic dense data sets used by Zhou and Khokar [14] , two more dense data sets, and also one real-life sparse data, the NETFLIX movie data. We evaluate speedup performance with varied parameter settings such as data sizes and processor numbers. The overall experimental results support that our implementations are scalable and able to obtain nearly linear speedup with respect to the considered parameter settings.

The rest of the paper is organized as follows. In section 2, we present the general framework of BCC. In section 3, we describe the parallelization of the framework. In section 4, we discuss the experimental results. Finally, we conclude with summary of the proposed work and future research in section 5.

Algorithm 1. Sequential Bregman Co-clustering (SBCC) Algorithm

```
1  SBCC($A$, $k$, $\ell$, $\rho$, $\gamma$)
   Input: $A \in \mathbb{R}^{m\times n}$, $k$, $\ell$, $\rho \in \{1,\cdots,k\}^{m\times 1}$, and $\gamma \in \{1,\cdots,\ell\}^{n\times 1}$
   Output: Clustering indicator vectors $\rho$ and $\gamma$
2  begin
3      Initialize cluster assignment of $\rho$ and $\gamma$                    $(INIT)$
4      Update cluster statistics and $A^R$ and $A^C$, to reflect $\rho$ and $\gamma$
5      $newobj \leftarrow$ largeNumber
6      $\tau \leftarrow 10^{-5}\|A\|^2$; {Adjustable parameter}
7      repeat
8          for $1 \le i \le m$ do
9              $\rho(i) \leftarrow \arg\min_{1\le r\le k} E_{V|u}\left[d_\phi\left(A, A^R\right)\right]$    $(RB)$
10         end
11         Update cluster statistics and $A^R$ to reflect new $\rho$
12         for $1 \le j \le n$ do
13             $\gamma(j) \leftarrow \arg\min_{1\le c\le \ell} E_{U|v}\left[d_\phi\left(A, A^C\right)\right]$    $(CB)$
14         end
15         Update cluster statistics and $A^C$ to reflect new $\gamma$
16         $oldobj \leftarrow newobj$
17         $newobj \leftarrow E_{U|v}\left[d_\phi\left(A, A^C\right)\right]$
18     until $|oldobj - newobj| > \tau$
19 end
```

2 Sequential Bregman Co-clustering Algorithm

Algorithm 1 describes the batch update algorithm of the BCC framework in [2], which contains the two Minimum Sum-Squared Residue Co-clustering (MSSRCC) algorithms [5] with basis 2 and basis 6, respectively, and Information Theoretic Co-clustering (ITCC) [8] with basis 5, as special cases. The two key components in formulating a co-clustering problem include (i) choosing a set of critical co-clustering-based statistics to be preserved and (ii) selecting an appropriate measure to quantify the information loss or the discrepancy between the original data matrix and the compressed representation provided by the co-clustering. For example, in Biclustering [3] and MSSRCC [5], the row and column averages of each co-cluster are preserved and the discrepancy between the original and the compressed representation is measured in terms of the sum of element-wise squared difference. In contrast, ITCC [8], applicable to data matrices representing joint probability distributions, preserves different set of summary statistics, i.e., the row and column averages and the co-cluster averages. Further, the quality of the compressed representation is measured in terms of the sum of element-wise I-divergence. Refer to [2] for the details of the matrices, the Bregman divergence d_ϕ, and the expected Bregman divergence $E\left[d_\phi\left(\cdot,\cdot\right)\right]$.

For the given data matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, number of row clusters k, and number of column clusters ℓ, it begins with the initial clustering in the row and column clustering indicator vectors ρ and γ at step (**INIT**). Each iteration involves finding the closest row (column) cluster prototype, given by a row of $\boldsymbol{A}^R$ and a column of $\boldsymbol{A}^C$, at step (**RB**) for every row and step (**CB**) for every column, respectively. Meanwhile, the row and column clustering is stored in ρ and γ, respectively. Step (**RB**) takes $O(mk\ell)$ (in the optimized way suggested in both [5] and [2]), since every row seeks for the best row cluster of the row prototype matrix $\boldsymbol{A}^R \in k \times \ell$. Similarly, we can show that step (**CB**) takes $O(nk\ell)$. Therefore, overall computation time is $O(mn + mk\ell + nk\ell)$.

Note that similar analysis for the parallel version of Euclidean co-clustering with basis 5 can be found in [9]. In fact, basis 1 of BCC makes use of only row cluster average and column cluster average, thus basis 1 requires much less computation time than the other bases because it doesn't utilize the co-cluster average. Note also that we can implement the BCC algorithms efficiently with the recipe in [2]. To appreciate this generalization, researchers (e.g., Dhillon et al. [8] and Banerjee et al. [2]) viewed partitional co-clustering as a lossy data compression problem, where given a specified number of rows and column clusters, one attempts to retain as much information as possible about the original data matrix in terms of statistics based on the co-clustering.

3 Distributed Bregman Co-clustering Algorithm

Given the matrix averages, the separability of the row and column update cost functions allows us to separately find the optimal assignment for each row and column. Using this inherent data parallelism, we propose a parallel version of the co-clustering algorithm (Algorithm 2).

To obtain a parallel version of the co-clustering algorithm, we note that there are three main steps in each iteration of the co-clustering algorithm: (1) computing cluster statistics (i.e., **line 5**, **line 14**, and **line 20** of Algorithm 2); (2) obtaining row cluster assignments (**RB**); and (3) obtaining column cluster assignments (**CB**). The key idea is to divide the rows and columns among the processors so that the steps (**RB**) and (**CB**) can be completely performed in parallel. For updating statistics, each processor first computes its partial contribution and then all values are reduced to obtain the overall cluster statistics (i.e., various matrix averages).

Furthermore, we divide processing steps at each row (or column) batch iteration as follows: (**step** 1) the existing row (or column) mean vectors and a subset of the row (or column) points are taken into each subnode in distributed systems; (**step** 2) in each subnode, the distance between each row (or column) point and each mean vector is computed and then the point is assigned to the closest cluster; and (**step** 3) all the assigned cluster labels are collected and new cluster mean vectors are computed. Note that steps 1 and 2 are equivalent to the "map" task and step 3 to the "reduce" task in [13].

We assume that input data matrix, either in a dense or in a sparse format, is equally partitioned among processors for easy implementation. To be more

Algorithm 2. Distributed Bregman Co-clustering (DBCC) Algorithm

```
1  DBCC($A_{I\cdot}$, $A_{\cdot J}^T$, $k$, $\ell$, $\rho_I$, $\gamma_J$)
   Input: $A_{I\cdot} \in \mathbb{R}^{\frac{m}{P}\times n}$, $A_{\cdot J}^T \in \mathbb{R}^{\frac{n}{P}\times m}$, $k$, $\ell$, $\rho_I \in \{1,\cdots,k\}^{\frac{m}{P}\times 1}$, and
          $\gamma_J \in \{1,\cdots,\ell\}^{\frac{n}{P}\times 1}$
   Output: Clustering indicator vectors $\rho_I$ and $\gamma_J$
2  begin
3      $P$ = MPI_Comm_size()
4      Initialize cluster assignment of $\rho_I$ and $\gamma_J$                    $(INIT)$
5      Update statistics $S_{IJ}$ to reflect $\rho_I$ and $\gamma_J$
6      MPI_Allreduce $(S_{IJ}, S, \text{MPI\_SUM})$
7      Update row and column cluster prototypes, $A^R$ and $A^C$ according to $S$
8      $newobj \leftarrow$ largeNumber
9      $\tau \leftarrow 10^{-5}\|A\|^2$; {Adjustable parameter}
10     repeat
11         for $1 \le i \le \frac{m}{P}$ do
12             $\rho_I(i) \leftarrow \arg\min_{1\le r\le k} E_{V|u}\left[d_\phi\left(A_{I(i)\cdot}, A_r^R\right)\right]$    $(RB)$
13         end
14         Update statistics $S_{IJ}$ to reflect $\rho_I$
15         MPI_Allreduce $(S_{IJ}, S, \text{MPI\_SUM})$
16         Update column cluster prototype $A^C$ according to $S$
17         for $1 \le j \le \frac{n}{P}$ do
18             $\gamma_J(j) \leftarrow \arg\min_{1\le c\le \ell} E_{U|v}\left[d_\phi\left(A_{\cdot J(j)}^T, A_c^{C\,T}\right)\right]$    $(CB)$
19         end
20         Update statistics $S_{IJ}$ to reflect $\gamma_J$
21         MPI_Allreduce $(S_{IJ}, S, \text{MPI\_SUM})$
22         Update row cluster prototype $A^R$ according to $S$
23         $oldobj \leftarrow newobj$
24         $newobj \leftarrow E_{U|v}\left[d_\phi\left(A, A^R\right)\right]$
25     until $|oldobj - newobj| > \tau$
26 end
```

specific, the dense data matrix $A \in \mathbb{R}^{m\times n}$ is equally partitioned among P processors so that each processor can own $\frac{m}{P}$ rows (i.e., $A_{I\cdot}$) and $\frac{n}{P}$ columns (i.e., $A_{\cdot J}^T$). Notice that Algorithm 2 requires both partial rows and columns for each processor. Since each processor only owns a fraction of the data matrix A, the computation of the partial contributions to the various cluster averages takes only $O\left(\frac{mn}{P}\right)$ operations while the accumulation of these contributions requires $O\left(Pk\ell\right)$ operations. The row and cluster assignment operation further require $O\left(\frac{mk\ell}{P} + \frac{nk\ell}{P}\right)$ operations. Therefore, the overall computation time is $O\left(\frac{mn}{P} + \frac{mk\ell}{P} + \frac{nk\ell}{P} + Pk\ell\right)$, which corresponds to near linear speedup, ignoring the communication costs. Note that as for Algorithm 1, basis 1 requires less computation time then the other cases.

Table 1. Synthetic Dense Datasets

Dataset	Rows $(= m)$	Columns $(= n)$	Elements	Actual Size
d2	$131,072(= 2^{17})$	$64(= 2^6)$	2^{23}	64MB
d4	$262,144(= 2^{18})$	$64(= 2^6)$	2^{24}	128MB
d5	$524,288(= 2^{19})$	$64(= 2^6)$	2^{25}	256MB
d17	$1,048,576(= 2^{20})$	$64(= 2^6)$	2^{26}	512MB
d18	$524,288(= 2^{19})$	$128(= 2^7)$	2^{26}	512MB
d19	$262,144(= 2^{18})$	$256(= 2^8)$	2^{26}	512MB
d20	$131,072(= 2^{17})$	$512(= 2^9)$	2^{26}	512MB
d21	$65,536(= 2^{16})$	$1,024(= 2^{10})$	2^{26}	512MB
d22	$32,768(= 2^{15})$	$2,048(= 2^{11})$	2^{26}	512MB
d30	$16,384(= 2^{14})$	$8,192(= 2^{13})$	2^{27}	1GB
d31	$16,384(= 2^{14})$	$16,384(= 2^{14})$	2^{28}	2GB

4 Experimental Results

We now empirically study the characteristics of the proposed parallel framework for co-clustering algorithms. Table 1 summarizes details dimensional information of the synthetic dense datasets. For a given number of data points m and number of dimensions n, we generate all the 11 synthetic datasets using the IBM data generator [10]. The first nine synthetic dense data sets were also used by Zhou and Khokar [14] and the remaining two additional dense data sets, d30 and d31 were newly generated. Note that Zhou and Khokar [14] generated these data sets so that the resulting data sets have precisely specifiable characteristics. In addition, we use the NETFLIX movie data to evaluate the scalability performance on real-life sparse data, whose dimension is of $17,770$ rows, $480,189$ columns, and $100,480,507$ non-zero values stored in the CCS sparse matrix format.

We evaluate the speedup performance with varied parameter settings such as data sizes, numbers of row and column clusters, and numbers of processors. Note that data sizes range widely from 64MB to 2GB. Therefore, smaller ones can be loaded in a single processor machine, however larger ones, especially d30 and d31, are too big to fit in a main memory of one machine and thus they spend quite a lot of time completing their job.

Notice in both Algorithms 1 and 2 that initial row and column clusterings are taken as inputs since there are various ways to obtain the initial clustering. For simplicity, throughout our experiments, execution time is measured in seconds and averaged over 20 random initial clusterings. We use the same initial row and column cluster assignments for both the sequential co-clustering algorithms and the parallel co-clustering algorithm so that we compare performance based on the same input conditions. We report four sets of experiments, where we vary m, n, k, ℓ, and P, respectively. For all our experiments, we have varied the number of row cluster $k = 1, 2, 4, 8, 16$, and 32 and the number of column cluster $\ell = 1, 2, 4, 8, 16$, and 32. However, because of space limitation, we only report the results with a specific parameter setting, $k = \ell = 32$, since these

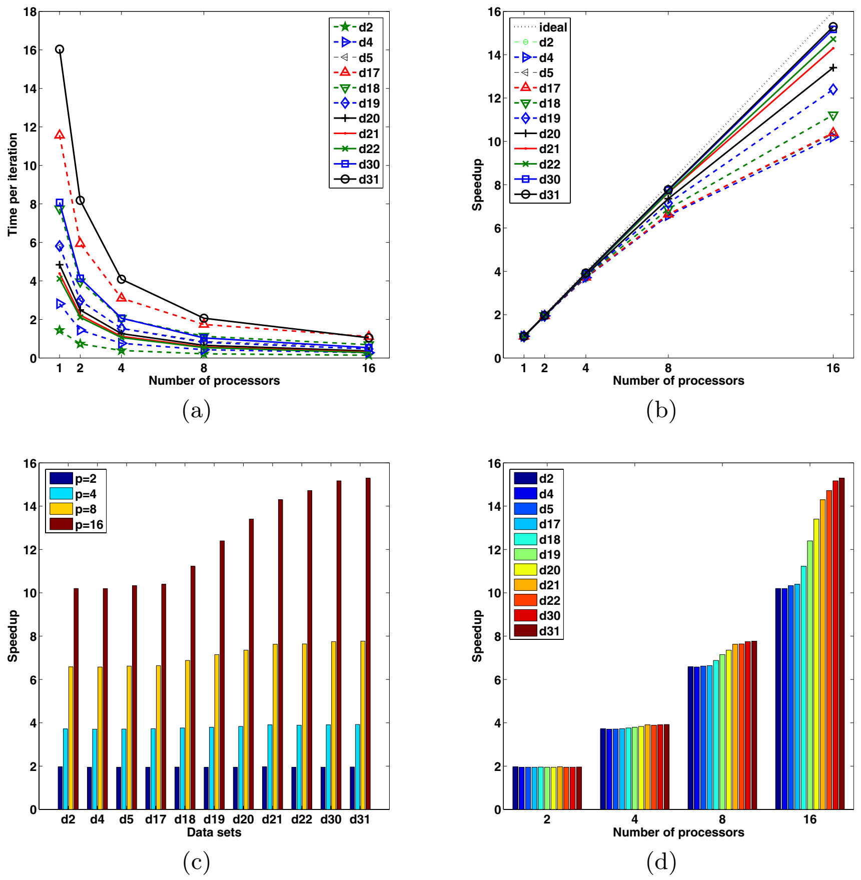


Fig. 1. Speedup of MSSRCCII (with basis 2) over varied data sets and processors

parameters clearly illustrate the characteristics of our parallel co-clustering algorithms. The other combinations also showed similar performance with small variations depending on data sets. Also, we have varied the number of processors $P = 1, 2, 4, 8$, and 16.

Relative speedup is defined as $\frac{T_1}{T_P}$, where T_1 is the execution time for coclustring a data set into k row clusters and ℓ column clusters on 1 processor and T_P is the execution time for identically co-clustering the same data set on P processors. Therefore, speedup depicts a summary of the efficiency of the parallel algorithm [7].

First, we study the speedup behavior when the number of data point m is varied. Specifically, we consider all the 11 data sets in Table 1. As discussed, we fixed the number of desired row clusters $k = 32$ and the number of desired column clusters $\ell = 32$. We also clustered each data set on $P = 1$, 2, 4, 8, and 16 processors. The measured execution times of all the 11 data sets are reported in plots (a) and (b) of Figures 1 and 2 for MSSRCC and ITCC, respectively. Observe

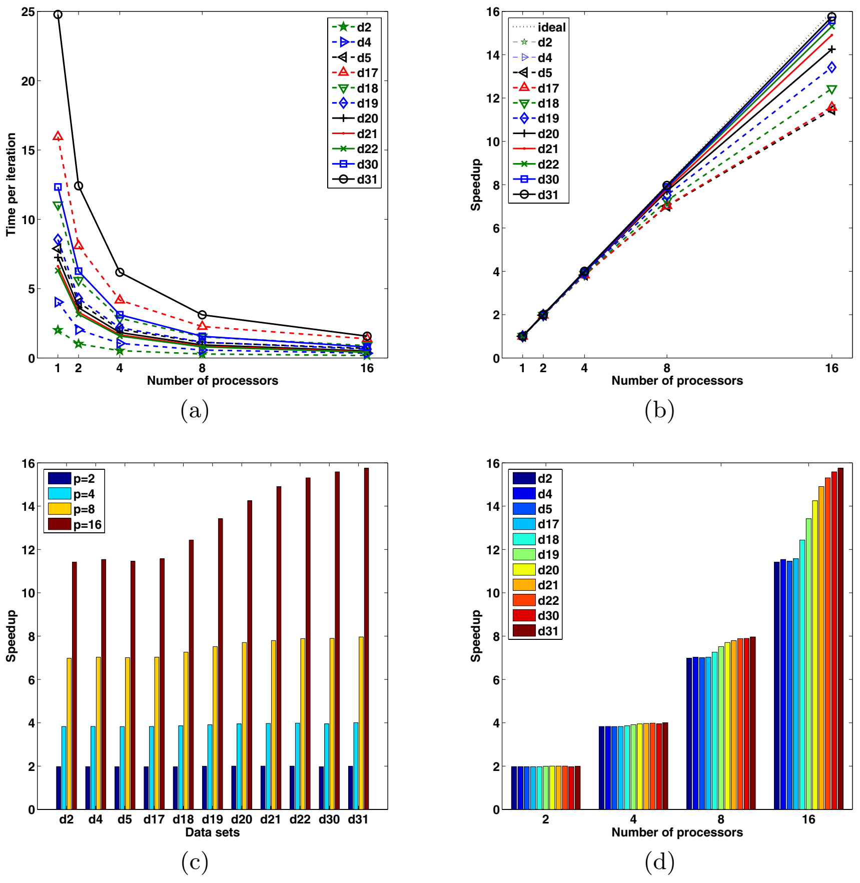


Fig. 2. Speedup of ITCC (with basis 5) over varied data sets and processors

that we measure execution time per iteration. In other words, we obtained plots by dividing the observed execution time by the number of iterations. This is necessary since each algorithm may require a different number of iterations for a different data set. The corresponding relative speedup results are also reported in plots (c) and (d) of Figures 1 and 2, where we can observe the following facts:

– Both algorithms achieve linear speedup for all the considered data sets. Speedups are linear with respect to the number of processors for all the cases. ITCC shows a little bit better performance than does MSSRCC.
– Relative speedup increases as data sizes increases with fixed number of processors. Speedups of both MSSRCC and ITCC come closer to the ideal speedup as data size increases. It verifies that our parallel co-clustering algorithms have a good sizeup behavior in the number of data points.

In particular, the plots (c) and (d) of Figures 1 and 2 illustrate the three different performance groups of the data sets as follows:

1. Group that shows similar speedup performance. We have very similar performance with small or similar sized data sets such as d2, d4, and d5. Notice that for these data, their column sizes are fixed (i.e., $n = 64$).
2. Group that shows increasing speedup performance, as row dimension decreases but column dimension increases. We have increasing speedups with a group of data sets whose row and column dimensions varies differently. To be more specific, for the data sets including d17 to d22, row dimensions are decreasing from 2^{20} down to 2^{15}, on the other hand, column dimensions are increasing from 2^{6} up to 2^{11}. By doing this way, we keep the overall matrix size (i.e., $m \times n$, fixed to 2^{26}, which is 512MB). As row sizes decrease but column sizes increase, the overall speedups are improved. The similar result has been reported in [14]. Since d22 has the smallest row size but the largest column size among the datasets in this group, it pays the least computation cost in the row cluster assignment phase. From this experiment, we observe that the cost of computing column updates dominates the computation time.

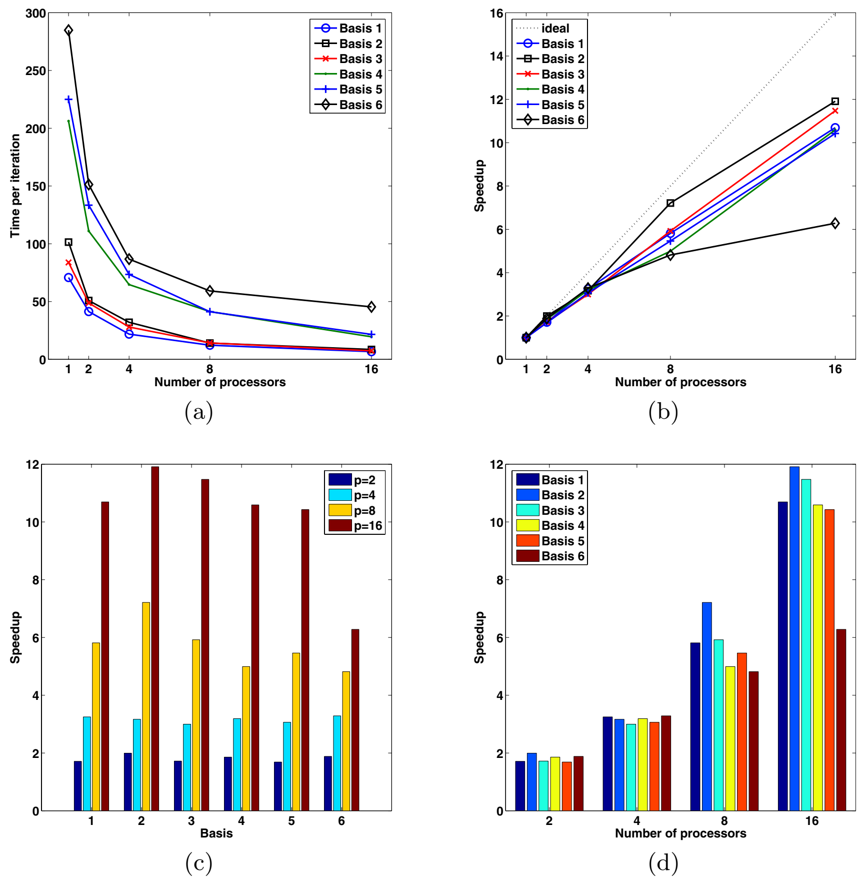


Fig. 3. Speedup of Euclidean co-clustering algorithms for NEFLIX data

3. Group that shows relatively increasing speedup performance, as data sizes increase (i.e., in both row and column dimensions). In plots (c) and (d) of Figures 1 and 2, this pattern is more clearly observed for larger data sets such as d30 and d31. It is because of the trade-off between communication time and computing time (i.e., communication overhead increases as data size grows). In this case, we might have more benefit if we use more processors.

Furthermore, Figures 1 and 2 demonstrate the relationship among speedups, data sizes, and number of processors. It shows that the relative speedup substantially increases with the ratio of the number of used processors increases regardless of the data size, when the number of processors is greater than 2. These speedups are linear with respect to the number of processors for most cases. For the two largest data sets, d30 and d31, we my obtain better speedups by exploiting more parallelism with more processors.

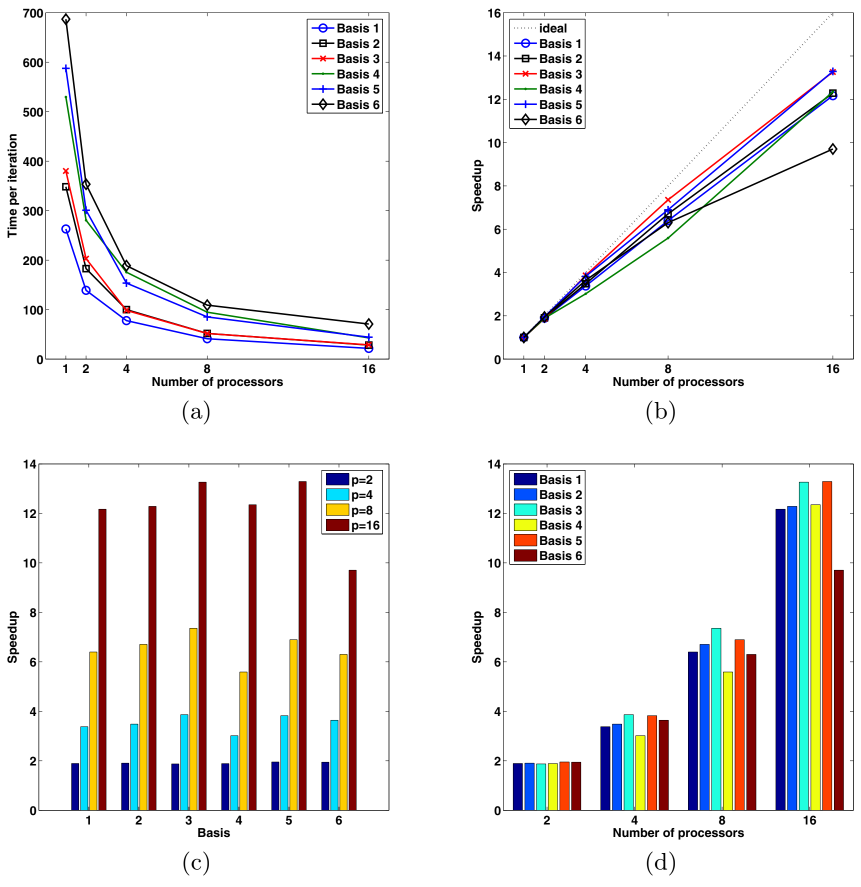


Fig. 4. Speedup of I-divergence co-clustering algorithms for NEFLIX data

For both dense and sparse matrices, we assume that input data matrix is equally partitioned among processors for easy implementation. Through this equal partitioning-based load balancing strategy, we are able to obtain near linear speedup for all the considered dense data matrices. Note that for space limitation we present the performance of distributed MSSRCC and ITCC on dense matrices in Figures 1 and 2, respectively, however all the other co-clustering algorithms result in the similar performance on all the considered dense matrices.

Figures 3 and 4 illustrate experimental results for sparse matrices. For this experiment, we use the NETFLIX movie data in the CCS format. We are able to obtain (sub-)linear speedup for most cases, whose performance is a little bit worse than that of the dense cases. Interestingly, with small-size data sets, the performances between dense and sparse cases are very close. However, as shown in plots (c) and (d) in Figures 3 and 4, with larger data sets and more processors, the performance gap between dense and sparse cases becomes bigger. In particular, both Euclidean and I-divergence co-clustering algorithms with basis 6 are not scaled well along with increased data size and processors. We ascribe this experimental results to the trade-off between communication time and computing time, since basis 6 requires more number of parameters than the other five bases.

5 Conclusion

Although there have been numerous studies on cluster and co-cluster mining, the study on parallel algorithms is limited to the traditional clustering problem. Recently, a few approaches are proposed but still focus on specific co-clustering algorithms. In this paper, we target the unified BCC framework and parallelize all the twelve co-clustering algorithms using message passing interface. The reasons of choosing Bregman co-clustering as a target in this research are manifold: (1) it is less complex since it is partitional, where all the rows and columns are partitioned into disjoint row and column clusters respectively; (2) it is more adaptable to varied domains since it is a unified framework that contains the six Euclidean co-clustering and the six I-divergence co-clustering algorithms, each of which targets on specific statistics of a given data matrix during co-clustering; and (3) it is more applicable and usable since given a specified row and column clusters, we can attempt to retain as much information as possible about the original data matrix in terms of the specific statistics for each of the algorithms.

With equal partitioning-based load balancing strategy, we are able to obtain near linear speedup for all the considered dense datasets. However, we are able to obtain (sub-)linear speedup for the sparse data. Therefore, for a sparse matrix, we need to employ more sophisticated partitioning schemes to ensure that the load can be evenly balanced in order to obtain better speedup.

The following is the list of some future research directions that we want to pursue: (1) we apply the parallel co-clustering algorithms to eleven synthetic datasets and one real-life dataset, and thus it is more desirable to apply the algorithms to larger real-world data sets; (2) we focus only on parallelizing batch update steps, however parallelizing local search step needs to be considered to get better local minima; (3) we parallelize all the co-clustering algorithms using MPI, however we can employ more advanced parallelization platforms such as "map-reduce" and "Hadoop"; (4) we emphasize more on experimental result than theoretical analysis and thus rigorous theoretical analysis similar to the analysis in [7] is desirable; and (5) performance comparison of the proposed approach with other existing co-clustering parallelization approaches are necessary to characterize different approaches.

References

1. Ahmad, W., Zhou, J., Khokhar, A.: SPHier: scalable parallel biclustering using weighted bigraph crossing minimization. Technical report, Dept. of ECE, University of Illinois at Chicago (2004)
2. Banerjee, A., Dhillon, I.S., Ghosh, J., Merugu, S., Modha, D.S.: A generalized maximum entropy approach to Bregman co-clustering and matrix approximation. Journal of Machine Learning Research 8, 1919–1986 (2007)
3. Cheng, Y., Church, G.M.: Biclustering of expression data. In: ISMB, vol. 8, pp. 93–103 (2000)
4. Cho, H., Dhillon, I.S.: Co-clustering of human cancer microarrays using minimum sum-squared residue co-clustering. IEEE/ACM Transactions on Computational Biology and Bioinformatics (IEEE/ACM TCBB) 5(3), 385–400 (2008)
5. Cho, H., Dhillon, I.S., Guan, Y., Sra, S.: Minimum sum squared residue based co-clustering of gene expression data. In: SDM, pp. 114–125 (2004)
6. Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, G., Ng, A., Olukotun, K.: Map-reduce for machine learning on multicore. In: NIPS (2006)
7. Dhillon, I.S., Modha, D.S.: A data clustering algorithm on distributed memory multiprocessors. In: Zaki, M.J., Ho, C.-T. (eds.) KDD 1999. LNCS (LNAI), vol. 1759, pp. 245–260. Springer, Heidelberg (2000)
8. Dhillon, I.S., Mallela, S., Modha, D.S.: Information-theoretic co-clustering. In: SIGKDD, pp. 89–98 (2003)
9. George, T., Merugu, S.: A scalable collaborative filtering framework based on co-clustering. In: ICDM, pp. 625–628 (2005)
10. IBM Quest synthetic data generation code for classification, http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html
11. Nagesh, H., Goil, S., Choudhary, A.: Parallel alogrithms for clustering high-dimensional large-scale datasets. In: Grossmen, R.L., Kamth, C., Kegelmeyer, P., Kumar, V., Namburu, R.R. (eds.) Data Mining for Scientific for Engineering Applications, pp. 335–356. Kluwer Academy Publishers, Dordrecht (2001)
12. Pizzuti, C., Talia, D.: P-AutoClass: scalable parallel clustering for mining large data sets. IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE) 15(3), 629–641 (2003)
13. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: HPCA (2007)
14. Zhou, J., Khokar, A.: ParRescue: scalable parallel algorithm and implementation for biclustering over large distributed datasets. In: ICDCS (2006)

Parallel Pattern Matching with Swaps on a Linear Array

Fouad B. Chedid

Department of Computer Science
Notre Dame University - Louaize
P.O. Box: 72 Zouk Mikael, Zouk Mosbeh, Lebanon
fchedid@ndu.edu.lb

Abstract. The Pattern Matching with Swaps problem is a variation of the classical pattern matching problem in which a match is allowed to include disjoint local swaps. In 2009, Cantone and Faro devised a new dynamic programming algorithm for this problem that runs in time $O(nm)$, where n is the length of the text and m is the length of the pattern. In this paper, first, we present an improved dynamic programming formulation of the approach of Cantone and Faro. Then, we present an optimal parallelization of our algorithm, based on a linear array model, that runs in time $O(m^2)$ using $\lceil \frac{n}{m-1} \rceil$ processors.

1 Introduction

The classical Pattern Matching problem (PM for short) is one of the most widely studied problems in computer science [9]. PM is defined as follows. Given a fixed alphabet Σ, a pattern $P \in \Sigma^*$ of length m and a text $T \in \Sigma^*$ of length $n \geq m$, PM asks for a way to find all occurrences of P in T. Part of the appeal of PM is in its direct real world applications. For example, applications in multimedia, digital libraries and computational biology have shown that variations of PM can be of tremendous benefit [17].

An early variant of PM appears in [11], where the authors allow a special "don't care" character that serves as a wild card. It matches every other alphabet symbol. Another early variant of PM, in which differences between characters of the pattern and characters of the text are allowed, appears in [14]. A difference is due to either a mismatch between a character of the text and a character of the pattern or a superfluous character in the text or a superfluous character in the pattern. The authors of [14] presented serial and parallel algorithms for finding all occurrences of the pattern in the text with at most k differences, for any fixed integer k. We mention that the literature contains a great amount of work on finding efficient algorithms for pattern matching with mismatches [1,4,10,13,15].

In this paper, we consider a variant of PM, named Pattern Matching with Swaps (PMS for short), in which a match is allowed to include disjoint local swaps. In particular, a pattern P is said to have a swapped match with a text T ending at location j if adjacent characters in P can be swapped, if necessary, so as to make P identical to the substring of T ending at location j. Here, all swaps

P = b **b a** b **a b** a b
T = a b **a b** b **b a** a b b a
j = 0 1 2 3 4 5 6 7 8 9 10

Fig. 1. An example of a swap-match

must be disjoint; that is, each character can be involved in at most one swap, and identical adjacent characters are not allowed to be swapped. This variant of PM has important applications in many fields such as computational biology, text and musical retrieval, data mining, and network security [8].

Example 1. We find it convenient to have an illustrative example for use throughout the paper, and the example of size $m = 8$ and $n = 11$ with the pattern and text shown in Fig. 1 is chosen for this purpose.

In our example, P has a swapped match with T ending at location $j = 8$ (swapped characters are shown in bold).

PMS was introduced in 1995 as an open problem in non-standard stringology [16]. Up until 1997, it had been an open problem whether PMS can be solved in time less than $O(nm)$. In [2,3], Amir *et al.* gave the first algorithm whose complexity beats the $O(nm)$ bound. More recently, Amir *et al.* [6] solved PMS in time $O(n \log m \log \sigma)$, where σ is the size of the alphabet ($= |\Sigma|$). We note that the above mentioned solutions to PMS are based on the Fast Fourier Transform (FFT) method [8]. The first attempt to solve PMS without using FFT is due to Iliopoulos and Rahman [12] who in 2008 devised an efficient algorithm, based on bit-parallelism, that runs in time $O((n + m) \log m)$ if the pattern size is comparable to the word size of the machine. In 2009, Cantone and Faro devised a new approach for solving PMS which resulted in an algorithm that is $O(nm)$. Moreover, they showed an efficient implementation of their algorithm, based on bit parallelism, that is $O(n)$, if the pattern size is comparable to the word size of the machine. Thus, for the first time, we have an algorithm that solves PMS for short patterns in linear time.

Let P_i denote the prefix of P of length $i + 1$. The main idea behind the work of Cantone and Faro is a new approach for finding all prefixes P_i of P, for $0 \leq i \leq m - 1$, that have swapped matches with T ending at some location j, for $0 \leq j \leq n - 1$. This will be denoted as $P_i \propto T_j$. The paper [8] defines the following collection of sets S_j, for $0 \leq j \leq n - 1$:

$$S_j = \{0 \leq i \leq m - 1 : P_i \propto T_j\}$$

Thus, the pattern P has a swapped match with T ending at location j if and only if $m - 1 \in S_j$. In the sequel, we use $P[i]$ to denote the $(i + 1)$th character of the string P; that is, the first character of P is denoted as $P[0]$.

To compute S_j, the authors of [8] define another collection of sets S'_j, for $0 \leq j \leq n - 1$, as follows.

$$S'_j = \{0 \leq i \leq m - 1 : P_{i-1} \propto T_{j-1} \text{ and } P[i] = T[j + 1]\}$$

Then, it is shown how to compute S_j in terms of S_{j-1} and S'_{j-1}, where S'_{j-1} is computed in terms of S_{j-2}. This formulation of the solution gives a $O(mn)$ time algorithm for PMS named Cross-Sampling.

In this paper, we provide a simpler implementation of Cross-Sampling. In particular, we present a much simpler approach to computing the sets S_j. Define the Boolean matrix S^i_j, for $0 \le i \le m-1, 0 \le j \le n-1$, as follows.

$$\begin{aligned} S^i_j &= 1 \text{ if } P_i \propto T_j \\ &= 0, \text{ otherwise.} \end{aligned}$$

Thus, the pattern P has a swapped match with T ending at location j if and only if $S^{m-1}_j = 1$. Then, we show how to compute S^i_j in terms S^{i-1}_{j-1} and S^{i-2}_{j-2}. This new formulation of the solution gives an improved presentation of Cross-Sampling that is easier to be comprehended and implemented. We name our algorithm Prefix-Sampling.

Then, we present an optimal parallelization of Prefix-Sampling, based on a linear array model, that runs in time $O(m^2)$ using $\lceil \frac{n}{m-1} \rceil$ processors. In our parallel algorithm, we need to know when a suffix (rather than prefix) S_i of length $i+1$ of a pattern P has a swapped match with a text T beginning (rather than ending) at some location j. Let w^r denote the reversal of a string w (w^r is w spelled backwards). Then, it is easy to see that S_i has a swapped match with T beginning at location j if and only if S^r_i has a swapped match with T^r ending at location $(n-1-j)$. Our parallel algorithm divides the text string T into $r = \lceil \frac{n}{m-1} \rceil$ parts $T_0, \ldots, T_{r-1}$. This idea is inspired by the work in [18] on suffix automata for parallel string matching. Then, for each pair of consecutive parts T_i, T_{i+1} of T, for $0 \le i \le r-2$, in parallel, we run Prefix-Sampling on T_i and T^r_{i+1}. This finds all prefixes P_i $(0 \le i \le |T_i|-1)$ of P that have swapped matched with T_i ending at some location j in T_i, and all suffixes $S_{i'}$ $(0 \le i' \le |T_{i+1}|-1)$ of P that have swapped matches with T_{i+1} beginning at some location j' in T_{i+1}. With this information available, P has a swapped match with T if the following conditions hold:

1. Prefix-Sampling(T_i) returns some prefix $P_i \propto T_j$, where j is the index of the last character in T_i.
2. Prefix-Sampling(T^r_{i+1}) returns some suffix $S_{i'}$ whose reversal $S^r_{i'} \propto T^r_{n-1-(j+1)}$. Here, $(j+1)$ is the index of the first character in T_{i+1}.
3. $|P_i| + |S_{i'}| = m$ or
4. $P_{i-1} \propto T_{j-1}$, $S^r_{i'-1} \propto T^r_{n-1-(j+2)}$, $|P_{i-1}| + |S_{i'-1}| = i + i' = m-2$, and $P[i] = T[j+1]$ and $P[i+1] = T[j]$.

This approach works because of the way we split the text T. That is, by dividing T into $r = \lceil \frac{n}{m-1} \rceil$ parts, we guarantee that each occurrence of the pattern P in T must span two consecutive parts.

Then, this paper concludes with some remarks on how this work can be extended to solve the Approximate Pattern Matching with Swaps problem (APMS for short). In APMS, we are to find for each text location $0 \le j \le n-1$ with a swapped-match of the pattern P, the number of swaps needed to obtain a match at that location.

The rest of the paper is organized as follows. Section 2 gives basic definitions. Section 3 reviews the Cross-Sampling algorithm for PMS. Our Prefix-Sampling algorithm is included in Section 4. Section 5 presents our optimal parallel Prefix-Sampling algorithm. Section 6 is the conclusion.

2 Problem Definition

Let Σ be a fixed alphabet and let P be a string over Σ of length m. The string P will be represented as a finite array $P[0 \ldots m-1]$. The prefix of P of length $i+1$ will be denoted as $P_i = P[0 \ldots i]$, for $0 \leq i \leq m-1$. We quote the following definition from [8]:

Definition 1. *A swap permutation of P is a permutation $\pi : \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m-1\}$ such that:*

1. *if $\pi(i) = j$ then $\pi(j) = i$ (characters are swapped).*
2. *for all $i, \pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters can be swapped)*
3. *if $\pi(i) \neq i$ then $P[\pi(i)] \neq P[i]$ (identical characters are not allowed to be swapped)*

The swapped version of P under the permutation π is denoted as $\pi(P)$; that is, $\pi(P)$ is the concatenation of the characters $P[\pi(i)]$, for $0 \leq i \leq m-1$. For a given text $T \in \Sigma^*$, we say that P has a swapped match with T ending at location j, denoted $P \propto T_j$, if there exists a swap permutation π of P such that $\pi(P)$ has an exact match with T ending at location j. For example, the swap permutation that corresponds to our example in Fig. 1 is $\pi(bbababab) = babbbaab$. This swap–match has two swaps: $\pi(1) = 2, \pi(2) = 1$ and $\pi(4) = 5, \pi(5) = 4$. In this case, we write $P \propto_2 T_8$.

The *Pattern Matching with Swaps Problem (PMS for short)* is the following:

INPUT: A text string $T[0 \ldots n-1]$ and a pattern string $P[0 \ldots m-1]$ over a fixed alphabet Σ.
OUTPUT: All locations j for $m-1 \leq j \leq n-1$ such that $P \propto T_j$.

The *Approximate Pattern Matching with Swaps Problem (APMS for short)* is the following:

INPUT: A text string $T[0 \ldots n-1]$ and a pattern string $P[0 \ldots m-1]$ over a fixed alphabet Σ.
OUTPUT: The number of swaps k needed for each $m-1 \leq j \leq n-1$, where $P \propto_k T_j$.

3 The Cross-Sampling Algorithm

In [8], Cantone anf Faro propose a new approach to solving PMS. Their resultant algorithm, named Cross-Sampling, runs in time $O(nm)$, where n and m are the lengths of the text and pattern, respectively. Moreover, the paper [8] describes an efficient implementation of Cross-Sampling based on bit parallelism [7] that solves PMS in time $O(n)$, if the pattern size is similar to the word size of the machine. This section reviews the basics of Cross-Sampling for PMS.

The main idea behind Cross-Sampling is a new approach to computing the swap occurrences of all prefixes of P in continuously increasing prefixes of T using dynamic programming. Cross-Sampling computes two collections of sets S_j and S'_j, for $0 \leq j \leq n-1$, defined as follows.

$$S_j = \{0 \leq i \leq m-1 : P_i \propto T_j\}$$
$$S'_j = \{0 \leq i \leq m-1 : P_{i-1} \propto T_{j-1} \text{ and } P[i] = T[j+1]\}$$

Thus, the pattern P has a swapped match with T ending at location j if and only if $m-1 \in S_j$. The following lemma, quoted from [8], provides a simple way for computing the sets S_j.

Lemma 1. *Let T and P be a text of length n and a pattern of length m, respectively. Then, for $0 \leq i \leq m-1, 0 \leq j \leq n-1$, $P_i \propto T_j$ if and only if one of the following two facts holds:*

- *$P[i] = T[j]$ and $P_{i-1} \propto T_{j-1}$.*
- *$P[i] = T[j-1], P[i-1] = T[j]$, and $P_{i-2} \propto T_{j-2}$.*

Based on Lemma 1, the following recursive definitions of S_{j+1} and S'_{j+1} are proposed. For all $0 \leq j < n-1$,

$$\begin{aligned} S_{j+1} = \{i \leq m-1 | (i-1 \in S_j \text{ and } P[i] = T[j+1]) \text{ or} \\ (i-1 \in S'_j \text{ and } P[i] = T[j])\} \cup \lambda_{j+1} \end{aligned}$$

$$S'_{j+1} = \{i < m-1 | i-1 \in S_j \text{ and } P[i] = T[j+2]\} \cup \lambda_{j+2}$$

where $\lambda_j = \{0\}$ if $P[0] = T[j]$, and $\emptyset$ otherwise. The base cases of these recursive relations are given by $S_0 = \lambda_0$ and $S'_0 = \lambda_1$.

Such recursive relations allow the computation of S_j and S'_j in an iterative fashion in time $O(mn)$. Here, S_j is computed in terms of S_{j-1} and S'_{j-1}. S'_j is computed in terms of S_{j-1}.

4 A Simpler Implementation of Cross-Sampling

We propose a simpler implementation of the Cross-Sampling algorithm. In particular, we propose a much simpler formulation of the dynamic programming relations used in the computation of the sets S_j.

Define the Boolean matrix S_i^j, for $0 \leq i \leq m-1, 0 \leq j \leq n-1$, as follows.

$$\begin{aligned} S_i^j &= 1, \text{ if } P_i \propto T_j \\ &= 0, \text{ otherwise.} \end{aligned}$$

```
Algorithm Prefix-Sampling $(P, m, T, n)$
1.   $S[m,n] \leftarrow \{0\}$ /* initially, all array elements are set to False
2.   for $j \leftarrow 0$ to $n-1$ do
3.       $S[0,j] \leftarrow P[0] = T[j]$
4.   for $j \leftarrow 1$ to $n-1$ do
5.       $S[1,j] \leftarrow (S[0,j-1] \wedge (P[1] = T[j]))$
6.                 $\vee((P[1] = T[j-1]) \wedge (P[0] = T[j]))$
7.   for $i \leftarrow 2$ to $m-1$ do
8.       for $j \leftarrow i$ to $n-1$ do
9.           $S[i,j] \leftarrow (S[i-1,j-1] \wedge (P[i] = T[j]))$
10.                  $\vee(S[i-2,j-2] \wedge ((P[i] = T[j-1]) \wedge (P[i-1] = T[j])))$
11.  for $j \leftarrow m-1$ to $n-1$ do
12.      if $S[m-1,j]$ then print j /* Here, $P \propto T_j$ */
```

Fig. 2. The Prefix-Sampling Algorithm for PMS

Thus, the pattern P has a swapped match with T ending at location j if and only if $S^j_{m-1} = 1$. The following recursive definition of S^j_i is inspired by Lemma 1. For $2 \leq i \leq m-1,\ i \leq j \leq n-1$, we have

$$\begin{aligned} S^j_i \leftarrow &(S^{j-1}_{i-1} \wedge (P[i] = T[j])) \\ &\vee (S^{j-2}_{i-2} \wedge ((P[i] = T[j-1]) \wedge (P[i-1] = T[j]))) \end{aligned} \tag{1}$$

The base cases for $i = 0$ and $i = 1$ are given by

$$\begin{aligned} S^j_0 \leftarrow &P[0] = T[j],\ \text{for } 0 \leq j \leq n-1 \\ S^j_1 \leftarrow &(S^{j-1}_0 \wedge (P[1] = T[j])) \\ &\vee ((P[1] = T[j-1]) \wedge (P[0] = T[j])),\ \text{for } 1 \leq j \leq n-1 \end{aligned}$$

This formulation for computing the sets S_j is much simpler than the formulation used in Cross-Sampling (See the previous section). The recursive relations in Equation 1 are readily translatable into an efficient algorithm for computing the elements of the matrix S^n_m iteratively. Our resultant algorithm, named Prefix-Sampling, is shown in Fig. 2. Clearly, this algorithm runs in time $O(mn)$. We

Table 1. A Sample Run of Prefix-Sampling

$i\backslash j$	0	1	2	3	4	5	6	7	8	9	10
0	0	1	0	1	1	1	0	0	1	1	0
1	0	0	0	0	1	1	0	0	0	1	0
2	0	0	0	1	0	0	1	0	0	0	1
3	0	0	0	0	1	0	1	0	0	0	0
4	0	0	0	0	0	0	0	1	1	0	0
5	0	0	0	0	0	0	1	0	1	1	0
6	0	0	0	0	0	0	0	1	0	0	1
7	0	0	0	0	0	0	0	0	1	0	1

have traced Prefix-Sampling on our example in Fig. 1. The result is shown in Table 1.

Table 1 shows that P has swapped matches with T ending at locations $j = 8$ and $j = 10$ in T. These values of j correspond to the True entries in the last row ($i = m - 1 = 7$) of the array.

5 Parallel Prefix Sampling

We present an optimal parallelization of Prefix-Sampling based on a linear array model. Our algorithm uses $\lceil \frac{n}{m-1} \rceil$ processors and runs in time $O(m^2)$.

First, we state the following lemma:

Lemma 2. *Let T and P be a text of length n and a pattern of length m, respectively. Let S_i denote the suffix of P of length $i+1$. Then, for $0 \leq i \leq m-1, 0 \leq j \leq n-1$, S_i has a swapped match with T beginning at location j if and only if $S_i^r \propto T_{n-1-j}^r$.*

Proof. It follows from Lemma 1 and the definition of the reversal of a string.

Thus, a suffix S_i of P has a swapped match with T beginning at location j if and only if the reversal S_i^r of S_i, or equivalently, the prefix of the reversal P^r of P of length $i+1$ ($= P_i^r$), has a swapped match with the reversal T^r of T ending at location $n-1-j$.

Our algorithm divides T into $r = \lceil \frac{n}{m-1} \rceil$ parts $T_0, \ldots, T_{r-1}$. The part of T of length h ending at location i is $T[i-h+1 \ldots i]$. For $0 \leq j \leq r-1$, let l_j denote the length of T_j. Observe that for $0 \leq j \leq r-2$, $l_j = m-1$, and $l_{r-1} = n\%(m-1)$. For each part T_j of T, for $0 \leq j \leq r-1$, in parallel, the algorithm uses processor numbered j to execute Prefix-Sampling(P, l_j, T_j, l_j) and Prefix-Sampling(P^r, l_j, T_j^r, l_j). This finds all prefixes P_i, for $0 \leq i \leq l_j - 1$, of P that have swapped matched with T_j ending at some location k, and all suffixes $S_{i'}$, for $0 \leq i' \leq l_j - 1$, of P that have swapped matches with T_j beginning at some location k'. Then, P has a swapped match with T if the following conditions hold:

1. Prefix-Sampling(P, l_j, T_j, l_j) returns some prefix $P_i \propto (T_j)_{l_j-1}$.
2. Prefix-Sampling($P^r, l_{j+1}, T_{j+1}^r, l_{j+1}$) returns some suffix $S_{i'}$ of P whose reversal $S_{i'}^r \propto (T_{j+1}^r)_{l_{j+1}-1}$.
3. $|P_i| + |S_{i'}| = i + i' + 2 = m$ or
4. $P_{i-1} \propto (T_j)_{l_j-2}$, $S_{i'-1}^r \propto (T_{j+1}^r)_{l_{j+1}-2}$, $|P_{i-1}| + |S_{i'-1}| = i + i' = m-2$, and $P[i] = T_{j+1}[0]$ and $P[i+1] = T_j[l_j - 1]$.

Our parallel prefix-Sampling algorithm for PMS, named Parallel-Prefix-Sampling, is shown in Fig. 3.

The run time of Step 1 is $O(m^2)$. The run time of Step 2 is $O(m)$. Thus, the overall running time of Parallel-Prefix-Sampling is $O(m^2)$. Its cost is $O(m^2) \cdot \lceil \frac{n}{m-1} \rceil = O(mn)$. Hence, this is an optimal parallelization of Prefix-Sampling.

```
Algorithm Parallel-Prefix-Sampling $(P, m, T, n)$
[0] Let $r = \lceil \frac{n}{m-1} \rceil$.
[1] For each processor $P_j$, for $0 \le j \le r-1$, in parallel, do
      $P_{j \neq (r-1)}$ considers the $jth$ part of the text $T_j = T[j(m-1) \dots (j+1)(m-1)-1]$.
      $P_{r-1}$ considers the last part of the text $T_{r-1} = T[r(m-1) \dots r(m-1) + n\%(m-1) - 1]$.
      Let $l_j$ equal the length of $T_j$, for $0 \le j \le r-1$.
      $P_j$ executes
           Prefix-Sampling$(P, l_j, T_j, l_j)$
           Prefix-Sampling$(P^r, l_j, T_j^r, l_j)$
[2] For each processor $P_j$ $(0 \le j \le r-2)$, in parallel, do
       for $i = 0$ to $m-1$ do
           Let $i' = m - 2 - i$
           if $(P_i \propto (T_j)_{l_j-1}$ and $P^r_{i'} \propto (T^r_{j+1})_{l_{j+1}-1})$ or
               $(P_{i-1} \propto (T_j)_{l_j-2}$ and $P^r_{i'-1} \propto (T^r_{j+1})_{l_{j+1}-2}$ and
                   $P[i] = T_{j+1}[0]$ and $P[i+1] = T_j[l_j - 1])$ then
                       $P \propto T_{(j+1)(m-1)+i'}$.
```

Fig. 3. The Parallel-Prefix-Sampling Algorithm for PMS

P	=	b b a b a b a	P^r	=	b a b a
T_0	=	a b a b b b a	T_1^r	=	a b b a
j	=	0 1 2 3 4 5 6	j	=	0 1 2 3

Fig. 4. Input to P_0 is shown on the left; input to P_1 is shown on the right

We have traced Parallel-Prefix-Sampling on our example in Fig. 1. Step 1 of the algorithm divides T into $r = \lceil \frac{n}{m-1} \rceil = \lceil \frac{11}{7} \rceil = 2$ parts T_0 and T_1 of length $m - 1 = 7$ and $n\%(m-1) = 4$, respectively. Then, it runs processors P_0 and P_1 in parallel executing Prefix-Sampling$(P, 7, T_0, 7)$ on P_0 and Prefix-Sampling$(P^r, 4, T_1^r, 4)$ on P_1. Fig. 4 shows the inputs to P_0 and P_1. The outputs of P_0 and P_1 are shown in Table 2 and Table 3, respectively.

Table 2 shows (See column numbered $j = 6$) that there are prefixes of length 3, 4, and 6 of P that have swapped matches with T_0 ending at the last character of T_0. Table 3 shows (See column numbered $j = 3$) that there are prefixes (suffixes) of length 2, 3, and 4 of P^r (P) that have swapped matches with T_1 ending (beginning) at the last (first) character of T_1^r (T_1). Step 2 of Parallel-Prefix-Sampling combines these results to conclude that $P \propto T_8$ and $P \propto T_{10}$.

Table 2. The Output of P_0

$i \backslash j$	0	1	2	3	4	5	6
0	0	1	0	1	1	1	0
1	0	0	0	0	1	1	0
2	0	0	0	1	0	0	1
3	0	0	0	0	1	0	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0

Table 3. The Output of P_1

$i' \backslash j$	0	1	2	3
0	0	1	1	0
1	0	1	0	1
2	0	0	1	1
3	0	0	0	1

6 Conclusion

In this paper, we provided a simpler implementation of a recent dynamic programming approach proposed for solving the Pattern Matching with Swaps problem. Also, we presented an optimal parallelization of our approach, based on a linear array model, that runs in time $O(m^2)$ using $\lceil \frac{n}{m-1} \rceil$ processors. We mention that the work presented here can be easily extended to solve the Approximate Pattern Matching with Swaps problem. In particular, our dynamic programming formulation for PMS (Equation 1 from Section 4) can be adapted for APMS as follows. Redefine S_i^j so that

$$\begin{aligned} S_i^j &= k+1, \text{ if } P_i \propto_k T_j \\ &= 0, \text{ otherwise.} \end{aligned}$$

Thus, the pattern P has a swapped match with T with k swaps ending at location j if and only if $S_{m-1}^j = k+1$. The following recursive definition of S_i^j is inspired by the following lemma quoted from [8]:

Lemma 3. *Let T and P be a text of length n and a pattern of length m, respectively. Then, for $0 \le i \le m-1, 0 \le j \le n-1$, $P_i \propto_k T_j$ if and only if one of the following two facts holds:*

- *$P[i] = T[j]$ and either $(i = 0$ and $k = 0)$ or $P_{i-1} \propto_k T_{j-1}$.*
- *$P[i] = T[j-1], P[i-1] = T[j]$, and either $(i = 1$ and $k = 1)$ or $P_{i-2} \propto_{k-1} T_{j-2}$.*

For $2 \le i \le m-1,\ i \le j \le n-1$, we have

$$\begin{aligned} S_i^j &= S_{i-1}^{j-1}, \text{ if } S_{i-1}^{j-1} \wedge (P[i] = T[j]). \\ &= S_{i-2}^{j-2} + 1, \text{ if } S_{i-2}^{j-2} \wedge ((P[i] = T[j-1]) \wedge (P[i-1] = T[j])). \\ &= 0, \text{ otherwise.} \end{aligned}$$

The base case for $i = 0$, for $0 \le j \le n-1$, is given by

$$S_0^j = (P[0] = T[j]).$$

The base case for $i = 1$, for $1 \le j \le n-1$, is given by

$$\begin{aligned} S_1^j &= 1, \text{ if } S_0^{j-1} \wedge (P[1] = T[j]) \\ &= 2, \text{ if } (P[1] = T[j-1]) \wedge (P[0] = T[j]) \\ &= 0, \text{ otherwise.} \end{aligned}$$

References

1. Abrahamson, K.: Generalized String Matching. SIAM Journal of Computing 16, 1039–1051 (1987)
2. Amir, A., Aumann, Y., Landau, G.M., Lewenstein, M., Lewenstein, N.: Pattern Matching With Swaps. In: 38th Annual Symposium on Foundations of Computer Science, pp. 144–153. IEEE Press, Los Alamitos (1997)
3. Amir, A., Aumann, Y., Landau, G.M., Lewenstein, M., Lewenstein, N.: Pattern Matching With Swaps. Journal of Algorithms 37, 247–266 (2000)
4. Amir, A., Lewenstein, M., Porat, E.: Faster String Matching With k Mismatches. In: 11th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 794–803 (2000)
5. Amir, A., Lewenstein, M., Porat, E.: Approximate Swapped Matching. Information Processing Letters 83(1), 33–39 (2002)
6. Amir, A., Cole, R., Hariharan, R., Lewenstein, M., Porat, E.: Overlap Matching. Inf. Comput. 181(1), 57–74 (2003)
7. Baeza-Yates, R., Gonnet, G.H.: A New Approach to Text Searching. Communications of the ACM 35(10), 74–82 (1992)
8. Cantone, D., Faro, S.: Pattern Matching With Swaps for Short Patterns in Linear Time. In: Nielsen, M., Kucera, A., Miltersen, P.B., Palamidessi, C., Tuma, P., Valencia, F.D. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 255–266. Springer, Heidelberg (2009)
9. Galil, Z.: Open Problems in Stringology. In: Galil, Z., Apostolico, A. (eds.) Combinatorial Algorithms on Words. Nato. Asi. Series, Advanced Science Institutes Series, Series F, Computer and Systems Sciences, vol. 12, pp. 1–8. Springer, Heidelberg (1995)
10. Galil, Z., Giancarlo, R.: Improved String Matching With k Mismatches. SIGACT News 17, 52–54 (1986)
11. Fisher, M.J., Paterson, M.S.: String Matching and Other Products. In: Karp, R.M. (ed.) SIAM–AMS Proceedings of Complexity of Computation, vol. 7, pp. 113–125 (1974)
12. Iliopoulos, C.S., Rahman, M.S.: A New Model to Solve the Swap Matching Problem and Efficient Algorithms for Short Patterns. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 316–327. Springer, Heidelberg (2008)
13. Karloff, H.: Fast Algorithms for Approximately Counting Mismatches. Information Processing Letters 48(2), 53–60 (1993)
14. Landau, G.M., Vishkin, U.: Efficient Parallel and Serial Approximate String Matching. Computer Science Department Technical Report #221, New York University (1986)
15. Landau, G.M., Vishkin, U.: Efficient String Matching With k Mismatches. Theoretical Computer Science 43, 239–249 (1986)
16. Muthukrishnan, S.: New Results and Open Problems Related to Non-Standard Stringology. In: Galil, Z., Ukkonen, E. (eds.) CPM 1995. LNCS, vol. 937, pp. 298–317. Springer, Heidelberg (1995)
17. Pentland, A.: Invited talk. NSF Institutional infrastructure Workshop (1992)
18. Supol, J., Melichar, B.: Suffix Automata and Parallel String Matching. In: London Algorithmics and Stringology (2007) (to appear)

Parallel Prefix Computation in the 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}@k.hosei.ac.jp
[2] Department of Computer Hardware
University of Aizu
Aizu-Wakamatsu 965-8580 Japan
w-chu@u-aizu.ac.jp

Abstract. In this paper, we propose an efficient algorithm for parallel prefix computation in recursive dual-net, a newly proposed network. The recursive dual-net $RDN^k(B)$ for $k > 0$ has $(2n_0)^{2^k}/2$ nodes and $d_0 + k$ links per node, where n_0 and d_0 are the number of nodes and the node-degree of the base network B, respectively. Assume that each node holds one data item, the communication and computation time complexities of the algorithm for parallel prefix computation in $RDN^k(B), k > 0$, are $2^{k+1} - 2 + 2^k * T_{comm}(0)$ and $2^{k+1} - 2 + 2^k * T_{comp}(0)$, respectively, where $T_{comm}(0)$ and $T_{comp}(0)$ are the communication and computation time complexities of the algorithm for parallel prefix computation in the base network B, respectively.

Keywords: Interconnection networks, algorithm, parallel prefix computation.

1 Introduction

In massively parallel processor (MPP), the interconnection network plays a crucial role in the issues such as communication performance, hardware cost, computational complexity, fault-tolerance. Much research has been reported in the literature on interconnection networks that can be used to connect parallel computers of large scale (see [1,2,3] for the review of the early work).

The following two categories have attracted a great research attention. One is the networks of hypercube-like family that have the advantage of short diameters for high-performance computing and efficient communication [4,5,6,7,8]. The other is the networks of 2D/3D meshes or tori that have the advantage of small and fixed node-degrees and easy implementations. Traditionally, most MPPs in the history including those built by NASA, CRAY, FGPS, IBM, use 2D/3D meshes or tori or their variations with extra diagonal links.

Recursive networks also have been proposed as effective interconnection networks for parallel computers of large scale. For example, the WK-recursive network [9,10] is a class of recursive scalable networks. It offers a high-degree of regularity, scalability, and symmetry and has a compact VLSI implementation.

Recently, due to advances in computer technology and competition among computer makers, computers containing hundreds of thousands of nodes have been built [11]. It was predicted that the MPPs of the next decade will contain 10 to 100 millions of nodes [12]. For such a parallel computer of very-large scale, the traditional interconnection networks may no longer satisfy the requirements for the high-performance computing or efficient communication.

For future generations of MPPs with millions of nodes, 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 bisection bandwidth, scalability, and efficient routing algorithms.

In this paper, we first describe a newly proposed network, called *Recursive Dual-Net* (RDN). The RDN is based on recursive dual-construction of a regular base-network. The dual-construction extends a regular network with n nodes and node-degree d to a network with $2n^2$ nodes and node-degree $d+1$. The RDN is especially suitable for the interconnection network of the parallel computers with millions of nodes. It can connect a huge number of nodes with just a small number of links per node and very short diameters. For example, a 2-level RDN with 5-ary, 2-cube as the base-network can connect more than 3-million nodes with only 6 links per node and its diameter equals to 22. The major contribution of this paper is to design efficient algorithm for parallel prefix computation in RDN.

The prefix computation is fundamental to most of numerical algorithms. Let $\oplus$ be an associative binary operation. Given n numbers $c_0, c_1, \ldots, c_{n-1}$, prefix computation is to compute all of the prefixes of the expression $c_0 \oplus c_1 \ldots \oplus c_{n-1}$.

The rest of this paper is organized as follows. Section 2 describes the recursive dual-net in details. Section 3 describes the proposed algorithm for parallel prefix computation in RDN. Section 4 concludes the paper and presents some future research directions.

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 $L(s,t)$ as the length of a shortest path connecting s and t. The diameter of G is defined as $D(G) = \max\{L(s,t) | s, t \in G\}$.

For any two nodes s and t in G, if there is a path connecting s and t, we say G is a connected graph. Suppose we have a symmetric connected graph B and there are n_0 nodes in B and the node degree is d_0. A k-level Recursive Dual-Net $RDN^k(B)$, also denoted as $RDN^k(B(n_0))$, 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).

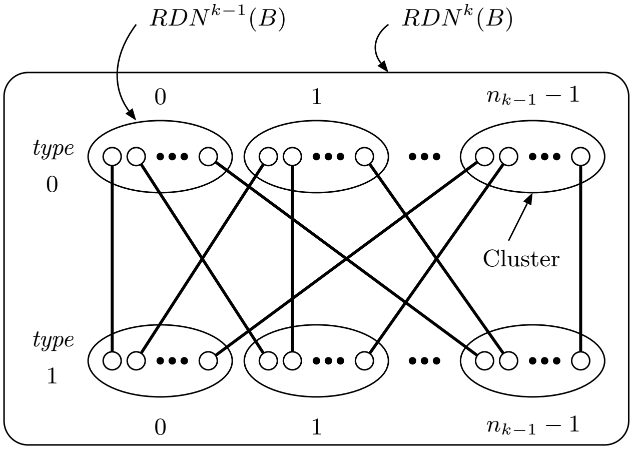


Fig. 1. Build an $RDN^k(B)$ from $RDN^{k-1}(B)$

Dual-construction: Let $RDN^{k-1}(B)$ be referred to as a *cluster* of level k and $n_{k-1} = |RDN^{k-1}(B)|$ for $k > 0$. 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. At level k, each node in a cluster has a new link to a node in a distinct cluster of the other type. We call this link *cross-edge* of level k. By following this rule, 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, $0 \le i, j \le n_{k-1} - 1$. In Figure 1, there are n_{k-1} nodes within each cluster $RDN^{k-1}(B)$.

We give two simple examples of recursive dual-nets with $k = 1$ and 2, in which the base network is a ring with 3 nodes, in Figure 2 and Figure 3, respectively. Figure 2 depicts an $RDN^1(B(3))$ network. There are 3 nodes in the base network. Therefore, the number of nodes in $RDN^1(B(3))$ is 2×3^2, or 18. The node degree is 3 and the diameter is 4.

Figure 3 shows the $RDN^2(B(3))$ constructed from the $RDN^1(B(3))$ in Figure 2. We did not show all the nodes in the figure. The number of nodes in $RDN^2(B(3))$ is 2×18^2, or 648. The node degree is 4 and the diameter is 10.

Similarly, we can construct an $RDN^3(B(3))$ containing 2×648^2, or 839,808 nodes with node degree of 5 and diameter of 22. In contrast, the 839,808-node

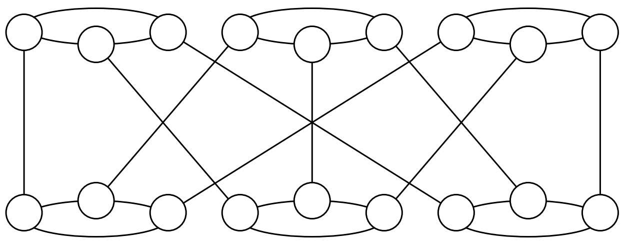


Fig. 2. A Recursive Dual-Net $RDN^1(B(3))$

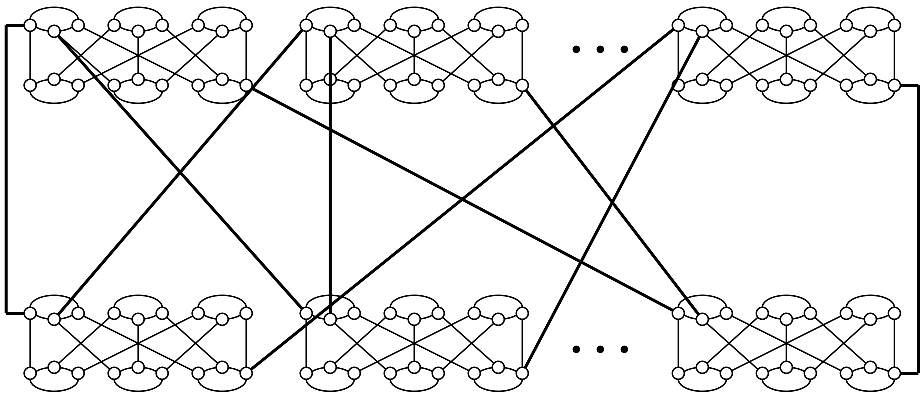


Fig. 3. A Recursive Dual-Net $RDN^2(B(3))$

3D torus machine (adopt by IBM Blue Gene/L [13]) configured as $108 \times 108 \times 72$ nodes, the diameter is equal to $54 + 54 + 36 = 144$ with a node degree of 6.

We can see from the recursive dual-construction described above that an $RDN^k(B)$ is a symmetric regular network with node-degree $d_0 + k$ if the base network is a symmetric regular network with node-degree d_0. The following theorem is from [14].

Theorem 1. *Assume that the base network B is a symmetric graph with size n_0, node-degree d_0, and the 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, $2^kD_0 + 2^{k+1} - 2$, and $\lceil (2n_0)^{2^k}/8 \rceil$, respectively.*

The *cost ratio* $CR(G)$ for measuring the combined effects of the hardware cost and the software efficiency of an interconnection network was also proposed in [14]. Let $|(G)|$, $d(G)$, and $D(G)$ be the number of nodes, the node-degree, and the diameter of G, respectively. We define $CR(G)$ as

$$CR(G) = (d(G) + D(G))/\lg |(G)|$$

The cost ratio of an n-cube is 2 regardless of its size. The CR for some $RDN^k(B)$ is shown in Table 1. Two small networks including 3-ary 3-cube and 5-ary 2-cube

Table 1. CR for some $RDN^k(B)$

Network	n	d	D	CR
10-cube	1,024	10	10	2.00
$RDN^1(B(25))$	1,250	5	10	1.46
$RDN^1(B(27))$	1,458	7	8	1.43
3D-Tori(10)	1,000	6	15	2.11
22-cube	4,194,304	22	22	2.00
$RDN^2(B(25))$	3,125,000	6	22	1.30
$RDN^2(B(27))$	4,251,528	8	18	1.18
3D-Tori(160)	4,096,000	6	240	11.20

are selected as practical base networks. For INs of size around 1K, we set $k = 1$, while for INs of size larger 1M, we set $k = 2$. The results show that the cost ratios of $RDN^k(B)$ are better than hypercubes and 3D-tori in all cases.

A presentation for $RDN^k(B)$ that provides an unique ID to each node in $RDN^k(B)$ is described as follows. Let the IDs of nodes in B, denoted as ID_0, be i, $0 \le i \le n_0 - 1$. 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. 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 $u_2 = v_1$. In general, ID_i, $1 \le i \le k$, can be defined recursively as follows: $ID_i = (b, ID_{i-1}, ID_{i-1})$, where $b = 0$ or 1. A presentation example is shown in Figure 4.

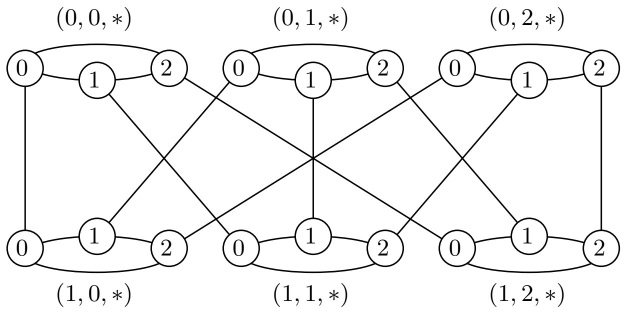


Fig. 4. $RDN^1(B(3))$ presentation

The ID of a node u in $RDN^k(B)$ can also be presented by an unique integer i, $0 \le i \le (2n_0)^{2^k}/2 - 1$, 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 $RDN^1(B(3))$ is $1 * 3^2 + 1 * 3 + 2 = 14$ (see figure 5). The ID of node (1,(0,2,2),(1,0,1)) in $RDN^2(B(3))$ is $1 * 18^2 + 8 * 18 + 10 = 324 + 144 + 10 = 478$.

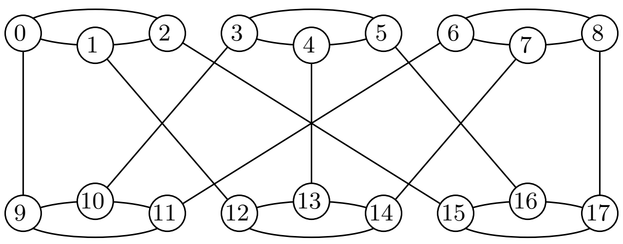


Fig. 5. $RDN^1(B(3))$ with integer node ID

3 Parallel Prefix Computation in Recursive Dual-Net

Let $\oplus$ be an associative binary operation. Given n numbers $c_0, c_1, \ldots, c_{n-1}$, parallel prefix computation [15,16] is defined as simultaneously evaluating all of the prefixes of the expression $c_0 \oplus c_1 \ldots \oplus c_{n-1}$. The ith prefix is $s_i = c_0 \oplus c_1 \ldots \oplus c_{i-1}$.

The parallel prefix computation can be done efficiently in recursive dual-net. Assume that each node i, $0 \leq i \leq n_k - 1$, in an $RDN^k(B)$ holds a number c_i. Let x_i and y_i are local variables in node i that will hold prefixes and total_sum at the end of the algorithm. The algorithm for parallel prefix (or diminished prefix which excludes c_i in s_i) computation in $RDN^k(B)$ is a recursive algorithm on k. We assume that the algorithm RDN_prefix(B, c, b) for prefix and diminished prefix computation in the base network ($b = 1$ for prefix and $b = 0$ for diminished prefix) is available. We describe it briefly below.

First, through a recursive call for every cluster of level k, we calculate the local prefix x_i and the local sum y_i in node i, where local prefix and local sum are the prefixes and the sum on the data items in each cluster of level k. To get the prefix of the data items in other clusters, we calculate the diminished prefix of all local sums of the clusters of the same type. This can be done by transferring the local sum to its neighbor via the cross-edge of level k, and then the prefix x'_i and the sum y'_i of all local sums of the same type can be computed by the nodes in every cluster of the other type via a recursive call.

After the second recursive call, the missing parts of the prefixes are ready for the nodes in clusters of type 0. Then, these values are transferred back to the nodes in the cluster of the original type via the cross-edge of level k and are added to its own local prefix. Finally, the algorithm adds the sum y'_i of data items in the nodes in clusters of type 0 to the current prefix of every node j in cluster of type 1. Notice that the value y'_i exists in every node j in the clusters of type 1 when the second recursive call is done.

The details are specified in Algorithm 1. Examples of prefix_sum in $RDN^1(B)$ and $RDN^2(B)$ are shown in Figure 6 and Figure 7, respectively.

Theorem 2. Assume 1-port, bidirectional-channel communication model. Assume also that each node holds a single data item. Parallel prefix computation in recursive dual-net $RDN^k(B), k > 0$, can be done in $2^{k+1} - 2 + 2^k * T_{comm}(0)$

Algorithm 1. RDN_prefix($RDN^k(B),\ c,\ b$)
Input: Recursive dual-net $RDN^k(B)$, an array of keys c with $|c| = n_k$, and a boolean variable b. Assume that node i holds c_i.
Output: node i holds $x_i = c_0 \oplus c_1 \ldots \oplus c[i]$ if $b = 1$, $c_0 \oplus c_1 \ldots \oplus c_{i-1}$ otherwise

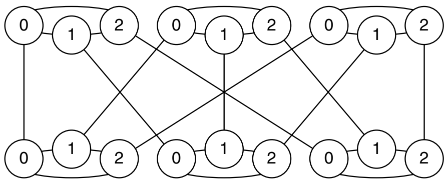
```
begin
    if $k = 0$ then RDN_prefix($B,\ c,\ b$)
    /* Assume that RDN_prefix($B,\ c,\ b$) is available. */
    else
        for $RDN_j^{k-1}(B),\ 0 \le j \le n_{k-1} - 1$, parallel do
        /* $j$ is the cluster ID. */
            RDN_prefix($RDN_j^{k-1}(B),\ c,\ b$);
            /* The values $x_i$ and $y_i$ at node $i$ are the local
                prefix and the local sum in the clusters of
                level $k$. */
        for node $i,\ 0 \le i \le n_k - 1$, parallel do
            send $y_i$ to node $i'$ via cross-edge of level $k$;
            $temp_i \leftarrow y_{i'}$;
        for $RDN_j^{k-1}(B),\ 0 \le j \le n_{k-1} - 1$, parallel do
            RDN_prefix($RDN_j^{k-1}(B),\ temp,\ 0$);
            /* Compute the diminished prefix of $temp$ */
            /* The results are denoted as $x'_i$ and $y'_i$. */
        for node $i,\ 0 \le i \le n_k - 1$, parallel do
            send $x'_i$ to node $i'$ via cross-edge of level $k$;
            $temp_i \leftarrow x'_{i'}$;
            $s_i \leftarrow s_i \oplus temp_i$;
        for node $i,\ n_k/2 \le i \le n_k - 1$, parallel do
            $s_i \leftarrow s_i \oplus y'_i$;
    endif
end
```

communication steps and $2^{k+1} - 2 + 2^k * T_{comp}(0)$ computation steps, where $T_{comm}(0)$ and $T_{comp}(0)$ are communication and computation steps for prefix computation in the base network, respectively.

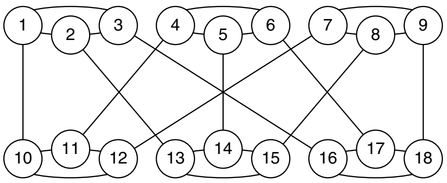
Proof. At Step 1, the local prefix in each cluster of level k is computed. At Steps 2 - 4, The part of the prefix located in other clusters of the same type is computed. Finally, at Step 5, for clusters of type 1, part of the prefix located in the clusters of type 0 is added to the nodes in the cluster of type 1. It is easy to see the correctness of the algorithm.

Next, we assume that the edges in $RDN^k(B)$ are bidirectional channels, and at each clock cycle, each node in D_n can send or get at most one message. In Algorithm 1, Step 1 and Step 3 are recursive calls, Step 2 and Step 4 involve one communication step each. Therefore, the complexity for communication satisfies recurrence $T_{comm}(k) = 2T_{comm}(k-1) + 2$. Solving the recurrences, we get $T_{comm}(k) = 2^{k+1} - 2 + 2^k * T_{comm}(0)$. Similarly, Steps 4 and 5 involve one computation step each. The recurrence for computation time satisfies the same concurrence.

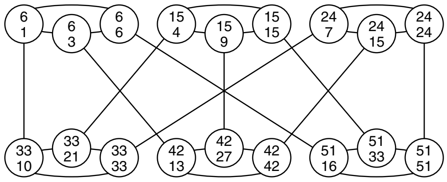
```
prefix_sum (1,2,3, 4, 5, 6, 7, 8, 9,10,11,12,13, 14, 15, 16, 17, 18)
         = (1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171)
```



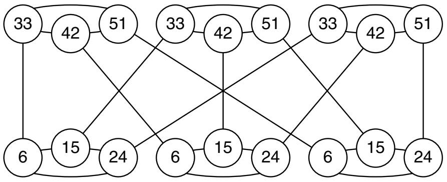
(a) Presentation of $RDN^1(B(3))$



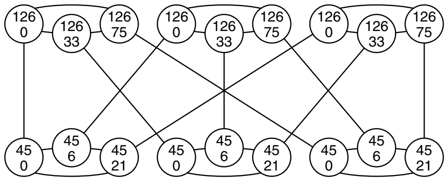
(b) Data distribution



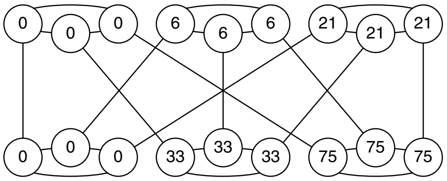
(c) $t[u]$ (top) and $s[u]$ (bottom)



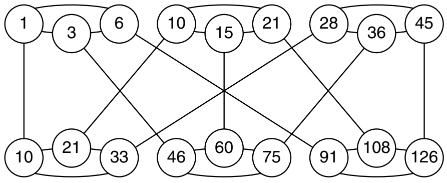
(d) Send $t[u]$ and receive $temp[u]$



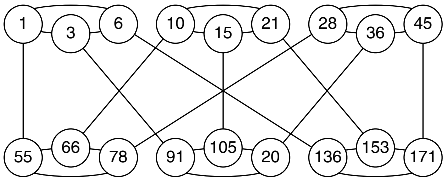
(e) $t'[u]$ (top) and $s'[u]$ (bottom)



(f) Send $s'[u]$ and receive $temp[u]$



(g) $s[u] \leftarrow s[u] \oplus temp[u]$

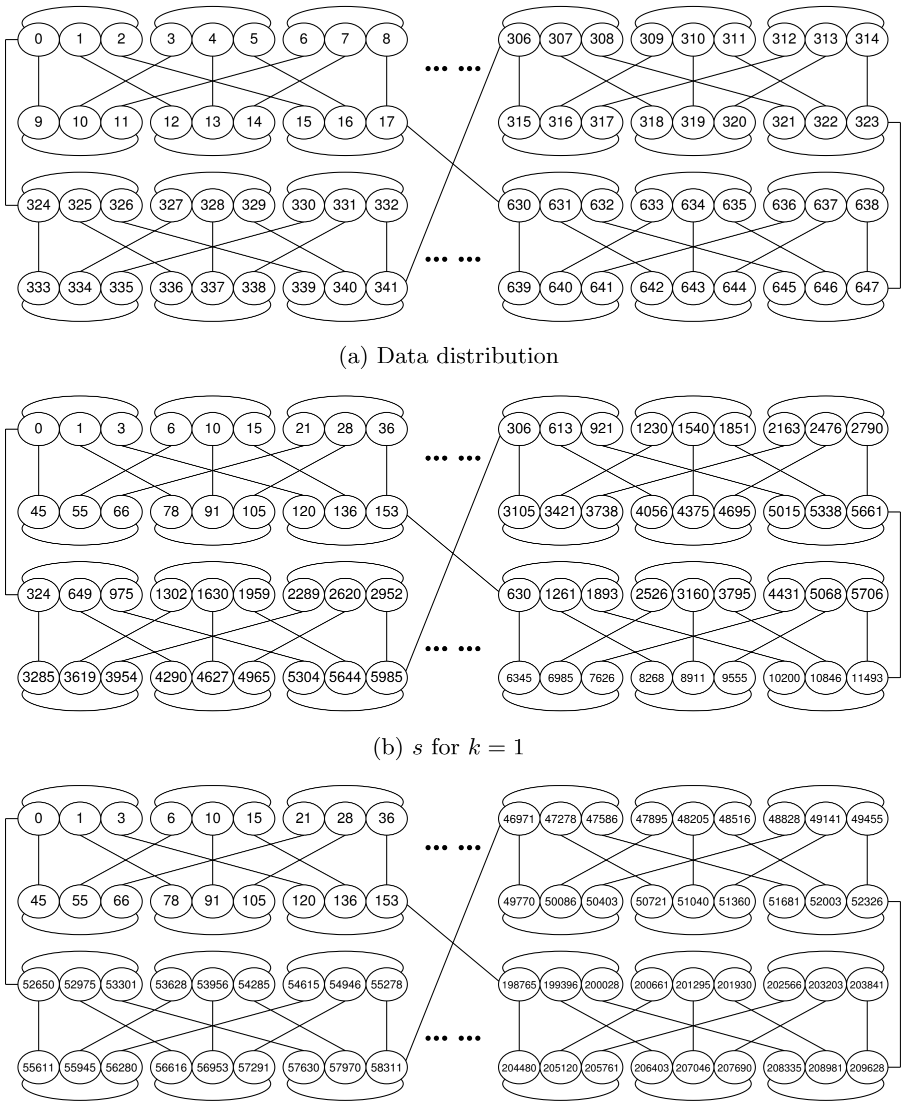


(h) In type 1, $s[u] \leftarrow s[u] \oplus t'[u]$ (final result)

Fig. 6. Example of prefix_sum in $RDN^1(B(3))$

Therefore, we conclude that the prefix computation in $RDN^k(B)$ for $k > 0$ can be done in $2^{k+1} - 2 + 2^k * T_{comm}(0)$ communication steps and $2^{k+1} + 2^k * T_{comp}(0)$ computation steps, where $T_{comm}(0)$ and $T_{comp}(0)$ are communication and computation steps for prefix computation in the base network, respectively. □

Extension of the parallel prefix algorithm to the general case where each node initially holds more than one data item is straightforward. Let the size of array



(a) Data distribution

(b) s for $k = 1$

(c) s for $k = 2$ (final prefix result)

Fig. 7. Example of prefix_sum in $RDN^2(B(3))$

c be $m > n$. The algorithm consists of three stages: In the first stage, each node do a prefix computation on its own data set of size m/n sequentially; In the second stage, the algorithm performs a diminished parallel computation on the RDN as describe in Algorithm 1 with $b = 0$ and c_i equals to the local sum; In third stage, for each node, the algorithm combines the result from this last computation with the locally computed prefixes to get the final result. We show the parallel prefix computation for the general case in theorem 3.

Theorem 3. Assume 1-port, bidirectional-channel communication model. Assume also that the size of the input array is m, and each node holds m/n_k numbers. Parallel prefix computation in recursive dual-net $RDN^k(B), k > 0$, can be done in $2^{k+1} - 2 + 2^k * T_{comm}(0)$ communication steps and $2m/n_k + 2^{k+1} - 3 + 2^k * T_{comp}(0)$ computation steps, where $T_{comm}(0)$ and $T_{comp}(0)$ are communication and computation steps for prefix computation in the base network with each node holds one single number, respectively.

Proof. The first and the third stages of the algorithm contains only local computations inside each node and the total number of computations are $(m/n_k) - 1$ and m/n_k, respectively. In the second stage, the algorithm performs parallel prefix computation in RDN with each node holding a single number. Following Theorem 1, it requires $2^{k+1} - 2 + 2^k * T_{comm}(0)$ communication steps and $2^{k+1} - 2 + 2^k * T_{comp}(0)$ computation steps. Therefore, we conclude that the parallel prefix computation of array of size $m > n_k$ in $RDN^k(B)$ requires $2^{k+1} - 2 + 2^k * T_{comm}(0)$ communication steps and $(2m/n_k + 2^{k+1} - 3) + 2^k * T_{comp}(0)$ computation steps. □

4 Concluding Remarks

In this paper, we showed an efficient algorithm for parallel prefix computation in recursive dual-net. Recursive dual-net is a potential candidate for the supercomputers of next generations. It has many interesting properties that are very attractive as an interconnection network of massively parallel computer. Design efficient algorithms for basic computational problems in an interconnection network is an important issue. The further research will include design of efficient searching and sorting algorithms for recursive dual-net.

References

1. Aki, S.G.: Parallel Computation: Models and Methods. Prentice-Hall, Englewood Cliffs (1997)
2. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, San Francisco (1992)
3. Varma, A., Raghavendra, C.S.: Interconnection Networks for Multiprocessors and Multicomputers: Theory and Practice. IEEE Computer Society Press, Los Alamitos (1994)
4. Ghose, K., Desai, K.R.: Hierarchical cubic networks. IEEE Transactions on Parallel and Distributed Systems 6, 427–435 (1995)
5. 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 (2000)
6. Li, Y., Peng, S., Chu, W.: Efficient collective communications in dual-cube. The Journal of Supercomputing 28, 71–90 (2004)
7. Preparata, F.P., Vuillemin, J.: The cube-connected cycles: a versatile network for parallel computation. Commun. ACM 24, 300–309 (1981)

8. Saad, Y., Schultz, M.H.: Topological properties of hypercubes. IEEE Transactions on Computers 37, 867–872 (1988)
9. Chen, G.H., Duh, D.R.: Topological properties, communication, and computation on wk-recursive networks. Networks 24, 303–317 (1994)
10. Vicchia, G., Sanges, C.: A recursively scalable network vlsi implementation. Future Generation Computer Systems 4, 235–243 (1988)
11. TOP500: Supercomputer Sites (2008), `http://top500.org/`
12. 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 (2008)
13. 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, 265–276 (2005), `http://www.research.ibm.com/journal/rd/492/tocpdf.html`
14. 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)
15. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison-Wesley, Reading (2003)
16. Hillis, W.D., Steele Jr., G.L.: Data parallel algorithms. Communications of the ACM 29, 1170–1183 (1986)

A Two-Phase Differential Synchronization Algorithm for Remote Files

Yonghong Sheng[1], Dan Xu[2], and Dongsheng Wang[1,3]

[1] Department of Computer Science and Technology, Tsinghua University, Beijing, P.R. China
shengyh04@mails.tsinghua.edu.cn

[2] School of Computer Science and Technology, Beijing University of Posts and Telecommunications, Beijing, P.R. China
xdbupt08@gmail.com

[3] Tsinghua National Laboratory for Information Science and Technology, Beijing, P.R. China
wds@tsinghua.edu.cn

Abstract. This paper presents a two-phase synchronization algorithm—*tpsync*, which combines content-defined chunking (CDC) with sliding block duplicated data detection methods. *tpsync* firstly partitions synchronized files into variable-sized chunks in coarse-grained scale with CDC method, locates the unmatched chunks of synchronized files using the edit distance algorithm, and finally generates the fine-grained delta data with fixed-sized sliding block duplicated data detection method. At the first-phase, *tpsync* can quickly locate the partial changed chunks between two files through similar files' fingerprint characteristics. On the basis of the first phase's results, small fixed-sized sliding block duplicated data detection method can produce better fine-grained delta data between the corresponding unmatched data chunks further. Extensive experiments on ASCII, binary and database files demonstrate that *tpsync* can achieve a higher performance on synchronization time and total transferred data compared to traditional fixed-sized sliding block method—*rsync*. Compared to *r*sync, *tpsync* reduces synchronization time by 12% and bandwidth by 18.9% on average if optimized parameters are applied on both. With signature cached synchronization method adopted, *tpsync* can yield a better performance.

1 Introduction

Delta compression and file synchronization technology are widely used in network data backup, software distribution, web application, P2P storage and mobile computing, etc. The key issue of file synchronization is how to update the file by using less time and with a minimal amount of network overhead between the source-end and the destination-end. Under low bandwidth and flow-sensitive network environment, such issues are especially important. Synchronization technology based on differential calculation and compression can satisfy the demands of remote file synchronization. Taking a file of 10 MB which has been modified only 16 KB as an instance, the process of synchronization can be done within 3 seconds if only the differential data is transferred, while it requires about 25 minutes for a full copy.

Researchers have already introduced a lot of algorithms for differential calculation and remote file synchronization. Delta compression algorithms, such as diff [1], [2], [3], vcdiff [4], xdelta [5], BSDiff [6], and zdelta [7], may be used to produce a succinct description of the differences between two similar files if both the old version and the new version of the file are locally available to the sender. However, in many cases, especially in distributed network environment, this assumption may be overly restrictive, since it is difficult or infeasible to know the copies of files other nodes hold. To synchronize remote files over network, researchers have put forward a lot of fast synchronization algorithms, such as Andrew Tidwell's *rsync* [8] algorithm, Microsoft's ActiveSync [9], Puma Technologies' IntelliSync [10] and Palm's HotSync [11]. *rsync* is one of the most famous and open source software. It is quite simple and clear, but is sensitive to the changing pattern of synchronized files and the block size parameter. Under the worst case, *rsync* needs to transfer the whole file even if the two files have much duplicated data, detailed analysis can been seen in Section 2.2. ActiveSync, IntelliSync and HotSync are efficient two-way synchronization tools between PC and Palm OS, Pocket PC and etc, but they are not suitable for synchronizing remote files. In order to save bandwidth under low-bandwidth network environment, LBFS [12] exploits similarities among all files on both the server and the client, and avoids sending the data which can already be found in the server's file system or the client's cache over the limited-bandwidth network. Based on LBFS, DFSR [13] uses three optimizing methods to achieve far smaller differential data amount transporting, which include local file similarity analyzing, large file recursive signature transporting and more precisely file chunking. But LBFS and DFSR both need to build index on local machine, bringing extra consumption on computing and storage resources, and the transport protocol is too complex. CVS [14] and SVN [15], the famous file version control tools, implement fast differential calculation by storing the complete old versions of files in client. They are suitable for the version control of ACSII file, such as source code files, but not efficient for binary files.

To satisfy the twofold demands of fast differential calculating as well as reducing the overhead over network, this paper proposes a two-phase synchronization algorithm—*tpsync*. In the first-phase, *tpsync* divides files into chunks based on content and calculates the chunks' signatures simultaneously at the source and the destination. Then *tpsync* locates unmatched chunks and calculates their differential data by fixed-sized sliding block duplicated data detection method, which is called the second-phase, drawing on the basic idea of *rsync*. *tpsync* can both quickly locate the changing range in files and generate much fine-grained delta data between unmatched chunks. Extensive experiments on different workloads shows that *tpsync* can both speed up synchronization and reduce overhead on network.

The rest of this paper is organized as follows: Section 2 presents the design and implementation in detail. Section 3 gives experimental evaluation and results. Section 4 concludes this paper with a summary of our contributions and future work.

2 Two-Phase Differential Synchronization Algorithm

File-level and block-level are two main granularities in duplicated data detection. WFD (whole file detecting) is a duplicated data detection method in file-level. WFD computes the hash value of the whole file, and then compares this value to the stored hash values. If equivalence is detected, then WFD assigns the file a pointer to the unique file, instead of allocating new space for it as for new files. The fine-grained duplicated data detection technology, usually at block level, can search the same blocks in a file, and store these blocks within only one copy. It usually uses FSP (fixed-sized partition) [16], VSP (variable-sized partition) [17], [18] or sliding block [19] duplicated data detection methods to detect and eliminate redundancies. FSP strategy employs a fixed block size to partition files into chunks, independent of the content of the files being processed. It is used in content-addressable systems such as Venti [20] and Oceanstore [21]. FSP is sensitive to the sequence of consecutive versions file's edition and change operation. Hence, this technology is not popular in practice. In order to overcome this limitation, researchers have put forward content-defined chunking technology, such Rabin's fingerprint [22], to divide files into variable-sized chunks [23].

2.1 CDC Based Duplicated Data Detection

The boundaries of variable-sized chunks are determined by Rabin's fingerprint algorithm. File's contents are seen through a fixed-sized (overlapping) sliding window from the beginning. At every position of the window, a fingerprint or signature of its contents, f, is computed using hash techniques such as Rabin's fingerprints. When the fingerprint meets a certain criteria, such as f mod D = r, where D is the divisor, and r is the predefined magic number(r<D), that position of the window defines the boundary of the chunk. This process is repeated until the complete file has been broken down into chunks. Figure 1 depicts Rabin's fingerprint chunking process. MD5, SHA-1 or higher SHA standard functions are used to compute each chunk's signature. Such signature is queried in index file to determine duplicated chunks. New chunks are written into disk and the index is updated with their chunks' IDs and signatures. Contend-defined chunking method can ensure changes made to consecutive versions are localized to a few chunks around the region of change.

2.2 Sliding Block Based Duplicated Data Detection

A sliding-block duplicated data detection approach is used in protocols like *rsync*, which is illustrated in Figure 3. The basic idea of *rsync* can be described as follows: First, the recipient, B, breaks its file F1 into non-overlapping, contiguous, fixed-sized blocks and transmits hashes of those blocks to A. In return, A begins to compute the hashes of all (overlapping) blocks of F2. If any of those hashes matches one from F1, A avoids sending the corresponding sections of F1, but telling B where to find the data in F1 instead.

According to the features of sliding block duplicated data detection technology, we know that it is sensitive to the block size parameter and the changing pattern of the file. In extreme conditions, even the changes are very slight, the whole file need to be retransferred. As the example shown in Figure 2, if the window size is set to 4, the whole version 2 would be retransmitted as no duplicated data detected. In fact, the duplicated data ratio is up to 75% between version 1 and 2.

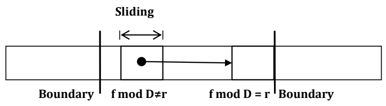


Fig. 1. Rabin's Fingerprint CDC Chunking Process

Version 1	ABCD	AXZD	BHFG	CNPZ
Version 2	ABC**K**	AXZ**S**	BHF**J**	CNP**Q**

Fig. 2. An Example for *rsync*'s Sensitivity to Block Size Parameter

If we set the sliding window size to 3 in above example, the only transferring data are (K, S, J, Q) and some extra indices information, which refers to the 'differential data' in *rsync*. This is also the reason why we choose smaller sliding window size when using sliding block duplicated data detection method to calculate the differential data in the second-phase.

2.3 Detailed Description of *tpsync*

This paper proposes *tpsync*, which combines the benefits of both Section 2.1 and 2.2. In large scale, *tpsync* exploits the similarity of fingerprint distribution in similar files, and also makes use of the advantage in computing the minor differential data by *librsync* between the corresponding unmatched chunks. As is shown in Figure 4, the second version has been modified upon the first by inserting a few bytes at a region near the beginning of the file. The sub-chunks are computed using a CDC implementation that identifies chunk boundaries by computing Rabin's fingerprints on a sliding window. Supposing the relative changes occurred in C_2 and C_2', then *librsync* is called to synchronize the differential between C_2 and C_2'.

The edit distance algorithm [24] is introduced in *tpsync* to locate unmatched chunks after files are partitioned by CDC. Given two strings A[1,m] and B[1,n], the edit distance of string A and B refers to the least edit operations in changing A to B. There are three common edit operations: insert, delete and replace. To find unmatched chunks with edit distance, *tpsync* first converts the two version files' signatures into two strings, then calculates the shortest transfer path by edit distance algorithm. Since the distributions of the files' fingerprints are

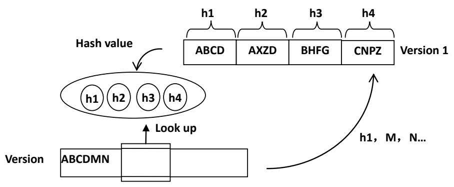


Fig. 3. *rsync* Algorithm on a Small Example

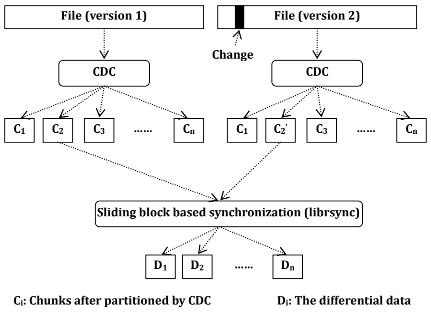

File (version 1)
File (version 2)
Change
CDC
CDC
C1 C2 C3 ...... Cn
C1 C2' C3 ...... Cn
Sliding block based synchronization (librsync)
D1 D2 ...... Dn
Ci: Chunks after partitioned by CDC
Di: The differential data

Fig. 4. An Example of *tpsync* Synchronization Process

basically similar after split, the strings transferred from signatures are basically similar. The edit distance algorithm is suitable for the transport of two similar strings, so it could also be applied in the second-phase of *tpsync*, detecting the unmatched chunks between the two files with a high degree of similarity. It is different from the traditional duplicated data detection technology, in which the index of signatures is required and the duplicated data is detected through the index.

Open source *librsync* module is used in *tpsync* to synchronize pairs of corresponding unmatched chunks in the second-phase. For one specified pair, *librsync* generates a delta file including literals and indices. In relation to the synchronized files, *librsync* may produce many delta files. Just as Figure 4 shows, D_1, D_2, ..., D_n are delta files generated by *librsync* with synchronizing two versions of files.

2.4 *tpsync* Algorithm Details

The tpsync algorithm details are shown in Figure 5.

2.5 Signature Cached Synchronization Method

Regardless of in *rsync* or *tpsync*, signature values on one end need to be sent to the other end. If storage space permitted, the signature of the older version can be stored in local machine. In this way, the differential data can be directly calculated with the signatures of the old version and the new version. The differences between the method of signature cached synchronization (SCS) and that of traditional one are shown in Figure 6.

As shown in Figure 6(a), in signature cached synchronization, the signature of the version 2, SIG(V_2), would be cached locally after synchronizing the version 2 to server. When the version 3 is synchronized to the server, what should be done is only to diff the version 3 and SIG(V_2) to calculate the delta data($\Delta 2$), and then send the delta data to the server. On the other hand, in the traditional method shown in the Figure 6(b), the differential data cannot be calculated

Algorithm 1: the first-phase: partition files with CDC

```
1: First_Phase_of_tpsync(basisfile, cdc_exp_chunk_size)
   Input: basisfile, old File; cdc_exp_chunk_size; expected chunk size of CDC
   Output: h_basis, List of basisfile chunks;
2: begin
3: List h_basis: =empty;
4: chunkMask: =calculateMask(cdc_exp_chunk_ size);
5: firstpos: =0;
6: foreach byte position X in basisfile do
7:      window: =substring(basisfile, X, substring_size);
8:    fp: =fingerprint(window);
9:    if fp % chunkMask==magic_value then
10:       chunk: =substring(f, firstpos, X-firstpos);
11:       hashvalue: =hash(chunk);
12:       h_basis.add(firstpos, X-firstpos, hashvalue);
13:       firstpos: =X;
14:   end if
15: end foreach
16: end
```

Algorithm 2: the second-phase: locates unmatched chunks and sync them with *librsync*

```
1: Second_Phase_of_tpsync(h_basis, h_newfile, bs)
   Input: h_basis, List of basisfile chunks; h_newfile, List of newfile chunks; bs, parameter of block size set in librsync
   Output: LD, list of delta data
2: begin
   //Calculate the edit distance between h_basis and h_newfile
3: m=h_basis.size(); n=h_newfile.size();
4: for (i=1; i<=m; i++) do
5:    for (j=1; j<=n; j++) do
6:       cost=(h_basis[m].hashvalue== h_newfile[n].hashvalue)?0:1;
7:       Distance[i][j]=min(D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+cost);
8:    end for
9: end for
   //Traverse the operations of insertion, deletion, replacement and copy according to the edit distance.
10: step=distance[m][n];
11: while (step!=0) do
12:    switch(step)
13:       case 1: // replace or copy
14:          if (step!=distance[m-1][n-1]) then
15:             delta=librsync(h_basis[m],h_newfile[n]);
16:             LD.add(delta);
17:          m−−; n−−;
18:          break;
19:       case 2: // insert
20:          LD.add (h_basis[m]);
21:            m−−;
22:          break;
23:       case 3: // delete
24:          Delete(h_basis[m]);
25:            n−−;
26:          step=distance[m][n];
27: end while
28: end
```

Fig. 5. *tpsync* Algorithm Details

until the signatures of version 2, SIG(V_2), have been received. The storage of the signatures of the old version needs to take some extra storage space. Taking *rsync* as an example, for a file of 100 MB, if the block size is 700 bytes and the signature is 20 bytes (128 bits strong checksum and 32 bits rolling checksum), it would consume extra 2.86 MB storage. *tpsync* only needs to store the signatures of the first-phase. If the expected chunk size of CDC in the first-phase is 2048 bytes, and it also takes 128 bits hash signature, then it will consume about 0.78 MB storage space.

3 Experimental Evaluation

3.1 Experimental Environment

We have implemented a prototype of *tpsync* in C++. The first-phase algorithm adopts CDC implementation of LBFS and the second-phase algorithm uses open source librsync [25] module. Our experiments were conducted on two Pentium(R) Dual-core E5200 2.50 GHZ computers with 4 GB DDR2 memory and 500GB Seagate ST3500620AS hard disks, running Ubuntu 9.10 connected by full-duplex 1Gps Ethernet. We use three different kinds of workloads, including ASCII text, binary and database files, to compare the performance of *tpsync* and *rsync*. ASCII text workloads include versions of gcc (4.4.2 and 4.3.4) and emacs (22.3 and 23.1). Another type of data widely upgraded and replicated is binary files. Binary files have different characteristics compared to ASCII files. We chose the entire contents of usr/lib directory in Ubuntu 9.04 and Ubuntu 9.10 as binary files. The database workload contains two snapshots, including data and log file, generated by Benchmark Factory 5.0 based on TPC-C [26]. Table 1 details the different workload characteristics giving the total uncompressed size and the number of files.

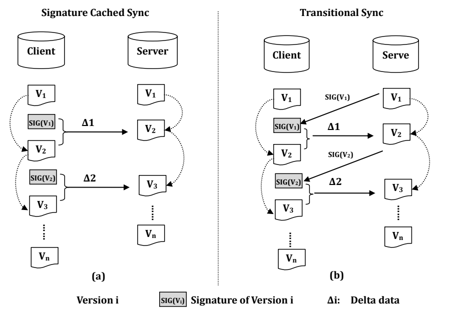


Fig. 6. Difference between Signature Cached Synchronization(SCS) and Traditional Synchronization Method

Table 1. Characteristic of the Different Workloads

Data Sets	Versions	File Numbers	Total Sizes(KBytes)
workload1	gcc(4.4.2)	62,298	383,423
	gcc(4.3.4)	58,212	364,173
workload2	emacs(23.1)	3907	146,746
	emacs(22.3)	3,523	136,046
workload3	usr/lib(9.10)	13,919	936,910
	usr/lib(9.04)	10,360	739,226
workload4	database(v1)	2	221,636
	database(v2)	2	221,636

Each type of workload has its own size distribution of unique files. Figure 7 shows the frequency histogram and the cumulative histogram of the unique file sizes of gcc 4.4.2, gcc 4.3.4, usr/lib (Ubuntu 9.04) and usr/lib (Ubuntu 9.10). Figure 7 indicates most unique files of gcc have sizes ranging from 256 bytes to 8 KB (2^8 to 2^{13} bytes) and most files are around 4K bytes, while usr/lib workload has low characteristic of central distribution. Figure 7 also shows that different versions of the same workload have similar unique file size distribution.

3.2 Results of *rsync* Synchronization

We first use *rsync* to synchronize different workloads. *rsync* uses a default block size of 700 bytes except for very large files where a block size of $\sqrt{n}$ is used, in

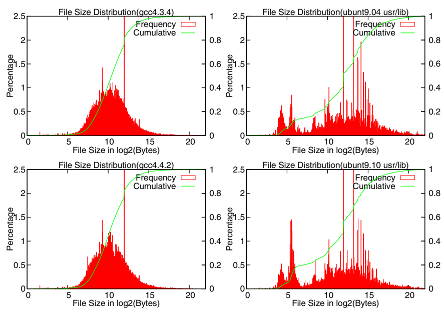


Fig. 7. Unique File Size Distributions of Different Workloads

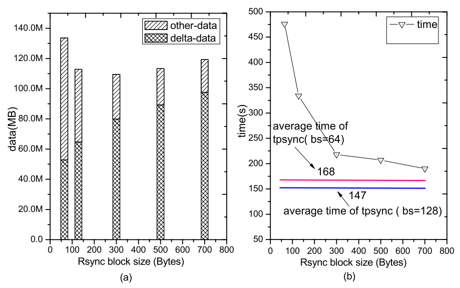


Fig. 8. Data Transmission and Synchronization Time of *rsync* on gcc

which n is the size of file in bytes. Figure 8 shows the synchronization results of *rsync* on gcc with default and pre-set parameters.

For *rsync* synchronization, we have carried out experiments for different block sizes: 64, 128, 300, 500 and 700 bytes respectively. The hit ratio is the proportion of matched data to the total file size; the differential data is produced by *librsync* module, including literal data and indices; extra communication data refers to the transmission data over network other than differential data.

Table 2 and Figure 8 show that the sensitive factor of *rsync* is the block size parameter. The smaller the block size is, the higher the hit ratio and the smaller the differential data it generates, but the amount of extra network communication data and the synchronization time increases. Taking gcc as an example, the block size of 64 bytes gives the hit ratio of 52.8% and the differential data size of 51,633 KB, which is just 54% of that of the default block size, 700 bytes. However, in this case, the data caused by extra network communication is 3.72 times and the synchronization time is about 2.5 times than that of the block size is chosen to the default value in *rsync*. Thus, although smaller block size can bring higher hit ratio and smaller differential data, it would also cause the increase of numbers of blocks because of smaller size of each block. Thus, the extra overhead will increase and the synchronization time will not decrease. The growth of synchronization time mainly comes from the increasing time of indexing, searching and matching of the signatures.

To text files, such as gcc and emacs, we can enhance the hit ratio and reduce the differential data by reducing the size of blocks. However, to binary files, like usr/lib, the hit ratio would not increase obviously when reducing the block size, while the synchronization time increases dramatically. Compared to default block size of 700 bytes, the hit ratio increased by 1.39%, while the synchronization time prolonged 3 times when the block size is set to 64 bytes. To database data set, the hit ratio would not increase more obviously, while the synchronization time increases drastically when the block size is reduced. Compared to default size, the hit ratio increased little, while the synchronization time prolonged nearly 10 times when the block size is set to 64 bytes, which has a higher growth than binary files.

Table 2. Results of *rsync* Synchronization on Different Workloads

Data Sets	Blocksize (Bytes)	Hit ratio	T-data(KB)		Total data (KBytes)	Sync time (s)
			d-data	e-data		
gcc4.4.2 / gcc4.3.4	64	52.8%	51,633	78,835	130,469	476
	128	43.1%	63,185	47,088	110,273	334
	300	45.6%	77,984	28,889	106,873	218
	500	43.0%	87.185	23,479	110,664	207
	700	40.8%	95,354	21,156	116,510	190
usr/lib(9.10) / usr/lib(9.04)	64	10.6%	466,689	167,067	633,755	260
	128	9.7%	488,083	86,564	574,647	161
	300	8.6%	510,449	40,412	550,861	104
	500	8.0%	523,935	26,671	550,606	93
	700	7.6%	533,437	20,783	554,220	89
database(v1) / database(v2)	64	84.7%	35,279	41,560	76,839	355
	128	83.6%	37,046	20,782	57,828	300
	300	81.5%	41,408	8,869	49,917	79
	500	80.0%	44,450	5,323	49,773	49
	700	78.7%	47,417	3,803	51,220	37

According to the result of the three types of workloads, we can see that, in *rsync*, if the block size is smaller, the delta data would be reduced, but the extra communication data and the calculation overhead will increase, which leads to the prolong of synchronization time, especially for binary files and databases.

3.3 Results of *tpsync* Synchronization

tpsync partitions the file into chunks with the method of CDC in the first-phase. CDC is tied to three parameters: exp_sz (expected_chunk_size), max_sz (maximum_chunk_size) and min_sz (minimum_chunk_size). In the following experiments, min_sz is set to the half of exp_sz while max_sz 2 times of exp_sz. In the second-phase, the fixed-sized sliding window is applied. Let bs denote the length of such fixed-sized sliding window. Since small bs value leads to fine-grained matching and generates small differential data, two bs values were selected: 64 and 128 bytes respectively in our *tpsync* experiments. Table 3 and Table 4 show the results of the synchronization tests on gcc, emacs and usr/lib under different exp_sz values. Table 5 gives out the results of synchronization of database when exp_sz was fixed, while bs varied from 64 bytes to 700 bytes.

Table 3. Result of Synchronization with *tpsync* (bs=64 bytes)

Workload	Parameter (exp_sz,bs)	Ph1-data (KBytes)	Ph2-data(KB)		Total data (KBytes)	Sync time (s)
			d-data	e-data		
gcc4.4.2 / gcc4.3.4	(2048,64)	2,993	66,823	41,671	111,487	177
	(4096,64)	2,231	66,788	49,265	118,285	175
	(6144,64)	1,991	66,525	52,872	121,388	171
	(8192,64)	1,879	66,673	55,797	124,350	156
usr/lib(9.10) / usr/lib(9.04)	(2048,64)	1,861	527,777	70,196	599,834	84
	(4096,64)	1,036	527,516	72,284	600,835	66
	(6144,64)	772	527,445	73,272	601,489	63
	(8192,64)	632	527,531	74,436	602,600	62

Table 4. Result of Synchronization with *tpsync* (bs=128 bytes)

Workload	Parameter (exp_sz,bs)	Ph1-data (KBytes)	Ph2-data(KB) d-data	Ph2-data(KB) e-data	Total data (KBytes)	Sync time (s)
gcc4.4.2 / gcc4.3.4	(2048,64)	2,993	78,628	20,922	102,543	154
	(4096,64)	2,231	78,608	24,706	105,546	154
	(6144,64)	1,991	78,343	26,504	106,839	140
	(8192,64)	1,879	78,396	27,964	108,240	138
usr/lib(9.10) / usr/lib(9.04)	(2048,64)	1,861	534,727	35,122	571,710	60
	(4096,64)	1,036	534,498	36,162	571,695	59
	(6144,64)	772	534,401	36,656	571,829	58
	(8192,64)	632	534,514	37,236	572,383	56

According to the analysis of the result of *tpsync*'s experiments on gcc, we have observed:

(1) when bs is fixed, different exp_sz values have slight effects on synchronization time. For instance, if bs is fixed at 64 bytes, the synchronization time ranges slightly from 156 seconds to 177 seconds with different exp_sz values.
(2) *tpsync* consists of three parts of data over the network: the communicating data of CDC in the first-phase (Ph1-data); the differential data in the second-phase (d-data) and the communication data other than the differential data in the second-phase (Ph2-data). When bs is fixed, if the exp_sz value increases, the Ph1-data would decline, the Ph2-data would increase, the e-data would stay the same and the total amount of data(Total data) transferred over network would increase, but slightly.
(3) When bs is 128 bytes, the synchronization time reduces by 13.6% compared to the condition when bs is 64 bytes, and the amount of data transferred over network reduces by 11%.

For *tpsync*, the synchronization time consists of five parts: the time of calculation of CDC in the first-phase (cdc-time), the time of calculating differential data in the second-phase (delta-time), the differential data patching time (patch-time), the copying time (copy-time) and other extra consuming time (other). The copy-time refers to the time consumed by directly copying files, which do not exist in the other side, from one side to the other. Figure 9 shows different parts of synchronization time of *tpsync* when bs are 64 and 128 bytes respectively. According to Figure 9, we can see the average cdc-time for gcc with different exp_size parameter makes up 37.8% of the total synchronization time. If such time can be reduced, the efficiency of *tpsync* can improve further. The method of signature cached synchronization this paper proposed aims for reducing the cdc-time, through fewer extra storage in exchange for a substantial reduction in synchronization time.

Compared to *rsync*, *tpsync* has less synchronization time, when the same bs lengths are selected in both algorithm. Taking gcc as an example, when bs is 64 bytes, synchronization time of *rsync* is 476 seconds, while that of *tpsync* is 177 seconds, which is the worst case for *tpsync* among all bs values; when bs is 128 bytes, *rsync*'s synchronization time is 344 seconds, while *tpsync*'s synchronization time is 154 seconds. As shown in Figure 8(b), *tpsync*'s average

Table 5. Result of Database Workload Synchronization with *tpsync* (exp_sz=8192 Bytes)

Workload	Parameter	Ph1-data	Ph2-data(KB)		Total data	Sync time
	(exp_sz,bs)	(KBytes)	d-data	e-data	(KBytes)	(s)
databasa(v1) / database(v2)	(8192,64)	553	34,983	26,959	62,494	32
	(8192,128)	553	36,980	13,482	51,016	29
	(8192,300)	553	41,750	5,757	48,060	24
	(8192,500)	553	45,106	3,456	49,116	23
	(8192,700)	553	48,138	2,471	51,162	23

synchronization times are 169 seconds and 190 seconds when bs are chosen to 64 bytes and 128 bytes respectively. They are both less than *rsync*'s best synchronization time, 190 seconds, when bs is set 700 bytes in *rsync*.

When bs is 64 bytes, the amount of data over network of *rsync* is 130 MB, while that of *tpsync* is 124 MB in the worst case (exp_sz=8192 bytes), as is shown in Figure 10 and Table 2, when bs is 128 bytes, the amount of data over network of *rsync* is 110 MB, while that of *tpsync* is 108 MB in the worst case (exp_sz=8192 bytes). Meanwhile, the maximum amount of data transmission over network is approximately 108MB when bs is 128 bytes in *tpsync*, which is less than *rsync*'s least transmission data 117 MB when bs is 700 bytes. Obviously, when the amounts of data over network are nearly the same, *tpsync* has a higher performance in synchronization time than *rsync*.

Results of *rsync*'s experiments also show that when *rsync* chooses the default block size of 700 bytes, its overall performance is the best, which has also been validated in paper [8]. Results of *tpsync*'s experiment show that when exp_sz is 2048 bytes and bs is 128 bytes, its overall performance is the best. Under this case, *tpsync* has less synchronization time by 12% and bandwidth savings over network by 18.9%, compared to *rsync*.

When exp_sz is fixed at 8192 bytes and bs are chosen to 64, 128, 300, 500 and 700 bytes respectively, the result of experiments shows that *tpsync*'s average amount of data over network is about 52 MB, while that of *rsync* is about 57

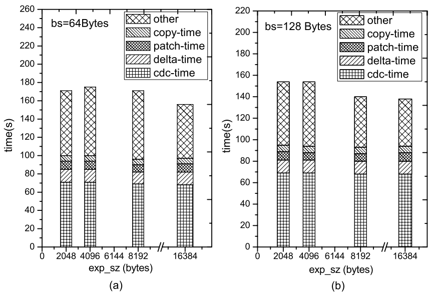


Fig. 9. Synchronizing Time of gcc with Different exp_sz Parameter in *tpsync*

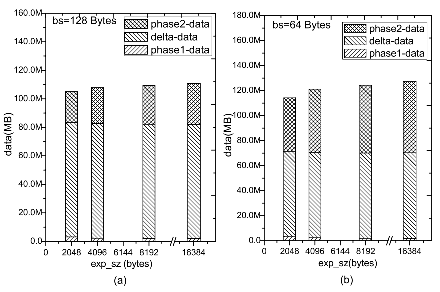


Fig. 10. Amount of Data Transferred over Network for gcc's Synchronization with Different exp_sz Parameter in *tpsync*

MB. When exp_sz is 8192 bytes and bs is 300 bytes, *tpsync*'s amount of data over network is about 48 MB and its synchronizing time is 24 seconds, both better than those of *rsync*, which are respectively 51 MB and 37 seconds (bs=700 Bytes).

Compared to *rsync*, *tpsync* generates nearly the same amount of data over network, but its synchronization time is much less than *rsync* especially with small bs value. The following four reasons account for such better performance:

(1) When calculating the signature, *rsync* works in a serial manner. It first calculates signatures on one side and then sends them to the other end. The other side then calculates and matches subsequently; while *tpsync* utilizes parallel calculation on both sides and then sends the signature from one side to the other to conduct signature match process.
(2) To files of different versions, *tpsync* utilizes edit distance algorithm to locate pairs of unmatched chunks between files, which is different from the index and match method used within *rsync*. In the first-phase, *tpsync* partitions files into chunks based on contents and calculates their corresponding signature. Because of the similarity of fingerprints of different versions of files, the edit distance algorithm can locate changed segments quickly.
(3) *tpsync* utilizes large window size to locate changed data segments, thus it has less extra communication overhead than *rsync*.
(4) To pairs of unmatched chunks of different files, *tpsync* calculates the differential data on small-grained chunks compared to the traditional way of *rsync* to calculate the differential data on the whole file. Thus, the second-phase of *tpsync* owns higher speed of synchronizing parts of the file than *rsync* synchronizing the whole file.

3.4 Results of Signature Cached Synchronization

According to the analysis of Figure 6, it is obvious that the synchronizing time can be reduced further if the transferring time of the CDC signature is reduced. Table 6 shows the result of synchronization time of gcc using SCS and without using SCS. Although cached signature consumes some extra storage space, it can improve synchronization efficiency further. Cached signature can save the computational overhead and the signature transferring time of the recipient side. Compared to the method without caching signature, signature cached synchronization method reduces its synchronization time by about 6.25% on average for gcc, as shown in Table 6.

Table 6. Result of Signature Cached Synchronization Method

Parameter	Ex-storage(MB)	SCS time(s)	No-SCS time(s)	Improvement(%)
(2048,128)	2,993	156	145	6.9%
(4096,128)	2,231	154	144	6.5%
(6144,128)	1,991	140	131	6.1%
(8192,128)	1,879	137	129	5.5%

4 Conclusion

The synchronization of remote files has been widely used in data back-up, web caching and etc. This paper presents *tpsync*, a two-phase synchronization algorithm, which introduces a two level redundancy elimination phases to reduce synchronization time and bandwidth savings over network. Compared to *rsync*, results of experiments show that *tpsync* has reduced synchronization time by 12% and bandwidth savings over network by 18.9%, when they are respectively under their own best parameters. When signature cached synchronization method is adopted, *tpsync* gets better performance further. In the future work, we still try to improve *tpsync*'s performance with optimum choice of these parameters according to the sizes and the type of each file and widths of networks, in order to balance the tradeoff between bandwidth savings and computation overheads.

Acknowledgements. This research is sponsored in part by the National High Technology Research and Development Program of China under Grant No.2009A-A01Z104, the Notational Natural Science Foundation of China under Grant No.60833004 and 60673145.

References

1. Ajtai, M., Burns, R., Fagin, R., Long, D., Stockmeyer, L.: Compactly encoding unstructured inputs with differential compression. Journal of the ACM (JACM) 49, 318–367 (2002)
2. Whitten, A.: Scalable Document Fingerprinting. In: The USENIX Workshop on E-Commerce (1996)
3. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Communications of the ACM 20, 350–353 (1977)
4. Korn, D., Vo, K.: Engineering a differencing and compression data format, pp. 219–228 (2002)
5. MacDonald, J.: File system support for delta compression. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, Master thesis (May 2000)
6. Percival, C.: Naive differences of executable code, Draft Paper, http://www.daemonology.net/bsdiff
7. Trendafilov, D., Memon, N., Suel, T.: zdelta: An efficient delta compression tool. Department of Computer and Information Science, Polytechnic University Technical Report (2002)
8. Tridgell, A.: Efficient algorithms for sorting and synchronization. PhD thesis, Australian National University (1999)
9. Meunier, P., Nystrom, S., Kamara, S., Yost, S., Alexander, K., Noland, D., Crane, J.: ActiveSync, TCP/IP and 802.11 b Wireless Vulnerabilities of WinCE-based PDAs. In: Proceedings of Eleventh IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, WET ICE 2002, pp. 145–150 (2002)
10. Whitepapers, P.: Invasion of the data snatchers (1999), http://www.pumatech.com/enterprise/wp-1.html

11. Palm: Palm developer knowledge base manuals (1999), http://palmos.com/dev/support/docs/palmos/ReferenceTOC.html
12. Muthitacharoen, A., Chen, B., Mazieres, D.: A low-bandwidth network file system. In: Proceedings of the eighteenth ACM symposium on Operating systems principles, pp. 174–187. ACM, New York (2001)
13. Teodosiu, D., Bjorner, N., Gurevich, Y., Manasse, M., Porkka, J.: Optimizing file replication over limited bandwidth networks using remote differential compression. Technical report, Microsoft Corporation (2006)
14. Grune, D.: Concurrent Versions System, a method for independent cooperation. Report IR-114, Vrije University, Amsterdam (1986)
15. Collins-Sussman, B., Pilato, C., Pilato, C., Fitzpatrick, B.: Version control with subversion. O'Reilly Media, Inc., Sebastopol (2008)
16. Policroniades, C., Pratt, I.: Alternatives for detecting redundancy in storage systems data. In: Proceedings of the 2004 USENIX Annual Technical Conference, pp. 73–86 (2004)
17. Jain, N., Dahlin, M., Tewari, R.: Taper: Tiered approach for eliminating redundancy in replica synchronization. In: Proceedings of the 4th Usenix Conference on File and Storage Technologies (FAST 2005) (2005)
18. Denehy, T., Hsu, W.: Duplicate management for reference data. Research Report RJ10305, IBM (2003)
19. Kulkarni, P., Douglis, F., LaVoie, J., Tracey, J.: Redundancy elimination within large collections of files. In: The USENIX Annual Technical Conference, General Track, 59–72 (2004)
20. Quinlan, S., Dorward, S.: Venti: a new approach to archival storage. In: Proceedings of the FAST 2002 Conference on File and Storage Technologies, vol. 4 (2002)
21. Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Weimer, W., Wells, C., et al.: Oceanstore: An extremely wide-area storage system. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Citeseer, pp. 190–201 (2000)
22. Rabin, M.: Fingerprinting by random polynomials. Technical report, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University (1981)
23. Bobbarjung, D., Jagannathan, S., Dubnicki, C.: Improving duplicate elimination in storage systems. ACM Transactions on Storage (TOS) 2, 424–448 (2006)
24. Levenshteiti, V.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet Physics-Doklady, vol. 10 (1966)
25. Martin Pool, D.B.: librsync, http://librsync.sourceforge.net
26. Council, T.: TPC BenchmarkTM C Standard Specification (2005)

A New Parallel Method of Smith-Waterman Algorithm on a Heterogeneous Platform

Bo Chen, Yun Xu, Jiaoyun Yang, and Haitao Jiang

Department of Computer Science, University of Science and Technology of China,
Key Laboratory on High Performance Computing, Anhui Province, China
cbo869@mail.ustc.edu.cn

Abstract. Smith-Waterman algorithm is a classic dynamic programming algorithm to solve the problem of biological sequence alignment. However, with the rapid increment of the number of DNA and protein sequences, the originally sequential algorithm is very time consuming due to there existing the same computing task computed repeatedly on large-scale data. Today's GPU (graphics processor unit) consists of hundreds of processors, so it has a more powerful computation capacity than the current multicore CPU. And as the programmability of GPU improved continuously, using it to do generous purpose computing is becoming very popular. In order to accelerate sequence alignment, previous researchers use the parallelism of the anti-diagonal of similarity matrix to parallelize the Smith-Waterman algorithm on GPU. In this paper, we design a new parallel algorithm which exploits the parallelism of the column of similarity matrix to parallelize the Smith-Waterman algorithm on a heterogeneous system based on CPU and GPU. The experiment result shows that our new parallel algorithm is more efficient than that of previous, which takes full advantage of the features of both the CPU and GPU and obtains approximately 37 times speedup compared with the sequential algorithm named OSEARCH implemented on Intel dual-core E2140 processor.

1 Introduction

General purpose computing on GPU (GPGPU) is one of the hottest trends in high performance computing. With the powerful computing capability, today's GPU is very suitable to solve the high parallel, computing-intensive problems, especially the scientific computation, such as bioinformatics [1,2,3,4], which is an important interdiscipline of computer science and biological science. To facilitate the programming on GPU for general purposes, Nvidia corporation has developed a new GPGPU platform named CUDA (compute unified device architecture) [5] which uses the language similar to standard C language to program rather than graphics API (application programming interface), such as OpenGL or DirectX. So it is easier to program on GPU using CUDA because it doesn't need to learn graphics knowledge.

Although today's GPU becomes a programmable data-parallel processor, if we can't fully exploit the potential parallelism of the problem or the problem is inherently serial, the performance of the program implemented on GPU will even be worse than that of CPU. In order to obtain the optimal performance, building a heterogeneous system integrated with CPU and GPU is a very good solution. For a given problem, the first step is to identify the parallel parts through analysis of the relationship of data dependency, the next to employ GPU to compute them in parallel, and the sequential parts are still computed by CPU. In this way, it can take advantage of the strength of both CPU and GPU.

In this paper, we will demonstrate how well the heterogeneous system integrated with CPU and GPU can accelerate the process of biological sequence alignment. And we designed a new parallel method of the Smith-Waterman algorithm [6] which can compute all the elements in the same column of the Smith-Waterman DP (dynamic programming) matrix independently of each other in parallel rather than Liu's algorithm [7] and Fumihiko Ino's algorithm [8] which compute all the elements in the same anti-diagonal independently of each other in parallel. Also we will do experiment to demonstrate that our new parallel algorithm is more efficient than the previous ones.

The remainder of this paper is organized as follows. Section 2 reviews the related work about the problem of sequence alignment. Section 3 describes this problem in details and proves why our new parallel method is reasonable. Section 4 implements and optimizes our parallel algorithm on a heterogeneous system. Section 5 gives an experimental result to evaluate the performance of our algorithm. Finally, concludes and points the direction of our further work.

2 Related Work

The Smith-Waterman algorithm [6] is a classic dynamic programming algorithm used to solve the problem of sequence alignment. However, it is a sequential and time-consuming algorithm which makes it impractical for a large number of sequence alignments. A more practical solution is that using a heuristic approach to generate a near optimal solution, such as FASTA [9] and BLAST [10] families, but they are approximation algorithms and can't give an optimal solution. OSEARCH and SSEARCH [2] are two Smith-Waterman implementations that are part of the FASTA program. All the algorithms above are implemented on CPU.

To the best of our knowledge, Liu's algorithm [7] is the first one which uses GPU to accelerate biological sequence alignment. It is based on that all the elements in the same anti-diagonal of the Smith-Waterman DP matrix can be computed independently in parallel and implements both pairwise sequence alignment and multiple sequences alignment on a single GPU. Fumihiko Ino's algorithm [8] is an extension of Liu's algorithm on multiple GPUs.

Although both of the above algorithms implemented on GPU obtain a considerable speedup compared with the serial algorithm implemented on CPU, there is still room for further improvement of the program performance. Firstly, the parallel granularity is the anti-diagonal of the Smith-Waterman DP matrix, so there is a workload imbalance case among kernels which invoked to parallel compute all elements in the same anti-diagonal due to the different length of the anti-diagonals. Secondly, limited by the GPU's architecture at that time, Liu employs texture memory to store the elements of the similarity matrix, as we known, texture memory is off-chip and the latency of accessing it is much higher than that of accessing the shared memory of current GPU [11].

In this paper, we take some steps to avoid the above factors affecting the performance. After carefully analyzing the relationship of data dependency, we can parallelize the column of the similarity matrix, so the problem of workload imbalance doesn't exist due to the each column has the same length. In addition, in order to reduce the latency of access memory extremely, we maximize the use of the on-chip memory such as registers and shared memory.

3 Sequence Alignment

There are two kinds of sequence alignment. One is global alignment which converts a sequence $A = a_1a_2 \ldots a_n$ to another sequence $B = b_1b_2 \ldots b_m$ by insertion or elimination of sequence elements, where n and m represent the length of A and that of B, respectively [8]. The other one is local alignment which recognizes the most similar part in the pair of sequences. Here the similarity represents the cost of alignment which correlated to the number of insertions and eliminations. So the local alignment is a special case of sequence alignment. Also it consists of pairwise sequence alignment and multiple sequences alignment (MSA). In this paper, we just discuss the problem of local sequence alignment.

3.1 Sequential Pairwise Sequence Alignment

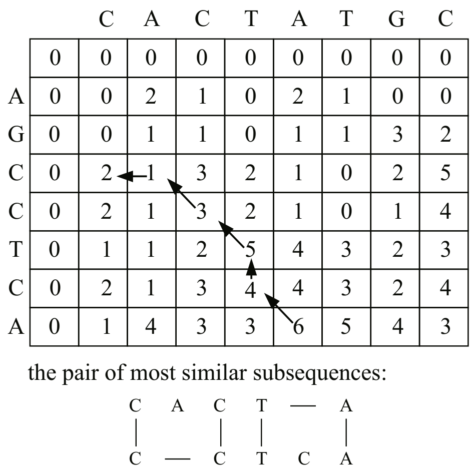
In general, given a biological sequence usually called query sequence and a database which contains many sequences, each one is called a target sequence, the purpose of pairwise sequence alignment is that we must extract the most similar or identical subsequences from query sequence and target sequence by aligning the query sequence with every target sequence in the database. The Smith-Waterman algorithm is the most classic algorithm for solving the sequence local alignment which uses a matrix called similarity matrix to store the similarity values of every pair subsequence of query sequence and target sequence from the first position. For example, let a sequence $A = a_1a_2 \ldots a_n$ be the query sequence, a sequence $B = b_1b_2 \ldots b_m$ be one of target sequence, matrix $H_{(n+1)\text{x}(m+1)}$ be the similarity matrix and element $H(i, j)$ represent the highest similarity between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$, namely a subsequences ending at element a_i and b_j, respectively, where $1 \leq i \leq n$ and $1 \leq j \leq m$ [8]. The similarity value $H(i, j)$ can be computed as follows:

$$\begin{aligned}
H(i,j) &= max\{0, E(i,j), F(i,j), H(i-1,j-1) + s(a_i,b_j)\}, \\
E(i,j) &= max\left\{H(i,j-1) - p, E(i,j-1) - q\right\}, \\
F(i,j) &= max\left\{H(i-1,j) - p, E(i-1,j) - q\right\}.
\end{aligned}$$

Where, $E(i,j)$ and $F(i,j)$ are the optimal similarity values of subsequences $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$ but with a gap in sequence A and B. The $s(a_i,b_j)$ is the substitution cost of converting one element a_i to another element b_j. The p and q represent the value of gap penalty. At the start, $H(i,0)=E(i,0)=H(0,j)=F(0,j)$, for all $i\in[0,n]$ or $j\in[0,m]$. Usually, the substitution cost $s(a_i,b_j)$ and gap penalty p, q are replaced with a constant value, such as $p=q=1$, $s(a_i,b_j)=2$ if $a_i=b_j$, otherwise $s(a_i,b_j)=-1$ [7]. In this way, the above formula of compute $H(i,j)$ can be simplified as:

$$H(i,j)=\begin{cases} max\{0,H(i,j-1)-1,H(i-1,j)-1, & \\ \qquad H(i-1,j-1)-1\}, & \text{if } a_i\neq b_j \\ H(i-1,j-1)+2, & \text{if } a_i=b_j \end{cases} \tag{1}$$

By the recursive formula (1), the similarity matrix of the query sequence A with the target sequence B can be computed. An example is showed in figure 1.



		C	A	C	T	A	T	G	C
	0	0	0	0	0	0	0	0	0
A	0	0	2	1	0	2	1	0	0
G	0	0	1	1	0	1	1	3	2
C	0	2	1	3	2	1	0	2	5
C	0	2	1	3	2	1	0	1	4
T	0	1	1	2	5	4	3	2	3
C	0	2	1	3	4	4	3	2	4
A	0	1	4	3	3	6	5	4	3

Fig. 1. This example shows the similarity matrix H computed for a query sequence $A=AGCCTCA$ and a target sequence $B=CACTATGC$ using Smith-Waterman algorithm. And tracing back from the maximum value of the similarity matrix ($H(7,5)=6$), we can get the pair of most similar subsequences: $CACTA$ and $CCTCA$.

3.2 Parallel Pairwise Sequence Alignment

From the figure 2(a), it can be seen that only the elements in the same anti-diagonal painted the same color can be computed simultaneously and Liu [7] and Fumihiko [8] just used this parallelism of anti-diagonal. However, through unrolling the recursive formula (1), we find that the relationship of data dependency among the same column can be eliminated, so the elements in the same column can be computed in parallel. Now we will show why it is reasonable.

According to formula (1), if the condition $a_i=b_j$ satisfies, $H(i,j)$ is determined by $H(i-1,j-1)$ instead of any other element of the jth column.

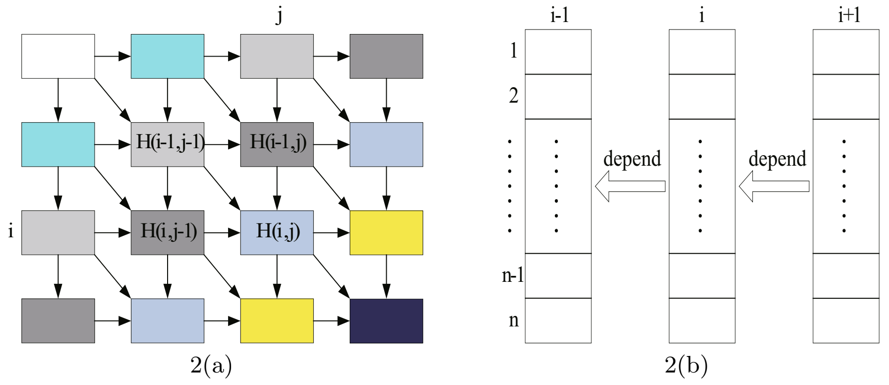


Fig. 2. 2(a) The relationship of data dependency in the similarity matrix. Element $H(i,j)$ of the similarity matrix depends on its left element $H(i,j-1)$, upper element $H(i-1,j)$ and the diagonal element $H(i-1,j-1)$. 2(b) The new relationship of data dependency in the similarity matrix. The elements of ith column only depend on the elements of $(i-1)$th $(i \geq 1)$ column.

Otherwise, $H(i,j)$ depends on the maximum one of $H(i,j-1)$, $H(i-1,j)$ and $H(i-1,j-1)$(we don't consider the constant value 0 temporarily). Here, it only needs to consider the case that $H(i-1,j)$ is maximum value because whether $H(i,j-1)$ or $H(i-1,j-1)$ is the maximum one, $H(i,j)$ doesn't depend on the elements of the jth column. As the previous hypothesis, $H(i,j) = max\{0, H(i-1,j)-1\}$. Considering the worst case that $H(i-1,j)-1 > 0$, namely $H(i,j) = H(i-1,j)-1$, we can further recursively compute $H(i-1,j)$ until there is no data dependency in the jth column or $i = 0$(initially $H(0,j) = 0$). Suppose a_k $(k > 0)$ is the first element encountered meeting with b_j in the process of the above recursion and $H(r,j)-1$ $(k \leq r \leq i-1)$ is the maximum one in every iteration, so

$$H(i,j) = max\{0, H(k-1,j-1)-(i-k)+2\}. \tag{2}$$

In order to prove the equation (2), we firstly use mathematical induction to prove the theorem 1.

Theorem 1. *If a_k is the first element meeting with b_j from a_i to a_1 and $H(r,j)-1$ $(k \leq r \leq i-1)$ is the maximum one in every iteration, then*

$$H(i,j) = H(k,j)-(i-k), \textit{ where } 1 \leq k < i. \tag{3}$$

Proof

- Initially, when $k = i-1$, $H(i,j) = H(k,j)-(i-k) = H(i-1,j)-1$, established.
- Suppose the equation (3) is established when $m = i-r$ $(k < m < i-1)$, namely

$$H(i,j) = H(m,j)-(i-m). \tag{4}$$

– Now we will prove that when $m = i-r-1$ the equation (3) is still established. According to the equation (4), furthermore, because a_m doesn't meet with b_j, so

$$H(m, j) = H(m-1, j) - 1. \tag{5}$$

Take the equation (5) to the equation (4), $H(i, j) = H(m-1) - (i-m) - 1 = H(m-1, j) - (i-(m-1))$, so the hypothesis was established, namely the theorem 1 is established. □

According to theorem 1, it is easy to prove the equation (2). Because of a_k meeting with b_j, so

$$H(k, j) = H(k-1, j-1) + 2. \tag{6}$$

Take the equation (6) to the equation (3) and considering the constant value 0, so

$$H(i, j) = max\{0, H(k-1, j-1) - (i-k) + 2\},$$

namely the equation (2) is established.

Through the above proof, we eliminate the data dependency among the same column of similarity matrix and transform the parallel granularity of anti-diagonal to that of column, as seen in figure 2(b). So the element $H(i, j)$ can be computed as:

$$H(i, j) = \begin{cases} max\{0, H(i, j-1) - 1, H(i-1, j-1) - 1, \\ \qquad H(k-1, j-1) - (i-k) + 2\}, & \text{if } a_i \neq b_j \\ H(i-1, j-1) + 2, & \text{if } a_i = b_j \end{cases}$$

$$\text{where} \quad k = max\{n, \text{ for } 0 < n < i \text{ } and \text{ } a_n = b_j\}.$$

When programming, an array called pre_dependence is used to store the index of the element in query sequence which is the first one meeting with a given element in target sequence from current position to 1.

4 Implement the Parallel Algorithm on a Heterogeneous System

4.1 Load Partition

In order to take advantage of the strengths of both CPU and GPU in our heterogeneous system respectively, we must identify the sequential parts and the parallel parts of the whole program, and make the former computed by GPU, the later computed by CPU. For the problem of sequence alignment, the parallelism derived from the computation of the similarity matrix which is also the most time consuming parts. To enhance the parallelism, like Liu's algorithm, we exploit both the task-level parallelism and the data-level parallelism. The task-level parallelism is that one thread block tackles the alignment of one target sequence with the query sequence, so there will be dozens of target sequences is aligned simultaneously because our GPU has 14 SMs (stream multiprocessor) and each one can process eight thread blocks in parallel at most [11].

The data-level parallelism is that one column of similarity matrix can be computed in parallel, namely one thread of a thread block aligns one character of target sequence with that of query sequence. This way of parallel computing is shown in figure 3. After all alignments completed, we can get a similarity value for every target sequence with the query sequence, and transfer them from global memory to host memory, then select the maximum one among them and compute the most similar subsequences on CPU, and this is the sequential parts which computed by CPU because the time consumed in this step can be ignored related to that of former step.

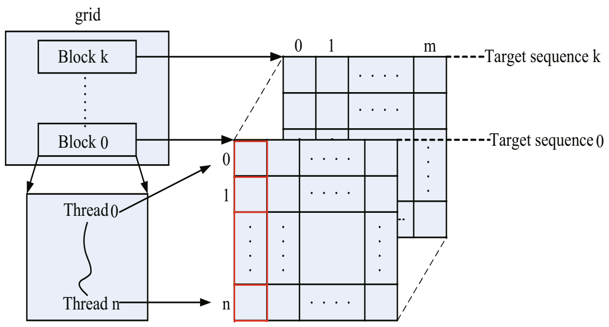


Fig. 3. The way of parallel process in kernel. The m and n represent the length of target sequence and query sequence respectively.

4.2 Pseudocodes of the Algorithm

We implement the new parallel algorithm of Smith-Waterman algorithm on a heterogeneous system integrated with CPU and GPU. The pseudocodes of the algorithm implemented both on CPU and GPU are illustrated in figure 4(a) and figure 4(b).

4.3 Some Optimization Strategies

Circularly Use Shared Memory. In order to find the optimal target sequence from the database which contains a subsequence having the highest similarity value with the subsequence of query sequence, it needs to check each target sequence and compute the corresponding similarity matrix. This is the most time consuming step, so we invoke a kernel executed on GPU to complete it. However, the shared memory of GPU is just 16KB which is too small to store the entire similarity matrix, unless using the global memory to replace it. Yet, the latency of accessing global memory is about 400 $\sim$ 600 clock cycles compared to that of shared memory's 4 cycle if there is no bank conflict [11]. Furthermore, the problem of local alignment is memory intensive, so access global memory frequently will affect the performance of the whole program greatly. Fortunately, there is no need to store the entire similarity matrix, just two columns of it

```
1. Randomly generate the query protein sequence
   and many target protein sequences;
2. Calculate the pre_dependence array for query
   sequence;
3. Copy query sequence and pre_dependence array
   to constant array, also target sequences to global
   memory on GPU;
4. Invoke kernel to compute the similarity value of
   query sequence with each target sequence on
   GPU;
5. Copy the sequence number with maximum
   similarity value back to CPU memory;
6. Compute the most similar subsequences on CPU.
```

4(a)

```
1. Allocate an array called target_seq on shared
   memory to store one target sequence;
2. Allocate two arrays named pre_col and cur_col
   on shared memory to store the elements of the
   column which was computed completely in the
   previous iteration and that will be computing in
   current iteration respectively;
3. Threads in the same block load one target
   sequence to array target_seq, one thread loads
   one element of the sequence;
4. For i=0 to m   // for each column
     Calculate the element of the current column
     according to the arrays pre_col and
     pre_denpendence;
     __syncthreads();
     Copy the elements of cur_col to pre_col
     preparing for the next iteration;
     Compute the maximum value of the current
     column using reduce-likely method;
5. Store the sequence number with maximum
   similar value to global memory.
```

4(b)

Fig. 4. 4(a) The pseudocode of the algorithm implemented on CPU. 4(b) The pseudocode of the algorithm implemented on GPU.

will be enough. The reason is that the parallel granularity is the column of the similarity matrix, namely, only the elements located at the current column can be computed simultaneously and they just depend on that of the previous column, the following columns can't be computed before the computation of current column is completed. So we use two arrays named pre_col and cur_col in the shared memory to store the column computed completely in the previous iteration and the column will be computed in the current iteration respectively. As seen in figure 5, the use of the two arrays is cyclic.

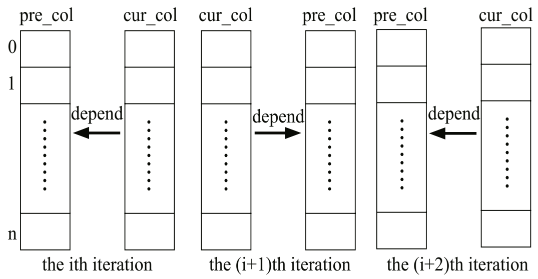


Fig. 5. Shows the cyclic use of arrays pre_col and cur_col in the different iterations

Use Reduce-Likely Method to Compute the Maximum Value. When the current column is computed completely, it needs to compute the maximum value of the column in order to determine whether it is a currently maximum value. We use a reduce-likely method to compute the maximum value of the current column which is very efficient. Figure 6 gives an illustration of this method.

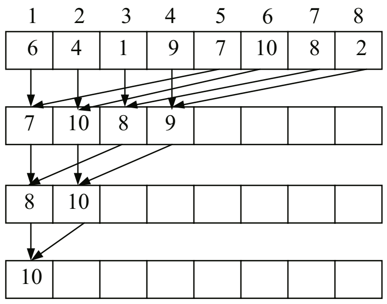


Fig. 6. This example shows the process of select the maximum value of an array which contains eight elements using reduce-likely method. When the element number of array equals 2^n, it only needs n iterations to get the maximum value and the ith iteration is executed parallel by $2^{(n-i)}$ threads.

Coalesced Access to Global Memory. The global memory is not cached, and the latency of accessing it is around 400 $\sim$ 600 clock cycles. So it is more important to take the appropriate access pattern to get maximum memory bandwidth because the bandwidth for non-coalesced accesses is around 2 $\sim$ 10 times lower than that for coalesced accesses. Fortunately, GPU is capable of reading 32-bit, 64-bit, or 128-bit words from global memory into registers in a single instruction, and the global memory bandwidth can reach the maximum value when simultaneous memory accesses by thread in a half-warp can be coalesced into a single memory transaction. The size of a memory transaction can be either 32 bytes (for compute capability 1.2 and higher only), 64 bytes, or 128 bytes [11]. The protein sequence consists of many specific characters, and the size of a character in global memory is 1 bytes. So if the data type of protein sequence is defined as char, reading or writing global memory can't achieve coalesced access. But we can use the build-in vector type uchar4 which makes four characters together to represent the protein sequence. In this way, it can obtain coalesced access to global memory due to the type uchar4 requires 4 bytes memory space and each thread in a half-warp access 4 bytes resulting in one 64-byte memory transaction.

5 Experiment Results

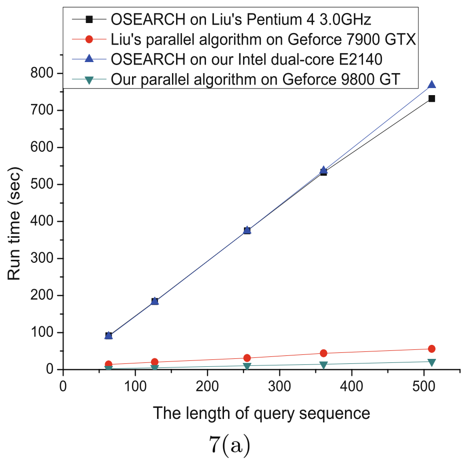
In order to show our parallel algorithm is more efficient than previous ones, we implemented the above algorithms on a heterogeneous system which consists of an Intel dual-core E2140 processor and an Nvidia 9800GT graphics card. In this experiment, we generated 176, 469 protein sequences as the target sequences randomly and the length of every sequence is 361, and also five query sequences with different length (from 63 to 511).

In addition, Liu's algorithm [7] and Fumihiko Ino's algorithm [8] implemented on GPU both are using the graphics API which increases the complexity of programming and decreases performance, thus we use the more convenient and efficient platform of Nvidia's CUDA to implement our algorithm instead. The experiment results can be seen from table 1 (all the time in table 1 is measured in second).

Table 1. OSEARCH is a straightforward implementation of Smith-Waterman algorithm. All data in the Liu's algorithm comes from [7]. And the speedup of Liu's algorithm represents the ratio that the runtime of OSEARCH implemented on Pentium 4 3.0GHz CPU to that of Liu's parallel algorithm implemented on Geforce 7900 GTX. Similarly, the speedup of our algorithm is the ratio that the runtime of OSEARCH implemented on Intel dual-core E2140 processors to that of our parallel algorithm implemented on Geforce 9800 GT.

Query sequence length	Liu's algorithm			Our algorithm		
	OSEARCH on Pentium 4 3.0GHz runtime	Geforce 7900 GTX runtime	Speedup	OSEARCH on Intel dual-core E2140 runtime	Geforce 9800 GT runtime	Speedup
63	91	14	6.5	90	2.3	39.1
127	184	20	9.2	183	4.9	37.3
255	375	31	12.1	375	10.7	35.0
361	533	44	12.1	537	14.3	37.6
511	732	56	13.1	768	21.6	35.6

Besides, we also compared our algorithm with Fumihiko Ino's [8] which extends Liu's algorithm to multiple GPUs. In [8], the authors scan SWISS-PROT database of release 51.0, which contains 241, 242 protein sequences and the average length is 367 which is the same with that of the query sequence. For a single scan of the database, the fastest running time is 18s on the Geforce 8800 GTX which contains 128 processor cores [8]. In experiment we also scan the same scale of the target sequences generated randomly, and the running time is 19.12s on Geforce 9800 GT which contains 112 processor cores. Intuitively, our algorithm taken more time than Fumihiko Ino's, but Fumihiko Ino's GPU has 16 processor cores more than ours. When considering the time spent by the signal processor core, our algorithm is more efficient than Fumihiko Ino's due to our signal processor core consumed 2141.44s compared with that of Fumihiko Ino's 2304.0s.



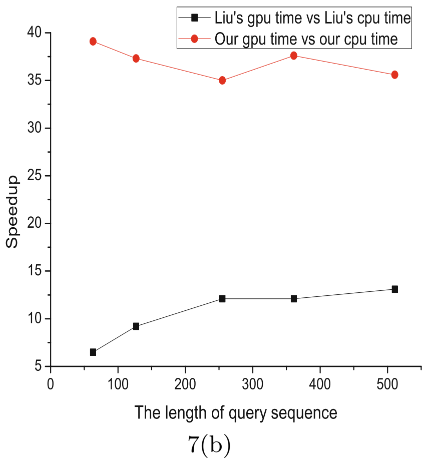


Fig. 7. 7(a) Shows the different running times of different implementations. 7(b) Illustrates the speedup of Liu's algorithm and our algorithm respectively. And the data comes from the table 1.

6 Conclusion

In this paper, we designed a new parallel method of Smith-Waterman algorithm to solve the problem of biological sequence alignment and implemented the algorithm on a heterogeneous system based on CPU and GPU through CUDA platform. The experiment results show that our new parallel algorithm is more efficient than previous ones. And the result of this work also indicates that a heterogeneous system integrated with CPU and GPU (or other computing devices, such as FPGA) has a powerful computation capability and superior generality, so it will be applied to more fields [12].

In the future, we will research the performance optimization of high performance computing on a heterogeneous system integrated with CPU and GPU, and further improve the performance of our algorithm through decreasing the number of access the global memory [13]. Furthermore, we also plan to solve more scientific computing problems on the heterogeneous platform.

Acknowledgments

We thank Yuzhong Zhao and Da Teng, who provided many helpful suggestions for our article.

Funding:

- The Key Project of The National Nature Science Foundation of China, under the grant No. 60533020 and No. 60970085.
- The Key Subproject of The National High Technology Research and Development Program of China, under the grant No. 2009AA01A134.

References

1. Yu, L., Xu, Y.: A Parallel Gibbs Sampling Algorithm for Motif Finding on GPU. In: 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 555–558 (2009)
2. Horn, D., Houston, M., Hanrahan, P.: ClawHMMer: A Streaming HMMer-Search Implementation. In: Proc. ACM/IEEE Conf. Supercomputing, SC 2005 (2005)
3. Liu, W., Schmidt, B., Voss, G.: Bio-Sequence Database Scanning on a GPU. In: Proc. 20th IEEE Int'l Parallel and Distributed Processing Symp (High Performance Computational Biology (HiCOMB) Workshop) (2006)
4. Liu, W., Schmidt, B., Voss, G.: GPUClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment. In: Proc. 13th Ann. IEEE Int'l Conf. High Performance Computing (HiPC 2006), pp. 363–374 (2006)
5. NVidia CUDA, http://www.nvidia.com/cuda
6. Smith, T., Waterman, M.: Identification of Common Molecular Subsequences. J. Molecular Biology 147, 195–197 (1981)
7. Liu, W., Schmidt, B., Voss, G.: Streaming algorithms for biological sequence alignment on GPUs. IEEE Trans. Parallel and Distributed Systems 18(9), 1270–1281 (2007)
8. Ino, F., Kotani, Y., Hagihara, K.: Harnessing the Power of idle GPUs for Acceleration of Biological Sequence Alignment. In: IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009, pp. 1–8 (2009)
9. Pearson, W.R.: Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith and Waterman and FASTA algorithms. Genomics 11, 635–650 (1991)
10. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. 215, 403–410 (1990)
11. NVIDIA Corporation. CUDA Programming Guide Version 2.0 (July 2008)
12. Owens, J.: HCW 2009 keynote talk: GPU computing: Heterogeneous computing for future systems. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–1 (2009)
13. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.-Z., Baghsorkhi, S.S., Hwu, W.-m.W.: Program optimization carving for GPU computing. J. Parallel Distrib. Comput., 1389–1401 (2008)

Improved Genetic Algorithm for Minimizing Periodic Preventive Maintenance Costs in Series-Parallel Systems

Chung-Ho Wang and Te-Wei Lin

Department of Power Vehicle and Systems Engineering
Chung Cheng Institute of Technology, National Defense University
No. 190, Sanyuan 1st St, Tahsi, Taoyuan, Taiwan 335. R.O.C.

Abstract. This work presents an improved genetic algorithm (IGA) for minimizing periodic preventive maintenance costs in series-parallel systems. The intrinsic properties of a repairable system, including the structure of reliability block diagrams and component maintenance priorities are considered by the proposed IGA. The proposed component importance measure considers these properties, identifies key components, and determines their maintenance priorities. The optimal maintenance periods of these important components are then determined to minimize total maintenance cost given the allowable worst reliability of a repairable system. An adjustment mechanism is established to solve the problem of chromosomes falling into infeasible areas. A response surface methodology is further used to systematically determine crossover probability and mutation probability in the GA instead of using the conventional trial-and-error process. A case study demonstrates the effectiveness and practicality of the proposed IGA for optimizing the periodic preventive maintenance model in series-parallel systems.

Keywords: Periodic preventive maintenance; Reliability; Importance measure; Genetic algorithms; Response surface methodology.

1 Introduction

Establishing a superior maintenance strategy for a complex repairable system requires that the maintenance priority of subsystems or components and their maintenance periods given limited maintenance resources be determined simultaneously. As preventive maintenance consumes human resources and time, and has associated costs, nonessential services or an inadequate maintenance schedule wastes the limited maintenance resources. Furthermore, to meet practical requirements, numerous studies have constructed maintenance models and optimization algorithms [7]. However, the complexity of optimizing a maintenance model in a series-parallel system increases significantly as the number of components in a system increases. In such situations, obtaining the exact global optimum using analytical approaches via mathematical inference is impractical. Therefore, meta-heuristic algorithms, such as a genetic algorithm (GA) [2][3], ant colony optimization [12]and simulated annealing [7], are commonly employed to optimize these models and approach the global optimum.

This study aims to propose an improved genetic algorithm (IGA) that efficiently optimizes the periodic preventive maintenance (PM) model for series-parallel systems. The improved mechanisms mainly overcome the weaknesses in conventional GA. Since the structure of a repairable series-parallel system markedly impacts system reliability, the properties of a preventive maintenance model in series-parallel systems subjecting to the allowable worst system reliability are considered to create the improved mechanisms. The proposed IGA has two stages. The first stage identifies important components for a repairable series-parallel system. A novel importance measure of components is developed to overcome the drawbacks of past importance measures of components [1][17][4] and appropriately assess the importance of components in a PM model during mission duration. The second stage determines the optimal maintenance periods of important components using the IGA search mechanism. An adjustment mechanism is established to make the chromosomes move back to feasible area in case the chromosomes fall into infeasible area. Doing so can enhance the exploring ability of conventional GA. The response surface methodology (RSM) [10] from the design of experiments is employed to systematically determine the crossover probability and mutation probability—instead of using the trial-and-error approach—to enhance IGA search capability. A case from the study by Bris et al. [2], subsequently applied by Samrout et al. [12], demonstrates the effectiveness and practicality of the proposed IGA in optimizing the periodic preventive maintenance model in series-parallel systems.

2 Maintenance Strategy of a Series-Parallel System

Maintenance is defined as activities that retain or restore the operational status of a system. Normally, maintenance can be classified as e-maintenance [11], corrective maintenance (CM) and preventive maintenance [8]. E-maintenance is a new concept of maintenance. It contains predictive prognostics and condition-based monitor, and integrates existing telemaintenance principles with Web services and modern e-collaboration principles. CM includes minimal repairs and corrective replacement when a system fails. PM includes simple preventive maintenance and preventive replacement when a system is operating. The maintenance policies of a repairable deteriorating system are (1) age-dependent PM policy, (2) periodic PM policy, (3) failure limit policy, (4) sequential PM policy, (5) repair limit policy, and (6) repair number counting and reference time policy[16]. Periodic PM is widely used in practice simply because of its ease of implementation and management. This maintenance policy, applied in a series-parallel system with multiple components, received much attention. For example, Tsai et al. [14] developed a periodic PM schedule for a system with deteriorating electro-mechanical components and optimized it using a GA. Leou [7] proposed a novel algorithm for determining a maintenance schedule for a power plant. This algorithm combines the GA with simulated annealing to optimize maintenance periods and minimize maintenance and operational cost. Tsai et al. [15] also proposed a preventive maintenance policy for a multi-component system. Maintenance activities for components in each stage of PM were determined by maximizing the availability of the system for maintenance. Busacca et al. [3] focused on a high-pressure injection system at a nuclear power plant to establish a multi-objective optimization model to obtain a maintenance strategy using GA. Bris et al. [2] proposed a periodic PM model that minimizes maintenance costs under the reliability constraint. The optimal maintenance period of each components after the first maintenance task for that component was determined using a GA. Samrout et al. [12] optimized the Bris et al. [2] using the same procedure, but the ant colony optimization was adopted to optimize the maintenance periods for all components.

3 Developed Importance Measures

This work extends the underlying concept of the Birnbaum importance measure [1] to a novel measure of component importance. The Birnbaum importance measure [1] considers the adverse effects of failed components on system reliability. However, component failure/operational probabilities are directly related to the reliability of individual components, which is a function of time. Failure/operational probabilities vary over time. Therefore, the developed importance measure considers this probability by calculating the expected values of component effects on system reliability to assess the importance of components for PM. The calculation of the expected values of effects is as follows.

$$I_j(t) = (R_S(1, r_j(t)) - R_S(t)) \times r_j(t) + (R_S(t) - R_S(0, r_j(t)) \times (1 - r_j(t))) \tag{1}$$

where $R_S(t)$ is the reliability of a system at time *t*; $R_S(1, r_j(t))$ is the reliability of a system when the *j*th component is operating at time *t*; $R_S(0, r_j(t))$ is the reliability of a system when the *j*th component fails at time *t*; $r_j(t)$ is the probability when the *j*th component is operating at time *t*; $(1 - r_j(t))$ is the probability when the *j*th component fails at time *t*; The value of $(R_S(1, r_j(t)) - R_S(t))$ indicates the positive effect when the *j*th component is operating at time *t*; $(R_S(t) - R_S(0, r_j(t))$ is the negative effect when the *j*th component is failed at time *t*; $I_j(t)$ is the expected value of system effect at time *t*. The integration of $I_j(t)$ values over mission duration is determined as the importance measure of components for PM, as follows.

$$I_j^T = \int_0^T I_j(t)dt \tag{2}$$

where I_j^T is the importance measure of the *j*th component during the mission duration *T*. The developed importance measures can be used in determining the maintenance priority of components for PM.

4 Proposed IGA

The following sections describe the proposed IGA for minimizing periodic PM cost in series-parallel systems.

4.1 Construction of Periodic PM Model of the Series-Parallel System

The PM cost model is based on the work of Bris et al. [2], as follows.

$$\text{Minimize} \quad C_{PM} = \sum_{k=1}^{K} \sum_{i=1}^{E_K} \sum_{j=1}^{n_{e(i,k)}} C_j(e(i,k)) \tag{3}$$

$$\text{Subject to} \quad R_S(t) \geq R_0 \tag{4}$$

$$R_S(t) = \prod_{k=1}^{K} [1 - \prod_{i=1}^{E_K} (1 - R_i(t))] \tag{5}$$

where C_{PM} is total maintenance cost; $e(i,k)$ is the *i*th component of the *k*th parallel subsystem; $n_{e(i,k)}$ is the total number of instances of maintenance of the *i*th component of the *k*th parallel subsystem; $C_j(e(i,k))$ is the cost of the *j*th instance of maintenance of the *i*th component in the *k*th parallel subsystem; E_K is the number of components in the given *k*th parallel subsystem; *K* is the number of parallel subsystems; R_0 represents the allowable worst reliability value, and $R_S(t)$ is the reliability of the system at time *t*.

4.2 Construction of IGA

This study adopts real encoding to represent maintenance periods. Hence, each gene represents the maintenance period for a component. A chromosome comprises the maintenance periods of all components. A different combination of maintenance periods of all components forms a different chromosome. The proposed IGA has the following two stages.

Stage 1: Identify the combinations of important components
The values of importance measure, I_j^T, for all components are calculated using Eqs. (1) and (2). Accordingly, the importance sequence of components is determined as follows:

$$\mathbf{S} = \bigcup_{i=1}^{n} \{s_i\} \tag{6}$$

where s_i is the *i*th important component; *n* is the number of components in a system; and $\mathbf{S}$ is a set of importance sequence of components. According to $\mathbf{S}$, the important components of a series-parallel system is determined in the following iteration procedure.

1. Given the first one component in $\mathbf{S}$, optimize the periodic preventive maintenance model using the GA to obtain a total maintenance cost if feasible solutions exist.
2. Add a component according to the importance priority in $\mathbf{S}$, optimize the periodic preventive maintenance model using the GA to obtain a total maintenance cost if feasible solutions exist.
3. Repeat iteration 2 until all the components are involved when optimizing the periodic preventive maintenance model.

The combination of components with the lowest total maintenance cost is identified as the important components of a series-parallel system. The equation for identifying the important components can be expressed as

$$C_{PM}(\mathbf{S}_m) = \min\left[C_{PM}(\mathbf{S}_j)\right],\ j = 1,\ 2,\ \ldots,\ m,\ \ldots,\ n \tag{7}$$

where $C_{PM}(\mathbf{S}_j)$ is the optimized total maintenance cost given the first j components in $\mathbf{S}$; $C_{PM}(\mathbf{S}_m)$ is the lowest maintenance cost of the first m components in $\mathbf{S}$, $1 \le m \le n$. These m important components are substituted in the second stage to establish initial GA population and thereby optimize their maintenance periods. The other unimportant components do not implement maintenance work for the mission duration.

Additionally, this study employs the faced center cube central composite design (FCCD) from the RSM [10] in which crossover probability and mutation probability are two experimental factors to systematically perform experiments. The optimal settings of crossover probability and mutation probability are therefore determined.

Stage 2: Optimize the maintenance periods.
The combination of the first m important components and the optimal parameters of crossover probability and mutation probability obtained from stage 1 form the basis to optimize the maintenance periods of a periodic PM model, performed by the following steps.

Step 1: Randomly generate maintenance periods of the first m important components to form initial population of chromosomes.
Step 2: Apply reproduce procedure, crossover procedure, and mutation procedure to breed the chromosomes for offspring.
Step 3: Perform the adjustment mechanism

An adjustment mechanism is established to make the chromosomes move back to feasible area in case the chromosomes fall into infeasible area. Doing so can enhance the exploring ability of conventional GA. In the constructed series-parallel maintenance model, the chromosomes with reliability lower than the allowable worst system reliability represent infeasible solutions. The proposed adjustment mechanism includes the following two procedures.

(1) Shorten the maintenance periods of components
The maintenance period of the component scheduled to be maintained at the time the occurrence of the lowest reliability is shortened to increase system reliability.

(2) Determine if the reliability greater than allowable worst system reliability
Recompute the system reliability after shortening the maintenance period of components. If the computed reliability does not meet the system minimum requirement, then repeat (1) and (2) until the reliability greater than system requirement, namely the chromosome moves back into feasible area.

Step 6: Perform elitist conservation strategy
If the total maintenance cost during each generation is lower than a recorded best fitness value (namely, the elitism), the elitism is replaced by the champion; otherwise, the chromosome with highest total maintenance cost (namely, worst fitness value) in each generation is replaced by the elitism.

Step 7: Terminate the IGA
Terminate the IGA and output the optimized maintenance periods of all components if the total iterations surpass a predetermined value or the fitness does not improve in continually maximum iterations.

5 Case Study

The PM model of a series-parallel system that was proposed by Bris et al. [2] is adopted herein to confirm the feasibility and practicality of the proposed algorithm. This system consists of four subsystems and 11 components. Components 1, 2, 3, 4 and 5 constitute the first subsystem. Component 6 is the second subsystem, which has a single unit. Components 7, 8, and 9 constitute the third subsystem. Components 10 and 11 consist of the fourth subsystem. Figure 1 displays the reliability block diagram. Table 1 presents the component parameters, including the probability distribution, mean time to failure (MTTF), and maintenance cost for each component. The mission duration of 50 years is simulated. The allowable worst system reliability is 0.9. The goal is to optimize the maintenance times and minimize the maintenance cost during the mission duration.

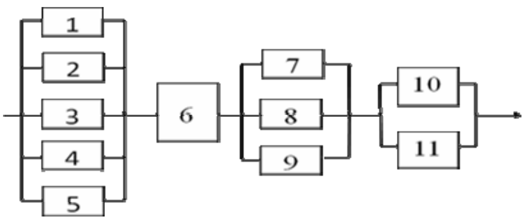


Fig. 1. Reliability block diagram [2]

Via the proposed IGA, this case was optimized in a stage-by-stage manner as follows.

Stage 1: Identify the combinations of important components.

The importance measures of 11 components, I_j^T values, are computed using Eqs. (1) and (2), and thereby form the importance sequence $\mathbf{S}$ = {5, 6, 11, 3, 1, 2, 10, 8, 9, 7, 4} using Eqs. (6). Table 2 lists the importance measures of all components for mission duration of 50 years. According to the proposed iteration procedure, the conventional GA is then applied with 5 repetitions under an initial population of 200

Table 1. Component parameters [2]

Number of components	Probability distribution	MTTF (years)	Maintenance cost
1	exponential	12.059	4.1
2	exponential	12.059	4.1
3	exponential	12.2062	4.1
4	exponential	2.014	5.5
5	exponential	66.6667	14.2
6	exponential	191.5197	19
7	exponential	63.5146	6.5
8	exponential	438.5965	6.2
9	exponential	176.0426	5.4
10	exponential	13.9802	14
11	exponential	167.484	14

chromosomes—crossover probability is 0.86 and mutation probability is 0.13 obtained from RSM—to identify the important components. The GA is terminated at 200 iterations. Hence, five total maintenance costs can be obtained to calculate the average total maintenance cost when feasible solutions exist. First, for the case involving the first important component, the fifth component, no feasible solution exists. For the case involving the first two components, components five and six, no feasible solution was found. This iteration procedure is performed by adding one component according to the importance priority in the importance sequence until all components are involved. Table 3 summarizes the optimized average total maintenance cost for all iterations. Via Eq. (7), the set of $\mathbf{S}_7$, iteration seven, which has the lowest average total maintenance cost of 218.5, was identified as the combination of important components, $\mathbf{S}_7 = \{5, 6, 11, 3, 1, 2, 10\}$. These important components, components 1, 2, 3, 5, 6, 10 and 11, are then substituted into stage two to establish an initial population and thereby optimize their maintenance periods. The components that are not included in $\mathbf{S}_7$ do not implement maintenance works for the mission duration.

Noticeably, the determination of GA with a crossover probability of 0.86 and mutation probability of 0.13 using RSM is determined based on the follow fitted model:

$$\hat{y} = 283.9 - 178.9x_1 - 53.8x_2 + 60.0x_1x_2 + 98.2x_1^2 + 46.2x_2^2 \qquad (8)$$

where $\hat{y}$ is the predicted total maintenance cost; and x_1 and x_2 are the crossover and mutation probabilities, respectively. The coefficient of determinant for this model is $R^2 = 0.933$.

Table 2. Importance measures and priority for T_M =50 years

Number of components	Importance measures	Priority
1	1.1957	5
2	1.1957	5
3	1.2268	4
4	0.0039	11
5	8.8588	1
6	6.0445	2
7	0.0948	10
8	0.1538	8
9	0.1359	9
10	0.9415	7
11	5.4180	3

Table 3. Combinations of important components

Iterations	S_m	The combinations	The average total maintenance cost
1	s_1	5	No feasible solutions
2	s_2	5, 6	No feasible solutions
3	s_3	5, 6, 11	427.8
4	s_4	5, 6, 11, 3	280.3
5	s_5	5, 6, 11, 3, 1	241.2
6	s_6	5, 6, 11, 3, 1, 2	225.3
7	s_7	**5, 6, 11, 3, 1, 2, 10**	**218.5**
8	s_8	5, 6, 11, 3, 1, 2, 10, 8	226.8
9	s_9	5, 6, 11, 3, 1, 2, 10, 8, 9	235.5
10	s_{10}	5, 6, 11, 3, 1, 2, 10, 8, 9, 7	244.7
11	s_{11}	5, 6, 11, 3, 1, 2, 10, 8, 9, 7, 4	253.1

Stage 2: Optimize maintenance periods
The randomly generated maintenance periods of seven components obtained from stage one form an initial population of chromosomes with seven genes. Each gene is a maintenance period of a component. Thus, the initial population of 200 chromosomes is established. The optimal settings of parameters obtained from RSM, including crossover probability is 0.86 and mutation probability is 0.13, are then substituted into the IGA search mechanism to approach the global optimum of the total maintenance cost solution. The magnitude of the adjustment mechanism to heighten system reliability by shortening component maintenance periods is set to 0.1 in this case. The IGA is terminated at 200 iterations. The optimized total maintenance cost for this case is 178.1. Table 7 lists the optimal maintenance periods of components for mission duration of 50 years. Figure 3 shows the corresponding reliability

Table 4. Optimized maintenance periods ($T_M = 50$ years, $R_S(t) \geq 0.9$)

Number of components	1	2	3	5	6	10	11
Maintenance periods	21.47	17.08	9.63	25.78	13.40	32.04	11.24

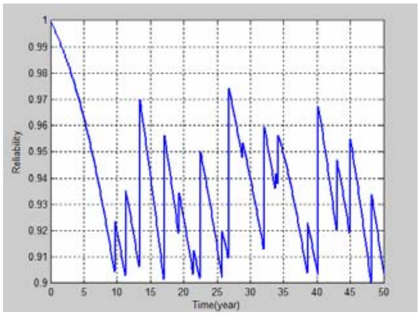


Fig. 2. Reliability curve ($T_M = 50$ years, $R_S(t) \geq 0.9$)

curve. All reliability values exceed 0.9, satisfying the constraint of allowable worst reliability.

The IGA optimization results are further compared to those obtained by Bris et al. [2] and Samrout et al. [12]. Table 5 summarizes comparison results. The optimized total maintenance cost in this study is reduced to 178.1 from 238, which was obtained by Bris et al. [2]—resulting in a reduction of 59.9 for mission duration of 50 years. Compared with that acquired by Samrout et al. [12], the optimized total maintenance cost is reduced to 178.1 from 224.2, a reduction of 46.1 for mission duration of 50 years. The proposed IGA outperforms the approaches developed by Bris et al. [2] and Samrout et al. [12].

Table 5. Comparisons of the total maintenance cost in mission duration of 50 years

	Bris *et al.*[2]	Samrout *et al.*[12]	This study (IGA)	Compare with Bris et al.[2]	Compare with Samrout et al.[12]
				Reduction	Reduction
$T_M = 50$	238.0	224.2	178.1	59.9	46.1

6 Conclusions

Some superior meta-heuristic algorithms and improved algorithms have been proposed to resolve large complex problems in recent years. However, due to the diversity of the problems, a customized algorithm typically outperforms a general algorithm in solving specific problems. This study proposes an improved GA to minimize total periodic PM cost. A novel importance measure of component is proposed for PM. Furthermore, an adjustment mechanism is established to move the chromosomes from infeasible to feasible areas in order to enhance the exploratory ability of conventional GA. A case adopted from a previous study demonstrates the problem solving efficacy of the proposed IGA. This algorithm can be extended to solve large problems with complex series-parallel systems comprised of many subsystems or components. Moreover, the proposed IGA can be modified for non-periodic maintenance and imperfect maintenance models.

References

1. Birnbaum, Z.: On the importance of different components in a multicomponent system: Multivariate Analysis 11. In: Krishnaiah, P.R. (ed.), Academic Press, London (1969)
2. Bris, R., Chatelet, E., Yalaoui, F.: New method to minimize the preventive maintenance cost of series–parallel systems. Reliability Engineering and system safety 82, 247–255 (2003)
3. Busacca, P., Marseguerra, M., Zio, E.: Multiobjective optimization by genetic algorithms: application to safety systems. Reliability Engineering and system safety 72, 59–74 (2001)
4. Fussell, J.: How to calculate system reliability and safety characteristics. IEEE Transactions on Reliability 24, 169–174 (1975)
5. Goldberg, D.: Genetic Algorithms in Search and Optimization. Addison-Wesley, Reading (1989)
6. Holland, J.: Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor (1975)
7. Leou, R.: A new method for unit maintenance scheduling considering reliability and operation expense. International Journal of Electrical Power and Energy Systems 28, 471–481 (2006)
8. Lie, C., Chun, Y.: An algorithm for preventive maintenance policy. IEEE Transactions on Reliability 35, 71–75 (1986)
9. Marseguerra, M., Zio, E.: Optimizing maintenance and repair policies via a combination of genetic algorithms and Monte Carlo simulation. Reliability Engineering and system safety 68, 69–83 (2000)
10. Montgomery, D.: Design and analysis of experiments. Wiley, Hoboken (2005)
11. Muller, A., Crespo Marquez, A., Iung, B.: On the concept of e-maintenance: review and current research. Reliability Engineering and system safety 93, 1165–1187 (2008)
12. Samrout, M., Yalaoui, F., Chatelet, E., Chebbo, N.: New methods to minimize the preventive maintenance cost of series–parallel systems using ant colony optimization. Reliability Engineering and system safety 89, 346–354 (2005)
13. Shieh, H., May, M.: Solving the capacitated clustering problem with genetic algorithms. Journal-Chinese Institute of Industrial Engineers 18, 1–12 (2001)
14. Tsai, Y., Wang, K., Teng, H.: Optimizing preventive maintenance for mechanical components using genetic algorithms. Reliability Engineering and system safety 74, 89–97 (2001)

15. Tsai, Y., Wang, K., Tsai, L.: A study of availability-centered preventive maintenance for multi-component systems. Reliability Engineering and system safety 84, 261–270 (2004)
16. Wang, H.: A survey of maintenance policies of deteriorating systems. European Journal of Operational Research 139, 469–489 (2002)
17. Rausand, M., Hoyland, A.: System reliability theory: models, statistical methods, and applications. Wiley-IEEE (2004)

A New Hybrid Parallel Algorithm for MrBayes

Jianfu Zhou, Gang Wang, and Xiaoguang Liu

Nankai-Baidu Joint Laboratory, Nankai University, Tianjin, China
jugeombu@gmail.com, wgzwp@163.com, liuxg74@yahoo.com.cn

Abstract. MrBayes, a popular program for Bayesian inference of phylogeny, has not been fast enough for Biologists when dealing with large real-world data sets. This paper presents a new parallel algorithm that combines the chain-partitioned parallel algorithm with the chain-parallel algorithm to obtain higher concurrency. We test the proposed hybrid algorithm with the two old algorithms on a heterogeneous cluster. The results show that, the hybrid algorithm actually converts more CPU cores into higher speedup compared with the two control algorithms for all of four real-world DNA data sets, therefore is more practical.

Keywords: MrBayes; hybrid; parallel; algorithm.

1 Introduction

MrBayes is a widely-used program for phylogenetic inference. It uses Bayes's theorem to estimate the posterior probability of a phylogenetic tree, which is called Bayesian inference of phylogeny [1,2]. The posterior probability, although easy to formulate, involves a summation over all trees and, for each tree, integration over all possible combinations of branch lengths and substitution model parameter values. The explicit solution of such a problem is computationally intractable for trees of biologically significance, so heuristics must be used to simplify the problem. MrBayes uses Markov chain Monte Carlo (MCMC) method to approximate the posterior probability of a phylogenetic tree. Standard implementations of MCMC can be prone to entrapment in local optima, so a variant of MCMC, known as Metropolis-coupled MCMC (MC^3), is proposed. It allows peaks in the landscape of trees to be more readily explored. However, both the standard MCMC and the MC3 methods are suffering from the unacceptable execution time when dealing with large data sets. Fortunately, some effective methods are available that can execute MrBayes in parallel. A chain-partitioned parallel algorithm for MrBayes [3] distributes Markov chains among processes. It is based on the fact that a relatively small amount of messages are needed to be exchanged among those chains during a run of MC^3. However, this algorithm has a fatal drawback. It can not break the upper bound of concurrency caused by the typically small number of chains during a run, which is enough for most of real-world applications. Another chain-parallel algorithm exclusively focuses on element-level parallelism in the Phylogenetic Likelihood Functions [5]. Although it solves the problem of the chain-partitioned algorithm, the chain-parallel algorithm has higher interaction overhead. This paper presents a new hybrid parallel algorithm for MrBayes, which tries to combine the advantages of the chain-partitioned and chain-parallel algorithms. Then, this paper empirically compares the proposed algorithm with the two previous algorithms on four real-world DNA data sets.

2 The Hybrid Algorithm

2.1 The Chain-Partitioned Parallel Algorithm

Metropolis-coupled MCMC (MC^3) algorithm runs one cold Markov chain with some heated Markov chains to sample the posterior probability distribution of a phylogenetic tree [2,4]. Relatively small amount of information is exchanged among these chains during a run. So we can distribute these chains among processes, and run them in parallel efficiently [3]. A process just performs all computation associated with chain(s) assigned to it. This algorithm exchanges heat values instead of complete state information between cold chains and heated chains to reduce communication overhead. Another advantage of this algorithm is only local instead of global synchronization is needed.

Considering its data partition method and synchronization mechanism, the chain-partitioned parallel algorithm is very suitable for Message-Passing Interface (MPI) implementing. Of course it can also be implemented in shared memory platforms (multi-core systems) easily and efficiently.

2.2 The Chain-Parallel Algorithm

According to the profiling of MrBayes's execution, the functions *CondLikeDown*, *CondLikeRoot* and *CondLikeScaler* spend more than 85% of the total execution time. Fortunately, the main part of each function is independent vector/matrix operations. So a chain-parallel algorithm that executes these operations by multiple threads in parallel is presented [5]. In this algorithm, the likelihood vector elements (therefore the associated computational work) are distributed over threads evenly. During a run of MrBayes, these three functions are called iteratively. Since the results of *CondLikeDown* or *CondLikeRoot* are used by *CondLikeScaler* in each iteration, and the result of *CondLikeScaler* is used by the next iteration, a global synchronization must be done in each iteration which is the major overhead of this algorithm.

Apparently, the chain-parallel algorithm is more suitable for the shared-memory mode than the message-passing mode.

2.3 The Proposed Hybrid Parallel Algorithm

The proposed hybrid parallel algorithm combines the advantages of the chain-partitioned and the chain-parallel algorithms. First, the chain-partitioned strategy is used, that is, each chain is assigned to a unique process. Then each chain is calculated by its owner process and auxiliary threads using chain-parallel strategy. For load balance, the number of chains (processes) assigned to a physical machine is approximately proportional to its relative computing power. Since cores in a machine are typically homogeneous, computational work in *CondLikeDown*, *CondLikeRoot* and *CondLikeScaler* of a chain is evenly distributed over its owner process and auxiliary threads.

In a word, the proposed algorithm breaks through the concurrency limit of the chain-partitioned algorithm caused by the relatively small number of Markov chains involved in MrBayes. It exploits two-level parallelism to make its concurrency greater than the total number of chains.

3 Experiments

This section compares the performance of the proposed hybrid parallel algorithm with the chain-partitioned and the chain-parallel algorithms on a heterogeneous cluster with a head node and five computation nodes. The head node has one Intel Core i7 920 processor and 2GB*3 of DDR3 1333 memory. The first computation node has one AMD Phenom II X4 945 processor and 2GB*2 of DDR3 1333 memory. The second one has one Intel Core 2 Quad Q6600 processor and 2GB*2 of DDR2 1066 memory. Both the third one and the fourth ones have one Intel Core 2 Duo E4400 processor and 1GB*2 of DDR2 800 memory. The fifth one has one Intel Core i7 920 and 2GB*3 of DDR3 1333 memory. These nodes are connected by gigabit Ethernet. The operating system is Red Hat Enterprise Linux 5. MPICH2 1.1 was used to compile and execute the hybrid algorithm and the chain-partitioned algorithm. For the chain-parallel algorithm program, GCC 4.3 was used. Table 1 shows the DNA data sets used in our experiments. These data sets come from real-world DNA data used in the research projects of the College of Life Sciences, Nankai University. Each data point is the average of 10 executions. Each execution computed 2 runs with 4 chains, which is a typical setting for MrBayes executions. All the executions used the 4by4 nucleotide substitution model and lasted 10000 generations.

We first performed a baseline test. As Fig. 1 on the next page shows, the original serial version of MrBayes runs fastest on the head node. In our experiments, the speedup of a parallel algorithm is defined as the result of the average running time of the original serial version of MrBayes on the head node divided by the average running time of a parallel algorithm program on our heterogeneous cluster for the same problem.

Table 1. Data sets used in our experiments

	Number of taxa	Number of characters
Data set 1	26	1546
Data set 2	37	2238
Data set 3	100	1619
Data set 4	111	1506

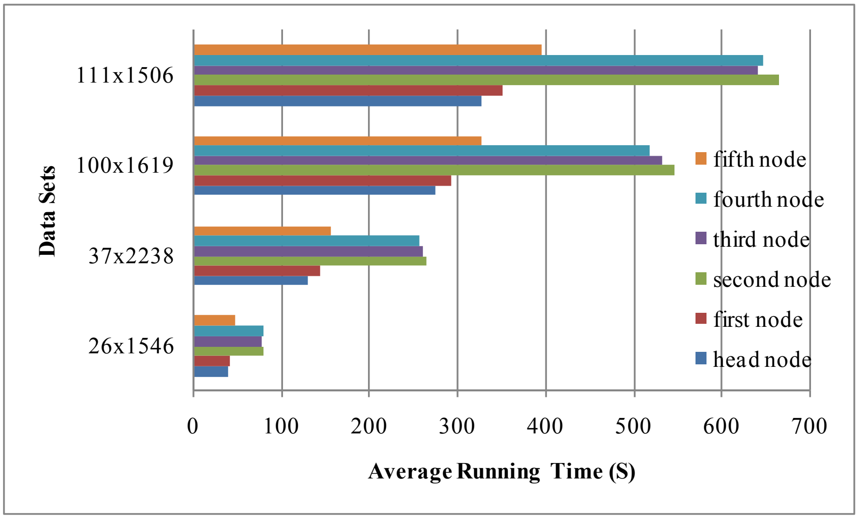


Fig. 1. Performance of different nodes

3.1 Comparison on the Same Number of Cores

Fig. 2 shows the performance of the proposed hybrid parallel algorithm and the chain-parallel algorithm on the head node. For the chain-parallel algorithm, the likelihood vector elements were assigned among 8 threads (including the main thread). Therefore, 8 (logical) CPU cores were used. For the hybrid algorithm, the 8 Markov chains were distributed evenly among 4 processes. Then each

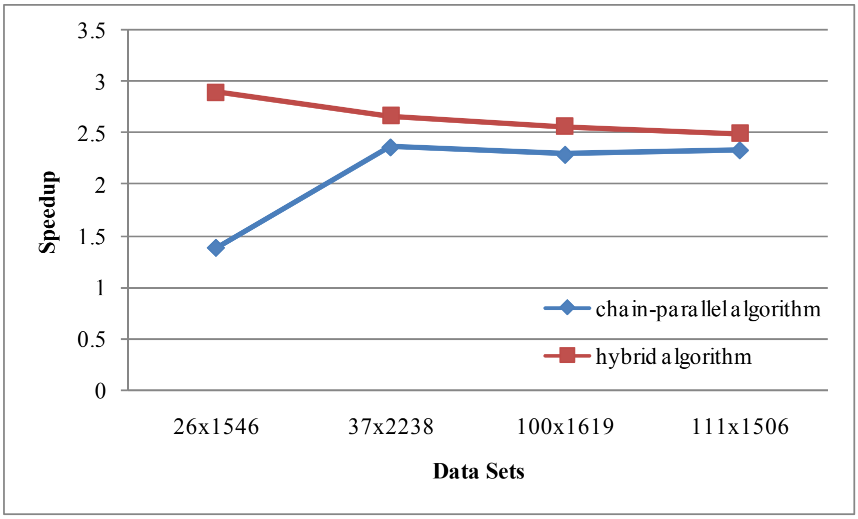


Fig. 2. The hybrid algorithm vs. the chain-parallel algorithm on a single node

process created a child thread. The likelihood vector elements were distributed to the process (namely the main thread) and its child thread. Hence the hybrid algorithm also used 8 cores.

Fig. 3 shows the performance of the proposed hybrid parallel algorithm and the chain-partitioned parallel algorithm. For the chain-partitioned algorithm, the 8 Markov chains were distributed evenly over 8 processes. 4 of these processes were assigned to the head node, and the remaining processes were distributed evenly between the first and the second nodes. For the hybrid algorithm, the 8 Markov chains were distributed evenly over 4 processes, of which 2 were assigned to the head node and the remaining were distributed evenly between the first and the second nodes. Then each process created a child thread and the likelihood vector elements were assigned between the process and its child thread evenly. Therefore, both algorithms used 8 cores.

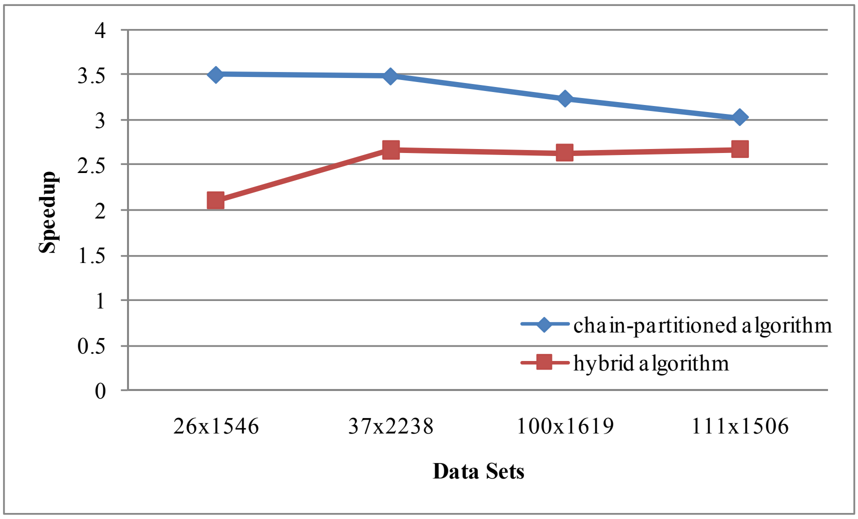


Fig. 3. The hybrid algorithm vs. the chain-partitioned algorithm on a cluster

We can see that the performance of the hybrid algorithm is between the chain-parallel algorithm and the chain-partitioned algorithm. This result is expected. The global synchronization in element-level parallelism causes much higher overhead than the local synchronization in chain-level parallelism. The hybrid algorithm mixes the two parallel formulations, therefore has mixed synchronization, and the other two both have pure synchronization pattern.

3.2 The Hybrid Algorithm on More CPU Cores

Although the chain-partitioned algorithm performs slightly better than the hybrid algorithm in the previous test, as mentioned above, it can not break through the concurrency limit caused by the total number of Markov chains involved. In

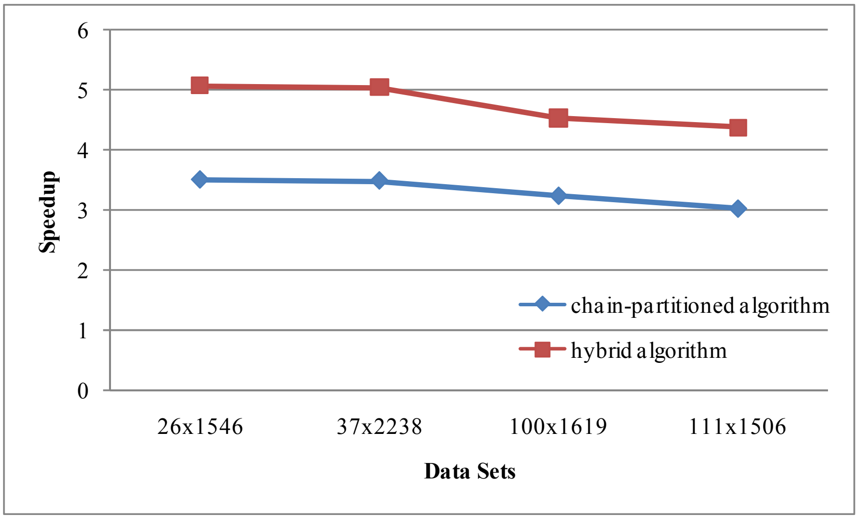


Fig. 4. High concurrency of the hybrid algorithm

our experiments, the total number of chains was 8. When the chain-partitioned algorithm assigned all the chains to 8 processes, it reached its maximum concurrency. There would be no more performance increase even if more processors are available. By contrast, the hybrid algorithm does not have this problem. Even if it has distributed the 8 chains among 8 processes, it can use the element-level parallelism to improve its concurrency. Moreover, the number of the likelihood vector elements is generally large. In our experiments, to achieve higher concurrency, we distributed all the chains among 8 processes, of which 4 were assigned to the head node and the remaining were distributed evenly between the first and the second nodes. Then each process created a child thread. The elements were distributed between the process and its child thread. The hybrid algorithm used 16 threads in total, thus reached the maximum concurrency (namely 16 cores) provided by these nodes. Fig. 4 shows that, the hybrid algorithm indeed uses extra CPU cores effectively and gains higher speedup than the chain-partitioned algorithm.

3.3 Load Balance for the Hybrid Parallel Algorithm

As Fig. 1 on page 105 shows, inside our heterogeneous cluster, the computing power of each node is different with each other. Without considering the difference, it will result in load imbalance and therefore a poor performance.

The relative computing power of cores on nodes is about:

$$Head : First : Second : Third : Fourth : Fifth = 2 : 2 : 1 : 1 : 1 : 2 \ . \qquad (1)$$

Comparison on the same number of cores. Fig. 5 on the following page shows the performance of the hybrid algorithm without or with load balance on the same number of cores. As a control group, the hybrid algorithm without load balance just assigned the 8 Markov chains evenly among 8 processes, which were distributed evenly to the head, the first, the second and the fifth nodes. Then each process created a child thread. The likelihood vector elements were distributed to the process and its child thread. Hence the computation associated with one chain was performed by two threads (namely a process and its child thread), both of which used one core respectively. So, 16 cores were used. We denote the "relative load" (load divided by computing power - roughly the equivalent of the running time) of node X by R_X:

$$\begin{aligned} & R_{head} : R_{first} : R_{second} : R_{fifth} \\ = & \frac{2\ chain}{8\ power} : \frac{2\ chain}{8\ power} : \frac{2\ chain}{4\ power} : \frac{2\ chain}{8\ power} \\ = & 1 : 1 : 2 : 1\ , \end{aligned} \tag{2}$$

which implies serious load imbalance.

The hybrid algorithm with load balance also assigned the 8 Markov chains among 8 processes. Considering the computing power of one core on each node, 3 of the 8 processes were assigned to the head node, 1 to the second node, and the remaining were divided evenly between the first and the fifth nodes. Each of the processes assigned to the head node, the first node and the fifth node just created a child thread respectively. And the process assigned to the second node created 3 child threads. For all of the four nodes, the likelihood vector elements were distributed to the process and its child thread(s). Therefore, for the head node, the first node and the fifth node, the computational work associated with

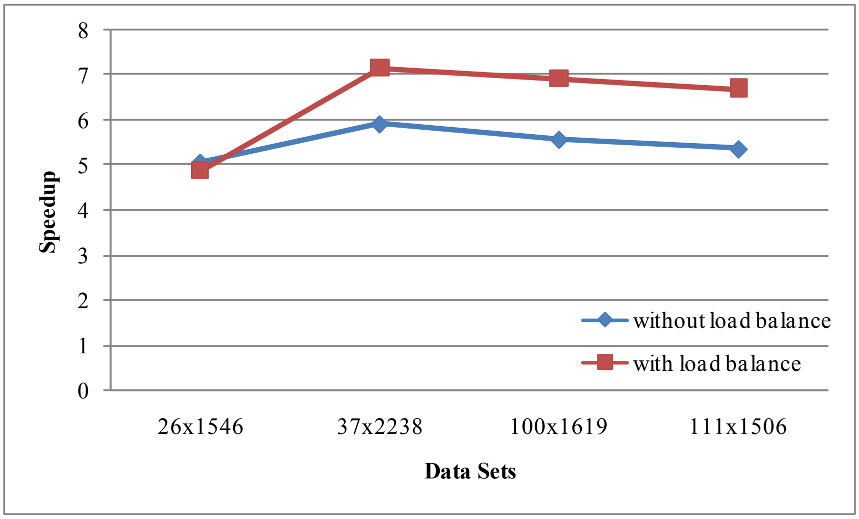


Fig. 5. Performance of the hybrid algorithm with or without load balance on the same number of cores

one chain was performed by two threads, both on one core respectively. For the second node, the computation associated with one chain was performed by four threads, each on one core respectively. Note that, although the head node has only 4 physical cores, it is arguably that each thread run on unique physical core since hyper-threading is supported by Intel i7 CPU. Therefore, this test also used 16 cores, and:

$$\begin{aligned} & R_{head} : R_{first} : R_{second} : R_{fifth} \\ = & \frac{3\ chain}{12\ power} : \frac{2\ chain}{8\ power} : \frac{1\ chain}{4\ power} : \frac{2\ chain}{8\ power} \\ = & 1 : 1 : 1 : 1\ , \end{aligned} \tag{3}$$

which implies approximate load balance.

The result shown in Fig. 5 on the preceding page verifies our analysis. Load balance strategy actually achieves a higher speedup. In particular, when dealing with the relatively large data sets (i.e. data set 3 and 4), load balance strategy increases performance by about 25%.

Comparison on different numbers of cores. Fig. 6 shows the speedup of the hybrid algorithm with or without load balance on different numbers of cores. For the hybrid algorithm without load balance, the 8 Markov chains were assigned evenly among 8 processes, of which 4 were distributed to the head node, and the remaining were divided evenly between the first and the second nodes. Then each process created a child thread. And the likelihood vector elements were distributed to the process and its child thread. So the computational work

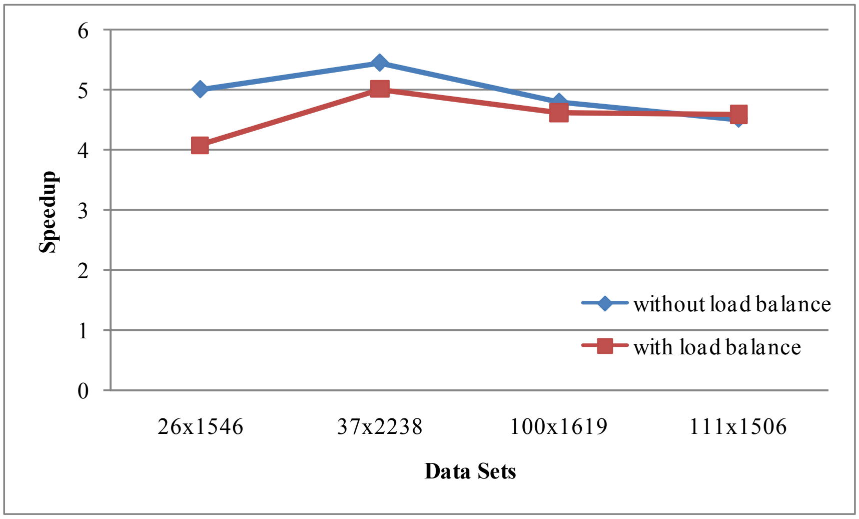


Fig. 6. Performance of the hybrid algorithm with or without load balance on different numbers of cores

associated with each chain was performed by two threads, both of which used one core respectively. Hence 16 cores were used, and:

$$R_{head} : R_{first} : R_{second} = \frac{4\ chain}{16\ power} : \frac{2\ chain}{8\ power} : \frac{2\ chain}{4\ power} = 1 : 1 : 2 \ , \tag{4}$$

which implies serious load imbalance.

For the hybrid algorithm with load balance, the 8 Markov chains were divided into 3 groups. The first group included 4 chains, while the remaining two groups included 2 chains respectively. Then the first group was assigned to the head node, while one of the remaining two groups was distributed to the first node, and the other to the second node. On the head node, the 4 chains were assigned evenly between 2 processes; on the first node, the 2 chains were assigned to 1 process; on the second node, the 2 chains were assigned evenly between 2 processes. And all of these processes created a child thread respectively. The likelihood vector elements were distributed to the process and its child thread. Therefore, on the head node and the first node, each thread group (a process and its auxiliary thread) calculated 2 chains in sequential; on the second node, each thread group calculates only one chain. Each thread run on a unique real core. So the hybrid algorithm with load balance only used 10 cores in total, which was less than the number of cores used by the hybrid algorithm without load balance. And:

$$R_{head} : R_{first} : R_{second} = \frac{4\ chain}{8\ power} : \frac{2\ chain}{4\ power} : \frac{2\ chain}{4\ power} = 1 : 1 : 1 \ , \tag{5}$$

which implies rough load balance.

As Fig. 6 on the previous page shows, although using less physical cores, the hybrid algorithm with load balance achieved approximately the same speedup as the hybrid algorithm without load balance. When dealing with the relatively large data set (i.e. data set 4), the hybrid algorithm with load balance achieves even a little higher speedup than the hybrid algorithm without load balance.

All in all, with load balance, the proposed hybrid parallel algorithm can yield better performance.

4 Related Works

As far as the authors know, except PBPI [7], that conducts multigrain Bayesian inference on the BlueGene/L, no result has been published on hybrid parallelization for MrBayes. PBPI basically represents a proof-of-concept work rather than a production level parallelization. It is not qualified for real-world analyses required urgently by Biologists. However, some results are known for the chain-partitioned parallelization or the chain-parallel parallelization for MrBayes, which both accelerate the execution of MrBayes for large data sets.

Gautam Altekar et al. [3] presented a parallel algorithm for Metropolis-coupled MCMC. This algorithm keeps the advantage to explore multiple peaks in the posterior distribution of trees while getting a shorter running time. This algorithm was implemented using both message passing parallel programming model and shared memory parallel programming model. Experiment results showed speedups in both programming models for small and large data sets.

Frederico Pratas et al. [5] proposed a chain-parallel algorithm for MrBayes and its Phylogenetic Likelihood Functions using different architectures. The experiments compared the scalability and performance achieved using general-purpose multi-core processors, the Cell/BE, and Graphics Processor Units (GPU). The results showed that the general-purpose multi-core processors resulted in the best speedup, and yet GPU and Cell/BE processors both got poor performance because of data transfers and the execution of the serial portion of the code.

5 Conclusion and Future Work

A new hybrid parallel algorithm has been proposed for MrBayes. On our heterogeneous cluster, when using more processors (namely 16 processors), the proposed algorithm without load balance runs maximum 3.67 times faster than the chain-parallel algorithm and maximum 1.447 times faster than the chain-partitioned parallel algorithm on four read-world DNA data sets. With load balance strategy, a further up to 25% performance increment was achieved. Therefore, the proposed hybrid parallel algorithm is very practical for many real biological analyses.

In this paper, a static, manual load balance method was used. Dynamic and automatic load balance algorithms are worth studying in the future. GPU for general purpose computing has shown its great power in many areas. Accelerating MrBayes using GPU is also planned.

Acknowledgement. This work was supported in part by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), RFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000). In order to get the four real-world DNA data sets, in addition to the chance to do this research, we would like to thank Professor Xie Qiang with the College of Life Sciences in Nankai University.

References

1. Huelsenbeck, J.P., Ronquist, F., Nielsen, R., Bollback, J.P.: Bayesian inference of phylogeny and its impact on evolutionary biology. Science 294, 2310–2314 (2001)
2. Huelsenbeck, J.P., Ronquist, F.: MrBayes: Bayesian inference of phylogenetic trees. Bioinformatics 17, 754–755 (2001)
3. Altekar, G., Dwarkadas, S., Huelsenbeck, J.P., Ronquist, F.: Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. Bioinformatics 20, 407–415 (2004)
4. Ronquist, F., Huelsenbeck, J.: MrBayes 3: Bayesian Phylogenetic Inference under Mixed Models. Bioinformatics 19, 1572–1574 (2003)

5. Pratas, F., Trancoso, P., Stamatakis, A., Sousa, L.: Fine-grain Parallelism Using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function. In: The 38th International Conference on Parallel Processing, Vienna, Austria, pp. 9–17 (2009)
6. Stamatakis, A., Ott, M.: Load Balance in the Phylogenetic Likelihood Kernel. In: The 38th International Conference on Parallel Processing, Vienna, Austria, pp. 348–355 (2009)
7. Feng, X., Cameron, K.W., Buell, D.A.: PBPI: a High Performance Implementation of Bayesian Phylogenetic Inference. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 75. ACM Press, New York (2006)
8. Larget, B., Simon, D.: Markov chain Monte Carlo algorithms for the Bayesian analysis of phylogenetic trees. Molecular Biology and Evolution 16, 750–759 (1999)
9. Yang, Z., Rannala, B.: Bayesian phylogenetic inference using DNA sequences: a Markov chain Monte Carlo method. Molecular Biology and Evolution 14, 717–724 (1997)
10. Mau, B., Newton, M., Larget, B.: Bayesian phylogenetic inference via Markov chain Monte Carlo methods. Biometrics 55, 1–12 (1999)
11. Ott, M., Zola, J., Stamatakis, A., Aluru, S.: Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pp. 1–11. ACM Press, New York (2007)

Research and Design of Deployment Framework for Blade-Based Data Center

Haiping Qu[1,2], Xiuwen Wang[1,2], Lu Xu[1], Jiangang Zhang[1], and Xiaoming Han[1]

[1] Institute of Computer Technology, Chinese Academy of Sciences,
100190 Beijing, China
[2] Graduate University of the Chinese Academy of Sciences,
100039 Beijing, China
quhp2314@gmail.com,
{wangxiuwen,xulu,zhangjiangang,hanxiaoming}@nrchpc.ac.cn

Abstract. For those outsourcing data centers hosting different applications or services, blade server is becoming the most popular solution. But the deployment of systems in such data centers will be a time-consuming and cumbersome task. Moreover, the service provided by the servers in data center changed frequently, which require a flexible service management system to reduce the burden of the system administrator. This paper presents the Bladmin (Blade Admin) system to provide dynamic, flexible and quick service and operation system deployment. It can build a Virtual Environment (VE) for each service. All the nodes in the VE serves for one certain kind of application and some nodes have to be shift from one VE to another to adopt the varying workloads. Experiment results demonstrate that system-level performance has been greatly improved and the running of one Paradigm job also show the efficiency of Bladmin framework.

Keywords: blade server, service deployment, virtual environment.

1 Introduction

Large outsourcing data centers that host many third party applications are receiving more and more attention recent years. These applications are mutually independent and require different guarantees of SLA (Service Level Agreement) performance. For such data centers, blade server indisputably ranks the most popular solution with which higher performance in unit cost and unit power as well as unit spatial performance can be obtained. Compared with general 1U frame server, blade server is easier for management and requires fewer TCO cost. Tests show that the processor density of blade server can be increased by four times, while the power consumption can be reduced by 20-30%. Gartner predicted that the production scale of blade will reach 2.30 million by 2011, and its percentage in the whole server market will reach nearly 22%.

With the extension of computation scale increasing, more effective rapid deployment and dispatchment has become the inevitable requirement for full utilization of calculation resources, energy-consumption reduction and cost maintenance as well as the basis to secure the high-efficiency and steady operation for data centers which are constructed with blade servers.

The rest of the paper is organized as follows. In Section 2, we first review the development of blade server, then the related works in the area of service deployment technology. In Section 3, we present the details of system architecture and correlations of software modules. Section 4 presents the work flow of the system through life cycle of VE. Section 5 demonstrates the experiment results in on-spot blade server data center. Concluding remarks and discussion about the future work are given in Section 6.

2 Related Work

2.1 Overview of Blade Server

Blade server refers to high-performance server in which multiple module-type server units are plugged in a frame-style case with standard height in order to achieve high usability and density [1]. Each module-type server unit is known as a blade which has CPU, memory, Ethernet controller and running independent operation system and private application. So a blade is an independent host system in reality.

First-generation blade server was launched in 2000 which aims at ultra-high calculation-intensity. However, to support this intensity, low-power CPU such as PIII and Transmeta Crusoe must be used. So high computing density and poor computing performance are the main characteristics of these blade servers. With the blade called "Blade Center" introduced by IBM in 2003 as the division line, the main characteristics of blade servers later no longer overstressed on calculation density; instead, a proper equilibrium point was selected between calculation density and single-blade performance. Under the precondition of improving calculation density properly, the performance of single blade server was raised; meanwhile, the integration, reliability and manageability of whole system were improved greatly.

2.2 Technology of Service Deployment

Service deployment of data center is a time-consuming and complicated work, especially with the dramatic increase of the number of computers managed by the system administrator. Therefore, the issue of service deployment has become a research focus in academe and industrial community. Related approaches can be classified into disk-mirror techniques and non-disk deployment techniques.

Disk-mirror techniques have become important owing to their simplicity and high efficiency, such as Ghost of Symantec [2], Tivoli of IBM [3] and COD (Cluster-On-Demand) of Duke University [4]. All these related systems generally preserve local storage equipment and accomplish local installation of disk mirrors from network through disk-mirror clone, network multicast and other techniques to realize large-scale and automatic deployment of services. Generally there is a centralized memory at the backstage of these systems for saving original disk mirrors, while the data specifically requested by each client are stored in the local disk of the client. Although client performance has little difference with the general system after deployment with this method, the disk-mirror data are required to be downloaded during deployment to local environment via network before startup, which affects the service deployment rate and increases the time of service switching; the requirements of those application environments requiring frequent dynamic adjustment can not be fully satisfied.

With the development of non-disk technologies and extensive application of networked storage such as NAS and SAN, it has been possible for non-disk service deployment system based on centralized storage to emerge. These systems include emBoot [5], Blutopia [6] and SonD [7]. In this approach, all client system and user data are saved in a backstage storage system in a centralized manner, and system services are initiated and offered to network through non-disk technologies. Its characteristic lies in immediate initiation via network without the process of data download to local environment, which greatly reduces the time for service deployment; meanwhile, centralized data storage facilitates data management and backup, and makes it possible to save storage resources. Besides, data services are offered via network directly, which facilitates dynamic service mapping and binding, and becomes more flexible in comparison with mirror technology.

3 Architecture of the Bladmin System

As a cluster management system developed to solve various problems existing in present blade server clusters, Bladmin (Blade Admin) is an extension version of SonD [7] and can create a VE (Virtual Environment) for each running service and dispatch physical resources (calculating and storing resources) according to catch the quick variation in workloads of service. It meets the SLA targets of service and improves serving capability of the whole system.

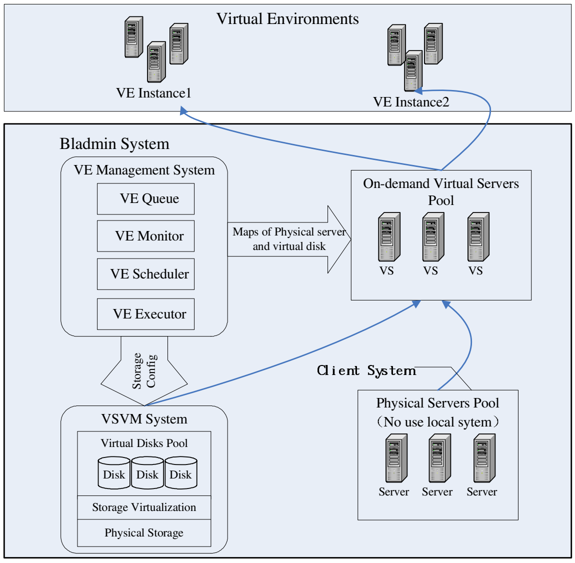


Fig. 1. Bladmin is composed of backstage VSVM System, forestage client system and VEMS

3.1 System Framework

As a typical service deployment management system based on centralized storage, Bladmin is composed of backstage VSVM [8], forestage client system [9] and VEMS (VE Management System), which is shown as Fig. 1. The main functions of VEMS involve monitoring of the whole system, establishment of dynamic mapping relation for the logical volumes in calculation resources and virtual disk pools to construct virtual server, and deploying services dynamically. The main functions of VSVM involve management of storage equipment and provision of network storage services for logical volumes of calculation resources. Calculation resources interact with VEMS during initiation and obtain mapped logical volumes through client system; meanwhile, VSVM exports the logical volume as being accessed to the calculation resources. Thus, one cycle of dynamic service deployment is accomplished.

3.2 VEMS Architecture

Realizing VE management is the major aim of VEMS and it is specifically subdivided into four modules: VE Queue for applying and approving VE; VE Monitor for monitoring VE operation; VE Scheduler for performing decision management on VE operation; VE Executor plays the role of decision execution unit, as shown in Fig. 2.

After a certain time elapsed, VEMS will check all VEs's statuses and judge the service's satisfaction, such a time period is called "control interval". As shown in Fig. 3, *Performance Monitor* acquires and stores original load data of system, and calculates the comprehensive load of current VE according to the load calculation formula of a service. *Workload Forecaster* receives and analyzes current workload

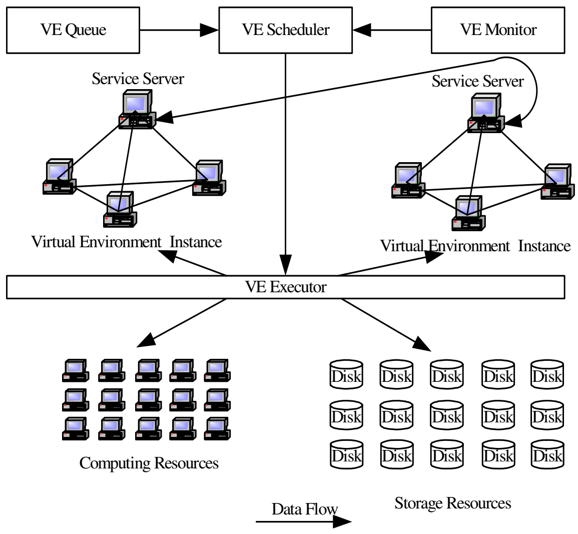


Fig. 2. The architecture of VEMS is subdivided into four modules: VE Queue, VE Monitor, VE Scheduler and VE Executor. Each VE has a Service Server to make scheduling decisions and dispatch requests from the incoming queue onto certain servers in VE.

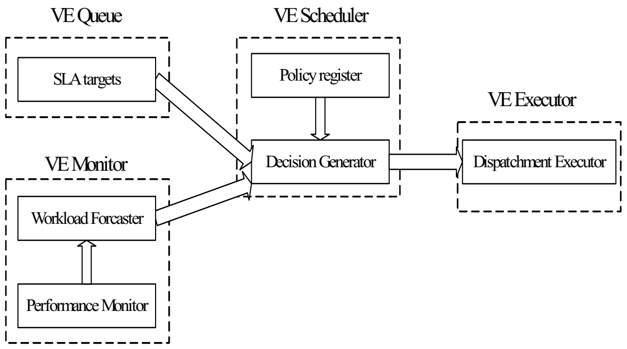


Fig. 3. Correlations of main function modules of VEMS

together with history workload information, and forecasts the load intensity in the next control interval.

Policy register provides an interface for service dispatching policy and defines the upper and the lower load thresholds as well as dispatching grades. The upper and the lower load thresholds are the basis to determine whether VE is overloaded or lightly loaded, and dispatching grades signify whether dynamic adjustment can be performed on the VE as well as adjustment granularity.

Decision Generator calculates the service level for each VE combining its *SLA targets* and comparison of current and future comprehensive load, and evaluates the candidate dispatching instructions according to dispatching grades and sends them concurrently to *Dispatchment Executor*.

Once receives the adjust instructions, *Dispatchment Executor* starts to perform VE adjustment action through change the mapping of blades and VEs.

3.3 Bladmin's Technical Characteristics

In this section, instead of describing Bladmin technicality in detail, only its technical characteristics are roughly introduced.

(1) **Usability.** Dynamic deployment based on IP storage is realized, with which blades are initiated through simulation of SCSI disks via network; supporting various existing OS versions of Windows and Linux, and application layers and file systems are completely compatible.

(2) **High performance.** High-performance backstage snapshot technology enables duplication of 128 service system mirrors within 30s; forestage nodes can achieve inter-VE switching within seconds so that real-time dynamic dispatchment of nodes becomes possible.

(3) **High usability.** Hot-standby deployment server; backstage storage system with support of inside Cache synchronization; heart-beat blade monitoring and support of automatic blade failure switching.

(4) **Expandability.** Modularized framework design and pluggable module design interface; equipment management based on standard SNMP; provision of expandable API programming interface.

4 Life Cycle of VE

The work flow of system is described following the sequence of VE life cycle.

4.1 VE Preparation

(1) **VE application.** Users submit job requests via VE Queue, including service attributes such as blade number, hardware limit, OS and running software for service template, and running time, as well as related dispatching policies registered via policy register.

(2) **VE approval.** Based on current situation of resource utilization and the workload forecast of next control interval, administrators examine the applied jobs via VE Queue and give decisions of consent or rejection.

4.2 VE Construction

VE construction includes three steps: template fabrication, service deployment and concurrent initiation.

(1) **Template fabrication.** The transfer tool provided by Bladmin can copy the software environment in local harddisk to backstage network disk conveniently and set it as a service template. Take Linux for example, the key points of transfer tool we called "ltranfer" include create connection with network disk, load needed modules, get and configure IP address of node, make the new initrd, copy and configure needed OS and applications from harddisk , remake grub and so on.

(2) **Service deployment.** VE Scheduler chooses corresponding blade nodes for prepared jobs and issues construction orders to VE Executor which clones multiple snapshots rapidly according to the service template and deploys the snapshots to blades. This is a mapping action between blades with VEs.

(3) **VE initiation.** VE Scheduler starts blades via network remotely, and the preconfigured application processes begin to work after the system is started according to the deployed relation. Then the service takes into action.

4.3 VE Monitoring

To get the service level of the running VE, VE Monitor need to do three things: monitor the current status of VE, calculate the comprehensive load of VE and forecast the future workload of the service.

VE Monitor adopts the open monitoring framework of Nagios [10] so that administrators can have an overall apprehension and grasp the working state of specific process in specific node. It reports the state information of hosts and services through external plug-in units and can inform administrators by various means (send messages or email) when incidents occur. Failure points can be well analyzed and positioned if a hierarchical resource model can be adopted.

(1) **Access to system load.** Three levels of load data in nodes, VE and data center can be acquired in real time through the monitoring mechanism of Nagios's host and host-group.

(2) **Ensure of service status.** Each VE has given service status to be monitored which specified by the VE user ahead of time. They can be monitored through the monitoring mechanism of Nagios's service and service-group.

To calculate the VE load calculation, a simple calculation mechanism is provided in default to perform weighted average on the CPU load on each node of the VE in order to obtain VE load, and the process authorization on monitored data is opened so that users can add alternative load calculation formulas that can better satisfy demands. To forecast the load intensity, an aggressive first-order method is adopted, which makes prediction based on the workload variation of past two intervals and always tends to assume heavier load in the next interval.

4.4 VE Adjustment

VE Scheduler performs corresponding dynamic adjustment on VE according to the monitored and forecasted data from VE Monitor and specific plug-in unit so as to satisfy specific service SLA targets. VE Scheduler provides an easily-expanding plug-in mechanism which can meet the requirement of specific service.

(1) **VE decision generation.** VE Scheduler provides three decision mechanisms: GROW, SHRINK and DO_NOTHING, and then generates corresponding decisions by comparing system load with upper and lower threshold values.

(2) **VE dispatching mechanism.** VE Scheduler provides two dispatching mechanisms for decision execution: EnforcePolicyHandler and Recommend Policyhandler. The former is responsible for issuing dispatching instructions and carrying out VE adjustment in real time; the latter only issues alerting suggestions to administrators which will decide whether actual adjustment should be performed.

4.5 VE Termination

VE will be normally exited when it reaches the running time submitted in VE application. Also VE will be forced terminated which the end time can be set by both users and administrators. VE termination involves deletion of related snapshots to release backstage memory and deletion of related VE records to release blade nodes.

5 Experiments

In order to evaluate the Bladmin system described in the previous sections, we applied it to blade-based data center in Geophysical Research Institute which belongs to Xinjiang Oil Co.'s Exploration and Development Institute. The system value and validity was verified through two on-spot tests which are VE construction time and running of Paradigm job.

5.1 Environment Construction

The architecture of test environment is shown in Fig. 4, and we can see that there are totally two sets of primary-standby Bladmin servers in the test environment: server1 (server2) and server3 (server4). The primary-standby servers perform

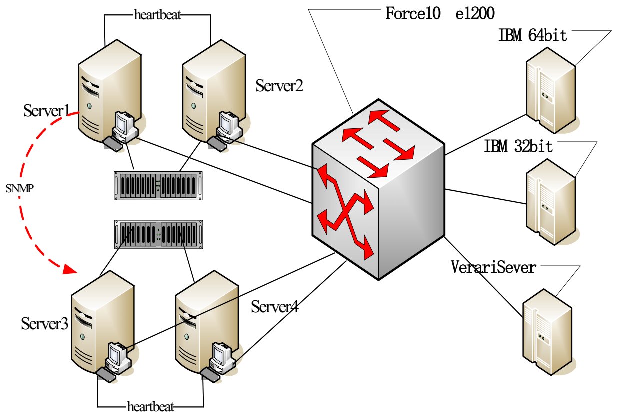


Fig. 4. Architecture of test environment

primary-standby switching through heartbeat when primary server is down, where server1 serves as the managing server and operates server3 via SNMP interface.

All these four servers are HP 1U Proliant DL360 and the configuration can be seen from Table 1. Each set of Bladmin server is configured a LSI 3992 Double Controller Disk Array. The disks are Hitachi FC 400G * 16 and are configured as RAID5 (15 * disks + 1 * Hotspare disk). We use three types of blade servers in the following tests which are IBM 32bit (BladeCenter 8832I1C), IBM 64bit (BladeCenter 8843IFG) and VerariServer(Verari BladeRack 2).

Table 1. Bladmin server configuration

Component	Specific
CPU	Intel Xeon 5310 1.6G*1
Memory	1G * 4
Ethernet Controller	Broadcom C373i Gigabit *3
Disk Subsystem	SAS 73G *2
OS	linux-2.6.17 x86_64

5.2 Test of VE Construction

To isolate applications, nodes are dedicated for a certain VE and to adapt to varying workload, some nodes need to be shift from one VE to another. If it takes too long to complete provisioning before the node can work normally, the SLA targets of service will be violated which leads to low efficiency. The speed of service switch is mainly determined by the time of service deployment, so in this section we run the test of VE construction time to decide whether the Bladmin system can support the quickly service deployment or not.

It can be seen from Section 4.2 that the VE construction includes three steps: template fabrication, service deployment and concurrent initiation. The original environment requires the local systems on nodes to produce mirrors which are put into a mirror server; then, the mirrors are sent out to local disks of nodes via network.

Fig. 5 shows the comparison of the VE construction time under the original environment and the Bladmin environment with 64 nodes of IBM 32bit and 10GB of template scale to deploy.

About 1h was needed to produce a 10GB mirror template under the original environment, and the operation was complex as well. The time for producing template under Bladmin environment was reduced greatly, and transfer of a 10GB local system disk required only around 30min.

The deployment time under the original environment is the time used to distribute mirrors. To 10GB OS image, it needs about 30min when distribute 6 blades of VerariServer and needs about 40min when 64 IBM blades were distributed. The deployment time under Bladmin environment is the time for creating snapshots, and the time for creating 64 snapshots is less than 5min.

Node initiation from local disk costs around 2min without reference to the number of nodes in concurrent initiation. In contrast, data in network disk need to be read under Bladmin environment, and the number of nodes for concurrent initiation may affect the initiation duration. Fig. 6 shows the times for node concurrence under

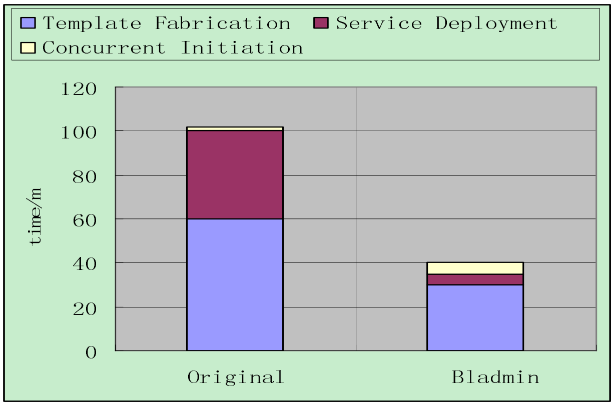


Fig. 5. Difference of VE construction time

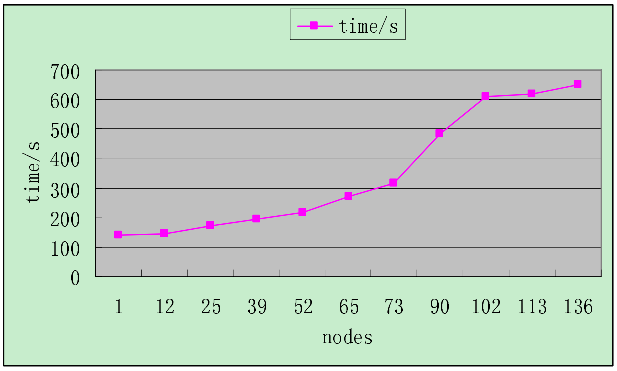


Fig. 6. Boot time for node concurrence under Bladmin

Bladmin, totally eleven groups were tested and increased at the pace of around twelve nodes. It can be seen that concurrent initiation of 64 blades cost around 5min, while simultaneous initiation of over one hundred blades required about 10min.

It can be seen from Fig. 5 and Fig. 6 that even in construction of VE with more than one hundred nodes, the construction rate under Bladmin was increased over one time in comparison with that under the original environment.

5.3 Test of Paradigm Job

For centralized storage, data centralization also leads to centralized system load, which affects system usability and expandability. Accordingly, this section presents Paradigm job (Inline57-160) which ran in the original environment and the Bladmin environment separately for verification.

In Bladmin environment, except for swap and scratch on which local harddisks are mounted, all other programs adopt network harddisks, including around 5.6GB of Paradigm applications in /opt. IBM 64bit and VerariServer blades were choose to run the job and three times of testing are 64 nodes in Original and Bladmin Environments, and 128 nodes in Bladmin Environment which can be seen from Table 2. When the node number was fixed, the time needed in the original and Bladmin was close; under the same Bladmin, the running time for 128 nodes was around one half of that for 64 nodes, and the node number bore a linear relation with the running time. Therefore, Bladmin can fully satisfy the requirement of Paradigm application running.

Table 2. Running time of Paradigm job

	time/h
Original Environment (64 nodes)	30.27
Bladmin Environment (64 nodes)	30.34
Bladmin Environment (128 nodes)	16.38

Fig. 7 depicts part of results of the CPU load of Bladmin server1 when the job ran on 128 nodes. It can be seen that the application mode produced slight pressure on server, and the CPU was generally idle, indicating that Bladmin system can support running of hundred-scale services and has excellent usability and expandability.

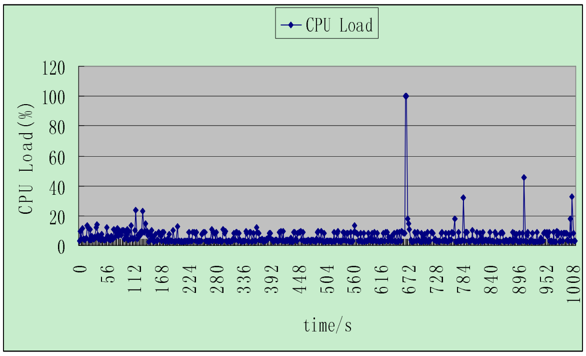


Fig. 7. CPU Load of Bladmin Server

6 Conclusions and Future Work

This paper presents the design, implementation and evaluation of the deployment framework for blade-based data center: Bladmin. It can construct wanted VE based on the job applications submitted by users automatically, rapidly and flexibly, and its dynamic service deployment mechanism improve the overall data center performance by shorting VE deployment time greatly. Its various technical characteristics satisfy the SLA targets of the service while reducing costs for management and maintenance, and improve the resource utilization rate of whole data center.

In the near future, we intend to further develop and deploy our Bladmin system into our blade server finally solution, which integrates multiple existing products of blade server companies. It also needs to further evolution of our Bladmin system both in real-world environment and prototype system. And we plan to integrate more load prediction techniques and further study of system performance model to fit real-world workload more precisely. Bladmin system is to be introduced in the idea of autonomous computing so as to dispatch nodes automatically among services according to service requests and system load, and to enhance the system's capability to counteract dynamic changes in service requests.

Acknowledgments. This work was supported by Major State Basic Research Development Program of China (No. 2004CB318205) and National High Technology Research and Development Program of China (No. 2007AA01Z402 and 2007AA 01Z184). We would also like to thank the anonymous reviewers for their comments.

References

1. Qiu, S., Wang, C., Zhen, Y.: Standardization Analysis of Blade Server. Information Technology and Standardization 11 (2006)
2. SYMANTEC, `http://sea.symantec.com/content/article.cfm?aid=99`
3. Tivoli, `http://www.rembo.com/index.html`
4. Chase, J., Grit, L., Irwin, D., Moore, J., Sprenkle, S.: Dynamic Virtual Clusters in a Grid Site Manager. In: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing(HPDC-12), Seattle, WA (2003)
5. Emboot, `http://www.emboot.com/products_netBoot-i.htm`
6. Oliveira, F., Patel, J., Van Hensbergen, E., Gheith, A., Rajamony, R.: Blutopia: Cluster Life-cycle Management. Technical Report, IBM
7. Liu, Z., Xu, L., Yin, Y.: Blue Whale SonD: A Service-on-Demand Management System. Chinese Journal of Computers 28(7), 1110–1117 (2005)
8. Yin, Y., Liu, Z., Yang, S.: Vsvm-enhanced: a volume manager based on the evms framework. In: Grid and Cooperative Computing Workshops, 2006, pp. 424–431 (2006)
9. Tang, H., Feng, S., Zhang, H.: NVD: the Network Virtual Device for HA-SonD. In: Proceedings of International Workshop on Networking, Architecture, and Storages, Shenyang, China (2006)
10. Nagios, `http://www.nagios.org`

Query Optimization over Parallel Relational Data Warehouses in Distributed Environments by Simultaneous Fragmentation and Allocation

Ladjel Bellatreche[1], Alfredo Cuzzocrea[2], and Soumia Benkrid[3]

[1] LISI/ENSMA Poitiers University,
Futuroscope, France
bellatreche@ensma.fr
[2] ICAR-CNR and University of Calabria,
Cosenza, Italy
cuzzocrea@si.deis.unical.it
[3] National High School for Computer Science (ESI),
Algiers, Algeria
s_benkrid@esi.dz

Abstract. *Parallel database technology* has already shown its efficiency in supporting high-performance *Online Analytical Processing* (OLAP) applications. This scenario implies achieving *query optimization* over *relational Data Warehouses* (RDW) on top of which typical OLAP functionalities, such as roll-up, drill-down and aggregate query answering, can be implemented. As a result, it follows the emerging need for a comprehensive methodology able to support the design of RDW over *parallel and distributed environments* in all the phases, including *data partitioning*, *fragment allocation*, and *data replication*. Existing design approaches have an important limitation: fragmentation and allocation phases are performed in an *isolated manner*. In order to overcome this limitation, in this paper we propose a new methodology for designing parallel RDW over distributed environments, for query optimization purposes. The methodology is illustrated on *database clusters*, as a noticeable case of distributed environments. Contrary to state-of-the-art approaches where allocation is performed *after* fragmentation, in our approach we propose *allocating fragments just during the partitioning phase*. Also, a naive replication algorithm that takes into account the heterogeneous characteristics of our reference architecture is proposed.

1 Introduction

In this paper, we focus the attention to the context of query optimization techniques over *relational Data Warehouses* (RDW) developed on top of *cluster environments* [14]. A RDW is usually modeled by means of a *star schema* consisting of a huge *fact table* and a number of *dimension tables*. *Star queries* are typically executed against RDW. Star queries retrieve aggregate information from *measures* stored in the fact table by applying *selection conditions* on joint dimension table columns, and they are extensively used as conceptual basis for more complex *OLAP queries*, which, in turn, are exploited to extract useful summarized knowledge from RDW for decision making purposes.

Unfortunately, evaluating OLAP queries over RDW typically demands for a high-performance that is difficult to ensure over large amounts of multidimensional data, even because such queries are usually complex in nature [2]. This complexity is mainly due to the presence of joins and aggregation operations over huge fact tables, which very often involve billions of tuples to be accessed and processed. In order to speed-up OLAP queries over RDW, several optimization approaches, mainly inherited from classical database technology, have been proposed in literature. Among others, we recall *materialized views* [12], *indexing* [20], *data partitioning* [5], *data compression* [8] etc. Despite this, it has been demonstrated that the sole use of these approaches singularly is not sufficient to gain efficiency during the evaluation of OLAP queries over RDW [21]. As a consequence, in order to overcome limitations deriving from these techniques, high-performance in database technology, including RDW [11,9], has traditionally been achieved by means of *parallel processing methodologies* [16].

Following this major trend, the most important commercial database systems vendors (e.g., Oracle, IBM, Microsoft, NCR, Sybase etc.) have recently proposed solutions able to support parallelism within the core layer of their DBMS. Unfortunately, these solutions still remain expensive for small and medium enterprises, so that *database cluster technology* represents an efficient low-cost alternative to tightly-coupled multiprocessing database systems [14]. A *database cluster* can be defined as a cluster of personal computers (PC) such that each of them runs an off-the-shelf sequential DBMS [14]. The set of DBMS relying in the cluster are then orchestrated by means of an ad-hoc middleware that implements parallel processing mechanisms and techniques, being this middleware able to support typical DBMS functionalities/services (e.g., storage, indexing, querying etc) in a transparent-for-the-user manner, just like end-users were interacting with a *singleton* DBMS. Starting from this low-cost technology solution, in our research we focus the attention on the application scenario represented by the so-called *parallel relational Data Warehouses* (PRDW) over database clusters, i.e. RDW that are developed on top of a cluster of databases that implements parallel processing mechanisms and techniques. Database clusters are thus assumed to be the reference distributed environment for our research.

Similarly to the traditional context of *distributed and parallel databases* [16], the design of a PRDW on a database cluster can be achieved by means of a *general* design methodology consisting by the following steps: (*i*) fragmenting the input data warehouse schema; (*ii*) allocating the so-generated fragments; (*iii*) replicating fragments in order to ensure high-performance during data management and query evaluation activities. By examining the active literature, a few work on how to design a PRDW on a database cluster exists [14,11]. These approaches can be classified into two main classes. The first class of proposals assume that data are already partitioned and allocated, and propose solutions to route OLAP queries across nodes of the database cluster in order to improve query performance [17,18]. The other class of proposals instead propose solutions to partition and allocate data across database cluster nodes [14]. Most importantly, the majority of approaches devoted to the design of a PRDW over a database cluster assume that all nodes of the cluster are *homogenous*, i.e. they have the same *processing power* and *storage capacity.* By looking at the peculiarities of the target application scenario, it is easy to understand how this assumption is not always true, as a cluster of PC with heterogeneous characteristics in terms of storage and processing capacity may exist. Therefore, it clearly follows the interest for PRDW design methodologies over database clusters characterized by *heterogeneous* nodes, which is the main goal of our research.

Data fragmentation[1] is a fundamental phase of any PRDW design methodology, and can also be considered as a *pre-condition* for PRDW design [1]. Data fragmentation can be of the following two kinds [16]: (*i*) *horizontal fragmentation*, according to which table instances are decomposed into *disjoint partitions*; (*ii*) or *vertical fragmentation*, according to which table instances are split into *disjoint sets of attributes.* Horizontal partitioning is the most popular solution used to design PRDW [1,21,22,11,14]. In previous PRDW design methodologies research efforts, horizontal partitioning algorithms do not control the number of generated fragments, except [1,6]. As a consequence, the number of fragments generated by the partitioning phase can be larger than the number of nodes of the database cluster. In turn, this causes flaws in the allocation and replication phases.

Allocation is the phase that places fragments generated by the partition phase across nodes of the database cluster. Allocation can be either *redundant*, i.e. with replication, or *non redundant*, i.e. without replication [16]. Some literature approaches advocate a *full replication* in order to ensure a high *intra-query parallelism* [14]. This solution demands for the availability of very large amounts of disk space, as each node must be ideally able to house the *entire* data warehouse. As a consequence, data updates become prohibitively expensive. On the basis of this main observation, we assert that replication must be *partial*, meaning that database cluster nodes house *portions* of the original data warehouse. Once fragments are placed and replicated, global OLAP queries against the target PRDW are re-written over fragments and evaluated on the *parallel machine.*

State-of-the-art PRDW design methodologies on database clusters proposals suffer from the following two main limitations. First, they focus the attention on homogenous database clusters, i.e. database clusters where nodes have the same *processing power* and *storage capacity.* Second, fragmentation and allocation phases are usually performed in an *isolated manner* (or *iteratively*), meaning that the designer first partitions his/her data warehouse using his/her favorite fragmentation algorithm and then allocates generated fragments on the parallel machine using his/her favourite allocation algorithm. This approach completely ignores the inter-dependency between fragmentation and allocation phases, which, contrary to this, can instead seriously affect the final performance of data management and OLAP query evaluation activities performed against

[1] In this paper, we use the terms “fragmentation” and “partitioning” interchangeably.

the PRDW. Starting from these breaking evidences, in this paper we propose and experimentally assess an innovative methodology for designing PRDW on database clusters overtaking the limitations above. To the best of our knowledge, our research is the first one in literature that addresses the issue of designing PRDW on heterogeneous database clusters via a combined fragmentation / allocation strategy.

2 Related Work

In this Section, we provide a brief overview on state-of-the-art approaches focusing on fragmentation and allocation techniques for supporting PRDW over database clusters [11,14,17,18].

Furtado [11] discusses partitioning strategies for *node-partitioned data warehouses*. The main suggestion coming from [11] 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 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. The fragmentation approach [11] does not take into account specific star query requirements, being such queries very often executed against data warehouses, and it does not consider the critical issues of controlling the number of generated fragments, like in [5,22].

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 plays a critical role in data allocation, and that, in this respect, 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 de-clustering 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. Just like [11], [14] does not control the number of generated fact table fragments.

To summarize, the most relevant-in-literature approaches related to our research are mainly oriented towards the idea of performing the fragmentation and allocation phases over database clusters in an isolate and iterative manner.

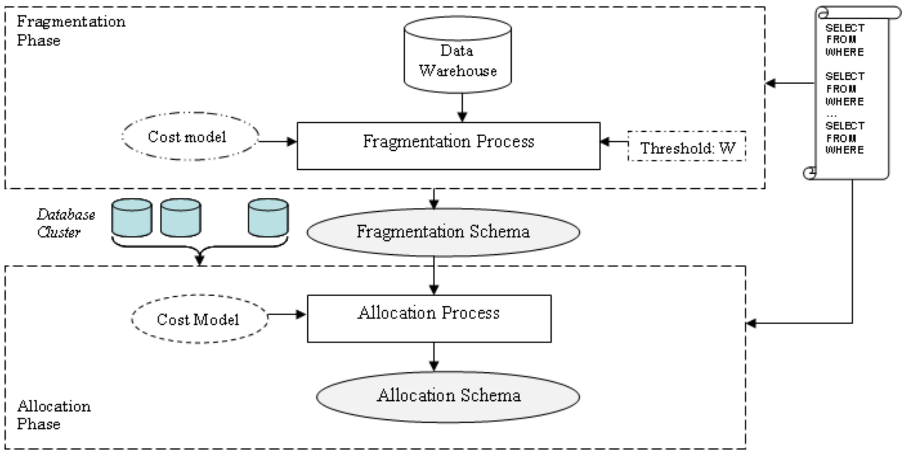


Fig. 1. Iterative PRDW Design over Heterogeneous Database Clusters

3 PRDW Design over Heterogeneous Database Clusters

In this Section, we introduce a rigorous formalization of the PRDW design problem on heterogeneous database clusters, which will be used as reference formalism throughout the paper. Formally, given:

- a data warehouse schema $\mathcal{DWS}$ composed by d dimension tables $\mathcal{D} = \{D_0, D_1, \ldots, D_{d-1}\}$ and one fact table $\mathcal{F}$ – as in [11,14], we suppose that all dimension tables are replicated over the nodes of the database cluster and are fully-available in main memories of cluster nodes;
- a database cluster machine $\mathcal{DBC}$ with M nodes $\mathcal{N} = \{N_0, N_1, \ldots, N_{M-1}\}$, each node N_m, with $0 \leq m \leq M-1$, having a *proper* storage S_m and *proper* processing power P_m, which is modeled in terms of the number of operations that N_m can process in the reference temporal unit;
- a set of star queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_{L-1}\}$ to be executed over $\mathcal{DBC}$, being each query Q_l, with $0 \leq l \leq L-1$, characterized by an *access frequency* f_l;
- 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 PRDW described by $\mathcal{DWS}$ over the heterogeneous database cluster $\mathcal{DBC}$ consists in *fragmenting the fact table $\mathcal{F}$ into N_F fragments and allocating them over different $\mathcal{DBC}$ nodes such that the total cost of executing all queries in $\mathcal{Q}$ can be minimized while storage and processing constraints are satisfied across nodes in $\mathcal{DBC}$, under the maintenance constraint $\mathcal{W}$.*

Based on the formal statement above, it follows that our investigated problem is composed of two sub-problems, namely data partitioning and fragment allocation. Each one of these problems is known to be *NP-complete* [5,19,13]. In order to deal with the PRDW design problem over database clusters, two main classes of methodologies are possible: *iterative design* methodologies and *combined design* methodologies. Iterative design methodologies have been proposed in the context of traditional distributed and parallel database design research. The idea underlying this class of methodologies consists in first fragmenting the RDW using *any* partitioning algorithm, and then allocating the so-generated fragments by means of *any* allocation algorithm. In 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 they are straightforwardly applicable to a large number of even-heterogenous parallel and distributed environments (e.g., *Peer-to-Peer Databases*). Contrary to this, their main limitation is represented by the the fact they neglect the inter-dependency between the data partitioning and the fragment allocation phase, respectively. Figure 1 summarizes the steps of iterative design methodologies.

To overcome limitations deriving from using iterative design methodologies, the combined design methodology we propose in our research consists in *performing the allocation phase/decision at fragmentation time, in a simultaneous manner*. Figure 2 illustrates the steps of our approach. Contrary to the iterative approach that uses two cost models (i.e., one for the fragmentation phase, and one for the allocation phase), the proposed combined approach uses only one cost model that monitors whether the *current* generated fragmentation schema is "useful" for the *actual* allocation process.

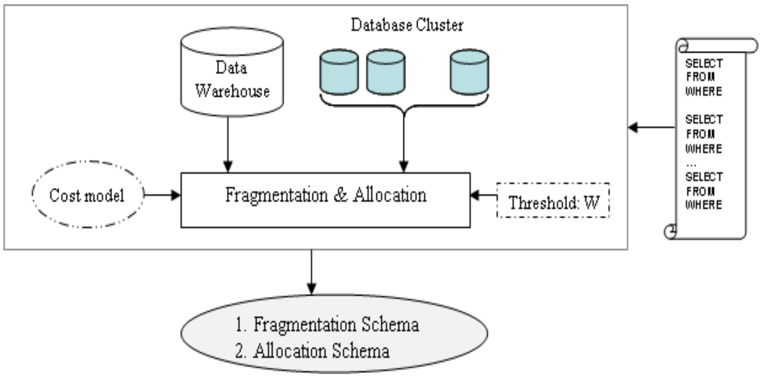


Fig. 2. Combined PRDW Design over Heterogeneous Database Clusters

4 A Combined PRDW Design Methodology over Heterogeneous Database Clusters

In this Section, we describe in detail our combined PRDW design methodology over heterogeneous database clusters. We first focus the attention on the critical aspect represented by the data partitioning phase, which, as stated in Section 1, is a fundamental and critical phase for any PRDW design methodology [1]. A particularity of our proposed methodology is represented by the fact that, similarly to [1,6], it allows the designer to control the number of generated fragments, which should be a mandatory requirement for *any* PRDW design methodology in cluster environments (see Section 1). Then, we move the attention on data allocation issues and, finally, we provide the main algorithm implementing our proposed methodology.

4.1 Data Partitioning

In our proposed methodology, we make use of horizontal (data) partitioning, which can be reasonably considered as the core of our PRDW design. Specifically, our data partitioning approach consists in fragmenting dimension tables D_j in $\mathcal{D}$ by means of *selection predicates* of queries in $\mathcal{Q}$, and then using the so-generated *fragmentation schemes*, denoted by $\mathcal{FS}(D_j)$, to partition the fact table $\mathcal{F}$. Formally, a selection predicate is of kind: $A_k\ \theta\ V_k$, such that: (*i*) A_k models an attribute of a dimensional table D_j in $\mathcal{D}$; (*ii*) V_k models an attribute value in the *universe of instances* of $\mathcal{DWS}$; (*iii*) θ models an *equality or comparison predicate* among attributes/attribute-values, i.e. $\theta \in \{=, <, >, \leq, \geq\}$. The fact table partitioning method that derives from this approach is known-in-literature under the term "*referential partitioning*", which has recently been incorporated within the core layer of the DBMS platform *Oracle11G* [10].

Based on the data partitioning approach above, the number of fragments N_F generated from the fact table $\mathcal{F}$ is given by the following expression:

$N_F = \prod_{j=0}^{d-1} \Phi_j$, such that Φ_j, with $0 \leq j \leq d-1$, denotes the number of horizontal fragments of the dimension table D_j in $\mathcal{D}$, and d denotes the number of dimension tables in $\mathcal{DWS}$. Such a decomposition of the fact table may generate a large number of fragments [21,3,5].

4.2 Naive Solution

In our proposed PRDW design methodology on database clusters, we introduce the concept of *fragmentation scheme candidate* of a dimensional table D_j in $\mathcal{D}$, denoted by $\mathcal{FS}_C(D_j)$. Intuitively enough, a fragmentation scheme candidate is a fragmentation scheme generated during the execution of the algorithm implementing the proposed methodology and that *may belong* to the final solution represented by the set of N_F fact-table fragments allocated across nodes of the target database cluster.

A critical role in this respect is played by the solution used to represent-in-memory fragmentation scheme candidates as this, in turn, impacts on the performance of the proposed algorithm. In our implementation, given a dimensional table D_j in $\mathcal{D}$, we model a fragmentation scheme candidate of D_j as a *multi-dimensional array* $\mathcal{A}_j$ such that rows in $\mathcal{A}_j$ represent so-called *fragmentation attributes* of the partitioning process (namely, attributes of D_j), and columns in $\mathcal{A}_j$ represent *domain partitions* of fragmentation attributes. Given an attribute A_k of D_j, a domain partition $\mathcal{P}_D(A_k)$ of A_k is a partitioned representation of the domain of A_k, denoted by $Dom(A_k)$, into *disjoint sub-domains* of $Dom(A_k)$, i.e. $\mathcal{P}_D(A_k) = \{dom_0(A_k), dom_1(A_k), \ldots, dom_{|\mathcal{P}_D(A_k)|-1}(A_k)\}$, such

that $dom_h(A_k) \subseteq Dom(A_k)$, with $0 \leq h \leq |\mathcal{P}_D(A_k)| - 1$, denotes a sub-domain of $Dom(A_k)$, and the following property holds:
$\forall\ h_p, h_q : h_p \neq h_q,\ dom_{h_p}(A_k) \bigcap dom_{h_q}(A_k) = \emptyset$. Given an attribute A_k of D_j, a number of alternatives for generating a domain partition $\mathcal{P}_D(A_k)$ of $Dom(A_k)$ exist. Among all the available solutions, in our proposed methodology we make use of the set of queries $\mathcal{Q}$ to this end (see Section 3). Coming back to the structural definition of $\mathcal{A}_j$, each cell of $\mathcal{A}_j$, denoted by $\mathcal{A}_j[k][h]$, stores an integer value that represents the number of attribute values of A_k belonging to the sub-domain $dom_h(A_k)$ of $Dom(A_k)$. It is a matter of fact to notice that $\mathcal{A}_j[k][h] \in [0 : |\mathcal{P}_D(A_k)|]$.

Based on the multidimensional representation model for fragmentation scheme candidates above, for each dimension table D_j in $\mathcal{D}$, the final fragmentation scheme of D_j, $\mathcal{FS}(D_j)$, is generated according to the following semantics:

- all cells in $\mathcal{A}_j$ of a fragmentation attribute A_k of D_j have *different* values $\mathcal{A}_j[A_k][h]$, then all sub-domains of $Dom(A_k)$ will be used to partition D_j;
- all cells in $\mathcal{A}_j$ of a fragmentation attribute A_k of D_j have the *same* value $\mathcal{A}_j[A_k][h]$, then the attribute A_k will not participate to the fragmentation process;
- a sub-set of cells in $\mathcal{A}_j$ of a fragmentation attribute A_k of D_j have the *same* value $\mathcal{A}_j[A_k][h]$, then the corresponding sub-domains of $Dom(A_k)$ will be merged into one sub-domain only, and then used to partition D_j.

Based on the formal model of fragmentation scheme candidates above, the naive solution to the PRDW design problem over database clusters we propose, which represents a first attempt of the algorithm implementing our methodology, makes use of a *hill climbing heuristic*, which consists of the following two steps [4]:

1. find an initial solution $\mathcal{I}_0$ – $\mathcal{I}_0$ may be obtained via using a *random* distribution for filling cells of fragmentation scheme candidates $\mathcal{A}_j$ of dimensional tables D_j in $\mathcal{D}$;
2. iteratively improve the initial solution $\mathcal{I}_0$ by using the hill climbing heuristic until no further reduction in the total query processing cost due to evaluating query in $\mathcal{Q}$ can be achieved, and the storage and processing constraints are satisfied, under the maintenance constraint $\mathcal{W}$.

It should be noted that, since the number of fragmentation scheme candidates generated from $\mathcal{DWS}$ is finite, the hill climbing heuristic will *always* complete its execution, thus finding the final solution $\mathcal{I}_F$.

On the basis of these operators running on fragmentation schemes of dimensional tables, the hill climbing heuristic still finds the final solution $\mathcal{I}_F$, while the total query processing cost can be reduced and the maintenance constraint $\mathcal{W}$ can be satisfied.

4.3 Data Allocation

The data allocation phase of our proposed PRDW design methodology on database clusters is performed simultaneously to the data fragmentation/

partitioning phase. Basically, each fragmentation scheme candidate generated by the algorithm implementing our methodology is allocated across nodes of the target database cluster, with the goal of minimizing the total query processing cost over *all* nodes, while satisfying the storage and processing constraints on *each* node. In more detail, during the allocation phase the following concepts/data-structures are used:

- *Fragment Placement Matrix* (FPM) $\mathcal{M}_P$, which stores the positions of a fragment across nodes (recall that fragment replicas may exist). To this end, $\mathcal{M}_P$ rows model fragments, whereas $\mathcal{M}_P$ columns model nodes. $\mathcal{M}_P[i][m] = 1$, with $0 \leq i \leq N_F - 1$ and $0 \leq m \leq M - 1$, if the fragment F_i is allocated on the node N_m, otherwise $\mathcal{M}_P[i][m] = 0$.
- *Fragment Size* $Size(F_i)$, which models the size of the fragment F_i in terms of the number of its instances across the nodes. $Size(F_i)$ is estimated by means of selection predicates. Since each node N_m in $\mathcal{N}$ has its own storage capacity S_m, the storage constraint associated to F_i across all nodes of the target database cluster can be formally expressed as follows:

$$\forall m \in [0 : M - 1] : \sum_{i=0}^{N_F - 1} \mathcal{M}_P[i][m] \times Size(F_i) \leq S_m \qquad (1)$$

- *Fragment Usage Matrix* (FUM) [15] $\mathcal{M}_U$, which models the "usage" of fragments by queries in $\mathcal{Q}$. To this end, $\mathcal{M}_U$ rows model queries, whereas $\mathcal{M}_U$ columns model fragments. $\mathcal{M}_U[l][i] = 1$, with $0 \leq l \leq L - 1$ and $0 \leq i \leq N_F - 1$, if the fragment F_i is involved by the query Q_l, otherwise $\mathcal{M}_U[l][i] = 0$. An *additional* column is added to $\mathcal{M}_U$ for representing the access frequency f_l of each query Q_l (see Section 3). In order to evaluate a query Q_l in $\mathcal{Q}$ on a node N_m in $\mathcal{N}$, N_m must store *relevant fragments* for Q_l. Based on our theoretical framework, a fragment F_i is relevant iff the following property holds: $\mathcal{M}_P[i][m] = 1 \wedge \mathcal{M}_U[l][i] = 1$, with $0 \leq i \leq N_F - 1$, $0 \leq m \leq M - 1$ and $0 \leq l \leq L - 1$.
- *Fragment Affinity Matrix* (FAM) $\mathcal{M}_A$, which models the "affinity" between two fragments F_{i_p} and F_{i_q}. To this end, $\mathcal{M}_A$ rows and columns both model fragments, hence $\mathcal{M}_A$ is a symmetric matrix. $\mathcal{M}_A[i_p][i_q]$, with $0 \leq i_p \leq N_F - 1$ and $0 \leq i_q \leq N_F - 1$, stores the sum of access frequencies of queries in $\mathcal{Q}$ that involve F_{i_p} and F_{i_q} simultaneously.

4.4 PRDW Design Algorithm

On the basis of the data partitioning phase and the data allocation phase described in Section 4.1 and Section 4.4, respectively, and the naive solution and improved solution to the PRDW design problem over database clusters provided in Section 4.2 and Section 4.3, respectively, for each fragmentation scheme candidate $\mathcal{FS}_C(D_j)$ of each dimensional table D_j in $\mathcal{D}$, the algorithm implementing our proposed methodology performs the following steps:

1. Based on the FUM $\mathcal{M}_U$ and the FAM $\mathcal{M}_A$, generate groups of fragments G_z by means of the method presented in [15].

2. Compute the size of each fragment group G_z, as follows: $Size(G_z) = \sum_i Size(F_i)$, such that $Size(F_i)$ denotes the size of the fragment F_i.
3. Sort nodes in the target database cluster $\mathcal{DBC}$ by descendent ordering based on their storage capacities and processing power.
4. Allocate "heavy" fragment groups on *powerful nodes* in $\mathcal{DBC}$, i.e. nodes with high storage capacity and high processing power, in a *round-robin* manner starting from the first powerful node. The allocation phase must *minimize* the total query processing cost due to evaluating queries in $\mathcal{Q}$ while *maximizing* the *productivity* of each node, based on the following theoretical formulation:

$$max \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} \sum_{i=0}^{N_F-1} \mathcal{M}_U[l][i] \times \mathcal{M}_P[i][m] \times Size(F_i) \tag{2}$$

   such that: (i) L denotes the number of queries against $\mathcal{DBC}$; (ii) M denotes the number of nodes of $\mathcal{DBC}$; (iii) N_F denotes the number of fragments belonging to the solution; (iv) $\mathcal{M}_U$ denotes the FUM; (v) $\mathcal{M}_P$ denotes the FPM; (vi) $Size(F_i)$ denotes the size of the fragment F_i. In formula (2), we implicitly suppose that the response time of any arbitrary query Q_l in $\mathcal{Q}$ is superiorly bounded by the time needed to evaluate Q_l against the *most-loaded node* in $\mathcal{DBC}$, thus we can consider it as a constant and omit it in formula (2).
5. Replicate on non-powerful nodes groups of fragments that require high computation time, in order to ensure a high performance.

5 Conclusions and Future Work

In this paper, we have introduced an innovative PRDW design methodology on distributed environments, while adopting database clusters as a specialized case. The final goal of our research consists in devising query optimization solutions for supporting resource-intensive OLAP over such environments. The proposed methodology encompasses a number of advancements over state-of-the-art similar approaches, particularly (i) the fact it considers heterogeneous cluster nodes, i.e. nodes having heterogeneous storage capacities and processing power, and (ii) the fact it performs the fragmentation and allocation phases simultaneously. This methodology can be applied in a various environments such as parallel machines, distributed databases, etc. Future work is mainly oriented towards performing comprehensive experimental studies on both synthetic and real-life data sets in order to show the efficiency of our proposed design methodology, and making it able to deal with next-generation *Grid Data Warehouse Environments* [7].

References

1. 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)

2. Bellatreche, L., Boukhalfa, K.: An evolutionary approach to schema partitioning selection in a data warehouse environment. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 115–125. Springer, Heidelberg (2005)
3. Bellatreche, L., Boukhalfa, K., Abdalla, H.I.: Saga: A combination of genetic and simulated annealing algorithms for physical data warehouse design. In: Bell, D.A., Hong, J. (eds.) BNCOD 2006. LNCS, vol. 4042, pp. 212–219. Springer, Heidelberg (2006)
4. Bellatreche, L., Boukhalfa, K., Richard, P.: Referential horizontal partitioning selection problem in data warehouses: Hardness study and selection algorithms. International Journal of Data Warehousing and Mining 5(4), 1–23
5. 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)
6. Cuzzocrea, A., Darmont, J., Mahboubi, H.: Fragmenting very large xml data warehouses via k-means clustering algorithm. International Journal of Business Intelligence and Data Mining 4(3-4), 301–328 (2009)
7. Cuzzocrea, A., Kumar, A., Russo, V.: Experimenting the query performance of a grid-based sensor network data warehouse. In: Hameurlain, A. (ed.) Globe 2008. LNCS, vol. 5187, pp. 105–119. Springer, Heidelberg (2008)
8. Cuzzocrea, A., Serafino, P.: LCS-hist: taming massive high-dimensional data cube compression. In: 12th International Conference on Extending Database Technology, EDBT 2009 (2009)
9. DeWitt, D.J.D., Madden, S., Stonebraker, M.: How to build a high-performance data warehouse, http://db.lcs.mit.edu/madden/high_perf.pdf
10. Eadon, G., Chong, E.I., Shankar, S., Raghavan, A., Srinivasan, J., Das, S.: Supporting table partitioning by reference in oracle. In: SIGMOD 2008 (2008)
11. Furtado, P.: Experimental evidence on partitioning in parallel data warehouses. In: DOLAP, pp. 23–30 (2004)
12. Gupta, H.: Selection and maintenance of views in a data warehouse. Technical report, Stanford University (1999)
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.: Improving parallel olap query processing in database clusters with data replication. Distributed and Parallel Database 25(1-2), 97–123 (2009)
15. Navathe, S.B., Ra, M.: Vertical partitioning for database design: a graphical algorithm. ACM SIGMOD, 440–450 (1989)
16. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (1999)
17. 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)
18. 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)
19. Saccà, D., Wiederhold, G.: Database partitioning in a cluster of processors. ACM Transactions on Database Systems 10(1), 29–56 (1985)

20. Sarawagi, S.: Indexing olap data. IEEE Data Engineering Bulletin 20(1), 36–43 (1997)
21. 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)
22. 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)

Function Units Sharing between Neighbor Cores in CMP

Tianzhou Chen, Jianliang Ma, Hui Yuan, Jingwei Liu, and Guanjun Jiang

College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, 310027, China
{tzchen,majl,libbug}@zju.edu.cn,
{richieln,ljw850216}@gmail.com

Abstract. Program behaviors reveal that programs have different sources requirement at different phases, even at continuous clocks. It is not a reasonable way to run different programs on constant hardware resources. So sharing feasible degree of hardware may get more benefits for programs.

This paper proposes architecture to share function units between neighbor cores in CMP to improve chip performance. Function units are central units on the core, it take little area and is not the performance critical part of core, but improving function units' utilization can improve other units' efficiency and core performance. In our design, share priority guarantees the local thread would not be influenced by threads in neighbor cores. Share latency is resolved by early share decision made and direct data path. The evaluation shows that the proposal is good for function unit intensive program and can drive other units more efficient.

1 Introduction

With the development of electric industry, more and more transistors can be used and the processor design is more complex. Many kinds of technologies are used to improve program performance, so that performance improvement becomes saturation. Single thread execution is not efficient enough for modern processors and processors try to execute multi-thread at the same time.

SMT (Simultaneous Multi-Threading) share hardware resources aggressively between threads, but meanwhile brings more resource competition. Instructions from more than one thread can be executed in any given pipeline stage at a time. SMT can share hardware to more than one thread and improve processor throughput. But concurrent threads, sharing the same on chip memory system, causes access conflicts and cache pollution. Bus and function units competition need extra manager unit. SMT increases chip design complexity. The number of concurrent thread is decided by chip designer, and chip complexity has limited the number to two for most SMT implementations.

CMP (Chip Multi-Processor) separates chip resources into cores to implement thread isolation, but cannot improve performance for single thread application effectively. CMP integrates more than one core to a chip. It can simplify chip design complexity, decrease chip frequency and voltage and implement real parallelism. But

CMP is not fit for legacy code very well. The legacy code bases on single thread processor and is executed in sequence. It cannot make use of the parallelism benefits of CMP. Meanwhile CMP usually has lower frequency and simpler architecture for each core. So CMP may get performance degradation for single thread.

SMT tries to share vast hardware resource between threads and CMP isolates threads on the chip from each other. These two technologies are two extreme of chip resource sharing to pursue high utilization ratio and isolation to pursue high parallelism. How to combine advantages and avoid flaws of SMT and CMP is an interesting topic.

FU (Function Unit) utilization ratio reflects chip resource usage indirectly. In a robust processor, different units should be able to consist with each other. For example, instruction fetch unit should consist with execution and commit unit; branch prediction unit should consist with branch rollback and execution unit; namely, it is unreasonable for a processor to have a wild instruction fetch bandwidth, but only few instruction execution unit, so better FU utilization ratio indirectly reflects better chip resource usage and better thread performance on chip.

Observation of program behavior provides more proofs. Compute-intensive programs require more ALUs and registers, but I/O-intensive programs require more I/O ports and Load/Store units. For a same program, different periods also have different behavior. Program needs more I/O at some periods and more computing at other periods. T. Sherwood et al classify similar periods as program phases [8]. Moreover, for any program at any clock, FU requirement is also unstable-some clocks fluctuant and some clocks lacking. These characters illustrate single SMT or CMP is not enough.

In this paper we propose a structure to share function units between neighbor cores in CMP. Function units in this paper include Integer ALU, Integer Multiplier/Divider, Load/Store unit, FP ALU and FP Multiplier/Divider. The share is compromising of SMT and CMP, which is not SMT-like aggressive share and is not CMP-like resource isolation. The proposed architecture tries to improve program performance and overcome flaws of SMT and CMP. Function unit only takes little chip die area and is not the bottle neck of current processor, but improving FU usage can lead to improve usage of other resource and improve thread performance.

The proposal bases on CMP architecture. Any core on chip can access the neighbor core's function units (except Load/Store units) and registers, and be accessed by neighbors. The key problem of function units sharing is share delay, which include delay of share decision making, instruction issue and data transmission. If the instruction issued to neighbor core is later than local issue, share would be meaningless.

2 Related Work

As the number of transistor is increasing on a die, how to use these resources more effective is a problem people explore. Single processor is out of date and SMT and CMP turn up. L. Hammond et al simulates three kind of superscalar, SMT and CMP processor to compare the efficiency of transistor using[1]. It indicates CMP is not only the easiest implementation, but also offer excellent performance. But Tullsen shows a different result[2]. It illustrates that both superscalar and fine-grain multithreaded architectures are limited in their ability to utilize the resources of a wide-issue processor, but SMT has potential advantages.

So SMT and CMP have their own excellence. J. Burns et al tests the area and system clock effects to SMT and CMP[4]. The result is that the wide issue SMT delivers the highest single thread performance with improved multi-thread throughput and multiple smaller cores deliver the highest throughput. Is a compromise of SMT and CMP good choice of billions transistors? Krishnan explores a hybrid design, where a chip is composed of a set of SMT processors[3]. The evaluation concludes that such a hybrid processor represents a good performance-complexity design point. But it is just physical combination and does not improve the substrate.

The new solution is feasible resources sharing between cores of a CMP chip. R. Dolbeau et al proposes a intermediate between SMT and CMP called CASH[5]. CASH is a parallel processor altering between SMT and CMP. It shares caches, branch predictors and divide units. But it does not exploit the complete dynamic sharing of resources enabled on SMT. Meanwhile CASH does not consider area effect.

Rakesh Kumar et al proposes conjoined-core chip multiprocessing to reduce die area and improve overall computational efficiency with resource sharing[6]. It shares crossbar, I-Cache, D-Cache and FPU. The goal of area saving is made, but the side effect is performance degradation.

The essence of resource sharing possibility is the resource require of program when it is running. T. Sherwood explores the program behavior and finds that programs change behaviors drastically, switching between periods of high and low performance[8]. But some periods have similar characters. He classifies similar behavior periods as program phases and, presents an architecture that can efficiently capture, classify and predict program phases[7]. The behavior differences let the chip resource usage be intensive or loose, so the loose phases can provide resource to intensive phases. J. Adam et al explores the value degree of use[9]. It implies that function units are idle sometimes and have the potential to be shared.

3 Motivation

The program performance is decided by hardware and itself. Hardware includes I/O speed, cache size, clock frequency, pipelining, function unit, and so on. Program itself means program logic and instruction dependences. For a compiled program, program logic and instruction dependences do not change any more, and the program performance is definitely decided by hardware structure.

FU utilization ratio reflects the core efficiency of a processor. An efficient core should be harmonious and any part of core would not be the bottle neck at general condition. So high FU utilization ratio means high core utilization ratio and improving FU utilization can improve the core efficiency. Figure 1 plots the relationship between FU and 4 other logic units on the core. Those units are representation of other units on core. In this graph, we set a 16 integer ALUs core as the baseline, and other logic units are configured to match. Then we configure the upper limit of integer ALU number, which can be used, to 8, 4 and 2 to observe the change of other units' utilization ratio during const time. In this figure, we ignore the influences from other FUs, such as Load/Store units and FP units.

From the graph, we can see that 4 units' utilization ratio degrade dramatically when number of IALU is 4 and 2, but when IALU is 8, the degradation is not clear.

The reason is the thread performance is mainly constrained by instruction dependence after IALU is more than 8. Lower IALU utilization ratio leads other units' utilization ratio to be low, and thus the core efficiency is low and many resources are wasted. On the contrary, improving FU utilization ratio can bring more benefits besides FU itself.

In this paper, we try to share FU between neighbor cores. The purpose is not only to improve FU usage, but more importantly is to improve most resource utilization on the core. Actually, FU only takes little chip area and is not the important area consumption unit and does not have complex logic or policy. But why dose we chose FU and is FU share possible in real program?

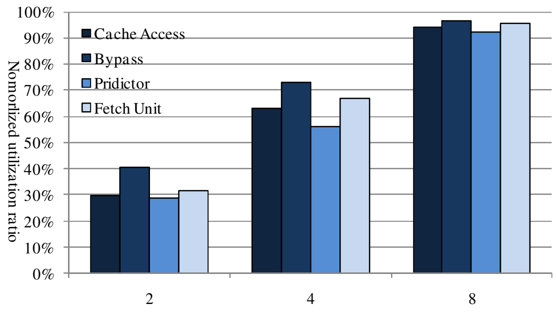


Fig. 1. Utilization ratio relationship between FU and other units

For a compiled program, program performance is constrained by hardware resources at first, but when the core has enough resources for the program, the performance is constrained by the program itself. This introduces a problem that cores are designed for wide range programs, but the program behaviors are always different for different ones. So resources are wasted for resource-relaxing programs and resources are not enough for resource-intensive programs in the same core. Taking FU as an example, some program needs more FUs and some have FU redundant under the same hardware environment.

It is obvious that different programs have different behaviors, but the same program also has different behaviors at different periods. T. Sherwood et al collects periods with similar behavior as program phase[8]. Table 1 presents the different FU utilization ratio of mesa's periods. The numbers in first row are unit number on core. On the whole, the FU utilization ratio is low. Integer ALU is best used at periods, but integer multiplier/divider never be used at first two periods, floatpoint multiplier/divider also be used infrequently at last three periods. This indicates many FUs are idle at program running time. Besides, FU usage is different at different clock cycles, even those clock cycles are in the same phase.

Figure 2 counts the average IALU taken for mesa and gcc at intervals of 1K clock cycles. The figure pictures the program behavior changes, meanwhile, mesa and gcc show opposite characters. mesa is IALU-intensive when gcc is IALU-relaxing, and gcc is IALU-intensive when mesa is not eager for IALU so much. But the curve marked as average fluctuates smoothly compared with mesa and gcc. So if those two program sections are running at two cores at same time, one core's IALUs are busy,

Table 1. FU unilization ratio in different program periods

IALU(2)	IMULT(1)	LS-UNIT(2)	FP-ALU(2)	FP-MULT(1)
53.95%	0.00%	15.80%	31.70%	73.30%
54.45%	0.00%	15.95%	31.75%	73.70%
72.50%	22.00%	16.80%	19.00%	6.00%
68.70%	27.10%	19.45%	25.25%	7.60%
66.85%	27.70%	19.10%	24.15%	8.00%

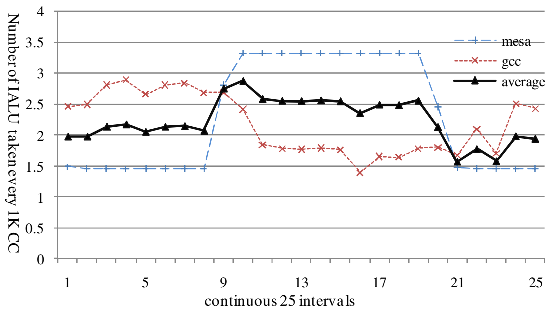


Fig. 2. Average IALU taken for mesa and gcc at intervals of 1K clock

when another core's IALUs have redundancy. Two cores sharing IALU to each other may get benefits reciprocally. Not only, it can use redundant IALU for IALU-intensive program to improve program performance, but also it is a butterfly effect to improve other units' efficiency on the core.

Resource sharing among cores is a way to improve resource usage and save chip area. Researches done before put focuses on cache or long latency unit, and short latency unit like IALU is hard to be shared, because sharing latency kill the advantages got by share. Then Table 1 shows FUs have potential to share, so how to share FUs between cores is work that will be done in this paper. FU is not the critical units on the core, so even FU share is possible, it still seems meaningless, but this point ignores an important fact that FU is central unit of all other units and improving FU utilization is to improve other units' utilization and core performance.

4 Architecture

This paper aims to share FUs between cores; the key problem of share is to solve latency brought by share. If program performance declines after sharing, share is meaningless. For example, the time to issue instruction to share is later than issued at local, share is not useful any more.

4.1 Main Framework and Dataflow

In this paper, the core on chip shares two neighbor cores and is shared by other two neighbor cores. The reason to only share neighbor cores is to cut down the share latency. Figure 3a presents the framework of core share. Arrows in figure is the share direction, core A is shared by core B and core C, meanwhile core A is sharing FUs of its left and downside cores.

Number on the arrow indicates the share priorities, first of all, local instructions have the highest priority to use local FUs, then instructions from core C has higher priority than core B to use A's FU. The intention of share with priority is to guarantee that local program would not be influenced by sharing and neighbor cores can use shared FUs orderly.

Figure 3b includes the internal framework of every core and dataflow between cores. The decision to share or not share FUs is made by Arbitrator. Firstly, Arbitrator needs to guarantee share latency is short and program performance doesn't degrade after sharing. Secondly, Arbitrator needs to decide which instructions would be issued to FU, according to share priority. Instructions from local thread have the highest priority to use FU, from right core have higher priority and from upside core have the lowest priority.

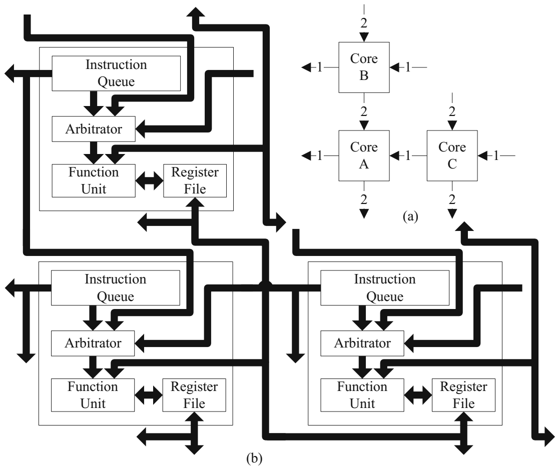


Fig. 3. Main framework and data flow of FU share

The dataflow is important between cores. In order to reduce transmission delay, the neighbor cores are connected directly to each other. Instruction Queue and Register File can be accessed by its left and downside cores directly; Function Unit can read and write its right and upside cores' Register File directly; and Arbitrator can get instructions from its right can upside cores directly.

4.2 Why Neighbor Two Cores

The share is meaningless if the start point of instruction execution on remote core is later than execution on local. Adding the data and instruction transmission, the original thread would suffer great degradation. So share latency has to be short enough. This includes two aspects—fast share decision made and short transmission latency.

The share decision is made by Arbitrator. Arbitrator in framework is a priority order circuit. Priority is the key factor to guarantee that local thread performance is not influenced by sharing and the remote instruction is executed only once. Firstly, the local thread has the highest priority to use all local FUs. Secondly, instructions sent to other core would be executed more than once if there is not constrained by priority. The more cores share with each other, the more operations the Arbitrator should do, and the longer latency the decision is made. So any core cannot be shared to too many cores, in this paper, we tested that a core shared by two cores is reasonable. The timing sequence of Arbitrator is presented in figure 5.

The FUs' execution period is short enough, after decision made by Arbitrator, instruction and data must be transmitted immediately. This means cores shared with each other should be as close as possible, so the share is built between neighbor cores. In order to transmit instruction and data with short delay, we set direct instruction and data path to access neighbor core Instruction Queue(IQ) and Register File(RF) like figure 3(b). We don't change the number read port of IQ, but we add two read port and one write to RF, because IQ has enough ports for all different kind FU and idle ports can be used to transmit instruction to neighbor cores, but RF is not like this.

5 Implementation

5.1 Core Detail

Figure 4 is the detail of figure 3b. Four bi-directional arrows in the figure are data paths with four neighbor cores, and the detailed connections are described in figure 3b. This section cares about the internal core infrastructure and workflow. The FUs shared in this paper doesn't include I/O unit, because I/O unit share is complex and it relate to cache, memory and cores connection.

The key factor to success FU share is share latency, the sharing instruction executed at shared core should not be later than local execution. In our implementation, all instructions in Instruction Queue (IQ) are judged whether it could be issued at next clock. Usually, instruction knows it can be issued when all its source register are free, but in this paper, we add a Register Status Unit (RSU) to mark how many cycles left for every register being free. For example, the latency of integer multiplier is 4 cycles and there is an instruction MUL R1, R2, R3, which just be issued, then RSU of register R1 will be set by 4. It means R1 will be free after 4 cycles. RSU will decrease after every cycle until the number in RSU is zero. Thus RSU indicates register will be free at next cycle when the related RSU is 1, so instruction can knows whether its source register would be ready and whether it can be issued at next cycles.

The Register Status Unit is to mortgage Arbitrator's decision latency and instruction and data transmission. In our design, the decision can be made in a cycle, but instruction and data transmission would be completed from one to two cycles according to the data-path and read and write port. The decision made and transmission is performed parallelism. So after having Register Status Unit, the Arbitrator can know instructions are ready to issue one cycle early. In this paper, we set integer ALU as the fastest function unit and it takes one cycle to execute. So decision made one cycle early can satisfy the fastest function unit on the core to execute instructions on shared core without share latency.

Instruction Queue is composed of three parts, the first one is instruction word, the second is NCR (Next Cycle Ready) flag, and the third one is CtI (Core to Issue) flag. NCR means instruction can be issued at next cycle which is the AND of two related register's RSU, and CtI means instruction would be issued to which core. CtI could be 00(local core), 01(left core), 10(downside core), 11(reserved), and it is set by local Arbitrator or sharing core Arbitrator. Arbitrator decides FUs assignment according to local, right and upside core NCR. Only there are redundant FU after all local ready instructions are issued, FU can be shared by right or upside core. If instruction in IQ is LOAD and STORE instruction, they will only be issued to local core.

Instructions issued to FUs are recorded by Reservation Station (RS). The number of entries in RS is the same as the number of FU. Entry is six-tuple like that:

OP, BUSY, CORE, DEST, SOURCE1, SOURCE2

OP is instruction operation code, BUSY represents FU is busy or not (if it is not zero, it means the FU operation would be done after BUSY cycles; if it is zero, it means FU is idle), CORE indicates where the instruction in this FU comes from (00, 01, 10), DEST are destination register, SOURCE1 and SOURCE2 are source registers.

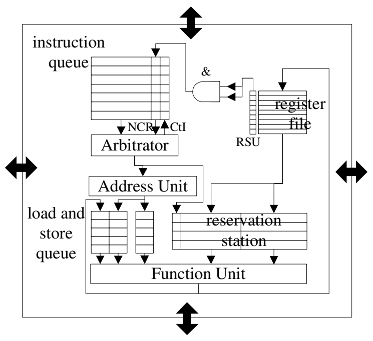


Fig. 4. Internal framework with detail

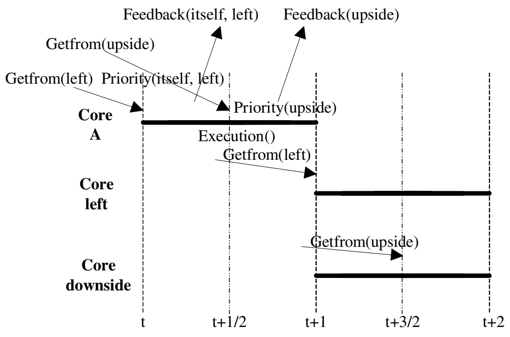


Fig. 5. A one cycle instance of Core A

5.2 Core Workflow

The workflow of core at any clock cycle is showed as figure 5. We analyses one cycle of Core A in the figure, and all core's work is same as A. At the start of cycle T, A's NCR is set according to RSU, meanwhile A knows which FU will be free at next cycle according to Reservation Station's BUSY. So Arbitrator can decide which instruction can be issue to FU at next cycle and the related CtI would be set. If there is rest FU at next cycle, A gets the right core's NCR and tries to share FU to right core. If share succeed, the right core's CtI would be set. This is the procedure Getfrom(left) ,Priority(itself, left), and Feedback(itself, left).

If there is local instruction that cannot be issued at next cycle because of lack of FUs and the left core has redundant FU, the next-cycle-ready instructions will be issued to the left core just like the right core issue instruction to A. This is the procedure of Core left's Getfrom(left).

If there still has redundant FU after it is shared by the right core, then A tries to share FU to upside core. If share succeed, the CtI of upside core will be set. This is Getfrom(upside), Priority(upside) and Feedback(upside). Similarly, if Core A still has instruction can be issued after sharing the left core; these instructions can share downside core's FU. This is Core downside's Getfrom(upside). When A is making issue decision about next cycle, A is executing instructions decided at previous cycle. Instruction and data transmission happens at next half cycle immediately after the decision made. So at the next cycle, instructions issued to neighbor cores can be executed. The following five instructions is an example for figure 5.

All this 5 instructions use integer ALU and we assume latency of integer ALU is 1 cycle. Those instructions start at clock T and all dependences with earlier instructions can be satisfied at clock T. Then if the core has enough IALU, 5 instructions can be finished at T+1. The execution is presented with case Enough IALU. But if the core only has 2 IALUs and the core cannot share other cores' IALU, the execution is presented at third row of table 2 and instructions can be finished at T+2. Then if core use share proposed in this paper, the situation will be changed. Firstly, at beginning of T-1, core knows instruction 1, 2, 3 will be ready at clock T. Then Arbitrator decides that instruction 1, 2 can be issued at cycle T, meanwhile it cannot offer any FU to be shared by its right and upside core, so it does not accept any other cores' instructions.

Secondly, at cycle T, if the left core have made the decision of local next-cycle-issue(cycle T+1) instructions and find there has redundant IALUs, so it can execute instruction 3 at cycle T+1, and the CtI of instruction 3 is set with 01. Similarly, if the downside core have made the decision of local and right core next-cycle-issue instructions and find there has redundant IALUs and CtI of instruction 3 is not be set, so it can execute instruction 3, and the CtI is set with 10. In those two cases, instruction 3 will be executed at left or downside core. Meanwhile, the core knows instruction 4, 5 would be ready at T+1 according to NCR and makes a decision to issue instruction 4, 5 at T+1. Meanwhile, instruction 1, 2 are executed at this cycle. The execution of this case is presented at forth row of table 2.

Table 2. An Example of Figure 5

Clock		T	T+1	T+2
Enough IALU		1, 2, 3	4, 5	-
2 IALUs	without sharing	1, 2	3, 4	5
	share succeeds	1, 2	4, 5(local)	-
	share fails	1, 2	3, 4	5

```
1 SUM     R2, R1, R2
2 AND     R3, R1, R3
3 OR      R4, R1, R4
4 SUM     R3, R2, R3
5 SUB     R5, R2, R5
```

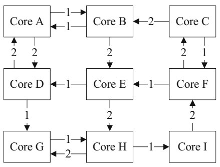


Fig. 6. Share relationship in evaluation

If the decision is instruction 3 cannot be issued to left or downside core, then the core decides that instruction 3, 4 are issued at T+1 when instruction 1, 2 are executing at cycle T. When instruction 3, 4 are executed at T+1, instruction 5 try to share left and downside core's IALU. If failed, it would be issued to local IALU at T+2. The execution of this case is the last row of table 2.

6 Evaluation

The timing simulator used was derived from the Simplescalar 3.0 tool set, and we modified sim-outorder.c and built a multicore architecture like figure 6. Arrows are the share direction and numbers are share priority. We break the "share left and downside core" rule in order to let cores on the edge of chip share other cores. It does not violate our share proposal. The baseline micro-architecture model of every core is detailed in table 3.

The test benchmark is SPEC2000 and all benchmarks are compiled based on PISA. We choose 11 benchmarks and they are 7 SPECINTs and 4 SPECFPs. Those benchmarks are representative of all benchmarks with and without FU-intensive.

T. Sherwood et al illustrated program has different phases. Our observation to phase is FU usage. We found that program is eager for FU at some phases and incompact for FU at some other phases. We call phases that are eager for FU as Resource

Table 3. Baseline microarchitecture model of every core

I L1 cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
D L1 cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 cache	256K 4-way set-associative, 64 byte blocks, 6 cycle latency
Branch Pred	Hybrid(default set)
Execution	Out-of-Order issue, Out-of-Order execution, In-Order commit
Function Unit	2 integer ALU, 1 integer MULT/DIV, 2 load/store units, 2 FP
Size(insts)	IQ:4, RUU:32, LSQ:4
Width(insts/c)	decode:4, issue:4, commit:4

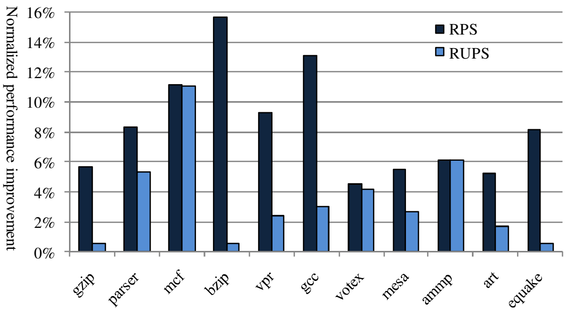


Fig. 7. Performance improvement for RPS and RUPS with proposal proposed in this paper

Popular Section (RPS) and the other phases as Resource Un-Popular Section (RUPS). Figure 7 is the result of performance improvement to RPS and RUPS for benchmarks with share. The baseline performance is a benchmark running at single core without sharing. We pick the RPS and RUPS of a benchmark and run those sections 9 times respectively at 9 different location of proposed architecture in figure 6. The result in figure 7 is the average performance improvement compared to the baseline performance. The average improvement of RPS is 8.4% and the average improvement of RUPS is 3.5%.

We can found that there is no benchmark performance degradation with share proposed in this paper. There are two reasons. Firstly, sharing instruction would never preempt shared core FUs when local instructions need to use those FUs. The priority of local instruction is higher than sharing instruction and only redundant FUs can be used to sharing instruction. Secondly, the start point to execute sharing instruction at shared cores would never be later than local execution. The improvement of RPS is always better than RUPS; this indicates RPS can get more benefits from FU share. The reason is RPS has better instruction level parallelism and need to issue more instruction at a cycle.

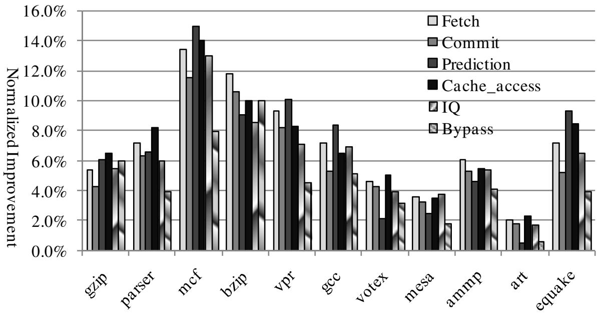


Fig. 8. Six units' utilization ratio is improvement comparing with baseline

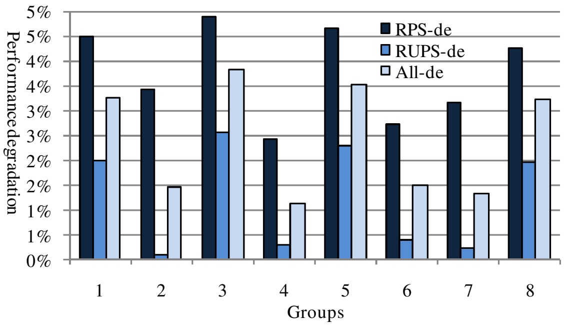


Fig. 9. Performance degradation in random assignment compared with static assignment

FU is not performance critical part and only take little of chip area, but FU is the central part of the core and the efficiency of FU reflects efficiencies of other units on the chip. Figure 8 is other units' improvement based on figure 7. In this figure, we do not distinguish RPS or RUPS, and it is the utilization improvement of whole benchmark. From the figure, we can see it is basically coherent with figure 7. Benchmark with high performance in figure 7 has high utilization improvement in figure 8, such as mcf and bzip. So FU sharing has benefits for whole chip performance.

The share efficiency is influenced by shared core. The program is hard to share cores which are running RPS. So the schedule from program to core affects the whole chip performance. We choose two methods to test this influence. The first method is a static way. We choose 5 RPS benchmarks to be executed at core A, C, E, G and I in figure 6 and choose 4 RUPS to be executed at rest cores. The mapping from benchmark to core for RPS and RUPS is random. The second method is all 9 chosen benchmarks are mapped randomly.

We test 8 groups. Statistics of every group include average IPC for RPS, average IPC for RUPS, average share instructions for RPS, average share instructions for RUPS and average IPC for all 9 benchmarks. The result shows the static assignment can get higher IPC than random assignment. Figure 9 is the performance degradation in random assignment compared with static assignment. We can find that RPS has a more serious impact than RUPS with different assignment. This is not to illustrate that the static assignment is better than random assignment, but to illustrate that a reasonable assignment can let whole chip be more effective.

Acknowledgement

This paper is supported by the Research Foundation of Education Bureau of Zhejiang Province under Grant No. Y200803333, the State Key Laboratory of High-end Server & Storage Technology(No. 2009HSSA10), National Key Laboratory OS Science and Technology on Avionics System Integration, the Special Funds for Key Program of the China No. 2009ZX01039-002-001-04, 2009ZX03001-016, 2009ZX03004-005.

References

1. Hammond, L., Basem, A.N., Olukotuna, K.: A Single-Chip Multiprocessor. Computer 30, 79–85 (1997)
2. Dean, M.T., Susan, J.E., Henry, M.L.: Simultaneous Multithreading Maximizing On-Chip Parallelism. In: Proceedings of the 22nd Annual Int. Sym. on Computer Arch., Santa Margherita Ligure (1995)
3. Venkata, K., Josep, T.: A Clustered Approach to Multithreaded Processors. In: Proceedings of the 12th International Parallel Processing Symposium (1998)
4. James, B., Gaudiot, J.: Area and System Clock Effects on SMT/CMP Processors. In: International Conference on Parallel Architectures and Compilation Techniques (2001)
5. Dolbeau, R., Seznec, A.: CASH: Revisiting hardware sharing in single-chip parallel processor. IRISA Report 1491 (2002)

6. Rakesh, K., Norman, P.J., Dean, M.T.: Conjoined-Core Chip Multiprocessing. In: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (2004)
7. Timothy, S., Suleyman, S., Brad, C.: Phase tracking and prediction. In: Proceedings of the 30th annual international symposium on Computer architecture (2003)
8. Timothy, S., Erez, P., Greg, H., Suleyman, S., Brad, C.: Discovering and Exploiting Program Phases. IEEE Micro (2003)
9. Butts, J.A., Sohi, G.S.: Characterizing and predicting value degree of use. In: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture (2002)
10. Francis, T., Yale, N.P.: Achieving Out-of-Order Performance with Almost In-Order Complexity. In: Proceedings of the 35th International Symposium on Computer Architecture (2008)
11. Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese, B.: Piranha: a scalable architecture based on single-chip multiprocessing. In: Proceedings of the 27th annual international symposium on Computer architecture (2000)
12. Engin, I., Meyrem, K., Nevin, K., Jose, F.M.: Core fusion: accommodating software diversity in chip multiprocessors. In: Proceedings of the 34th annual international symposium on Computer architecture (2007)

A High Efficient On-Chip Interconnection Network in SIMD CMPs

Dan Wu, Kui Dai*, Xuecheng Zou, Jinli Rao, and Pan Chen

Department of Electronic Science and Technology,
Huazhong University of Science and Technology,
1037 Luoyu Road, Wuhan, China
{dandan58wu,josh.maxview,escxczou,ary.xsnow,chenpan.maxview}@gmail.com

Abstract. In order to improve the performance of on-chip data communications in SIMD (Single Instruction Multiple Data) architecture, we propose an efficient and modular interconnection architecture called Broadcast and Permutation Mesh network (BP-Mesh). BP-Mesh architecture possesses not only low complexity and high bandwidth, but also well flexibility and scalability. Detailed hardware implementation is discussed in the paper. And the proposed architecture is evaluated in terms of area cost and performance.

1 Introduction

Moore's Law continues with technology scaling, and the integration capacity of billions of transistors has already existed today. However, there is growing evidence that wire delays and power dissipation become the dominant constraints for present and future generations of microprocessors with shrinking feature sizes and increasing clock frequencies [1-4]. Chip multiprocessors (CMPs) become the mainstream by using the increasing transistor budgets to integrate multiple processors on a single die while mitigating the negative effects of wire delays and the power consumption. Parallelism is an energy-efficient way to achieve performance [5]. Thus, to convert the rich transistor resources into application performance, different degrees of parallelism are exploiting and those on-chip processors are elaborately managed to provide powerful parallel processing capacity.

Single Instruction Multiple Data (SIMD) architectures are able to exploit the data-level parallelism, and the replicated structure offers not only efficient and fast processing performance at relatively low power, but is also very scalable. The communication among PEs has significant impact on the efficiency of parallel computing in many parallel programs. Therefore, the interconnection structure is very important in SIMD systems. This paper emphasizes on the architecture and performance of the interconnection network in SIMD CMPs and proposes an efficient interconnection architecture called Broadcast and Permutation Mesh

* Corresponding author.

(BP-Mesh). BP-Mesh architecture is a modular structure and can be hierarchically constructed to two-tier high-bandwidth, low-latency and scalable on-chip network (NoC). At local level, 4 by 4 PEs are closely grouped as one Cell and are connected by a two-dimensional (2-D) Mesh network with one broadcast and permutation network (BPN) attached at each row and column, respectively. At global level, Cells are also connected by a 2-D Mesh network with each row and column completely connected by one BPN. Hence, each PE could communicate with others (intra-Cell or inter-Cell) in a very flexible way with low hop latency. The tiled architecture of PE array effectively approaches the challenges of wire delay and the "*power wall*" and exhibits well scalability.

The implementation results show that the additional area cost of the proposed BPN network is rather small. In order to evaluate the performance of BP-Mesh architecture, two parallel application kernels such as dense matrix multiplication and Fast Fourier Transform (FFT) are considered. By setup the experimental environment with a prototype design system and a verified simulator, we use the metric *scalability* to demonstrate the scalable property of the proposed BP-Mesh architecture, and the performance improvements of the implementation of double-precision floating-point 1-D FFT on the BP-Mesh SIMD over conventional mesh SIMD can be 14.28%∼71.63%. Moreover, the experimental results show that the BP-Mesh based SIMD multiprocessor system has strong competitive strength in the fields of high performance computing.

The rest of paper is organized as follows. Related works are shown in Section 2, followed by the introduction and implementation details of the architecture of BP-Mesh in Section 3. The experimental results are presented in Section 4. Finally, conclusion and future works are discussed in Section 5.

2 Related Works

Conventional SIMD machines are constructed as either the traditional two-dimensional mesh or the modified mesh structure for the inter-PE communications, such as Illiac IV [6], MPP [7], and DAP 500 series [8]. MasPar [9] architecture has an X-Net interconnection network connecting each PE with its 8 nearest neighbors in a 2-D mesh. The main component of MorphoSys [10] is a SIMD fashion 8×8 RC (Reconfigurable Cells) Array with 2-D mesh and complete row/column connectivity per quadrant.

IMAP-VISION chip of the IMAP system [11] contains 32 PEs that connected in series to configure a one-dimensional SIMD processor array. Each PE has one shared register where it can send or receive data to and from, and has only the ability to access to its neighbors at left and right. A general purpose computational accelerator, Clearspeed CSX600 [12], comprises 96 processing cores operating in SIMD manner and communicating with one another via a linear network named *swazzle*.

Imagine [13] is a SIMD processor which consists of eight arithmetic clusters (PEs) with each PE including six ALUs to execute VLIW instructions and eight clusters are fully connected. But the study in [14] has shown that the area of the inter-PE communication unit (the interconnection network) in Imagine will dominate the total area if the number of PEs beyond 64.

GRAPE-DR [15] is another example of high performance SIMD processor with 512 processing cores integrated on the chip. To gain the compact integration density, there is no communication network between PEs. PEs are organized into blocks and the blocks are connected to a reduction network.

According to these previous works, mesh network is the most popular topology for its low complexity and planar 2D-layout properties. However, the performance is degraded with rapidly increasing diameter which is not suitable for larger networks. Linear network, though is simple and power efficient, allows only the communication between directly connected PEs which makes the communication with a non-direct neighbored PE a bottleneck. Fully connected network has the highest flexibility, but occupies large die area and is power inefficient. Furthermore, it scales poorly. The broadcast/reduction network in GRAPE-DR is highly efficient to solve the problems such as N-body particle-based simulations, but it is not powerful enough in some other parallel applications like sorting and FFT.

In this paper, we propose a modular network topology taking advantage of the communication locality and provides both efficiency and flexibility. The topology can be used as building blocks to construct a hierarchical interconnection network. The motivation of BP-Mesh architecture is inspired by the researches in [16] and [17]. Hierarchical topology is energy efficient, and high radix topology such as CMesh [16] is a good choice for locality. The optimized BP-Mesh network presents high bandwidth, low local latency as well as excellent scalability.

Our proposed network topology is similar to MorphoSys, but is different in that the row/column connectivity in MorphoSys is mainly used to broadcast context words to implement the configuration whereas the BPN network in our design is used to broadcast or exchange data information during the computation in a fast and flexible way.

3 The Architecture of the BP-Mesh Network

In a SIMD multiprocessor, all PEs execute the single stream of instructions concurrently, while each PE operates on data from its own local memory. An interconnection network connects PEs as the communication fabric for the exchange of data and control information. The efficient exchange across the network is crucial to overall system performance. Thus, the choice of the interconnection network is very important. Different data parallel applications require different communication patterns for maximum performance. It is, therefore, necessary that the network structure is efficient and flexible for both regular and irregular communications to accommodate different applications. Scalability is another significant characteristic of the interconnection network to scale the performance with the increasing number of PEs.

All communications in SIMD architecture take place at the same step under the explicit control of instructions, so there has no message contesting problems and the router logic is much simpler than the one in general NoC. In this section, we propose the architecture of BP-Mesh network and discuss its detailed hardware implementation.

Mesh topology has been widely used for its simplicity and regularity. Whereas it scales poorly in terms of performance and power efficiency with the increase of the number of PEs and average hop count. To approach these bottlenecks, a pipelined broadcast and permutation network (BPN) is employed to provide low hop latency and high parallelism of data transfer, and the combination of BPN and mesh network can be constructed hierarchically to gain benefits of exploiting communication locality and energy optimization. BPN network enables data transferred from one PE to another or other indirect-connected PE/PEs in a one-to-one or one-to-multiple mode, as well as enables data swapped between PEs simultaneously. The "mesh + BPN" constitutes a new network topology named BP-Mesh. The architecture of two-tier hierarchical BP-Mesh connecting 256 PEs is depicted in Fig. 1.

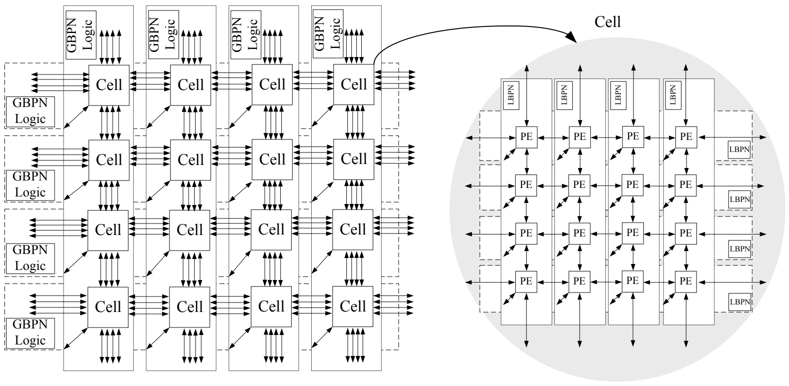


Fig. 1. Hierarchical BP-Mesh Architecture

At local level, 16 PEs are closely grouped as a Cell unit and organized as a 4 by 4 mesh array. Furthermore, a secondary network (BPN) with the functions of data broadcasting and permutation among PEs is added at each row and column of PE array, respectively. With this modular network topology, 4 by 4 Cells are connected at the same way to construct the higher level interconnection structure for global PE communications. The symmetric structure with BPN at each row/column supports more convenient communications. BP-Mesh architecture possesses not only low complexity and high bandwidth, but also well flexibility and scalability. The cooperation of mesh and BPN network provides high efficient inter-PE communication for various data parallel applications. Mesh network takes charge of neighbored communications while BPN network mainly deals with hop communication, data broadcast (multicast) or data exchange between PEs in parallel.

For detailed hardware implementation, each PE directly communicates with its four neighbors (left/right/up/down) through the mesh connection in the network. On each clock cycle, PE core can perform a register-to-register transfer to one of its neighbors in one direction, while simultaneously receiving data from the other neighbor in the opposite direction. The structure of BPN network is shown in Fig. 2. It includes three main parts: the control logic, multiplexer (MUX) sets, and the shared registers (SR). The number of multiplexers, as well as the number of inputs for each multiplexer, is corresponding to the number of PEs in the same row/column. Rather few numbers of shared registers are used in the BPN to temporally store data during the pipelined operations. Control logic module is the backbone of BPN. After configuring the contents of the configuration register, the control logic generates control signals for each network pipeline stage. When data has arrived, where it comes from, which shared register is its destination, and what are the control bits for each multiplexer to transfer data to desired PEs are all under the control of the control logic.

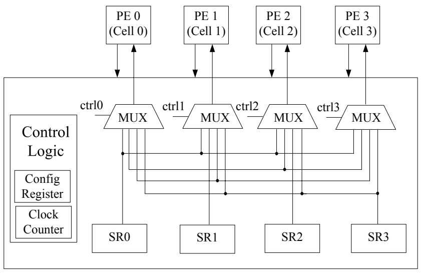


Fig. 2. The Structure of Broadcast and Permutation Network

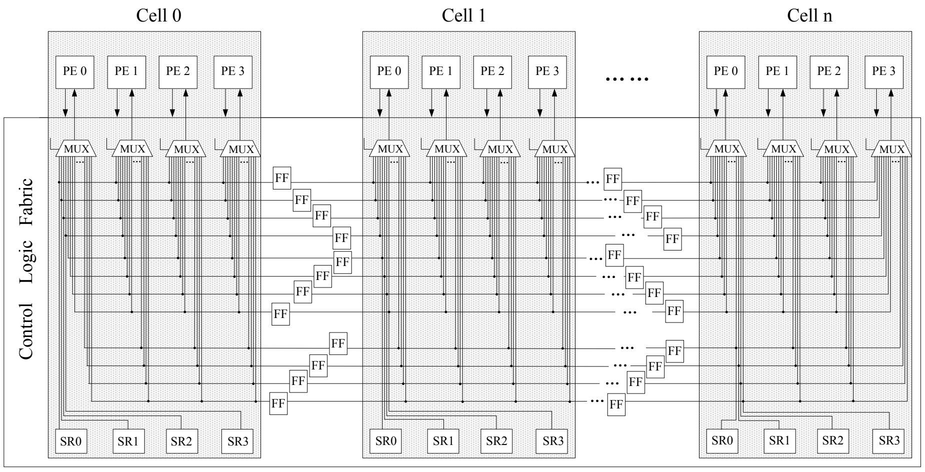


Fig. 3. The Structure of BPN Network for Cell Array in Flatten Way

The BPN in each row or column of the Cell array can be implemented as either the broadcast/permutation communications among PEs (in a flatten view) or among Cells, according to the specified SIMD instruction set architecture and the manufacture process. The first method (Method *I*) can be implemented as in Fig. 3. The number of inputs of each MUX scales with the number of the connected PEs, and the longest transfer path between shared register and the multiplexer is increased accordingly. To address the wire delay problem, flip-flops are inserted across the Cells. The second method (Method *II*) can be implemented as the same way as we have described above (see Fig. 2), except that the granularity has been changed from PE to Cell and the control logic is responsible to establish physical links from one PE in a Cell unit with another indirect-connected PE in a different Cell unit.

Though Method *I* has the advantage of simpler control logic, with the increasing number of PEs and wider data path, the routing congestion will be the main issue to practically implement this kind of network instead of the cost of gate count. Method *II* is superior in term of scalability with more PEs integrated on one single chip. Moreover, its hierarchical levels are identical and unified.

4 Experimental Results

This section presents some initial experimental results of the proposed BP-Mesh network. Based on a real design case, we setup the evaluation platform and environment. We first evaluate the area cost of the additional BPN network. Then we test the function and performance of the proposed BP-Mesh network.

4.1 Experimental Platform and Environment

The BP-Mesh architecture proposed in Section 3 has been implemented on a SIMD multiprocessor–ESCA (Engineering and Scientific Computation Accelerator). ESCA acts as an attached device of a general purpose processor to accelerate the compute-intensive parallel computing in high performance scientific and engineering applications. The instructions run on ESCA processor are 128-bits wide, and have a fixed RISC-like format. Up to 256 PEs can be integrated onto one chip and the data communications among PEs are explicitly controlled by the specified fields of the network instructions. Each PE has one private local memory and four function units: the floating-point Multiply-and-Add (FMAC), the integer Multiply-and-Add (IMAC), the arithmetic logic unit (ALU) and the comparison unit (CMP). The word length of PE is 64-bits, and the supported data types include IEEE 754 compatible double-precision and single-precision floating-point, and the 64/32/16/8-bits integer. Instructions can be executed in vector mode with maximum vector length 64. Dual-buffering mechanism has been adopted to alleviate the external shared memory bandwidth pressure by overlapping the external memory access latency with the computation of function unit.

Until now, we have completed the hardware design of PE, BP-Mesh network, control unit and I/O communication mechanism. A prototype chip has been implemented with Chartered 0.13 μm high-performance (HP) process in a 6mm by

Table 1. Key Parameters of ESCA Prototype Chip

Number of PEs	4×4
On-Chip-Network	2D-Torus + BPN at row/column
Private Local Memory Capacity of PEs	Each 4KB, total 64 KB
Working Frequency	500 MHz
Number of Pipeline Stages of FMAC	5
Number of Pipeline Stages of ALU	1
Number of Pipeline Stages of BPN	3

6mm die size. Some key parameters of the chip are shown in Table 1. Since only 16 PEs are integrated on the chip, we implement the interconnection network at local level and use Torus network instead of the Mesh network to achieve better communication performance for edged PEs.

We have also developed a cycle-accurate simulator for ESCA. And the correctness of the simulator is verified by the prototype chip. The pipeline stages of BPN at global level have increased to 7 in ESCA simulator, since the data communicated among the Cells need to be transferred through the shared registers both in the local level and in the global level and the global interconnect wire requires adding pipelines.

The experimental environment is made up of two parts:

– The ESCA-based test system (shown in Fig. 4), which consists of one ESCA chip, one FPGA chip and four SRAM chips. FPGA chip is the bridge of the SRAM and ESCA. Data and programs to be executed on ESCA have been stored in the SRAM. The I/O interface on FPGA is responsible for reading and writing the QDR SRAM and ESCA with DMA mode. Instructions are first loaded into the instruction cache and then issued by the control unit to ESCA. The communication bandwidth for the instruction stream is reduced with vector mode. The working frequencies of the QDR SRAM and FPGA are both 250MHz.
– The cycle-accurate simulator. Due to the limited number of PEs on ESCA prototype chip, we use the simulator to evaluate the performance of hierarchical BP-Mesh network.

4.2 Area Cost

The routing strategy of the network is explicit and simple in our SIMD CMP which has been implemented with hard-wired logic in ESCA. To estimate the additional logic of BP-Mesh over mesh network, that is, the cost of BPN network, we synthesize a 5-port SpecVC router (with 4 VCs per port and 4-flit-deep cyclic buffers for each VC) [18] used in general NoCs under the same conditions (Chartered 0.13μm HP process@500MHz) with the BPN logic. The synthesis results are **33637.3772461** for our BPN network and **340715.712746** for the 5-port SpecVC router. Therefore, the area cost of BPN network is no more than 10% of that of the 5-port router, and the experiences from the physical design

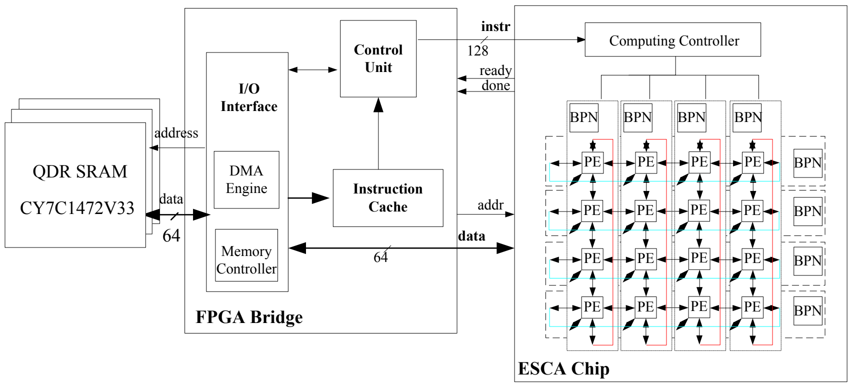


Fig. 4. ESCA Chip Test System Block Diagram

of ESCA chip have proved that the routing resource needed for the BP-Mesh architecture is acceptable.

4.3 Performance Evaluation and Comparison

In this part, we evaluate the performance of the proposed BP-Mesh network and compare it with the performance of conventional mesh network and some other referenced CMPs. We first describe the implementations of two selected data-parallel scientific computing application kernels on ESCA. Then we use *scalability* to demonstrate the scalable property of BP-Mesh architecture, and use the metric *speedup* to compare the BP-Mesh network with mesh topology to show the performance improvement. At last the two kernel performances implemented on ESCA prototype system are presented and compared with some other CMPs to indicate the high efficiency of the BP-Mesh architecture and its implementation platform in a comprehensive way.

Application Kernels. In order to evaluate the performance of BP-Mesh architecture, two data parallel application kernels: dense matrix multiplication and Fast Fourier Transform (FFT) are considered. They are basic kernels in parallel computing. Moreover, these two kernels have close relations with data communications among PEs.

The basic operation of matrix multiplication is C=AB. We choose Cannon algorithm [19] to implement the dense matrix-matrix multiplication and assume A and B are both square matrix of size 1024×1024. The data type used in this paper is single-precision floating-point to compare its performance later with GPUs. Both matrix A and B are divided into blocks for computation due to the limited PE local memory capacity. More specifically, sub-matrix A is (8×4) and stored by column, sub-matrix B is (4×16) and stored by row. After sub-matrices of A have been stored in the local memories of PEs, the sub-matrices of B can be broadcasted. Then the standard Cannon algorithm is computed with columns/rows of sub-matrices of A and B transferred among PEs via the BP-Mesh network. By applying the dual-buffering technique, data are communicated between the ESCA and external shared SRAM concurrently with the matrix computation operations. Thus the communication overhead is effectively hidden.

We choose the basic radix-2 decimation-in-time (DIT) 1-D FFT for the FFT kernel since it's simple to implement and the performance results can be compared with other counterparts. Assume ESCA can manipulate up to N data points for its limited on-chip memory capacity, and there are P processing elements (PEs) on chip. All data points are divided into N/P groups and cyclically (a non-consecutive sequence) distributed into the private local memories of P PEs. In addition, twiddle factors with a few redundancies are stored in the same way as the data points. Therefore, the first $\log P$ stages are the butterfly transform with data communications between PEs. Twiddle factors required for each butterfly operation are broadcasted from the owner PEs to the destination PEs. All data are exchanged synchronously among PEs via the BPN network. Then SIMD parallel computations are executed by PEs in vector mode. During the latter $\log(N/P)$ stages, both twiddle factors and data points needed for the butterfly computation have been stored in the private local memory of the same PE, and the transform computations can be done without any communication overhead. In this paper, we implement this parallel FFT algorithm with two kinds of data types: 1) double-precision floating-point for the evaluation of the scalability and efficiency of BP-Mesh architecture; 2) 16-bit fixed-point for the performance comparison with a published work with mesh network topology.

Scalability. We use ESCA simulator to evaluate the performance of dense matrix multiplication and 1-D FFT with varied PE numbers.

The sustained performance of the implementation of matrix multiplication on ESCA with the number of PEs varying from 1 to 256 is illustrated in Fig. 5. With the increase of the number of PEs, the sustained performance increases linearly. The evaluated matrix size is 1024×1024.

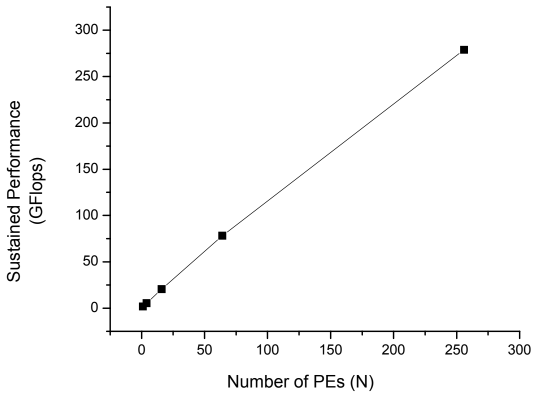


Fig. 5. Dense Matrix Multiplication Performance Evaluation

Table 2 shows the cycle counts for 1-D FFT application with data size varying from 64-point to 16384-point and the PE configuration from 4×4 to 16×16. The data type used here is double-precision floating-point. With the increasing of the number of PEs, the completion time of the Fourier transform for the same point size decreases. The *Italic* items in Table 2 demonstrate the condition that the size of FFT applications exceeds the maximum private local memory capacity of PE array. Thus the intermediate computing results should be exchanged between the off-chip shared SRAM memory and on-chip local memory frequently which dramatically degrades the performance.

Table 2. Cycle Counts with Different PEs at Different FFT Size

FFT Size	Cycle Count with different PEs		
	4×4	8×8	16×16
64	516	372	408
128	846	508	476
256	1528	718	544
512	2978	1128	682
1024	6092	1970	932
2048	*34749*	3740	1422
4096	*104279*	7494	2424
8192	*276233*	*64169*	4504
16384	*423435*	*163839*	*8908*

Speedup. Now we evaluate the performance improvement of BP-Mesh architecture over the mesh network. *Speedup* is defined as the ratio of the difference between the time required to implement the kernel on the conventional mesh SIMD and the time taken to implement the same kernel on BP-Mesh SIMD, to the time required to implement the kernel on the conventional mesh SIMD.

The speedup of implementing double-precision floating-point 1-D FFT on ESCA with or without BPN network can be 14.28%∼17.56% with FFT size varying from 64 to 1024 on 16 PEs (tested on ESCA-based test system) as shown in Fig. 6(a), and can be 14.28%∼71.63% with 4 points/PE while the number of PE varies from 16 to 256 (tested by ESCA simulator) as shown in Fig. 6(b). With the increasing of hop distance, the advantage of the BP-Mesh architecture is more obvious.

Performance Comparison with Other CMPs. In order to indicate the high efficiency of BP-Mesh architecture in a general way, we now compare the performances of dense matrix multiplication and 1-D FFT implemented on ESCA prototype system with some other CMPs.

Graphics Processing Units (GPUs) are high-performance many-core processors in which the stream processing units are arranged into SIMD arrays to provide high parallel computation capability and are playing increasing role in scientific computing applications. We choose the GPU platforms [20] both from

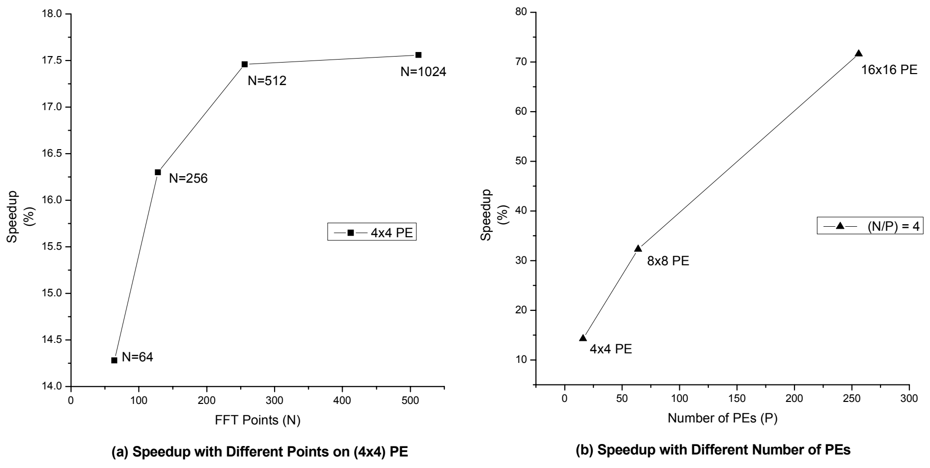


Fig. 6. Speedup of Floating-point 1-D FFT

Table 3. Performance Comparison for 1024 × 1024 Matrix Multiplication

Platform [23]	Core/Memory Clock (MHz)	Processing Time (s)
NV 5900 Multi	350/500	0.713
NV 6800 Multi	350/500	0.469
ATI 9800 Multi	500/500	0.445
ATI X800 Multi	500/500	0.188
ESCA Chip	500/500	**0.183**

NVIDIA and ATI (now is AMD) with the clock frequency at the same order of magnitude with ESCA chip. The performances of the matrix multiplication of 1024 × 1024 matrices implemented on those GPU platforms and the ESCA chip are measured by processing time (in seconds) and listed in Table 3. We also exclude the overhead of repacking input matrices in system memory as in [20] and count the time from the beginning of matrices A and B loaded into ESCA local memories to the end of final computation results (matrix C) stored back to the external shared SRAM memory. It can be concluded that ESCA with the implementation of BP-Mesh network outperforms the referenced GPUs.

We compare the performance of 1-D FFT implemented on ESCA-based test system with that of the implementation on a referenced mesh connected multi-core NoC[21] (with 16 PEs on the chip), in which the data type used is 16-bit fixed-point and the performance is presented in term of cycle count. We calculate the clock cycles elapsed to implement different point size of FFT algorithm (from 64-points to 4096-points) on ESCA prototype system and the comparison with the results in [21] is illustrated in Fig. 7. With the support of sub-word parallelism in ALU, the performance of 1-D FFT implementation on ESCA processor is about 7 times faster than the results in [21] on the average.

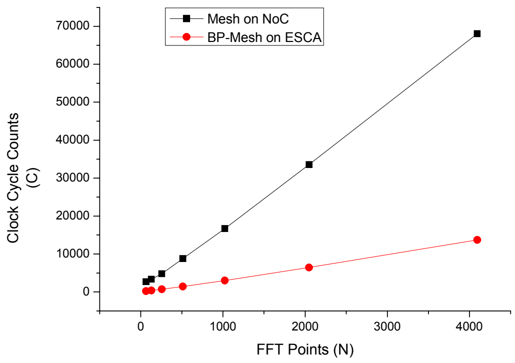


Fig. 7. Performance Comparison of fixed-point FFT with a Mesh NoC[24] (4×4 PEs)

5 Conclusion

In this paper, we present an effective interconnection architecture for inter-PE communications in SIMD CMPs. BP-Mesh architecture inherits the low complexity and high bandwidth advantages of mesh connection, while mitigates the large hop latency and provides flexibility and parallelism for different communication patterns with a broadcast and permutation network. The basic BP-Mesh structure is modular and symmetric, as well as scalable and high efficient. Detailed hardware implementation has been discussed. Based on a real design, we evaluate the area cost and performance of the BP-Mesh architecture. The scalability of the proposed architecture has been verified by a RTL-level consistent simulator with more PEs integrated. The speedups in term of performance improvement over conventional mesh connection have been tested with different problem size and different number of PEs. Also, the performances of the prototype chip implemented with our proposed BP-Mesh architecture have been compared with some other CMPs to exemplify the high efficiency of the network.

More extensive testing with different applications, especially the irregular communication pattern applications, is ongoing. The design of on-chip interconnection network has proven to have a significant effect on overall system energy consumption [22], since it is implemented using global wires with long delays and high capacitance properties. We'll estimate the energy consumed on the BP-Mesh network and use the experience of designing a heterogeneous on-chip interconnection network [23] [24] with varying latency, bandwidth and power characteristics to achieve a better energy-efficient design.

Acknowledgements. The work of this paper was supported by the National Science Foundation of China under grant No. 60773024 and No. 60973035, The Central Special Fund for College Basic Scientific Research And Operation (C2009Z028J), The 863 Hi-Tech Research and Development Program of China (2009AA01Z105) and Special Research Fund of The Ministry of Education And

Intel For Information Technology (MOE-INTEL-08-05). We thank all members of the ESCA project. Also we would like to thank reviewers for their helpful feedback.

References

1. Bohr, M.T.: Interconnect scaling - the real limiter to high performance ULSI. In: IEEE International Electron Devices Meeting, pp. 241–244 (1995)
2. Matzke, D.: Will physical scalability sabotage performance gains? IEEE Computer 30, 37–39 (1997)
3. Wolfe, A.: Intel clears up post-tejas confusion. VARBusiness (May 17, 2004), http://www.varbusiness.com/sections/news/breakingnews.jhtml?articleId=18842588
4. Agarwal, V., Hrishikesh, M.S., Keckler, S.W., Burger, D.: Clock rate versus IPC: The end of the road for conventional microarchitectures. In: Proc. Of IEEE 27th International Symposium on Computer Architecture (ISCA-27), pp. 248–259 (2000)
5. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low-power CMOS digital design. IEEE Journal of Solid-State Circuits 27, 473–484 (1992)
6. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., et al.: The Illiac IV computer. IEEE Transactions on Computers C-17, 746–757 (1968)
7. Batcher, K.E.: Design of a massively parallel processor. IEEE Transactions on Computers C-29, 836–840 (1980)
8. Parkinson, D., Hunt, D.J., MacQueen, K.S.: THE AMT DAP 500. In: Proc. Of the 33rd IEEE International Conference of Computer Society, pp. 196–199 (March 1988)
9. Nickolls, J.R.: The design of the MasPar MP-1: A cost effective massively parallel computer. In: Proc. Of the 35th IEEE International Conference of Computer Society, pp. 25–28 (March 1990)
10. Singh, H., Lee, M.-H., Lu, G., Kurdahi, F.J., et al.: MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Transactions on Computers 49, 465–481 (2000)
11. Fujita, Y., Kyo, S., Yamashita, N., Okazaki, S.: A 10 GIPS SIMD processor for PC-based real-time vision applications. In: Proc. Of the 4th IEEE International Workshop on Computer Architecture for Machine Perception (CAMP 1997), pp. 22–32 (October 1997)
12. ClearSpeed Whitepaper: CSX Processor Architecture, http://www.clearspeed.com/newsevents/presskit
13. Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., et al.: Imagine: Media processing with streams. IEEE Micro 21, 35–46 (2001)
14. Fatemi, H., Corporaal, H., Basten, T., Kleihorst, R., Jonker, P.: Designing area and performance constrained SIMD/VLIW image processing architectures. In: Blanc-Talon, J., Philips, W., Popescu, D.C., Scheunders, P. (eds.) ACIVS 2005. LNCS, vol. 3708, pp. 689–696. Springer, Heidelberg (2005)
15. Makino, J., Hiraki, K., Inaba, M.: GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In: Proc. Of the 2007 ACM/IEEE Conference on Supercomputing (SC 2007), pp. 1–11 (2007)
16. Balfour, J., Dally, W.J.: Design tradeoffs for tiled CMP on-chip networks. In: Proc. Of the 20th Annual International Conference on Supercomputing (ICS 2006), pp. 187–198 (June 2006)

17. Das, R., Eachempati, S., Mishra, A.K., Narayanan, V., Das, C.R.: Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs. In: Proc. of IEEE 15th International Symposium on High Performance Computer Architecture (HPCA 2009), pp. 175–186 (Febuary 2009)
18. Banerjee, A., Wolkotte, P.T., Mullins, R.D., Moore, S.W., Smit, G.J.M.: An energy and performance exploration of network-on-chip architectures. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 17, 319–329 (2009)
19. Cannon, L.E.: A cellular computer to implement the kalman filter algorithm. Ph.D. thesis, Montana State University (1969)
20. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: Proc. Of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 133–137 (August 2004)
21. Bahn, J.H., Yang, J., Bagherzadeh, N.: Parallel FFT algorithms on network-on-chips. In: Proc. Of the 5th International Conference on Information Technology: New Generations, pp. 1087–1093 (April 2008)
22. Kumar, R., Zyuban, V., Tullsen, D.M.: Interconnections in multi-core architectures: understanding mechanism, overheads and scaling. In: Proc. Of the 32nd International Symposium on Computer Architecture (ISCA 2005), pp. 408–419 (June 2005)
23. Cheng, L., Muralimanohar, N., Ramani, K., Balasubramonian, R., Carter, J.B.: Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In: Proc. of the 33rd International Symposium on Computer Architecture (ISCA 2006), pp. 339–351 (2006)
24. Flores, A., Aragon, J.L., Acacio, M.E.: An energy consumption characterization of on-chip interconnection networks for tiled CMP architectures. Journal of Supercomputing 45, 341–364 (2008)

Network-on-Chip Routing Algorithms by Breaking Cycles

Minghua Tang[1] and Xiaola Lin[2,*]

[1] School of Information Science and Technology, Sun Yat-sen University, Guangzhou 510275, China
fractal218@126.com

[2] School of Information Science and Technology, Sun Yat-sen University, Guangzhou 510275, China and
Key Laboratory of Digital Life (Sun Yat-sen University), Ministry of Education, Guangzhou 510275, China
linxl@mail.sysu.edu.cn

Abstract. In this paper, we propose a methodology to design routing algorithms for Network-on-Chip (NoC) which are customized for a set of applications. The routing algorithms are achieved by breaking all the cycles in the application specific channel dependency graph(ASCDG). Thus, the result routing algorithms are ensured to be deadlock-free. The proposed method can overcome the shortcomings of the method in [1] which heavily depends on the order of the cycles being treated and has high computational complexity. Simulation results show that the Routing Algorithms by Breaking Cycles (*RABC*) improves the performance significantly.

1 Introduction

In the recent years, the so called Network-on-Chip (NoC) communication infrastructure has been proposed to replace the bus based communication architecture [2,3]. Basing on this paradigm many research groups have proposed different NoC architectures [4,5,6,7,8]. One NoC architecture differs from others by the network topology and routing algorithm [9,10]. We mainly deal with routing algorithm in this paper.

A large number of NoC routing algorithms either adding virtual channels (VCs) [11,12] or not [13,14,15,16,17] have been proposed. On the whole, the VCs added routing algorithms have higher routing adaptivity than those without VCs. However, the VCs may consume considerable precious on chip resources.

Routing algorithms without VCs can be implemented by two distinct methods. Firstly, the routing algorithm is implemented by algorithmic method. If every pair of cores in the network communicates then this is the suitable choice. Nevertheless, there are many specialized NoC systems where the communication happens between part of the pairs of cores. And the routing algorithms can incorporate the communication information to improve their adaptivity [1]. These

* Corresponding author.

routing algorithms are suited to be implemented by the second method which uses routing table.

With the routing table support the routing algorithms can be constructed according to the applications and dynamically updated. However the routing table may significantly consume the precious on chip resources. Fortunately, many researchers have proposed effective techniques to shorten the routing table and maintain the deadlock freedom of the routing algorithms [18,19].

M. Palesi *et al.* [1] present a methodology to design routing algorithms for specific applications. The routing algorithms which do not need VCs are implemented by routing table. And the result routing table is compressed to improve the overall performance with the technique in [18]. It breaks all the cycles in the channel dependency graph (*CDG*) to make the routing algorithms deadlock free.

There are two main shortcomings of the methodology in [1]. Firstly, the routing algorithms are got by sequentially breaking all the cycles of *CDG*. Consequently, the performance of the routing algorithms depends on the sequence of the cycles in *CDG* (the detail is given in Section 3). Secondly, in the worst case, all possible combinations of the cycles in *CDG*, have to be exhaustively processed to find a feasible minimal routing for all communicating pairs [1]. The computational complexity is O(n!), where n is the number of the cycles.

In this paper, we propose the method to develop routing algorithms by breaking cycles (*RABC*) to solve the problems of *APSRA*. The basic idea is that the cycles in *CDG* is considered as a whole but no longer one by one. As a result, the routing algorithms are not dependent on the order of the cycles. And the computational complexity is decreased to O(n), n is the number of the cycles.

The rest of the paper is organized as follows: In Section 2, we first summarize some major related work. Some basic concepts and the principle of the *APSRA* are briefly introduced in Section 3. Then we illustrate the design methodology and the various algorithms of *RABC* in Section 4. The experiment and the performance achievement of the method under various traffic scenarios are detailed in Section 5. Finally in Section 6 we draw some conclusions.

2 Related Work

In the NoC, under the given network topology, the overall performance of the network depends on the switching technique, routing algorithm and flow control strategy etc. The main topic of this paper is the routing algorithms. The routing algorithm determines the number of pathes that can be traversed by the packets.

The routing algorithms can be classified into two categories: deterministic routing and adaptive routing. In the deterministic routing, a pair of communicating cores has only one path, which is determined by the source address and destination address. Although it is simple to implement, the overall performance degrades sharply if the network has nonuniform traffic. Dimension-Order Routing (DOR) [20] is an example of deterministic routing.

In contrast, in the adaptive routing the packets have a large number of selectable paths. Consequently, the packets can choose the fastest path to the destination according to the network status. The overall performance increases significantly since the traffic is evenly distributed in the network.

Glass and Ni [13] present the turn model to develop adaptive deadlock free routing algorithms for NoC. It shows that by prohibiting just enough turns the routing algorithms gets deadlock free since no cycle can be formed. The main drawback is that the adaptivity of the routing function is not uniformly distributed.

Chiu [16] improves the turn model to propose the Odd-Even (OE) turn model which make the routing adaptivity more evenly distributed. The Odd-Even turn model ensures the deadlock freedom of the routing algorithms by restricting the locations where the packets take turns. The OE turn model significantly outperforms the turn model due to the evenly distributed routing adaptivity.

Hu and Marculescu [17] present a routing scheme to take the advantage of both deterministic and adaptive routing algorithms. The router works in the deterministic mode when there is no congestion in the network and switches to adaptive mode when the network is heavily loaded. However, it is difficult to precisely estimate the network contention status.

Although the adaptive routing algorithm provides lots of paths it does not points out which one is the best. So, in order to fully take advantage of the adaptive routing algorithm it is necessary to develop an effective selection strategy which is responsible to select the proper path. When a packet needs to be forwarded the router calls the routing function to calculate the set of admissible output ports *(AOP)* then calls the selection function to choose the best one. Ascia *et al.* [21] propose a selection strategy named *Neighbors-on-Path (NoP)*, which aims to choose the channel that allows the packet to be routed to its destination as fast as possible. However, the *NoP* selection will encounter inefficiency when combined with Odd-Even routing algorithm. M. Tang and X. Lin [22] improve the NoP selection to design the *Advanced NoP (ANoP)* selection strategy which is suitable for Odd-Even routing algorithm.

3 Introduction of Application Specific Routing Algorithms (*APSRA*)

To design NoC routing algorithms, the application specific routing algorithms (*APSRA*) is the first methodology which makes use of characteristics of communication traffic. To make the paper self-contained, we briefly describe the basic idea of the *APSRA* methodology.

The framework of the *APSRA* methodology is shown in Table 1. It computes the routing table (RT) according to the inputs of *CG*, *TG*, and *M*.

The first input is the communication graph (*CG*) which is a directed graph. The vertexes of *CG* represent the tasks of the application and each directed arc in *CG* stands for the communication of the two linked tasks.

The second input of topology graph (*TG*) which is also a directed graph is the NoC architecture. Each vertex is a node of the on chip network and each directed arc represents a physical unidirectional channel between two nodes.

Table 1. APSRA framework

```
1 APSRA(in: CG, TG, M, out: RT)
2 {
3  R=MinFullAdaptiveRouting(CG,TG,M);
4  ASCDG=BuildASCDG(CG, TG, M, R);
5  CYCLES=GetCycles( ASCDG );

6  RET = BreakCycles(CYCLES, ASCDG,
    CG, TG, M, R);

7  ExtractRoutingTable( RET, R, RT );
8 }
```

The input of *M* is the mapping function which responsibility is mapping the *CG's* tasks onto the nodes of *TG*. The mapping result determines which pairs of NoC nodes communicate and which never communicate. *APSRA* exploits the information to help improving the routing adaptivity.

The main *APSRA* algorithms can be partitioned into three phases.

The first phase (from line 3 to line 5) which contains three steps completes the task of preprocessing. In the first step (line 3) of this phase a fully adaptive and minimal routing function is created. This routing function is not deadlock free currently. In the second step (line 4) the channel dependency graph (*CDG*) [23] is firstly constructed based on the previous routing function. Then the *Application Specific Channel Dependency Graph (ASCDG)* [1] is extracted from *CDG*. The *ASCDG* only contains those direct dependencies which are generated by the application specific communication. In the third step (line 5) the set *CYCLES* of all the cycles in the *ASCDG* are extracted.

The critical job is fulfilled in the second phase (line 6). Every cycle in *CYCLES* needs to be broke to make the routing algorithm deadlock free. If a cycle's size is n then there are n dependencies in this cycle. It is sufficient to break the cycle if any one of the n dependencies is removed. As for which one is the best choice, the heuristic in [1] claims to remove the dependency which minimally impacts routing adaptivity. The removed dependency is a restricting in the ultimate routing function.

In the last phase (line 7) the routing table is constructed by the information from the routing function and the *CG*.

The inefficiency of the BreakCycles procedure lies in that it depends on the order in which the cycles are processed.

On the one hand, if a cycle can not be broken, that is, removing any dependency will lead to a non-minimal routing function then it must try another order of breaking the cycles. In the worst case all possible combinations of cycles have to be exhaustively probed. Thus the computational complexity is O(n!), where n is the number of the cycles.

On the other hand, the performance of final routing algorithms also heavily depends on the order in which the cycles are treated. Figure 1 is the performance of two routing algorithms. The curve labeled descend stands for the routing algorithm by descendingly sorting the cycles according to the size of the cycles. Whereas the curve labeled not-sorted represents the routing algorithm in which the cycles are not sorted. The sequence of the cycles enormously impacts the performance of the routing algorithms. But we do not in advance know which order will produce the best performance routing function.

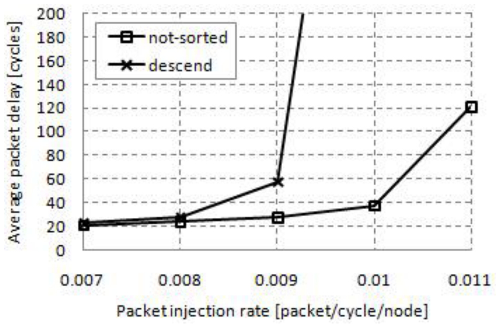


Fig. 1. Impact of cycle order on performance

4 Main *RABC* Algorithm

To overcome the shortcomings of the *APSRA* algorithm, we propose the *Routing Algorithms by Breaking Cycles (RABC)* methodology. The *RABC* framework is like that of *APSRA* in Table 1. The difference lies in the algorithm for the BreakCycles procedure. The algorithms for the *RABC* BreakCycles procedure is shown in Table 2.

The *APSRA* chooses the best suitable dependency among the candidates in only one cycle. In contrast, *RABC* selects the best suitable dependency among all the dependencies. So, the first step (line 3) is to find all the dependencies (*DEPENDS*) in the *ASCDG*.

If there are cycles not yet be broken then some dependencies need to be selected to be removed (line 4). In line 5, each dependency is graded. The score of a dependency equals the number of cycles it can break.

The *MAXDEPEND* which has the highest score and keeps the routing function minimal is found in line 6. This dependency is stored as a restriction (*RESTRICTION*) in the ultimate routing function (line 8). Before that the cycles broken by the *MAXDEPEND* is deleted from *CYCLES* (line 7).

The final routing algorithms is guaranteed to be deadlock free and minimal when all the cycles in the *ASCDG* are broken [1].

Table 2. RABC BreakCycles Procedure

```
RABCBreakCycles( CYCLES, ASCDG, CG, TG, M, R )
{
   DEPENDS = GetDependencies( ASCDG );
   while( CYCLES is not empty ){
     SCORES = GetScore( DEPENDS, CYCLES );
     MAXDEPEND = MaxDepend( DEPENDS, CYCLES );
     DeleteBrokeCycles( CYCLES, MAXDEPEND );
     StoreMaxDepend( MAXDEPEND, RESTRICTION );
   }
}
```

It is important to notice that not every dependency can be prohibited to keep the result routing algorithm minimal. As shown in Figure 2, if the dependency $l_{12} \rightarrow l_{24}$ has been forbidden then the the dependency $l_{13} \rightarrow l_{34}$ can not be prohibited to ensure that there is minimal path from p1 to p4. On the other hand, we can show that if the dependency $l_{13} \rightarrow l_{34}$ is not restricted then the routing function is minimal. For example, the dependency $l_{34} \rightarrow l_{42}$ is chosen to be forbidden. The path from p1 to p4 is minimal. Since the path from another arbitrary node pi to p1 is not affected by that dependency, the path is still minimal. Consequently the minimal path from pi to p4 is still preserved.

The same analysis is applied to other pair of nodes. It indicates that the ultimate routing algorithm is minimal if the contradictory dependencies such as $l_{12} \rightarrow l_{24}$ and $l_{13} \rightarrow l_{34}$ are not prohibited at the same time.

There are three main advantages of the *RABC* algorithm. Firstly, the new algorithm of breaking cycles treats all cycles as a whole but not one by one

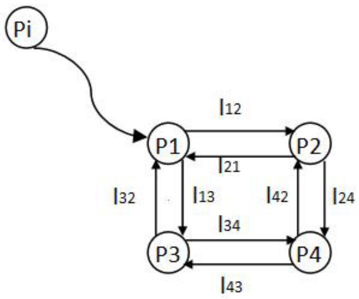


Fig. 2. Preserve minimal path

like the previous algorithm. As a result the performance of the ultimate routing function does not depend on the order in which the cycles are stored in *CYCLES* set. Secondly, its computational complexity is very low. Suppose the number of cycles is n, the function of GetScore (line 5) and DeleteBrokeCycles (line 7) needs to process every cycle. So their computational complexities are O(n). While the operation of MaxDepend (line 6) and StoreMaxDepend (line 8) is irrelevant to n. Their computational complexities are O(1). Moreover the running times of the while loop (line 4) is also irrelevant to n. As a whole the computational complexity of the proposed algorithm is lowered to O(n). Thirdly, the adaptivity of the routing function is increased greatly.

5 Performance Evaluation

We first introduce the simulation configurations and then give the simulation results.

5.1 Experimental Configurations

We extend the NoC simulator *Noxim* [24] developed in SystemC which is flit-accurate and open source to evaluate the performance of the *RABC* method. The performance of *RABC* is compared with that of *APSRA*. The packet injection rate (*pir*) is taken as the load parameter. The performance metrics are *average packet latency* and *throughput* which are defined as follows respectively:

$$Average\ packet\ latency = \frac{1}{K}\sum_{i=1}^{K} lat_i$$

where K refers to the total number of packets arrived in their destinations and lat_i is the latency of *ith* packet.

$$throughput = \frac{total\ received\ flits}{(number\ of\ nodes) * (total\ cycles)}$$

where *total received flits* represents the number of flits taken in by all destinations, *number of nodes* is the number of network nodes, and *total cycles* ranges from the start to the end of the simulation.

Each simulation goes on for 20,000 cycles following 1,000 cycles of initialization. The simulation at each *pir* is iterated a number of times to ensure accuracy. The configurations of the simulation are shown in Table 3.

In this paper, we use three synthetic traffic scenarios to evaluate the performance of the *RABC* methodology. In the transpose1 traffic, node (i, j) only sends message to the symmetrical node (N-1-j, N-1-i), N is the size of the network. The hotspot traffic scenarios, which are thought to be more realistic are also considered. In these traffic, some nodes are specified as hotspot nodes, which receive the 20 percent hotsopt traffic except the normal traffic. In the first hotspot traffic, which is shortened *hs-c*, the four hotspot nodes [(3, 3), (4, 3), (3, 4), (4, 4)] situate at the center of the mesh network. And the second hotspot traffic scenario, which is abbreviated as *hs-tr*, has the four hotspot nodes [(6, 0), (7, 0), (6, 1), (7, 1)] positioned at the top right corner of the mesh.

Table 3. Experimental configurations

Topology	Mesh-based
Network size	8×8
Port buffer	Four flits
Switch technique	Wormhole switching
Selection strategy	Buffer level
Traffic scenario	transposed and hotspot
Packet size	Eight flits
Traffic distribution	Poisson
Simulator	Noxim

5.2 Results

Routing adaptivity is an important metric for estimating the performance of the adaptive routing algorithms [1]. High adaptivity routing algorithm has the potential to provide more paths for the communication. Formally, it is defined as the ratio of the number of the permitted minimal paths to the whole number of minimal paths from the source to the destination node. Then the adaptivity for all the communications is averaged to get the average adaptivity of the routing algorithm.

Figure 3 compares the average routing adaptivity (normalized to *APSRA*) of the *APSRA* and *RABC* routing algorithms from network size 5×5 to 8×8 under uniform traffic. The improved routing adaptivity of the *RABC* over *APSRA* ranges from 15.7% to 28.2%.

The simulation results for transpose1 traffic scenario are presented in Figure 4. The maximum load that is sustainable by the network is greatly increased by the *RABC* routing algorithm. At the same time the throughput is also improved.

The results for *hs-c* and *hs-tr* are respectively depicted in Figure 5 and Figure 6. The packet latencies are decreased while the throughput is maintained in both cases.

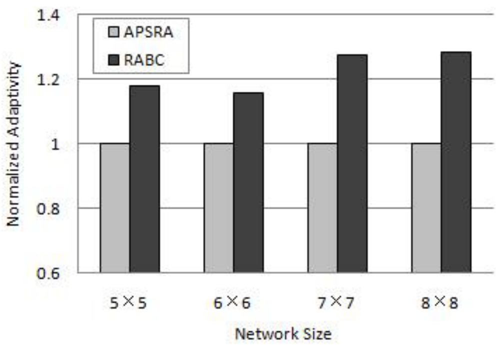


Fig. 3. Routing adaptivity of the APSRA and RABC routing algorithms

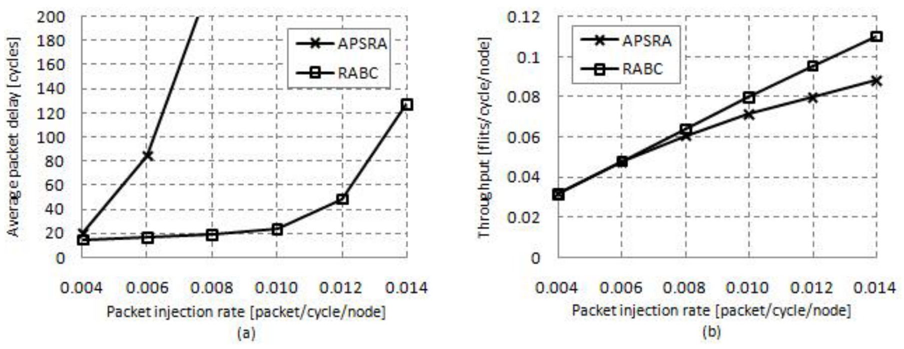


Fig. 4. (a) Latency and (b) throughput variation under transpose1 traffic scenario

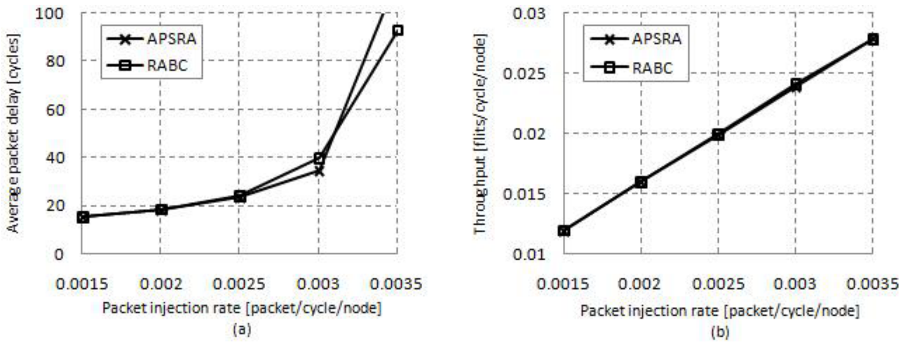


Fig. 5. (a) Latency and (b) throughput variation under hs-c traffic scenario

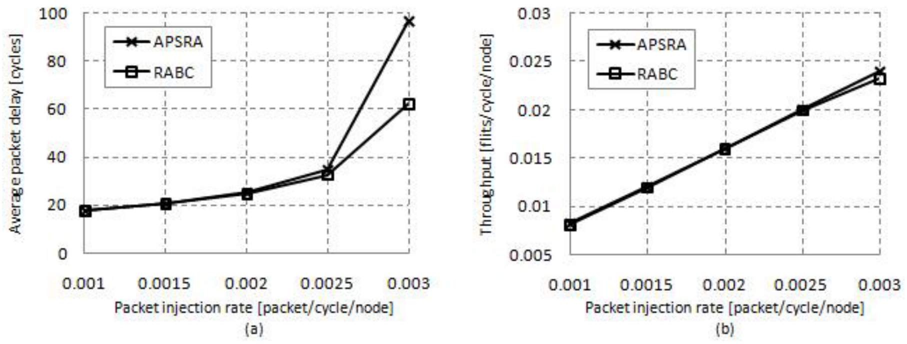


Fig. 6. (a) Latency and (b) throughput variation under hs-tr traffic scenario

6 Conclusions

In this paper, we propose the *RABC* to design application specific routing algorithms for NoCs. *RABC* makes use of the communication information of the application to construct the *ASCDG* like *APSRA* methodology. In order to achieve deadlock free routing algorithms the *RABC* considers the cycles in *ASCDG* as a whole. The *RABC* has three advantages over *APSRA*. Firstly, the performance of the ultimate routing algorithm does not depend on the order in which the cycles are processed. Secondly, the computational complexity of the algorithm itself is greatly decreased. At last, the adaptivity of the routing algorithm is significantly increased.

Acknowledgments

This work was supported in part by NSFC under Projects 60773199, U0735001, and 985 II fund under Project 3171310.

References

1. Palesi, M., Holsmark, R., Kumar, S., Catania, V.: Application Specific Routing Algorithms for Networks on Chip. IEEE Trans. Parallel and Distributed Systems. 20, 316–330 (2009)
2. Dally, W.J., Towles, B.: Route Packets, Not Wires: On-Chip Interconnection Networks. In: Design Automation Conf., pp. 684–689 (2001)
3. Benini, L., De Micheli, G.: Networks on Chips: A New SoC Paradigm. Computer 35, 70–78 (2002)
4. Millberg, M., Nilsson, E., Thid, R., Kumar, S., Jantsch, A.: The Nostrum backbone - a communication protocol stack for networks on chip. In: VLSI Design Conference, pp. 693–696 (2004)
5. Rijpkema, E., Goossens, K., Wielage, P.: A router architecture for networks on silicon. In: 2nd Workshop on Embedded Systems (2001)
6. Kumar, S., Jantsch, A., Soininen, J.P., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K., Hemani, A.: A Network on Chip Architecture and Design Methodology. In: Proc. IEEE CS Ann. Symp. VLSI, pp. 105–112 (2002)
7. Karim, F., Nguyen, A., Dey, S.: An Interconnect Architecture for Networking Systems on Chips. IEEE Micro. 22, 36–45 (2002)
8. Pande, P.P., Grecu, C., Ivanov, A., Saleh, R.: Design of a Switch for Network on Chip Applications. In: Proc. IEEE Intl. Symp. Circuits and Systems, vol. 5, pp. 217–220 (2003)
9. Marculescu, R., Bogdan, P.: The Chip Is the Network: Toward a Science of Network-on-Chip Design. Foundations and Trends in Electronic Design Automation, 371–461 (2009)
10. Bjerregaard, T., Mahadevan, S.: A Survey of Research and Practices of Network-on-Chip. ACM Computing Surveys, 1–51 (2006)
11. Mello, A., Tedesco, L., Calazans, N., Moraes, F.: Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC. In: Proc. 18th Symp. Integrated Circuits and System Design, pp. 178–183 (2005)

12. Pullini, A., Angiolini, F., Meloni, P., Atienza, D., Srinivasan MuraliRaffo, L., De Micheli, G., Benini, L.: NoC Design and Implementation in 65 nm Technology. In: Proc. First Intl. Symp. Networks-on-Chip, pp. 273–282 (2007)
13. Glass, C.J., Ni, L.M.: The Turn Model for Adaptive Routing. J. Assoc. for Computing Machinery 41, 874–902 (1994)
14. Chien, A.A., Kim, J.H.: Planar-Adaptive Routing: Low-Cost Adaptive Networks for Multiprocessors. J. ACM 42, 91–123 (1995)
15. Upadhyay, J., Varavithya, V., Mohapatra, P.: A Traffic-Balanced Adaptive Wormhole Routing Scheme for Two-Dimensional Meshes. IEEE Trans. Computers 46, 190–197 (1997)
16. Chiu, G.M.: The Odd-Even Turn Model for Adaptive Routing. IEEE Trans. Parallel and Distributed Systems 11, 729–738 (2000)
17. Hu, J., Marculescu, R.: DyADSmart Routing for Networks-on-Chip. In: Proc. 41st Design Automation Conf., pp. 260–263 (2004)
18. Palesi, M., Kumar, S., Holsmark, R.: A Method for Router Table Compression for Application Specific Routing in Mesh Topology NoC Architectures. In: Proc. Sixth Intl. Workshop Systems, Architectures, Modeling, and Simulation, pp. 373–384 (2006)
19. Bolotin, E., Cidon, I., Ginosar, R., Kolodny, A.: Routing table minimization for irregular mesh NoCs. In: Proc. Conf. Des., Autom. Test Eur., pp. 942–947 (2007)
20. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufman, San Mateo (2004)
21. Ascia, G., Catania, V., Palesi, M., Patti, D.: Implementation and Analysis of a New Selection Strategy for Adaptive Routing in Networks-on-Chip. IEEE Trans. Computers 57, 809–820 (2008)
22. Tang, M.H., Lin, X.L.: An Advanced NoP Selection Strategy for Odd-Even Routing Algorithm in Network-on-Chip. In: Hua, A., Chang, S.-L. (eds.) ICA3PP 2009. LNCS, vol. 5574, pp. 557–568. Springer, Heidelberg (2009)
23. Dally, W.J., Seitz, C.L.: Deadlock-free message routing in multiprocessor interconnection networks. IEEE Trans. Comput. 36, 547–553 (1987)
24. Sourceforge.net, Noxim: Network-on-chip simulator (2008), http://noxim.sourceforge.net

A Fair Thread-Aware Memory Scheduling Algorithm for Chip Multiprocessor*

Danfeng Zhu, Rui Wang, Hui Wang, Depei Qian, Zhongzhi Luan, and Tianshu Chu

Sino-German Joint Software Institute
School of Computer Science and Engineering,
Beihang University,
100191, Beijing, P.R. China
{danfeng.zhu,rui.wang,hui.wang,depei.qian,
zhongzhi.luan}@jsi.buaa.edu.cn

Abstract. In Chip multiprocessor (CMP) systems, DRAM memory is a critical resource shared among cores. Scheduled by one single memory controller, memory access requests from different cores may interfere with each other. This interference causes extra waiting time for threads and leads to negligible overall system performance loss. In conventional thread-unaware memory scheduling patterns, different threads probably experience extremely different performance; one thread is starving severely while another is continuously served. Therefore, fairness should also be considered besides data throughput in CMP memory access scheduling to maintain the overall system performance. This paper proposes a Fair Thread-Aware Memory scheduling algorithm (FTAM) that ensures both the fairness and memory system performance. FTAM algorithm schedules requests from different threads by considering multiple factors, including the source thread information, the arriving time and the serving history of each thread. As such FTAM considers the memory characteristic of each thread while maintains a good fairness among threads to avoid performance loss. Simulation shows that FTAM significantly improves the system fairness by decreasing the unfairness index from 0.39 to 0.08 without sacrificing data throughput compared with conventional scheduling algorithm.

1 Introduction

Chip multiprocessor (CMP) [1,2] architecture is now believed to be the dominant trend for future computers. In CMP systems multiple independent processing cores share the same DRAM memory system. Different threads running on different cores send memory access requests simultaneously and may interfere with each other while accessing the shared resources. As the number of cores on a chip increases, the pressure on the DRAM system also increases. The limited bandwidth therefore becomes the bottleneck of the whole system and poses a severe resource management issue on memory access scheduling.

* This work is supported by National Science Foundation of China under grant No. 60873053, No. 60673180 and National Hi-tech R&D Program (863 program) of China under grant No. 2009AA01Z107.

Conventional DRAM memory controllers are designed for maximizing the data throughput obtained from the DRAM. It is essentially a sequential scheduling scheme that simply gathers all the memory access requests from different threads into one queue and employs existing scheduling algorithm onto this queue. They do not take the source thread of requests into consideration and thus are thread-unaware. This unawareness leads to a severe unfairness among the cores. Unfairly prioritizing some threads over others may increase the access latency and lead to a substantial performance loss of CMP systems. If a thread generates a stream of requests that access the same row in the same bank, another thread that requests for data in another row will not be served until the first thread's requests are completed. This results in big performance loss as some cores have to spend lots of time waiting for their requests to be served.

Therefore, for a scalable CMP system, memory access scheduling techniques that control and minimize inter-thread interference are necessary. Not only the effectiveness of memory scheduler should be taken into consideration but also the fairness should be provided among threads.

In this paper, we propose a new memory scheduling algorithm, the Fair Thread-Aware Memory scheduling algorithm (FTAM) for CMP systems. FTAM takes into account the locality each thread exhibits and provides fairness among cores without sacrificing overall memory system throughput. Source-thread, Arriving-time and Serving-History are the three metrics for computing each thread's priority in memory scheduling.

The rest of this paper is organized as follows: Section 2 gives related works on memory scheduling algorithm of multi-core processor; Section 3 introduces the proposed FTAM algorithm; we gives the simulation methodology and simulation results in section 4 and section 5, respectively; finally, we conclude in Section 6.

2 Related Works

2.1 DRAM Architecture

A modern DRAM chip is a 3-D device with dimensions of bank, row and column, as shown in Fig.1. The memory access identifies the address that consists of the bank, row and column fields. Each bank has a single row-buffer, which can contain at most one row. The data in a bank can be accessed only from the row-buffer. So an access to the memory may require three transactions before the data transfer: bank pre-charge, row activate and column access. A pre-charge charges and prepares the bank. An activate copies an entire row data from the array to the row-buffer. Then a column access can access the row data. The amount of time it takes to service a DRAM request depends on the status of the row-buffer and falls into three categories:

- *Row hit*: when the requested row is currently open in the row-buffer;
- *Row closed*: if there is no open row in the row-buffer;
- *Row conflict*: if the requested row differs from the current one in the row-buffer.

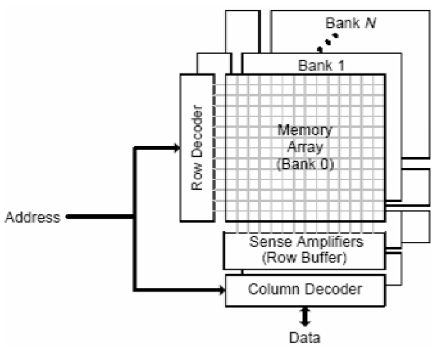


Fig. 1. A Modern 3-D DRAM Chip

2.2 DRAM Memory Controller

A memory controller is the interface between processors and memory system, translating memory requests into memory commands. The basic structure of a modern memory controller consists of a memory scheduler, a request buffer, and write/read buffers.

The memory scheduler is the core of the memory controller. The scheduler reorders requests based on the implemented scheduling policy. Memory schedulers have two levels. The first level is the per-bank schedulers. Each bank owns its per-bank scheduler, prioritizing requests to this bank and generating a sequence of DRAM commands (while respecting the bank timing constraints). The second level is the across-bank channel scheduler. It receives the banks' ready commands and issues the one with the highest priority (while respecting the timing constraints and scheduling conflicts in the DRAM address and data buses). The request buffer contains each memory request's state, e.g., the address, type and identifier of the request. The write/read buffers contain the data being written to/read from the memory.

The controller's function is to satisfy memory requests by issuing appropriate DRAM commands. Yet how to optimize the memory controller to alleviate the gap between the processing ability and memory bandwidth is not an easy task. First of all, state-of-the-art DDR2 SDRAM chips may have over 50 timing constraints [3] including both local (per-bank) and global (across banks due to shared resources between banks) timing constraints. All these must be obeyed when scheduling commands. Second, the controller should intelligently reorder the memory access requests it receives in the request buffer by its prioritizing rules to optimize system performance. Scheduling decision must be carefully made because it has significant impact on not only DRAM throughput and latency but also the whole system performance.

Modern memory controllers use FR-FCFS [4,5] policy to schedule memory access requests. FR-FCFS prioritizes memory requests that hit in the row-buffer over others. If no request hits the row-buffer, older requests are prioritized over younger ones. For single-threaded systems FR-FCFS policy was shown to provide the best average performance. But this scheduling algorithm is thread-unaware, i.e., it does not take into account the interference among different threads on scheduling decisions. Although it may achieve high system throughput, some threads may suffer from extreme starvation. For example if one thread continuously issues requests that have a very high row-buffer hit rate, then FR-FCFS scheduling will unfairly prioritize these request in series, and other threads have to be delayed for long time and sacrifice their processing power. Therefore, in CMP systems FR-FCFS cause unfairness among cores, as different threads running together on the same chip can experience extremely different memory system performance. This is a big performance loss for CMP systems. So in CMP systems scheduling policy should consider more than just throughput.

2.3 Related Works on Memory Scheduling Algorithms

Many memory scheduling policies were proposed to optimize memory scheduling, but most of them are thread-unaware. Recent years the focus has been shifted to providing fairness to multi-core and multi-threaded processors. Several memory scheduling policies with features adapted to multi-threaded environment have been proposed.

Zhu et al. [6] evaluated thread-aware memory access scheduling schemes and found that thread-aware DRAM access scheduling schemes may improve the overall system performance by up to 30%, on workload mixes of memory-intensive applications. The considered thread states include the number of outstanding memory requests, the reorder buffer occupancy, and the issue queue occupancy. While this work mainly focused on SMT system and was tested in a single core processor, it has not specifically considered the scenario of CMP system. Nesbit et al. [7] proposed a memory scheduler based on the network fair queuing (FQ) concept. Its objective is to ensure that each thread is offered its allocated share of aggregate memory system and distribute any excess memory bandwidth to threads that have consumed less excess bandwidth in the past. Mutlu et al. [8] proposed a Stall-Time Fair Memory scheduler (STFM) to provide quality of service to different threads sharing the DRAM memory system. This scheme can reduce the unfairness in the DRAM system while also improving system throughput (i.e. weighted speedup of threads) on a variety of workloads and systems. The goal of the proposed scheduler is to "equalize" the DRAM-related slowdown experienced by each thread due to interference from other threads, without hurting overall system performance. However quite a lot of additional work is introduced in this STFM policy to estimate the slowdown of each thread thus complicated the implementation. Zheng et al. [9] evaluated a set of memory scheduling policies and proposed a memory efficiency-based scheduling scheme called ME-LREQ for multi-core processors. Based on the conventional least-request scheme, ME-LREQ considers the memory-efficiency of application running on each core to improve the memory bandwidth and latency. Mutlu et al. [10] proposed a parallelism-aware batch scheduling policy for CMP systems. Based on the finding that inter-thread interference can destroy bank-level parallelism of individual threads and leads to throughput degradation, this parallelism-aware policy optimizes conventional FR-FCFS scheme for intra-thread bank-parallelism. Fang et al. [11] presents a core-aware scheduling scheme similar to our work, but provide no fairness consideration for different threads.

Different from these ideas, our algorithm is a balance between fairness and throughput. We pick multiple metrics which are simple but tested useful to achieve this balance: the Source-thread, the Arriving-time, and the Serving-history of these threads. Requests from different threads are grouped in different queues and an arbiter window is used to flexibly control queues, and compute the priority of each request according to the above three factors. These metrics are very easy to get and maintain, so our proposed algorithm is very easy to implement and only need extreme low hardware consumption.

3 Fair Thread-Aware Memory Scheduling Algorithm

In this section, we introduced a novel fair thread-aware memory access scheduling for CMP systems, Fair Thread-Aware Memory Scheduling Algorithm (FTAM). In FTAM, memory requests from different threads are maintained in different queues according to their arriving time. An Arbiter Window is placed on the head of all the queues and the headmost requests are grouped inside the arbiter window to be prioritized before the outside ones. In the arbiter window, we use multiple factors to calculate their priority including *Source-thread, Arriving-time* and *Serving-history.*

Source-thread prioritizes requests from the same thread as the former scheduled one over others guaranteeing the scheduling efficiency and memory system throughput. Arriving-time prioritizes old requests over young requests avoiding starvation. Serving-history prioritizes requests from the thread that have been serviced least of times over others providing fairness among all the threads. Once the thread with highest priority is serviced, the queue moves forwards and the above procedure is repeated. The use of the arbiter window and the synthesis priority metrics assures both performance and fairness of FTAM.

3.1 FTAM Scheduling Policy

Our proposed scheduling algorithm FTAM is an efficient memory scheduling solution which incorporates thread information into scheduling decisions and provides high CMP system performance while keeping fairness among different threads. In FTAM, memory access requests from different treads are in different queues according to their source thread. An arbiter window is used to group the headmost requests of each queue as the candidates for next serving.

Let n represent the number of threads. For the n candidate requests in the arbiter window, FTAM calculates their priority based on following three factors: source-thread, arriving-time and serving-history. The request with the highest priority is moved out of the window and serviced first. Then its queue moved forward to the arbiter window and the scheduler repeats the above procedure. If two or more requests in the window have the same priority then FC-FRFS rule is applied.

3.2 Factors for Priority Calculation

We choose three factors to calculate the thread's priority: Source-thread, Arriving-time and Serving-History.

– *Source-thread:* Requests from the same thread probably have better locality than the ones from different threads, so prioritizes requests from the same thread as the former scheduled one over others is a guarantee for higher row-hit thus to decrease memory access latency and improve the memory system throughput.

– *Arriving-time:* Always scheduling requests from the same thread results in starvation of other threads. High row-hit rate may be achieved though, the overall system performance is hurt because some cores wastes long time waiting. So we use Arriving-Time to prioritize old requests over young requests avoiding possible starvation of some threads.
– *Serving-history:* This metric prioritizes requests from thread that has been serviced least times over others providing fairness among all the threads.

We use *waiting time* as the measurable parameter to stand for arriving-time. Let WT_i represents the waiting time of thread *i*. In terms of serving-history we use *accumulated serving times* as the measurable parameter counting how many requests of this thread has been serviced. Let AST_i represents the accumulated serving times of thread *i*.

We define the weight of the three factors in the calculation of priority as follows:

α : the weight of waiting time;
β : the weight of accumulated serving times;
γ : the weight of source thread factor (with high row-hit probability).

3.3 Algorithm Description

1. Queue requests according to their source thread.
2. Refresh the accumulated serving times of each thread.
3. Calculate the priority for each request located in the arbiter window.
   The priority calculation formula of thread *i* is:

$$P_i = \alpha * WT_i + \beta * AST_i + \gamma \tag{1}$$

After the calculation, the request with highest priority will be selected to be serviced before others.

3.4 Example

In this section, we give a simplified example illustrating how FTAM works. Let's assume that we have four cores running threads T1, T2, T3 and T4. The last serviced request is from T2. At the point of scheduling they have been serviced 25, 20, 5 and 10 times respectively since the threads starts, and the headmost memory access requests of these four threads, R1, R2, R3 and R4 have been waiting for 30ns, 10ns, 20ns and 20ns. As shown in Fig.2, memory requests from each thread are grouped in their request queues. The accumulated serving times are stored for each thread. The headmost requests of all the threads are grouped in an arbiter window for priority calculation.

For simplicity, we assume $\alpha = 1$, $\beta = -1$, and $\gamma = 10$ in (1). So the priority of R1, R2, R3 and R4 are:

$Priority_{R1}$ = 1*30 + -1*25 + 0 = 5
$Priority_{R2}$ = 1*10 + -1*20 + 10 = 0
$Priority_{R3}$ = 1*20 + -1*5 + 0 = 15
$Priority_{R4}$ = 1*20 + -1*10 + 0 = 10

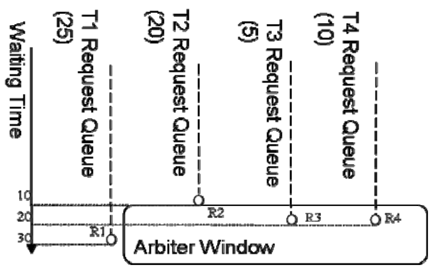


Fig. 2. Priority calculation with FTAM

So R3 from thread T3 is serviced first for it has the highest priority among all the requests in the arbiter window. While using FCFS, R1 will be first serviced and using FR-FCFS, R2 will be serviced first.

3.5 Implementation Analysis

The algorithm only needs a very small extra space to store the waiting time for each memory access request and the accumulated serving times for each thread. Assuming the request buffer size of the controller is *L*, 4*L* bytes are needed to store the arriving time for each request. For an n-core processor, 2*n* bytes are needed to store the accumulated serving times for each thread. Compared with other complicate algorithms, FTAM has the least hardware resource requirement. Meanwhile FTAM does not need extra pre-calculation for information collection.

3.6 Pseudo Code of FTAM

Following is the pseudo code of our FTAM memory access request scheduling algorithm:

```
INPUT: request queues according to the source thread
QUEUE[number_of_threads]
OUPUT:the next request to be issued
BEGIN
  priority <- 0
  min_priority <- MAX
  min_where <- -1
  i <- 0
  While i<number_of_threads
    If ( QUEUE[i] not empty) Then
       If i == the thread number called last time Then
          priority <- Same_Thread
       priority <- priority - the request's wait
       periods(cycles)
       priority <- priority + the thread's called time
       Select min_priority, min_where
  End While
  Service the request at min_where
  last called thread's number < min_where
```

```
    thread[min_where]'s called time + 1
    Select min_call <- min{thread[i]'s called time,
                               0<=i<number_of_threads}
    While i<number_of_threads
      thread[i]'s called time -= min_call
    End While
  END
```

4 Simulation Setup

4.1 Simulation Methodology

First we define the metrics to measure the performance of the FTAM algorithm, and compare it with the FR-FCFS. The metrics take both system fairness and memory access speed into consideration.

Our fairness metric is the unfairness index of the system, which is the standard deviation of each thread's interference ratio. Assuming that the individual execution time of a thread is $T_{exclusive}$ when the memory access is exclusive, the execution time of a thread is T_{shared} when multiple threads run simultaneously, and the extra stall time caused by inter-thread interference that a thread experiences is $T_{interfere}$, so we have $T_{shared} = T_{exclusive} + T_{interfere}$. A thread's interference ratio, IR, is defined as IR = $T_{interfere}$ / $T_{exclusive}$. This interference ratio reflects the impact caused by inter-thread interference in memory access on the thread. Bigger the difference between each thread's interference ratio is, more unfair the system is. Our goal is to minimize the gap between the interference ratio of different threads to maintain a good fairness in the CMP system. So the unfairness index of the system, F, is defined as the standard deviation of each thread's interference ratio, as shown in (1), where IR_i is the interference ratio of thread *i*, is the average value of interference ratio of all the threads and n is the number of the threads. A smaller unfairness index indicates better system fairness. The unfairness index is 0 in a perfectly fair system.

$$F = \sqrt{\frac{\sum_{i=1}^{n}\left(IR_i - \overline{IR}\right)^2}{n}} \tag{2}$$

With respect to the throughput measurement, we use the average memory accessing latency. The memory access delay time from all the threads is recorded and their average value is computed for both FTAM and FR-FCFS algorithm.

4.2 Simulation Setup

We use M5 [12] as the base architectural simulator and extend its memory scheduling part to simulate the conventional scheduling algorithm and our proposed FTAM algorithm. The detailed parameters are listed in Table 1.

In our experiments, each processor core is single-threaded and runs a distinct application. We use FFT and Cholesky benchmark in the SPLASH-2 [13] suite, and use two test application we developed, a data density application named DDT and a computation density application named CDT. The value of α, β and γ is set as 1, -1 and 10.

Table 1. Simulation Setup

Parameters	Value
Processor	4 core, 2GHz,,x issue per core,y-stage pipleline
Functional Unit	4 ALU
L1 cache(per core)	64KB, Inst/64KB Data,2-way,64B line, hit latency: 1 cycle Inst/3-cycle Data
L2 cache(share)	256KB,4-way,64B line,15-cycle hit latency
Memory	4/2/1-channels,2-DIMMs/channel,2-ranks/DIMM,8-banks/rank,9-devices/rank
Memory contoller	64-entry buffer,15ns overhead

5 Simulation Result

To evaluate the performance of FTAM, firstly each application is executed alone and the $T_{exclusive}$ is recorded, then the four applications are executed concurrently respectively scheduled by FR-FCFS and FTAM and the T_{shared} of each thread is recorded. Thus we get the $T_{interfere}$ of each thread under both FR-FCFS and FTAM scheduling.

Fig.3 shows the execution time of the four applications under FR-FCFS and FTAM. From the results we can see that the execution time of different threads are quite various under FR-FCFS scheduling, ranging from 32ms to 72ms, as FR-FCFS probably schedules requests from one thread continuously for higher row-hit and results in long waiting time of the others. While under FTAM all the application runs with roughly equal chance to get the memory access but meanwhile tries to maintain high row-hit possibility. So though FFT and DDT takes even a little bit longer execution time than under FR-FCFS, significant decrease is achieved in Cholesky and CDT, the execution time varies within a smaller range from 37ms to 51ms, showing that a significant performance loss is saved on these two cores.

Fig.4 and Fig.5 shows the different impact of inter-thread interference in memory access under FTAM and FR-FCFS scheduling. Fig.4 is the interference time and Fig.5 is the interference ratio of these four applications under our FTAM scheduling algorithm and FR-FCFS. From the results we can see that under FR-FCFS scheduling the

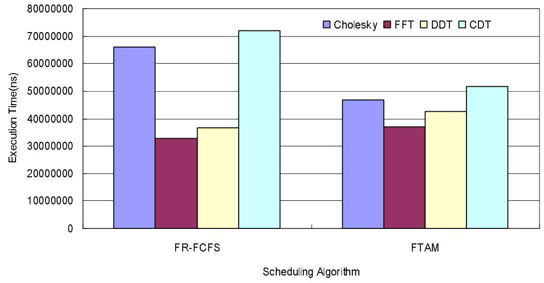


Fig. 3. Comparison of fairness between FTAM and FR-FCFS

interference time and interference ratio different threads experience are quite different, ranging from 8.6ms to 36ms and from 35.92% to 106.84% respectively, as FR-FCFS never considers fairness. While under FTAM though FFT and DDT have even a little bit longer interference time and larger interference ratio than under FR-FCFS, significant decrease is achieved in Cholesky and CDT, and the difference of the interference time and interference ratio among these four threads are quite small. The ranges are from 13ms to 17ms and from 40.31% to 58.11% respectively. It is obvious that FTAM significantly reduces the interference time for some threads and evens the interference ratio among different threads.

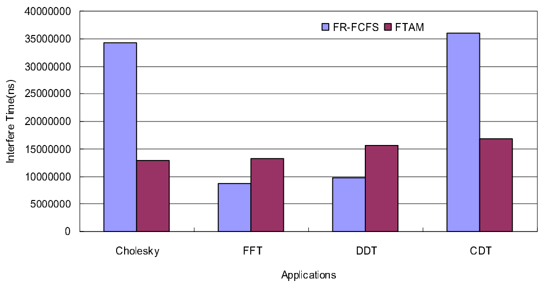


Fig. 4. Comparison of interference time between FTAM and FR-FCFS

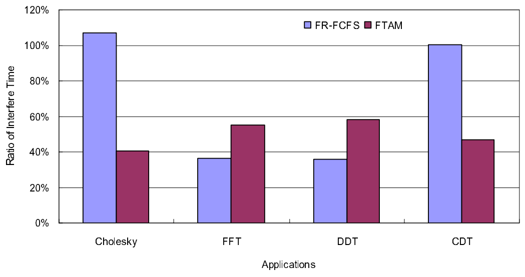


Fig. 5. Comparison of interferece time ratio between FTAM and FR-FCFS

From these figures we can get the *unfairness index* of the system under FTAM and FR-FCFS, F_{FTAM} and $F_{FR\text{-}FCFS}$. Calculated by (1), we have $F_{FTAM} = 0.08$ and $F_{FR\text{-}FCFS} = 0.39$, proving that FTAM provides much better system fairness among threads.

Though FTAM provides better fairness than FR-FCFS, the system throughput should never be ignored. Fig.6 shows the comparison of throughput measured in average memory access latency between FTAM and FR-FCFS. From the results we can see that under FTAM though the average memory access latency is increased in the case of FFT and DDT than FC-FRFS, it is reduced in Cholesky and CDT. The total average memory access latency is increased from 22.5ns to 22.85ns. This negligible 0.05ns shows that FTAM has very little negative impact on the total system throughput.

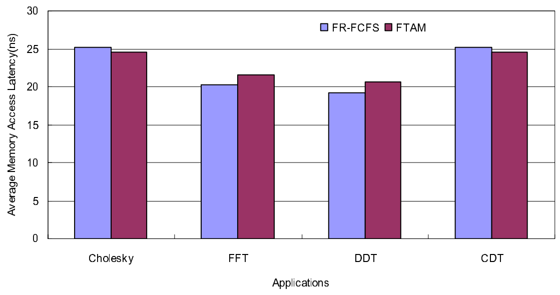


Fig. 6. Comparison of memory accessing time between FTAM and FR-FCFS

6 Conclusion

In this paper, we present a memory scheduling algorithm, FTAM, for CMP systems. FTAM prioritizes memory access requests based on their source-thread, arriving-time and serving-history. With the concerning of these three above factors simultaneously, FTAM algorithm provides a good balance between system throughput and threads fairness. Simulation results show that FTAM algorithm can achieve better fairness than the widely used FR-FCFS algorithm without sacrificing data throughput. With the tuning on parameters, FTAM can be switched to fulfill different requirement with respect to the fairness and throughput.

References

1. Krewell, K.: Best Servers of 2004: Multicore is Norm. Microprocessor Report (January 2005), `http://www.mpronline.com`
2. Olukotun, K., Nayfeh, B.A., et al.: The case for a single-chip multiprocessor. In: Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (October 1996)
3. Micron. 512Mb DDR2 SDRAM Component Data Sheet: MT47H128M4B6-25 (March 2006), `http://download.micron.com/pdf/-datasheets/dram/ddr2/512MbDDR2.pdf`
4. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D.: Memory access scheduling. In: Proc. of the 27th ACM/IEEE International Symposium on Computer Architecture, ISCA (2000)
5. Rixner, S.: Memory controller optimizations for web servers. In: MICRO-37 (2004)
6. Zhu, Z., Zhang, Z.: A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In: Proc. of the 11th International Symposium on High-Performance Computer Architecture (2005)
7. Nesbit, K.J., Aggarwal, N., Laudon, J., Smith, J.E.: Fair Queuing Memory Systems. In: Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, Los Alamitos (2006)
8. Mutlu, O., Moscibroda, T.: Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In: Proc. of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, Los Alamitos (2007)

9. Hongzhong, Z., Jiang, L., Zhao, Z., Zhichun, Z.: Memory Access Scheduling Schemes for Systems with Multi-Core Processors. In: 37th International Conference on Parallel Processing, ICPP 2008 (2008)
10. Mutlu, O., Moscibroda, T.: Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. SIGARCH Comput. Archit. News 36(3), 63–74 (2008) (ISCA 2008)
11. Fang, Z., Sun, X.-H., Chen, Y., Byna, S.: Core-Aware Memory Access Scheduling Schemes. In: Proc. of IEEE International Parallel and Distributed Processing Symposium, IPDPS 2009 (2009)
12. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The M5 simulator: Modeling networked systems. IEEE Micro 26(4), 52–60 (2006)
13. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: The 22nd Annual International Symposium on Computer Architecture, ISCA 1995 (1995)

Efficient Partitioning of Static Buses for Processor Arrays of Small Size

Susumu Matsumae

Saga University, Saga 840-8502, Japan
matsumae@is.saga-u.ac.jp

Abstract. This paper shows an efficient partitioning of static row/column buses for tightly coupled 2D mesh-connected processor arrays (mesh for short) of small size. With additional $O\left(\frac{n}{m}\left(\frac{n}{m}+\log m\right)\right)$ time slowdown, it enables the mesh of size $m \times m$ with static row/column buses to simulate the mesh of larger size $n \times n$ with reconfigurable row/column buses ($m \leq n$). This means that if a problem can be solved in $O(T)$ time by the mesh of size $n \times n$ with reconfigurable bus, then the same problem can be solved in $O\left(T \cdot \frac{n}{m}\left(\frac{n}{m}+\log m\right)\right)$ time on the mesh of smaller size $m \times m$ without reconfigurable function. This time-cost is optimal when the relation $n \geq m \log m$ holds (e.g., $m = n^{1-\epsilon}$ for $\epsilon > 0$).

1 Introduction

The mesh-connected processor array (mesh for short) is one of fundamental parallel computational models. The architecture is suitable for VLSI implementation and allows a high degree of integration. However, the mesh has a crucial drawback that its communication diameter is quite large due to the lack of broadcasting mechanism. To overcome this problem, many researchers have considered adding broadcasting buses to the mesh [1,2,3,4,5,6].

Consider a linear processor array of n processing elements (PEs) where each adjacent PEs can communicate each other via the local communication link. If it has a global bus spanning the entire array, it takes only one step to perform a data broadcast operation, while it would take $O(n)$ steps if it has no global bus. Also, if it has a global bus spanning the entire array, it takes $O\left(n^{1/2}\right)$ steps to perform a fundamental prefix semi-group computations, while it would take $O(n)$ steps again if it has no global bus. Thus, the power of broadcasting capability is inevitable for the mesh architecture to be an efficient computational model.

To avoid write-conflicts on a global bus and to make effective use of it, the global bus may be partitioned into smaller bus segments. It is an interesting problem that decides the effective/optimal length used for such partitioning of a bus. As for the simulation problem discussed in [7], we successfully proved that the problem can be solved optimally in $O\left(n^{1/3}\right)$ steps if we partition every row/column buses into sub-buses of length $n^{2/3}$. However, in general, the effective/optimal bus length depends on the problem to be solved, and hence can not be fixed in advance.

Recently, the dynamically reconfigurable bus systems gain much attention due to its powerful computational power [8]. It can dynamically change its bus configuration during the execution of programs. The reconfigurable function enables the models to make efficient use of broadcast buses, and to solve many important, fundamental problems efficiently, mostly in a constant or polylogarithmic time. Such reconfigurability, however, makes the bus systems complex and causes negative effects on the communication latency of global buses [9].

In this paper, we investigate an efficient partitioning of static row/column buses for the mesh of small size. In [10], we have shown that the mesh of size $n \times n$ with static row/column buses can simulate the mesh of the same size $n \times n$ with reconfigurable row/column buses with additional $O(\log n)$ step slowdown. This means that if a problem can be solved in $O(T)$ time by the mesh of size $n \times n$ with reconfigurable row/column buses, then the same problem can be solved in $O(T \cdot \log n)$ time on the mesh of the same size $n \times n$ without reconfigurable function. Here in this paper, we extend the result to the case where the simulating mesh is of smaller size. It is practical to consider such a case, because we cannot always have the mesh of arbitrary large size in hand. We prove that the mesh of size $m \times m$ with static row/column buses to simulate the mesh of larger size $n \times n$ with reconfigurable row/column buses with additional $O\left(\frac{n}{m}\left(\frac{n}{m} + \log m\right)\right)$ slowdown $(m \leq n)$. This time-cost is optimal when the relation $n \geq m \log m$ holds (e.g., $m = n^{1-\epsilon}$ for $\epsilon > 0$), because the time-cost becomes $O\left(\frac{n^2}{m^2}\right)$ which matches the trivial lower bound derived from the ratio of the number of simulated processors to that of simulating ones.

This paper is organized as follows. Section 2 explains several models of mesh-connected parallel computers with global buses. Section 3 defines the problem formally and introduces the basic approach to solve it. Section 4 describes our algorithm. And finally, Section 5 offers concluding remarks.

2 Models

An $n \times n$ mesh consists of n^2 identical SIMD processors or processing elements (PEs) arranged in a two-dimensional grid with n rows and n columns. The PE located at the grid point (i, j), denoted as PE$[i, j]$, is connected via bi-directional unit-time communication links to those PEs at $(i \pm 1, j)$ and $(i, j \pm 1)$, provided they exist $(0 \leq i, j < n)$. PE$[0, 0]$ is located in the top-left corner of the mesh. Each PE$[i, j]$ is assumed to know its coordinates (i, j).

An $n \times n$ *mesh with separable buses* (MSB) and an $n \times n$ *mesh with partitioned buses* (MPB) are the $n \times n$ meshes enhanced with the addition of broadcasting buses along every row and column. The broadcasting buses of the MSB, called *separable buses*, can be dynamically sectioned through the PE-controlled switches during execution of programs, while those of the MPB are statically partitioned in advance. (Figure 1 and 2).

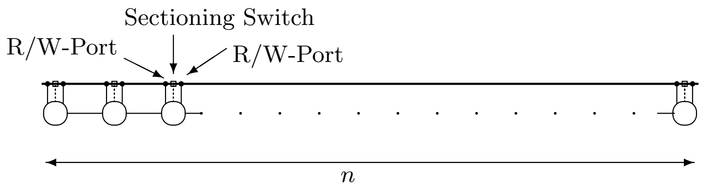


Fig. 1. A separable bus along a row of the $n \times n$ MSB. Each PE has access to the bus via the two ports beside the sectioning switch.

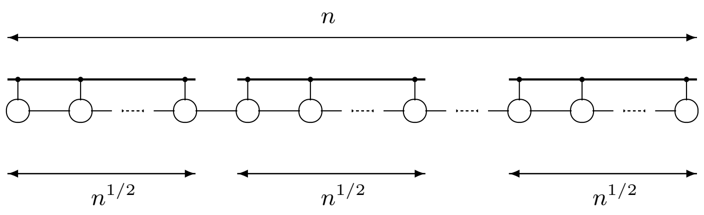


Fig. 2. A partitioned bus along a row of the $n \times n$ MPB. Here, the bus is equally partitioned by length $\sqrt{n}$.

A single time step of the MSB and the MPB is composed of the following three substeps:

Local communication substep: Every PE communicates with its adjacent PEs via local links.

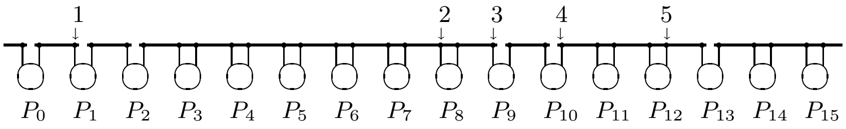
Broadcast substep: Every PE changes its switch configurations by local decision (this operation is only for the MSB). Then, along each broadcasting bus segment, several of the PEs connected to the bus send data to the bus, and several of the PEs on the bus receive the data transmitted on the bus.

Compute substep: Every PE executes some local computation.

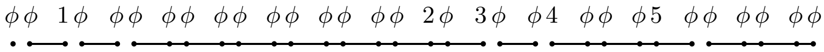
The bus accessing capability is similar to that of Common-CRCW PRAM model. If there is a write-conflict on a bus, the PEs on the bus receive a special value $\perp$ (i.e., PEs can detect whether there is a write-conflict on a bus or not). If there is no data transmitted on a bus, the PEs on the bus receive a special value ϕ (i.e., PEs can know whether there is data transmitted on a bus or not).

3 Problems

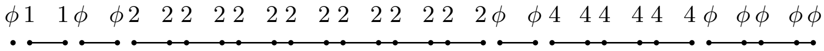
In this paper, we consider *(scaling) simulation* of the $n \times n$ MSB by the $m \times m$ MPB ($m < n$). To simplify the exposition, we assume that $n \bmod m = 0$. We define the processor mapping as follows: each PE$[i, j]$ of the $m \times m$ MPB simulates PE$[x, y]$ of the $n \times n$ MSB ($i\frac{n}{m} \leq x < (i+1)\frac{n}{m}$, $j\frac{n}{m} \leq y < (j+1)\frac{n}{m}$). We assume that the computing power of PEs, the bandwidth of local links, and that of broadcasting buses are equivalent in both simulated and simulating meshes. Throughout the paper, we assume that the simulation is done by step-by-step, that is, we consider how to simulate any single step of the MSB by using the MPB.


1
2
3
4
5
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}

(a) Broadcasts to be simulated. The processors P_1, P_9, P_{10}, and P_{12} respectively send data


$\phi\phi$ 1ϕ $\phi\phi$ $\phi\phi$ $\phi\phi$ $\phi\phi$ $\phi\phi$ $\phi\phi$ 2ϕ 3ϕ $\phi 4$ $\phi\phi$ $\phi 5$ $\phi\phi$ $\phi\phi$ $\phi\phi$

(b) The corresponding port-connectivity graph G with initial labels


$\phi 1$ 1ϕ $\phi 2$ $2\,2$ $2\,2$ $2\,2$ $2\,2$ $2\,2$ $2\,2$ 2ϕ $\phi 4$ $4\,4$ $4\,4$ 4ϕ $\phi\phi$ $\phi\phi$

(c) After connected-component labeling, vertices in each component C is labeled by the smallest initial label of all the vertices in C, with regarding ϕ as the greatest element

Fig. 3. Broadcasts on a separable bus along a row of the $n \times n$ MSB are simulated by connected-component labeling of the port-connectivity graph. Here, $n = 16$.

In what follows, we focus on how to mimic the broadcast substep of the MSB using the MPB, because the local communication and the compute substeps of the MSB can be easily simulated in $O\left(\frac{n^2}{m^2}\right)$ steps by the MPB.

Here, the problem of simulating the broadcast substep of the MSB is explained by connected-component labeling (CC-labeling) for a *port-connectivity graph* (pc-graph). See Figure 3 for an example. Vertices of the pc-graph correspond to read/write-ports of PEs, and edges stand for the port-to-port connections. Each vertex is initially labeled by the value which is sent through the corresponding port by the PE at the broadcast substep. If there is no data sent through the port, the vertex is labeled by ϕ. The CC-labeling is done in such a way that vertices in each component C is labeled by the smallest initial label of all the vertices in C, with regarding ϕ as the greatest value. These labels are called *component labels*. Obviously, the simulation of the broadcast substep of the MSB can be achieved in $O(T)$ steps on the MPB if the CC-labeling of the corresponding port-connectivity graph can be executed in $O(T)$ steps by the MPB.

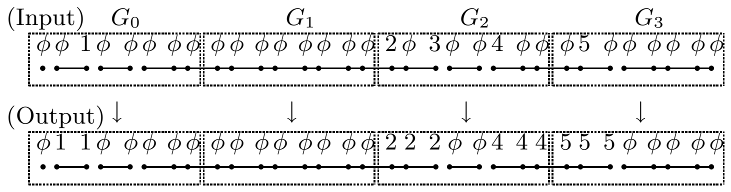
In this paper, we solve the CC-labeling problem by divide-and-conquer strategy composed of the following three phases:

Phase 1: { *local labeling*}

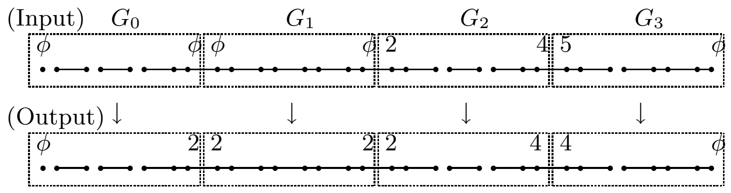
Divide the pc-graph into subgraphs, and label vertices locally within each of the subgraphs. The labels are called *local component labels*. In each subgraph, check whether the two vertices located at the boundary of the subgraph is connected to each other or not.

Phase 2: { *global labeling of boundary vertices*}
Label those vertices located at the boundary of each subgraph with component labels.
Phase 3: { *local labeling for adjustment*}
Update vertex labels with component labels within each of subgraphs for the consistency with Phase 2

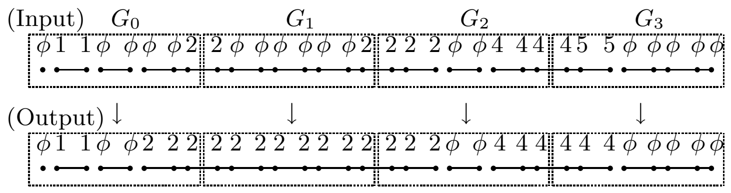
See Figure 4 for an example.


(Input)
G_0
G_1
G_2
G_3
$\phi\phi$ 1ϕ $\phi\phi$ $\phi\phi$ | $\phi\phi$ $\phi\phi$ $\phi\phi$ $\phi\phi$ | 2ϕ 3ϕ ϕ4 $\phi\phi$ | ϕ5 $\phi\phi$ $\phi\phi$ $\phi\phi$
(Output)
ϕ1 1ϕ $\phi\phi$ $\phi\phi$ | $\phi\phi$ $\phi\phi$ $\phi\phi$ $\phi\phi$ | 2 2 2ϕ ϕ4 4 4 | 5 5 5ϕ $\phi\phi$ $\phi\phi$

Phase 1: Connected-component labeling is locally executed within each subgraph G_i. Vertices are labeled by local component labels


(Input)
G_0
G_1
G_2
G_3
ϕ ϕ | ϕ ϕ | 2 4 | 5 ϕ
(Output)
ϕ 2 | 2 2 | 2 4 | 4 ϕ

Phase 2: Every boundary vertices is labeled by component label


(Input)
G_0
G_1
G_2
G_3
ϕ1 1ϕ $\phi\phi$ ϕ2 | 2 ϕ $\phi\phi$ $\phi\phi$ ϕ2 | 2 2 2ϕ ϕ4 4 4 | 4 5 5ϕ $\phi\phi$ $\phi\phi$
(Output)
ϕ1 1ϕ ϕ2 2 2 | 2 2 2 2 2 2 2 2 | 2 2 2ϕ ϕ4 4 4 | 4 4 4ϕ $\phi\phi$ $\phi\phi$

Phase 3: Connected-component labeling is executed within each subgraph G_i for the consistency with Phase 2. Every vertex is labeled by component label

Fig. 4. An example for the connected-component labeling algorithm. The algorithm consists of three phases.

4 Scaling-Simulation of the MSB by the MPB

In this section, we prove that any one step of the $n \times n$ MSB can be simulated in $O\left(\frac{n}{m}\left(\frac{n}{m} + \log m\right)\right)$ steps by the $m \times m$ MPB $(m \leq n)$.

To begin with, we introduce the following theorem:

Theorem 1. *[10] Any step of the $n \times n$ MSB can be simulated in $O(\log n)$ steps by the $n \times n$ MPB.* ■

Each row separable bus of the $n \times n$ MSB is simulated by the $m \times m$ MPB as follows (the case for a column separable bus is similar). To simulate the broadcasts taken along a row separable bus of the simulated $n \times n$ MSB, the CC-labeling problem of the corresponding pc-graph is solved by the simulating $m \times m$ MPB. Here, we divide the pc-graph into m disjoint subgraphs of width $\frac{n}{m}$[1] so that each subgraph is locally stored in a single PE of the MPB. Then, Phase 1 and 3 of CC-labeling can be executed in $O\left(\frac{n}{m}\right)$ steps locally in each PE by a sequential algorithm that scans the vertex information from left to right and then from right to left. Phase 2 is essentially the same as the problem of labeling a pc-graph of width m on the $m \times m$ MPB, and hence it can be done in $O(\log m)$ steps (Theorem 1). Since each row of the MPB has to simulate the assigned $\frac{n}{m}$ row separable buses of the MSB, as a whole the time cost becomes $O\left(\frac{n}{m}(\frac{n}{m} + \log m)\right)$ steps. Hence, we have the following lemma:

Lemma 1. *Any broadcasts taken along rows of the $n \times n$ MSB can be simulated in $O\left(\frac{n}{m}\left(\frac{n}{m} + \log m\right)\right)$ steps by the $m \times m$ MPB ($m \leq n$).* ■

Local communication and compute substeps can be simulated in $O\left(\frac{n^2}{m^2}\right)$ steps locally in each PE of the $m \times m$ MPB. Broadcast substep is simulated by first simulating broadcasts along rows and then simulating those along columns. As a whole, the simulation completes in $O\left(\frac{n^2}{m^2} + \frac{n}{m}\left(\frac{n}{m} + \log m\right)\right)$ steps (Lemma 1). Now, we can obtain the main theorem of this paper:

Theorem 2. *Any step of the $n \times n$ MSB can be simulated in $O\left(\frac{n}{m}\left(\frac{n}{m} + \log m\right)\right)$ steps by the $m \times m$ MPB ($m \leq n$).* ■

5 Concluding Remarks

We have presented an algorithm that simulates the $n \times n$ MSB on the $m \times m$ MPB in $O\left(\frac{n}{m}\left(\frac{n}{m} + \log m\right)\right)$ steps ($m \leq n$). If the relation $n \geq m \log m$ holds (e.g., $m = n^{1-\epsilon}$ for $\epsilon > 0$), the time-cost is optimal because it matches the lower bound $\Omega\left(\frac{n^2}{m^2}\right)$ derived from the ratio of the number of simulated PEs and that of simulating ones.

From a practical viewpoint, our scaling simulation algorithm can simulate the MSB model in which the concurrent write is resolved by the MIN rule [11] where the minimum among the sent values is received when a write-conflict occurs. This is because our algorithm simulate the broadcast operation of the MSB by connected-component labelling of the corresponding *port-connectivity*

[1] We say that a subgraph of pc-graph is of width w if it has $2w$ vertices corresponding to the R/W-ports of w consecutive PEs.

graph [11]. This fact may make up the slowdown required for the simulation because the MSB with MIN-bus model is very powerful, for example, it can find the minimum among the values distributed over the mesh in a constant time.

Acknowledgements

This work was partly supported by the MEXT Grant-in-Aid for Young Scientists (B) (20700014).

References

1. Prasanna-Kumar, V.K., Raghavendra, C.S.: Array processor with multiple broadcasting. J. of Parallel Distributed Computing 4, 173–190 (1987)
2. Maeba, T., Sugaya, M., Tatsumi, S., Abe, K.: Semigroup computations on a processor array with partitioned buses. IEICE Trans. A J80-A(2), 410–413 (1997)
3. Miller, R., Prasanna-Kumar, V.K., Reisis, D., Stout, Q.F.: Meshes with reconfigurable buses. In: Proc. of the fifth MIT Conference on Advanced Research in VLSI, Boston, pp. 163–178 (1988)
4. Wang, B., Chen, G.: Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. IEEE Trans. Parallel and Distributed Systems 1(4), 500–507 (1990)
5. Maeba, T., Tatsumi, S., Sugaya, M.: Algorithms for finding maximum and selecting median on a processor array with separable global buses. IEICE Trans. A J72-A(6), 950–958 (1989)
6. Serrano, M.J., Parhami, B.: Optimal architectures and algorithms for mesh-connected parallel computers with separable row/column buses. IEEE Trans. Parallel and Distributed Systems 4(10), 1073–1080 (1993)
7. Matsumae, S., Tokura, N.: Simulating a mesh with separable buses. Transactions of Information Processing Society of Japan 40(10), 3706–3714 (1999)
8. Vaidyanathan, R., Trahan, J.L.: Dynamic Reconfiguration. Kluwer Academic/Plenum Publishers (2004)
9. Maeba, T., Sugaya, M., Tatsumi, S., Abe, K.: An influence of propagation delays on the computing performance in a processor array with separable buses. IEICE Trans. A J78-A(4), 523–526 (1995)
10. Matsumae, S.: Polylogarithmic time simulation of reconfigurable row/column buses by static buses. In: Proc. of IEEE International Parallel and Distributed Processing Symposium, IPDPS 2010 (2010)
11. Matsumae, S., Tokura, N.: Simulation algorithms among enhanced mesh models. IEICE Transactions on Information and Systems E82–D(10), 1324–1337 (1999)

Formal Proof for a General Architecture of Hybrid Prefix/Carry-Select Adders

Feng Liu[1], Qingping Tan[1], Xiaoyu Song[2], and Gang Chen[3]

[1] National Lab of Parallel Distributed Processing, Hunan, Chian
[2] ECE department, Portland State University, Portland, OR, USA
[3] Lingcore Lab, Portland, OR, USA

Abstract. In this paper, we present a general architecture of hybrid prefix/carry-select adder. Based on this architecture, we formalize the hybrid adder's algorithm using the first-order recursive equations and develop a proof framework to prove its correctness. Since several previous adders in the literature are special cases of this general architecture, our methodology can be used to prove the correctness of different hybrid prefix/carry-select adders. The formal proof for a special hybrid prefix/carry-select adder shows the effectiveness of the algebraic structures built in this paper.

1 Introduction

Addition is a crucial arithmetic function for most digital systems. In the area of computer arithmetic, a great number of addition algorithms have been proposed. With the shrinking feature size and the growing density, the costs of VLSI circuit manufacturing are soaring, so do the costs of circuit errors. Therefore, the logic correctness of binary addition algorithm becomes extremely important.

In the past few years, some adders based on hybrid carry-lookahead and carry-select scheme have been reported [1,2,3,4]. In the hybrid scheme, two conditional addition results are pre-computed for each specific block and each global carry is used to select one of the two results [3]. Therefore, the delay of the local carry generation is eliminated in the hybrid adder architecture.

To improve the speed, based on the algorithm of hybrid carry-lookahead/ carry-select adder, some new adders with the hybrid parallel prefix adder/ carry-select scheme are designed, such as [5]. For simplicity, the hybrid carry-lookahead/carry-select adders and its variations are called hybrid prefix/carry-select adders. Unfortunately, their algorithms are always described based on a few elegant properties, which are often accepted by intuition. Formal analysis on their correctness is limited. The details of their proofs are incomplete or hard to find. In this paper, we propose a general architecture for the design of hybrid prefix/carry-select adder. Based on the traditional notions such as propagated carry and generated carry, a detailed proof of the general architecture's correctness is described. Using this general architecture, we formalize a special hybrid prefix/carry-select adder which is implemented as a part of the IBM POWER6 microprocessor's floating-point unit. The formal proof of this special adder's correctness shows the effectiveness of the proposed algebraic structures. In our approach, the representation of the algorithm is given in the form of first-order recursive equations, which are widely adopted in computer arithmetic community. It is our belief that these proofs would be helpful to get a better understanding about the nature of the adder's algorithms. Another aim is to make it convenient for further formal investigations.

The rest of this paper is organized as follows. Section 2 is a review of some related works. In Section 3, we present a general architecture for hybrid prefix/ carry-select adders and develop a framework for its formal proof. In Section 4, we formalize a special hybrid prefix/carry-select adder and describe its formal proof. Section 5 concludes the paper.

2 Related Work

Some hybrid prefix/carry-select adders have been designed in the past few years. T. Lynch et al [1] proposed a 64-bit spanning tree carry-lookahead adder (STCLA) with a hybrid carry lookahead/carry-select structure. A follow-up work of the STCLA which used Manchester chain of various lengths was reported by [2]. By replacing the ripple-carry blocks with parallel prefix blocks, Tyagi proposed a hybrid parallel prefix/carry-select adder scheme and derived a lemma to reduce the area with intra-block rippling [6]. C. Arijhan et al [7] proposed a new hybrid scheme using irregular parallel-prefix and carry-select structures. A general form of a new lemma and its proof were presented. Yuke Wang et al [4] also proposed a new hybrid carry- lookahead/carry-select scheme based on Ling's carries and complemented the group carry terms to gain speed. All of the above works used some traditional formal notations to describe the algorithms. But there were no complete formalization and correctness proofs.

On the other hand, concerning the correctness proof, J. O'Donnell and G. Rnger [8] employed high-order functional combinators to represent carry lookahead adder and the correctness is established by algebraic transformations. D. Kapur and M. Subramaniam described adder's algorithms using the language of RRL and verified the correctness by term rewriting in a rewrite-rule based theorem prover [9]. M. Sheeran recently made a profound formal investigation into parallel prefix computation using the Haskell language [10]. R. Hinze employed the Haskell as the meta language and introduced an algebra of scans which can be used in formal analysis of parallel prefix circuits [11]. These works use many jargons of formal verification specialists and mainly focus on the correctness proof of prefix adders. Jungang Han and G. Stone formulated the Conditional Sum addition algorithm and used Gordon's HOL system to do the formal verification of the CMOS implementation of the Conditional Sum adder [12]. K. M. Elleithy and M. A. Aref presented a rule-based framework for formal hardware verification [13]. Basic circuits such as carry-select adder and carry lookahead carry are verified. They didn't concern the correctness proof of hybrid prefix/carry-select adder. In this paper, we adopt the first-order recursive equations to formalize the algorithm of hybrid prefix/carry select adders and to prove their correctness.

3 Hybrid Prefix/Carry-Select Adder

3.1 Preliminaries

We use the following notations and definitions in this paper. Let the symbols 0 and 1 denote Boolean False and True, or digital number Zero and One; the symbol $\wedge$ denotes the Boolean AND; $\vee$ denotes the Boolean OR; $\oplus$ denotes the Boolean Exclusive OR. A binary number of length $n (n \geq 0)$ is an ordered sequence of binary bits where each bit can assume one of the values 0 or 1. Let the operands in addition are two N-bit binary numbers $x = (x_{n-1}x_{n-2}...x_0)$ and $y = (y_{n-1}y_{n-2}...y_0)$. The sum is $s = (s_{n-1}s_{n-2}...s_0)$ and $c = \{c_n, c_{n-1}, ..., c_0\}$ is the set of carries where c_0 is the initial incoming carry, c_i denotes the carry from the bit position $i-1$. x_i, y_i, s_i denote the binary bits of x, y, s at position i.

The correctness of an integer adder can be expressed by:

$$\sum_{i=0}^{n-1} x_i 2^i + \sum_{i=0}^{n-1} y_i 2^i + c_0 = \sum_{i=0}^{n-1} s_i 2^i + c_n 2^n \tag{1}$$

The main idea behind prefix addition is to calculate all carries in parallel. To describe its algorithm, some standard notions such as propagated carry $P_i = x_i \oplus y_i$ and generated carry $G_i = x_i \wedge y_i$ are introduced as in Koren [14]. For the algorithm of parallel prefix adders, we also need to introduce the notations of group-propagated carry $P_{i:j}$ and group-generated carry $G_{i:j}$. Their definition can also be found in [14].

$P_{i:j}$ and $G_{i:j}$ can be used to develop a general representation of the prefix's algorithms as follows:

$$c_i = G_{i-1:j} \vee (P_{i-1:j} \wedge c_j), s_i = P_i \oplus c_i \tag{2}$$

where $0 \leq j \leq i-1 \leq n-1$.

G. Chen et al [15] gave an elegant correctness proof of equation (2). To make the calculation of $P_{i:j}$ and $G_{i:j}$ simple, the so-called fundamental carry operator $\circ$ is introduced by [16]. G. Chen et al [15] also proved that the fundamental carry operator enjoys two properties: associativity and idempotency. These properties allows us to compute the $P_{i:j}$, $G_{i:j}$ as follows:

$$(G_{i:j}, P_{i:j}) = (G_{i:m}, P_{i:m}) \circ (G_{v:j}, P_{v:j}) = (G_{i:m} \vee \{P_{i:m} \wedge G_{v:j}\}, P_{i:m} \wedge P_{v:j}) \tag{3}$$

where $i \geq v \geq m-1 \geq j \geq 0$.

Assume that the N-bit operands of an adder are divided into non-overlapping groups of possible different lengths. For the gourp of bit positions $i, i-1, ..., j$ (with $i \geq j$), a carry select adder takes the operand bits of this group as inputs and generates two sets of outputs. One set assumes that the incoming carry into the group is 0 while the other assumes that it is 1. In carry select adder, the corresponding sum $s_i, s_{i-1}, ..., s_j$ and the outgoing carry c_{i+1} are selected by incoming carry into this group c_j as follows:

$$\begin{aligned} s_m &= (s_m^0 \wedge \overline{c_j}) \vee (s_m^1 \wedge c_j) \quad (m = j, j+1, ..., i) \\ c_{i+1} &= (c_{i+1}^0 \wedge \overline{c_j}) \vee (c_{i+1}^1 \wedge c_j) \end{aligned} \tag{4}$$

s_m^0 (or s_m^1) is the sum bit at the m bit position and c_{i+1}^0 (or c_{i+1}^1) is the group outgoing carry under the condition that the incoming carry into the group is 0 (or 1). $\overline{c_j}$ is the one's complement code of c_j.

3.2 Formalization of General Architecture

We propose a general architecture for the hybrid prefix/carry-select adder in Fig.1. The prefix adder unit takes x, y and the incoming carry c_0 as the inputs and generates global carries at different positions. Usually, tree architectures are used as prefix adder unit for fastest computation of the global carries. The prefix adder unit does not generate all carries, instead only part of the global carry signals. Let $x_{i:j}$, $y_{i:j}$, $s_{i:j}$ denote the bit pieces $(x_i x_{i-1} ... x_j)$, $(y_i y_{i-1} ... y_j)$ and $(s_i s_{i-1} ... s_j)$ respectively, where $i \geq j$. The carry-select adders compute the sum bits to be selected. Each global carry selects one of the two results of the corresponding carry-select adder. The general block diagram of each carry-select adder in Fig.1 is shown in Fig.2.

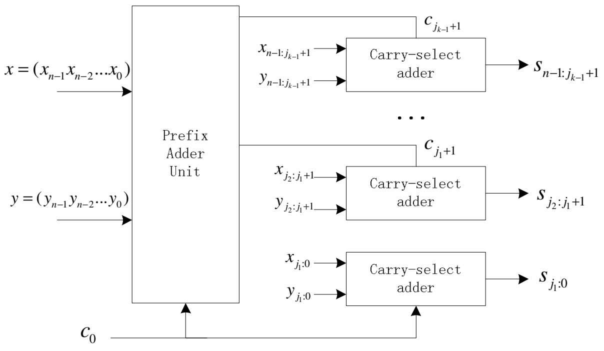


Fig. 1. General architecture of hybrid Prefix/Carry-Select adder

In Fig.2, we use $csa_{i:j}$ to denote the carry select adder. It constitutes of two prefix adders: the *Conditional_0 adder* and the *Conditional_1 adder*. The *Conditional_0 adder* takes $x_{i:j}$, $y_{i:j}$, the incoming carry 0 as the inputs and generates the sum $s_{i:j}^0$, the output carry c_{i+1}^0; the *Conditional_1 adder* takes $x_{i:j}$, $y_{i:j}$, the incoming carry 1 as the inputs and generates the sum $s_{i:j}^1$, the output carry c_{i+1}^1. In hybrid prefix/carry-select adder, the *Conditional_0 adder* and *Conditional_1 adder* can be implemented as any "correct prefix adder". So, the following correctness statements are hold for them:

$$\begin{aligned}\sum_{m=j}^{i} x_m 2^m + \sum_{m=j}^{i} y_m 2^m + 0 &= \sum_{m=j}^{i} s_m^0 2^m + c_{i+1}^0 2^{i+1} \\ \sum_{m=j}^{i} x_m 2^m + \sum_{m=j}^{i} y_m 2^m + 2^j &= \sum_{m=j}^{i} s_m^1 2^m + c_{i+1}^1 2^{i+1}\end{aligned} \tag{5}$$

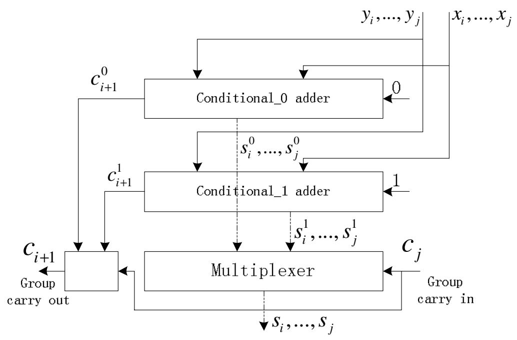


Fig. 2. Carry-Select adder

The above formulas can be used to prove the correctness of carry-select adder's algorithm. Since the correctness of carry-select adder has been discussed by other researchers. In this paper, we just list this fact as follows:

Lemma 1 (Correctness of Carry Select Adder [12]). *For the Carry Select Adder (CSA) whose architecture is shown by Fig.2, let* $x_{i:j} = (x_i ... x_j)$*,* $y_{i:j} = (y_i ... y_j)$*,* c_j *be the input arguments and the outputs be defined by equation (4), then we have:*

$$\sum_{m=j}^{i} x_m 2^m + \sum_{m=j}^{i} y_m 2^m + c_j 2^j = \sum_{m=j}^{i} s_m 2^m + c_{i+1} 2^{i+1} \tag{6}$$

For the addition of two N-bit binary numbers, the operands $x = (x_{n-1} x_{n-2} ... x_0)$, $y = (y_{n-1} y_{n-2} ... y_0)$ are possibly divided into some non-overlapping groups of different lengths. The carry select adder $csa_{i:j}$ can only deal with the sub-group of bit positions $i, i-1, ..., j$. Assume that the incoming carry is c_j, by equation (4), we know the output carry is $c_{i+1} = (c^0_{i+1} \wedge \overline{c_j}) \vee (c^1_{i+1} \wedge c_j)$. By equation (2), we know for the same group, the prefix adder calculates the output carry as follows: $c_{i+1} = G_{i:j} \vee (P_{i:j} \wedge c_j)$. In the following, we will study the relationship between these two expressions.

Lemma 2. *For the group of bit positions* $i, i-1, ..., j$*, let* $c'_{i+1} = (c^0_{i+1} \wedge \overline{c_j}) \vee (c^1_{i+1} \wedge c_j)$ *be the output carry of carry select adder* $csa_{i:j}$*, let* $c_{i+1} = G_{i:j} \vee (P_{i:j} \wedge c_j)$ *be the carry calculated by the general prefix algorithm, then:* $c'_{i+1} = c_{i+1}$.

Proof. If $c_j = 0$, we have: $c_{i+1} = G_{i:j} \vee (P_{i:j} \wedge c_j) = G_{i:j}$. By equation (4), we have: $c'_{i+1} = (c^0_{i+1} \wedge \overline{c_j}) \vee (c^1_{i+1} \wedge c_j) = c^0_{i+1}$. Since *Conditional_0 adder* is implemented by prefix adder, and the incoming carry is 0, so, we have: $c^0_{i+1} = G_{i:j} \vee (P_{i:j} \wedge 0) = G_{i:j}$. Therefore, $c'_{i+1} = c_{i+1}$ for the case $c_j = 0$. The case $c_j = 1$ can be proved similarly. So, $c'_{i+1} = c_{i+1}$.

Lemma 2 is the basis of using prefix adder to compute the global carries in hybrid prefix/carry-select scheme. In the following, we will explain how to formalize the hybrid adder's algorithm.

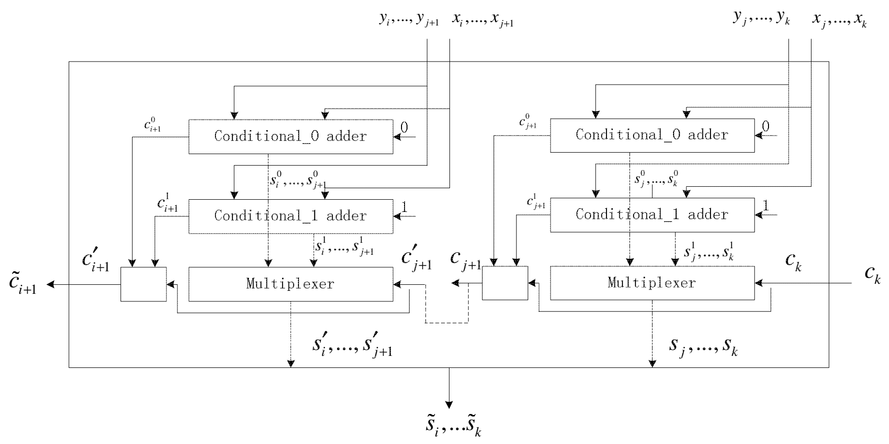


Fig. 3. Connection of adjacent carry select adders

For two adjacent bit pieces $s_{i:j+1} = (s_i s_{i-1} \ldots s_{j+1})$, $s_{j:k} = (s_j s_{j-1} \ldots s_k)$, the bit pieces concatenation operator "$\diamond$" can be defined as follows: $s_{i:j+1} \diamond s_{j:k} = (s_i s_{i-1} \ldots s_k) = s_{i:k}$. We use $adder_{j:k}$ to denote an adder whose addends are $x_{j:k} = (x_j x_{j-1} \ldots x_k)$ and $y_{j:k} = (y_j y_{j-1} \ldots y_k)$. With the help of bit pieces concatenation operator, we define adder concatenation operator as follows:

Definition 1 (Adder concatenation operator). *Let $adder_{i:j+1}$ and $adder_{j:k}$ be two adders. Let c_k, $s_{j:k} = (s_j s_{j-1} \ldots s_k)$ be the incoming carry and sum of $adder_{j:k}$; $c'_{j+1} = G_{j:k} \vee (P_{j:k} \wedge c_k)$ and $s'_{i:j+1} = (s'_i s'_{i-1} \ldots s'_{j+1})$ be the incoming carry and sum of $adder_{i:j+1}$. Then the adder concatenation operator $/\!\!/$ is defined as follows: the new adder $\widetilde{adder}_{i:k} = adder_{i:j+1} /\!\!/ adder_{j:k}$ takes $x_{i:j+1} \diamond x_{j:k}$, $y_{i:j+1} \diamond y_{j:k}$ and the incoming carry c_k as inputs and generates $(\widetilde{s}_i \widetilde{s}_{i-1} \ldots \widetilde{s}_k) = s'_{i:j+1} \diamond s_{j:k}$ as the sum bits, $\widetilde{c}_{i+1} = c'_{i+1}$ as the output carry.*

The adder concatenation operator can be used to connect two adjacent carry select adders. Fig.3 shows the block diagram of $csa_{i:j+1} /\!\!/ csa_{j:k}$. If we use adder concatenation operator to connect more than one carry select adders, we should ensure that the resulting adder is correct.

Lemma 3. *Assume there are n adjacent carry select adders $csa_{i:i_1}$, $csa_{(i_1-1):i_2}$, ..., $csa_{(i_{n-1}-1):k}$, where $i \geq i_1 > \ldots > i_{n-1} > k$. Let $\widetilde{csa}_{i:k} = csa_{i:i_1} /\!\!/ \ldots /\!\!/ csa_{(i_{n-1}-1):k}$ be the concatenation of them. Then, $\widetilde{csa}_{i:k}$ is correct:*

$$\sum_{m=k}^{i} x_m 2^m + \sum_{m=k}^{i} y_m 2^m + c_k 2^k = \sum_{m=k}^{i} \widetilde{s}_m 2^m + \widetilde{c}_{i+1} 2^{i+1}$$

where $x_{i:k}$, $y_{i:k}$ are the operands of $csa_{i:k}$, c_k is the incoming carry, $\widetilde{s}_{i:k}$ is the sum, $\widetilde{c}_{i+1}$ is the output carry.

Proof. Induction on the numbers of adder concatenation operators: $z = n - 1$.

Base case. (z=0). There is only one carry select adder $csa_{i:i_1}$. So, $\widetilde{csa}_{i:i_1} = csa_{i:i_1}$. By lemma 1, $\widetilde{csa}_{i:i_1}$ is correct.

Induction case. Assume that the induction hypothesis is true for $z = j$ ($\widetilde{csa}_{i:i_j}$ is correct):

$$\sum_{m=i_j}^{i} x_m 2^m + \sum_{m=i_j}^{i} y_m 2^m + c_{i_j} 2^{i_j} = \sum_{m=i_j}^{i} s^{'}_m 2^m + c^{'}_{i+1} 2^{i+1}$$

where $s^{'}_{i:i_j}$ is the sum of $\widetilde{csa}_{i:i_j}$; $c^{'}_{i+1}$ is the output carry of $\widetilde{csa}_{i:i_j}$.

We need to show that the assertion is valid for $z = j+1$: $\widetilde{csa}_{i:i_{j+1}} = \widetilde{csa}_{i:i_j} \mathbin{/\!\!/} csa_{(i_j-1):i_{j+1}}$

By lemma 1, we know carry select adder $csa_{(i_j-1):i_{j+1}}$ is correct. By lemma 2 and Definition 1, for $csa_{(i_j-1):i_{j+1}}$, we have:

$$\begin{aligned}\sum_{m=i_{j+1}}^{i_j-1} x_m 2^m + \sum_{m=i_{j+1}}^{i_j-1} y_m 2^m + c_{i_{j+1}} 2^{i_{j+1}} &= \sum_{m=i_{j+1}}^{i_j-1} s_m 2^m + c_{i_j} 2^{i_j} \\ &= \sum_{m=i_{j+1}}^{i_j-1} s_m 2^m + [G_{(i_j-1):i_{j+1}} \vee (P_{(i_j-1):i_{j+1}} \wedge c_{i_{j+1}})] 2^{i_j}\end{aligned} \quad (7)$$

For $\widetilde{csa}_{i:i_{j+1}} = \widetilde{csa}_{i:i_j} \mathbin{/\!\!/} csa_{(i_j-1):i_{j+1}}$, by Definition 1, we know that the incoming carry of $\widetilde{csa}_{i:i_j}$ is set to be $G_{(i_j-1):i_{j+1}} \vee (P_{(i_j-1):i_{j+1}} \wedge c_{i_{j+1}})$.

Due to the induction hypothesis, we have:

$$\begin{aligned}&\sum_{m=i_j}^{i} x_m 2^m + \sum_{m=i_j}^{i} y_m 2^m + c_{i_j} 2^{i_j} \\ &= \sum_{m=i_j}^{i} x_m 2^m + \sum_{m=i_j}^{i} y_m 2^m + [G_{(i_j-1):i_{j+1}} \vee (P_{(i_j-1):i_{j+1}} \wedge c_{i_{j+1}})] 2^{i_j} \\ &= \sum_{m=i_j}^{i} s^{'}_m 2^m + c^{'}_{i+1} 2^{i+1}\end{aligned} \quad (8)$$

Let $\widetilde{s}_{i:i_{j+1}}$ be the sum of $\widetilde{csa}_{i:i_{j+1}}$, $\widetilde{c}_{i+1}$ be the output carry of $\widetilde{csa}_{i:i_{j+1}}$. By Definition 1, we know that $\widetilde{s}_{i:i_{j+1}} = s^{'}_{i:i_j} \diamond s_{(i_j-1):i_{j+1}}$, $\widetilde{c}_{i+1} = c^{'}_{i+1}$, So, plus the left sides and right sides of equation (7) and equation (8) respectively, we get:

$$sum^{i}_{m=i_{j+1}} x_m 2^m + \sum_{m=i_{j+1}}^{i} y_m 2^m + c_{i_{j+1}} 2^{i_{j+1}} = \sum_{m=i_{j+1}}^{i} \widetilde{s}_m 2^m + \widetilde{c}_{i+1} 2^{i+1}$$

The induction case is proved. So, $\widetilde{csa}_{i:k} = csa_{i:i_1} \mathbin{/\!\!/} csa_{(i_1-1):i_2} \cdots \mathbin{/\!\!/} csa_{(i_{n-1}-1):k}$ is correct.

The adder concatenation operator enjoys associativity when it is used to connect carry select adders.

Lemma 4. *Let $csa_{i:j+1}$, $csa_{j:k+1}$ and $csa_{k:p}$ be carry select adders, where $i \geq j+1 > j \geq k+1 > k \geq p$, then:*

$$(csa_{i:j+1} \ /\!\!/ csa_{j:k+1}) \ /\!\!/ csa_{k:p} = csa_{i:j+1} \ /\!\!/ (csa_{j:k+1} \ /\!\!/ csa_{k:p})$$

Proof. By Definition 1, it is known that the central of carry select adder concatenation operator $/\!\!/$ is to set the proper incoming carry for each carry select adder. Therefore, we prove its associativity by looking insight into the incoming carry of each carry select adder. Let $\widetilde{csa}_{i:p} = (csa_{i:j+1} \ /\!\!/ csa_{j:k+1}) \ /\!\!/ csa_{k:p}$ and $\widetilde{csa}_{i:k+1} = csa_{i:j+1} \ /\!\!/ csa_{j:k+1}$. For $\widetilde{csa}_{i:k+1}$, the incoming carry of $csa_{j:k+1}$ is c_{k+1} and the incoming carry of $csa_{i:j+1}$ is $c_{j+1} = G_{j:k+1} \vee (P_{j:k+1} \wedge c_{k+1})$. Since $\widetilde{csa}_{i:p} = \widetilde{csa}_{i:k+1} \ /\!\!/ csa_{k:p}$, therefore, for $\widetilde{csa}_{i:p}$, the incoming carry of $csa_{k:p}$ is c_p, the incoming carry of $csa_{j:k+1}$ is $c_{k+1} = G_{k:p} \vee (P_{k:p} \wedge c_p)$, the incoming carry of $csa_{i:j+1}$ is as follows:

$$c_{j+1} = G_{j:k+1} \vee (P_{j:k+1} \wedge c_{k+1}) = G_{j:k+1} \vee \{P_{j:k+1} \wedge [G_{k:p} \vee (P_{k:p} \wedge c_p)]\}$$

Let $\widetilde{csa}^{'}_{i:p} = csa_{i:j+1} \ /\!\!/ (csa_{j:k+1} \ /\!\!/ csa_{k:p})$ and $\widetilde{csa}^{'}_{j:p} = csa_{j:k+1} \ /\!\!/ csa_{k:p}$. For $\widetilde{csa}^{'}_{j:p}$, the incoming carry of $csa_{k:p}$ is c_p and the incoming carry of $csa_{j:k+1}$ is $c^{'}_{k+1} = G_{k:p} \vee (P_{k:p} \wedge c_p)$. Since $\widetilde{csa}^{'}_{i:p} = csa_{i:k+1} \ /\!\!/ \widetilde{csa}^{'}_{k:p}$, therefore, for $\widetilde{csa}^{'}_{i:p}$, the incoming carry of $csa_{k:p}$ is c_p, the incoming carry of $csa_{j:k+1}$ is $c^{'}_{k+1} = G_{k:p} \vee (P_{k:p} \wedge c_p)$, the incoming carry of $csa_{i:j+1}$ is $c^{'}_{j+1} = G_{j:p} \vee (P_{j:p} \wedge c_p)$. Therefore, for $\widetilde{csa}_{i:p}$ and $\widetilde{csa}^{'}_{i:p}$, the incoming carry are both c_p, and $c_{k+1} = c^{'}_{k+1} = G_{k:p} \vee (P_{k:p} \wedge c_p)$. We just need to prove that $c_{j+1} = c^{'}_{j+1}$. By equation (3), we have $(G_{j:p}, P_{j:p}) = (G_{j:k+1}, P_{j:k+1}) \circ (G_{k:p}, P_{k:p})$, $G_{j:p} = G_{j:k+1} \vee (P_{j:k+1} \wedge G_{k:p})$, $P_{j:p} = P_{j:k+1} \wedge P_{k:p}$. So, we have:

$$\begin{aligned}
c^{'}_{j+1} &= G_{j:p} \vee (P_{j:p} \wedge c_p) = [G_{j:k+1} \vee (P_{j:k+1} \wedge G_{k:p})] \vee [(P_{j:k+1} \wedge P_{k:p}) \wedge c_p] \\
&= G_{j:k+1} \vee \{(P_{j:k+1} \wedge G_{k:p}) \vee [(P_{j:k+1} \wedge P_{k:p}) \wedge c_p]\} \\
&= G_{j:k+1} \vee \{(P_{j:k+1} \wedge G_{k:p}) \vee [P_{j:k+1} \wedge (P_{k:p} \wedge c_p)]\} \\
&= G_{j:k+1} \vee \{P_{j:k+1} \wedge [G_{k:p} \vee (P_{k:p} \wedge c_p)]\} = c_{j+1}
\end{aligned}$$

Since all the incoming carries for each carry select adder in $\widetilde{csa}_{i:p}$ and $\widetilde{csa}^{'}_{i:p}$ are equal, therefore, we have:

$$(csa_{i:j+1} \ /\!\!/ csa_{j:k+1}) \ /\!\!/ csa_{k:p} = csa_{i:j+1} \ /\!\!/ (csa_{j:k+1} \ /\!\!/ csa_{k:p})$$

When using adder concatenation operator $/\!\!/$ to connect carry select adders, the incoming carries into each carry select adder are very important. They are the global carries are used to select one of the two results of each specific carry select adder block. The associativity of $/\!\!/$ is actually based on the associativity of the group carry of prefix adder which makes it possible to calculate the carry signals at different bit positions in parallel. Therefore it can speed up the computation of global carries in hybrid prefix/carry select adder. The associativity of adder concatenation operator also makes it possible to use these global carries to select the correct results for each carry select adder block in parallel. It ensures that the scheme of hybrid prefix/carry select adder can work correctly. We believe that this is the key of the hybrid prefix/carry select adder.

Therefore, the general algorithm of hybrid prefix/carry select adder can be formalized using adder concatenation operator.

Definition 2. *For a N-bit hybrid prefix/carry select adder $adder_{(n-1):0}$ which divides the operands $x_{(n-1):0}$, $y_{(n-1):0}$ into $k(k \geq 1)$ non-overlapping groups of possibly different lengths. Each group is implemented by a carry select adder. The global carries are computed by a prefix adder. Then this hybrid prefix/carry select adder is defined as follows:*

$$adder_{n-1:0} = csa_{i_k:(i_{k-1}+1)} \,/\!\!/ csa_{i_{k-1}:(i_{k-2}+1)} \,/\!\!/ \dots \,/\!\!/ csa_{i_1:0}$$

where $n-1 = i_k > i_{k-1} > \dots > i_i \geq 0$.

Using the notations and algebraic properties discussed above, the correctness of hybrid prefix/carry-select adder whose architecture is shown in Fig.1 can be proved easily.

Theorem 1 (Correctness of Hybrid Adder). *The hybrid prefix/carry select adder which is formalized in Definition 2 is correct.*

Proof. By Definition 2, lemma 1, lemma 3 and lemma 4, the correctness of hybrid prefix/carry-select adder can be proved immediately.

4 Formal Proof of Special Hybrid Adders

In the following, we choose a typical hybrid prefix/carry select adder as a case study to show the effectiveness of our proof framework.

In IBM POWER6 microprocessor, a 128-bit end-around-carry (EAC) adder is an important part of the floating-point unit [5]. The 128-bit binary adder is divided into three sub-blocks: four 32-bit adder blocks, the end-around-carry logic block and the final sum selection block. Each 32-bit adder block is implemented as a hybrid prefix/carry select adder. Fig.4 shows the architecture of the 32-bit adder block.

Each 32-bit adder block itself is partitioned into three sub-components. The first are four 8-bit adder blocks. Each 8-bit adder block is a carry select adder in which the *Conditional_0 adder* and *Conditional_1 adder* are implemented as 8-bit prefix-2 Kogge-Stone trees [17]. The second sub-component is a Kogge-Stone tree with sparseness of 8 which generates the global carry signals. The last sub-component is a sum selection block.

Fig.5 shows the block diagram of 8-bit adder block. Since Kogge Stone tree belongs to prefix adders, its correctness proof has been presented in [17], therefore, due to lemma 1, the adder block shown in Fig.5 is correct:

$$\sum_{m=j}^{j+7} x_m 2^m + \sum_{m=j}^{j+7} y_m 2^m + c_j 2^j = \sum_{m=j}^{j+7} s_m 2^m + c_{j+8} 2^{j+8}$$

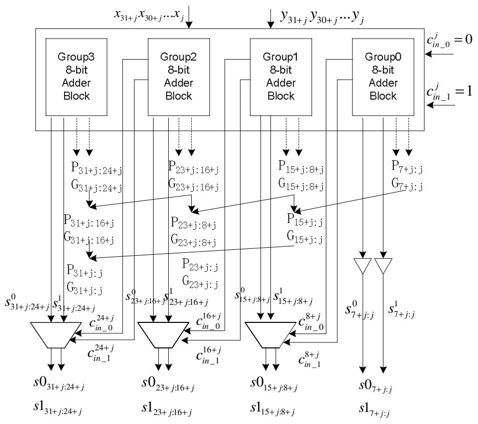

$x_{31+j}x_{30+j}\dots x_j$
$y_{31+j}y_{30+j}\dots y_j$
$c^j_{in\_0}=0$
$c^j_{in\_1}=1$
Group3 8-bit Adder Block
Group2 8-bit Adder Block
Group1 8-bit Adder Block
Group0 8-bit Adder Block
$P_{31+j:24+j}$ $G_{31+j:24+j}$
$P_{23+j:16+j}$ $G_{23+j:16+j}$
$P_{15+j:8+j}$ $G_{15+j:8+j}$
$P_{7+j:j}$ $G_{7+j:j}$
$P_{31+j:16+j}$ $G_{31+j:16+j}$
$P_{23+j:8+j}$ $G_{23+j:8+j}$
$P_{15+j:j}$ $G_{15+j:j}$
$P_{31+j:j}$ $G_{31+j:j}$
$P_{23+j:j}$ $G_{23+j:j}$
$s^0_{31+j:24+j}$ $s^1_{31+j:24+j}$
$s^0_{23+j:16+j}$ $s^1_{23+j:16+j}$
$s^0_{15+j:8+j}$ $s^1_{15+j:8+j}$
$s^0_{7+j:j}$ $s^1_{7+j:j}$
$c^{24+j}_{in\_0}$ $c^{24+j}_{in\_1}$
$c^{16+j}_{in\_0}$ $c^{16+j}_{in\_1}$
$c^{8+j}_{in\_0}$ $c^{8+j}_{in\_1}$
$s0_{31+j:24+j}$ $s1_{31+j:24+j}$
$s0_{23+j:16+j}$ $s1_{23+j:16+j}$
$s0_{15+j:8+j}$ $s1_{15+j:8+j}$
$s0_{7+j:j}$ $s1_{7+j:j}$

Fig. 4. 32-bit adder Block

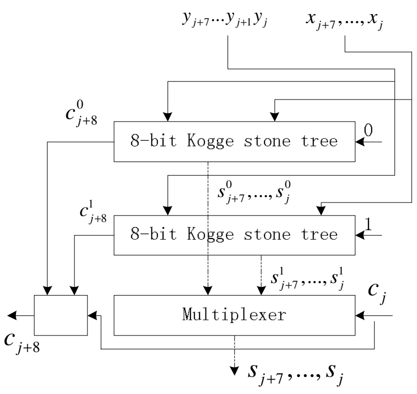


Fig. 5. 8-bit adder block

The 32-bit adder block shown in Fig.4 is built on the 8-bit adder block shown in Fig.5. For the adder in Fig.4, the inputs are two 32-bit binary numbers which are divided into four 8-bit groups as the inputs of each 8-bit adder block. These four 8-bit adder blocks are connected and the second-level kogge stone is used to calculate the global carry signals which are used to select the correct result. In fact, in IBM's design, the 32-bit adder block itself composes a carry select adder which generates two conditional sums: $s0_{31+j:j}$ with assuming that the incoming carry into the 32-bit adder block is $c^j_{in\_0} = 0$ and $s1_{31+j:j}$ with assuming that the incoming carry is $c^j_{in\_1} = 1$. Fig.6 shows the block diagram of the 32-bit adder block with incoming carry is 0.

Since IBM uses Kogge Stone tree which is on the second level in Fig.4 to compute the global carries, so the carries can be expressed using equation (2). We can use adder concatenation operator to formalize the adder shown in Fig.6. Therefore, by theorem 1, it is easy to prove the correctness of adder block of Fig.6:

$$\sum_{m=j}^{31+j} x_m 2^m + \sum_{m=j}^{31+j} y_m 2^m + c^j_{in\_0} 2^j = \sum_{m=j}^{31+j} s0_m 2^m + c^{32+j}_{in\_0} 2^{32+j}$$

For the case that the incoming carry $c^j_{in\_1} = 1$, we can do the similar formal analysis and show its correctness:

$$\sum_{m=j}^{31+j} x_m 2^m + \sum_{m=j}^{31+j} y_m 2^m + c^j_{in\_1} 2^j = \sum_{m=j}^{31+j} s1_m 2^m + c^{32+j}_{in\_1} 2^{32+j}$$

In this way, the correctness of the adder block shown in Fig.4 can be proved using the framework developed in this paper.

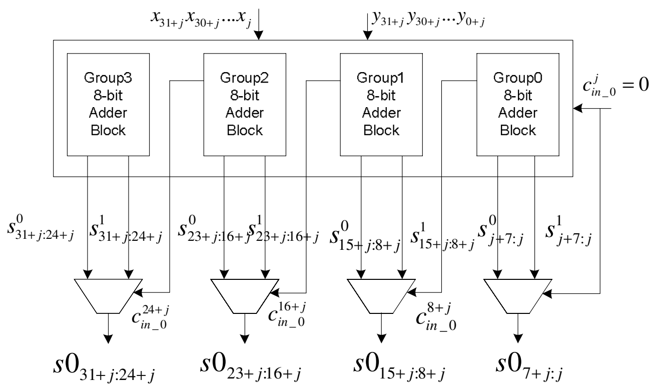


Fig. 6. 32-bit adder block with incoming carry is 0

5 Conclusion

In this paper, we propose a general architecture for the design of hybrid prefix/carry-select adder. Based on the traditional notions such as propagated carry and generated carry, we develop a proof framework for the correctness of the general architecture. Several previous adders in the literature are all special cases of this general architecture. They differ in the way Boolean functions for the carries are implemented. Therefore, using our framework, we can formalize special hybrid prefix/carry-select adders and prove their correctness. We choose a hybrid adder which is implemented in IBM POWER6 microprocessor's floating-point unit as a case study. The formal proof for the correctness of the IBM's hybrid adder shows the effectiveness of our algebraic structures.

References

1. Lynch, T., Swartzlander, E.: A spanning tree carry lookahead adder. IEEE Trans. Comput. 41, 931–939 (1992)
2. Kantabutra, V.: A recursive carry-lookahead/carry-select hybrid adder. IEEE Trans. Comput. 42, 1495–1499 (1993)
3. Kwon, O., Swartzlander Jr., E.E., Nowka, K.: A Fast Hybrid Carry-Lookahead/Carry-Select Adder Design. In: Proceedings of the 11th Great Lakers symposium on VLSI, pp. 149–152 (2001)
4. Wang, Y., Pai, C., Song, X.: The Design of Hybrid Carry-Lookahead/Carry-Select Adders. IEEE Transactions on Circuits and Systems II:Analog and Digital Signal Proessing 49, 16–24 (2002)
5. Yu, X.Y., Fleischer, B., Chan, Y.H., et al.: A 5GHz+ 128-bit Binary Floating-Point Adder for the POWER6 Processor. In: Proceedings of the 32nd European Solid-State Circuits Conference, pp. 166–169 (2006)
6. Tyagi, A.: A reduced-area scheme for carry-select adders. IEEE Trans. on Computers. 42, 1163–1170 (1993)

7. Arjhan, A., Deshmukh, R.G.: A Novel Scheme for Irregular parallel-prefix adders. In: Proceedings of IEEE Southeastcon 1997, ‘Engineering new New Century’, pp. 74–78 (1997)
8. O’Donnell, J., Rnger, G.: Derivation of a carry lookahead addition circuit. Journal of Functional Programming 14, 127–158 (2004)
9. Kapur, D., Subramaniam, M.: Mechanical verification of adder circuits using rewrite rulelaboratory. Formal Methods in System Design 13, 127–158 (1998)
10. Sheeran, M.: Hardware design and functional programming: a perfect match. Journal of Universal Computer Science 11, 1135–1158 (2005)
11. Hinze, R.: An algebra of scans. In: Kozen, D. (ed.) Proceedings of the Seventh International Conference on Mathematics of Program Construction, pp. 186–210 (2004)
12. Han, J.: Stone, G.: The implementation and verification of a conditional sum adder. Technical Reports, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada (1988)
13. Elleithy, K.M., Aref, M.A.: A Rule-based Approach for High Speed Adders Design Verification. In: 37th Midwest Symposium on Circuits and Systems, pp. 274–277 (1994)
14. Koren, I.: Computer Arithmetic Algorithms, 2nd edn. A.K.Peters, Natick (2002)
15. Chen, G., Liu, F.: Proofs of correctness and properties of integer adder circuits. IEEE Trans. on Computers 59, 134–136 (2010)
16. Brent, R.P., Kung, H.T.: A regular layout for parallel adders. IEEE Trans. on Computers C-31, 260–264 (1982)
17. Kogge, P.M., Stone, H.S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. IEEE Trans. on Computers C-22, 786–793 (1973)

An Efficient Non-blocking Multithreaded Embedded System

Joseph M. Arul, Tsung-Yun Chen, Guan-Jie Hwang, Hua-Yuan Chung, Fu-Jiun Lin, and You-Jen Lee

Fu Jen Catholic University, Department of Computer Science and Information Engineering, Hsin Chuang, Taipei, Taiwan, R.O.C.
arul@csie.fju.edu.tw

Abstract. Most embedded systems are designed to perform one or fewer specific functions. It is important that the hardware and the software must closely interact to achieve maximum efficiency in all of these realms and overcome the drawbacks found in each aspect. This research focuses on designing a totally new Instruction Set Architecture (ISA) as well as hardware that can closely tie together with the new emerging trend such as multithreaded multicore embedded systems. This new ISA can efficiently execute simple programs in a much efficient way as well as have a better cache performance with less cache misses due to the way the program is split into non-blocking multithreading paradigm. This particular research is aimed to compare the performance of this new non-blocking multithreaded architecture with the ARM architecture that is commonly used in an embedded environment. It has a speedup of 1.7 in general compared to the MIPS like ARM architecture.

1 Introduction

Multithreading has become a mainstream trend in many multicore architectures. Besides, to achieve high level parallelism, it creates several simultaneous multithreading executions, which can be scheduled on different cores at the same time. In control flow execution paradigm, a memory access may occur anywhere inside one thread, which might cause the processor to stall for the memory access, thus increasing the overall execution time of the program. Hence, it is necessary to identify memory accesses and to use non-blocking thread where the data is pre-loaded before the execution. By using non-blocking multithreaded model, the system makes sure that the data needed by the thread are brought into memory and remain as long as it is needed by a thread. As mentioned in [1] the hardware/software component must be loaded into an embedded system which is known as Component Based Development (CBD) that is appropriate for embedded systems.

1.1 Multicore Multithreaded Embedded System

In the year 2008, June 12th issue by William wongś article [2] on electronic design for embedded system announced "Multicore Multithreaded goes Embedded". This design will be able to consume less power by running multiple cores simultaneously at a slower speed. In this architecture, each core contains dual-port cache tag memory, allowing simultaneous access by each processing element. However, the cache coherence mechanism and memory interface allows only one processing element to access at a time. In multicore architectures, cache misses can have great impact. The details of how this architecture is built by Intel can be found on the webpage [2]. This architecture has a maximum speed of 800 MHz. It has typically two to four cores, Virtual Processing Element (VPE) with 32-Kbyte L1 caches about 3.3 mm^2.

Rest of the paper is organized as follows: section two presents some background and suitable architectures that have been discussed by various research communities. Section three presents the unique features that are used in this architecture. Section four presents the environment in which the experiments were conducted. Section five presents the results and comparison of this architecture with ARM that is commonly used. Final section will present the conclusion.

2 Background and Related Research

Each architecture aims to enhance one aspect of embedded processors such as memory efficiency, power efficiency or silicon area consumption. The “Synchroscalar” architecture [3] which has a multicore embedded processor mainly evaluates its performance for multimedia applications. The synchroscalar architecture uses number of Processing Elements (PE) up to 16 PE where each processing element can be mapped to the application in a synchronous dataflow (SDF) model. The synchronous dataflow model has been supported in many architectures. Synchroscalar architecture focuses on increasing efficiency rather than performance. Synchroscalar architecture uses 256 bits wide bus grouped into 8 32 bits separable vertical buses separated in between each of the tiles. The main advantage of this architecture is that, different portions of applications can be mapped to different processors. However, one drawback of this architecture is the area consumption that would increase as we try to reduce the power consumption by increasing the tile size.

Stanford has proposed register pointer architecture for efficient embedded architecture [4]. In this architecture, the register is accessed indirectly through the register pointer. In the conventional architecture, register file capacity can not be increased without restoring the longer instruction words. By using register pointer, without increasing the instruction word, it allows large register file usage. Thus, number of registers used can be increased. By having large number of registers, number of loads and stores can be increased. Many techniques such as loop unrolling, register renaming and reuse registers require large number of registers, in order to improve the performance of the execution of the program. It mainly provides large register file capacity and flexibility in register renaming. It only extends the ARM instruction set to support register pointer architecture. For this research, they modified the ARM version of the Simplescalar architecture to implement register pointer architecture (RPA). The results show that the RPA leads to 46% average performance improvement and 32% average reduction in energy consumption. However, it increases the code size.

The IEEE Micro May/June 2009 issue was mostly dedicated to the articles on embedded multicore processors and systems. There are many potential approaches to solve the numerous multicore related issues. This issue mainly focuses on solving some of the issues related to multicore multithread embedded system. In this issue, the guest editor, Markus Levy, writes, "unlike desktop PC or server applications, multicore devices used in embedded systems are as devices as the stars in the universe" [5]. Hence, it has lot of issues that need to be addressed and solved. Thomas bergs article [6] on "Maintaining I/O data Coherence in Embedded Muticore Systems" focuses on solving the communication problem that exists in passing data between the I/O peripheral devices and on chip CPUs. One of the main issues is solving the problem that exists in multithreaded multicore architectures is the bottleneck associated with the shared memory and other shared resources such as CPUs, multithreads and so on. This particular research and aim is to focus on addressing the proper usage of multithread and memory access delays involved in multicore architectures.

This research [7] presents software standards for the multicore era. The authors say that, even though the multicore hardware platform emerges faster, it is necessary to define some software standards for these multicore systems. This paper also states that it is impossible to achieve parallelism using standards defined for symmetric multiprocessing on a heterogeneous multicore. In the multicore architecture, top priority must be inter-core communication. Hence, the consortiums working group has completed the multicore's communication API.

3 Non-blocking Multithreaded Embedded System

Decoupled architectures have not commonly been used in general purpose computing due to their inability to tolerate control intensive code that exists in wide range of scientific applications. However, with the multithreading concept that is becoming more common, it can overcome the loss of decoupling dependencies. Combining multithreading and decoupling can provide memory and control latency hiding, parallel instruction execution, and efficient resource utilization with a minimal amount of complexity. In this particular architecture, there are two processes, which are known as Synchronization Process (SP) and Execution Process (EP) as shown in figure 1. Only SP is responsible for data access by executing the LOAD instructions as well as storing of data by executing the STORE instructions. Only SP is linked to the memory unit. It is possible to do in multithreaded architectures, where the application is divided by the compiler such as, all the LOAD and STORE instructions which are responsible for fetching and storing of data from memory. Every application is divided into several threads, where each thread has three portions to execute. As you can see in figure 1, SP has two queues and EP has one queue where all the threads go through these three queues at least ones. Some threads may go through these queues more than ones, depending on the program execution. The left queue on SP is basically for the initiated threads awaiting SP to fetch and load the data needed for the thread. When all the data needed for the thread is loaded, it is moved to

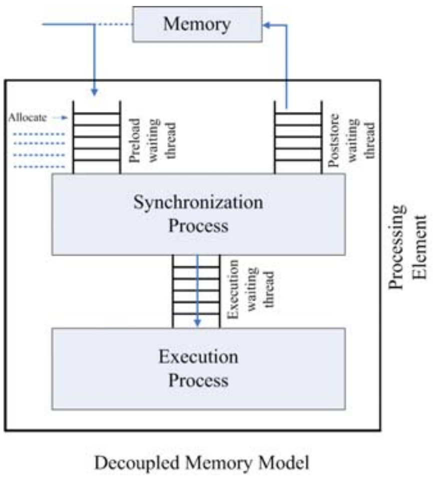


Fig. 1. Decoupled Architecture for Embedded System

the EP queue for execution. It fetches the threads from its queue and executes all the instructions. All the data needed for the execution are already loaded onto the thread. It is called non-blocking multithreaded architecture, since all the data needed for a thread are loaded onto the thread before execution. When a thread completes execution, it is placed onto the right queue of SP.

3.1 Execution Paradigm of This Non-blocking Multithreaded Architecture

In this system, the instruction of every thread cannot retain functional properties. Figure 2 shows the nature of a simple program that can be scheduled into multiple threads by the compiler. It is a simple program, where the main thread loads all the data (x, y, a and b) needed for this simple program. Thread 1 loads all the data and creates thread 2, 3 and 4 during execution phase of the program. Thread 2 receives X and Y to perform multiplication operation. Similarly, thread 3 receives A and B to perform addition operation. Thread 1 needs to pass the data to thread 2 and 3 in order to perform their tasks. One way of passing the data between threads is to store the data into the relative frame memory of each thread. When thread 1 finishes its execution, it stores X and Y into the frame memory of thread 2. When Thread 2 starts execution, it loads its data from frame memory. Similarly, thread 1 stores A and B into the frame memory of Thread 3. When thread 3 starts execution, it loads A and B from its frame memory. However, efficient way of passing of data from one thread to the other would be storing into its relative register context. When thread 1 finishes execution, it stores X and Y into the registers of thread 2. Thus, it eliminates the load operations of thread 2. Similarly, thread 3 can also eliminate the load operations which can perform the execution operation without much delay. The

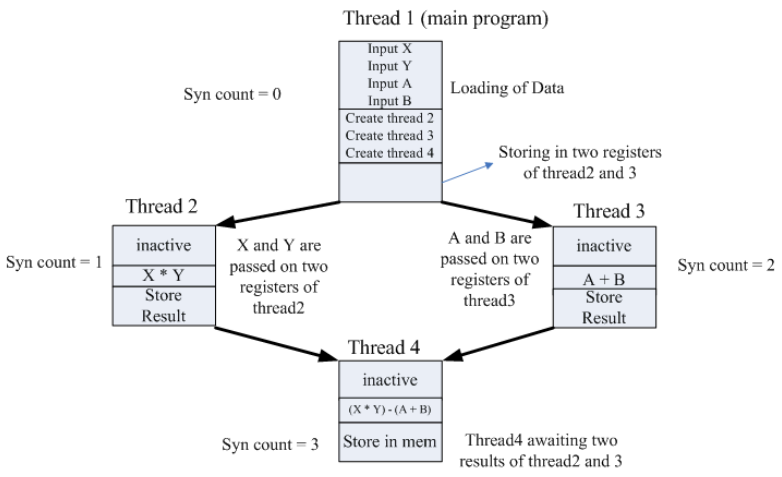


Fig. 2. Execution process of an application using several Non-blocking multithread

whole application can save several cycles, and as a result, it can increase the performance of the whole application program. The memory access operation would be reduced in this efficient way of passing of data in a multithreaded architecture.

3.2 Scheduling Unit

To create a thread in this architecture, FALLOC instruction is used, whose main function is to allocate frame reference by a Frame Pointer (FP) X for a thread to use. At the same time, a frame is initialized by storing Instruction Pointer (IP) and a Synchronization Pointer (SC). Each thread, when being created, will be assigned a SC, which is to record the value that each thread is supposed to receive. With the reception of a value, SC is accordingly reduced by one value until SC finally turns zero. When SC reaches zero, it suggests that the reception of all values of a thread's need is completed. In order to accelerate frame allocation, we do the allocating of the frame necessary for the thread, fixing the frame size at a certain length and setting the allocated frame in the stack. It is the SU that controls the stack. Whenever frame allocation is to be carried out for the thread to use, SU will come out of the stack to pop a frame. On the other hand, when a thread reaches the end, FFREE instruction will be performed. During that time SU will push the frame into the stack for the following thread to use. FALLOC and FFREE instructions are set to take two instruction cycles while FORKEP and FORKSP instructions are set to be four instruction cycles. The instruction cycles of FALLOC, FFREE, FORKEP and FORKSP are set to two due to the platform Sparcle [8] as a frame of reference. Besides, this architecture is set to be with multiple EP and SP, and SU is to do the scheduling of pre-load (PLC) and post-store (PSC).

4 ARM Architecture

The ARM is a 32 bit RISC special purpose embedded Instruction Set Architecture (ISA) started in 1983 as a development project by Acron computers Ltd. It is most widely used microprocessor and microcontroller for embedded systems. ARM processors are used extensively in modern electronics such as PDA, iPods, mobile phones and other handheld devices. The StrongARM was a collaborative project between DEC and ARM to create a faster processor based on the existing one. As a result, StrongARM is a faster and somewhat different version of ISA. For the purpose of research and comparison of the newly designed non-blocking multithreaded embedded systems, SimIt ARM was used. SimIt ARM is a free open source software available under GNU general public license. This simulator supports both system and user level ARM programs. It supports basic ARM memory management and I/O devices. SimIt ARM simulator and the non-blocking multithreaded architecture are both developed using C++. The non-blocking multithreaded architecture was developed in order to compare the performance of the existing control flow MIPS like ARM architecture. Similar benchmarks were used in both of the simulators to compare the performance as well as the data cache in some programs. The non-blocking multithreaded architecture cache performance seems to be better than the ARM architecture, due to the nature of splitting the program into several threads as well as the efficient way of passing of data in a multithreaded architecture. In order to have a fair comparison on both simulators, the same parameters are identically set on them and same cache sizes were set for both environments. The instruction cycle count is set to 1 in both simulators. When a thread finishes its execution, it uses FFREE to release the thread and its related resources. The experiments on the next section were run on SimIt ARM as well as the new non-blocking architecture using a different ISA that is suitable for the non-blocking multithreaded multicore embedded system.

5 Performance Evaluation of Non-blocking Multithreaded Architecture

Table 1 presents the data collected after running the summation program. From the table we can note that the non-blocking multithreaded architecture has better performance than the simple ARM architecture using the control flow like programs. With the usage of various data sizes from 10,000 to 80,000 we can see that the speedup remains close to 2. This is a simple benchmark with one simple loop to add the loop variable and sum the result. In case of ARM architecture, every time the index value is added to the total, it is stored in a new memory address and it needs to retrieve the total value from the memory. Several memory accesses to the same value have to be stalled in case of cache miss. However, in the non-blocking multithreaded architecture, since it is using the multithreaded architecture and implementing efficient way of passing of data, the new value is passed to the thread in its register contexts rather than storing into the frame

Table 1. Comparison of Cycle count for the Summation Program

N	NBM (clock cycle)	ARM (clock cycle)	Speedup
10000	86228	177242	2.055504
20000	172228	347242	2.016176
30000	258228	517242	2.003044
40000	344228	687242	1.996473
50000	430228	857242	1.99253
60000	516228	1027242	1.9899
70000	602228	1197242	1.988021
80000	688228	1367242	1.986612

Table 2. Address Book Data Retrieval Program

N	NBM (clock cycle)	ARM (clock cycle)	Speedup
100	3731	5395	1.708016
500	16131	27552	1.765325
1000	31631	55839	1.771297
1500	47131	83483	1.774313
2000	62631	111127	1.775569
2500	78131	138727	1.776879
3000	93631	166371	1.777817
3500	109131	194015	1.42851
4000	155168	221659	1.778757
4500	140131	249259	1.778643
5000	155631	276812	1.708016

memory and retrieving. In this case, it not only improves the performance of this architecture, but also less memory access helps towards cache performance of this architecture. For some of the benchmarks, cache performance is also included in the later part of this section in order to evaluate the cache miss ratio between ARM and this new non-blocking multithreaded architecture.

Table 2 presents the results of running a database retrieval program, which is often used in many embedded systems such as cell phone, PDA and other small devices. The above program presents the database with limited number of fields such as last name, first name and phone number and so on. For this particular experiment, worst case scenario was chosen to observe the number of cycles needed while running this program. Rest of the programs were run using efficient way of passing of data in a multithreaded architecture as explained in the previous example. From table 2 one can note that the speedup of roughly 1.7, which remains the same for all the data size except 3500. For this program, to search for the particular data in a database, sequential search was used. Most embedded systems would have rather less amount of data at any time, hence sequential search would be sufficient to use. It also clearly shows that the multithreaded architecture would be suitable for an embedded system that is also a multithreaded multicore system.

Table 3 shows the results of sequential search program. For the non-blocking architecture, random numbers are stored in an 'istructure' memory. For the non-blocking architecture the 'istructure' memory is used, since it uses data flow like engine. For the ARM architecture, the random numbers are stored in an array. Since, search is one of the necessary operations that has to be performed in a handheld device too, like the computer systems; this benchmark is also taken into consideration for the performance. Even though the sequential search is not an efficient search algorithm, it can be used in handheld devices where minimum amount of data is stored due to its memory constraints. This program does not show great improvement since the involvement of the frame memory access as

Table 3. Sequential Search Program

N	NBM (clock cycle)	ARM (clock cycle)	Speedup
1000	29247	30402	1.039491
5000	145247	152325	1.048731
10000	290247	304723	1.049875
15000	435247	457070	1.050139
20000	580247	609419	1.050275
25000	725247	761521	1.050016
26000	754247	791920	1.049948

Table 4. Greatest Common Divisor program

N	NBM (clock cycle)	ARM (clock cycle)	Speedup
100	33637	85911	2.554062
200	67452	172179	2.552615
300	101162	258178	2.552124
400	135257	345315	2.553029
500	169037	431662	2.553654
600	203342	519256	2.553609
700	237367	606092	2.553396
800	272022	694610	2.553507
900	306642	783071	2.553698
1000	341857	873214	2.554325

Table 5. Prime Factorization Program Written in a Non-Recursive Fashion

N	NBM (clock cycle)	ARM (clock cycle)	Speedup
10000	7657	25891	3.38135
50000	54284	130066	2.396028
100000	143458	303799	2.117686
150000	268795	523756	1.948533
200000	431803	791983	1.83413
250000	605260	1073899	1.774277
300000	819940	1407533	1.716629

Table 6. Image Zooming Program

N	NBM (clock cycle)	ARM (clock cycle)	Speedup
8x8	29374	20941	0.712909
16x16	114486	140676	1.228762
32x32	453286	554583	1.223473

well as the 'istructure' memory which has a basic operation of store ones, retrieve as many times as you want. Similar to the above benchmarks, GCD program was run on both non-blocking multithreaded and ARM architecture. For this program, the speedup remains constant (about 2.6) for all data size that are used. The results for the GCD program are presented in table 4.

Table 5 presents the results of running prime factorization program. This program finds the entire prime factor given an N. It is similar to finding all the prime numbers; however, it differs a bit from the finding prime number. For example, given N=10, it can be multiplied by 2 * 5 =10. Similarly, for a given N, it finds the entire prime factor that can be used to obtain the result. From the table it can be observed that when N is small, we can get a speedup of 3.3. As the data size is increased, the speedup comes up to 1.7. Most of the embedded systems are designed to run rather small programs than large scientific programs. Hence, for our experiments too, we did not consider running large scientific programs. Table 6 presents the results of image zooming program. Since the memory size is limited in an embedded system, only small size of data can be used for this program. Hence, the data sizes vary from 8x8 to 32x32 array size. For this program too, unrolling

Table 7. Cache Hit Ratio Shown in Percentage for Address book database Program

N	NBM (dcache hit ratio)	ARM (dcache hit ratio)	Diff. in Cache Hit Ratio
100	96.0784	70.4	25.6784
500	98.008	71.502	26.506
1000	98.2036	72.696	25.5076
1500	98.269	73.814	24.455
2000	98.3017	74.825	23.4767
2500	98.3213	75.745	22.5763
3000	98.3678	76.584	21.7838
3500	98.3724	77.352	21.0204
4000	98.42	78.059	20.361
4500	98.3785	78.712	19.6665
5000	98.3806	79.321	19.0596

Table 8. Cache Performance of Sequential Search Program (dcache)

N	NBM (dcache hit ratio)	ARM (dcache hit ratio)	Diff. in Cache Hit Ratio
1000	98.4	50.206	48.194
5000	98.42	55.043	43.377
10000	98.43	59.874	38.556
15000	98.4333	63.735	34.6983
20000	98.435	66.891	31.544
25000	98.436	69.52	28.916
26000	98.4346	69.993	28.4416

of loop was performed in order to increase the instruction level for each thread. As the array size is increased, the speedup in performance shows improvement of 1.2. In many handheld devices, image manipulation as well as enlarging and zooming of images are very common. Hence, this program was picked to see the impact of performance of this non-blocking architecture against the ARM embedded architecture. One of the main characteristics of non-blocking architecture is the spawning of many threads, each thread having more workload improves the performance of a multithreaded architecture.

Even though cache miss cycles were set same in both the architectures, ARM architecture shows that the hit ratio is very low. Difference in hit ratio ranges between 19 to 25. This could also contribute significantly to the execution performance of the non-blocking mulithreaded architecture. Table 7 shows the cache hit ratio for ARM and non-blocking multithreaded architecture. When the data size is 5000, it still has the difference of 19%. In the new architecture, the hit ratio remains almost 96% to 98%. Similarly, we can note from table 8 for the sequential search program. In this case, the array is stored in 'istructure' or array memory. Every time the array has to be searched, it fetches the data from memory and stores in cache memory. We can see that the multithreaded architecture, cache hit ratio remains constant about 98.4%. It has very less cache miss ratio. For the ARM architecture, the cache hit ratio improves slowly. However, the improvement is not that great. Hence, we can conclude that the non-blocking multithreaded architecture which divides the program into several threads not only improves cache hit ratio, but also improves the overall performance of the program.

6 Conclusion

From section 5 we can conclude that, the new architecture can be suitable for a multithreaded multicore embedded system with limited memory size. This architecture reduces the memory access compared to the existing architectures. As a result, it also shows fewer cache misses compared to ARM architecture. On the average, the benchmark results show that it is to get a speedup of 1.5 or more. Since this new architecture has limited number of registers, the future aim is to use instruction pointer architecture as proposed by some researchers which can allow us to use several registers. There are several SOC researches using VHDL like language to measure the exact hardware complexity and the execution time accurately. One of our future aim is to develop this architecture using VHDL like language to calculate the exact amount of hardware complexity and the performance in microseconds.

References

1. Juncao, L., et al.: Embedded Architecture Description Language. In: The 32nd Annual IEEE International conference on Computer Software and Applications (2008)
2. Multicore, Multithreaded Goes Embedded, `http://electronicdesign.com/Articles/ArticleID/19022/19022.html`
3. Oliver, J., Rao, R., Sultana, P., Crandall, J., Czernikowski, E., Jones, L.W.I., Franklin, D., Akella, V., Chong, F.T.: Synchroscalar: a multiple clock domain, power-aware, tile-based embedded processor. In: The 31st Annual International Symposium on Computer Architecture (2004)
4. JongSoo, P., Sung-Boem, P., Balfour, J.D., Black-Schaffer, D., Kozyrakis, C., Dally, W.J.: Register Pointer Architecture for Efficient Embedded Processors. In: Design, Automation & Test in Europe Conference & Exhibition (2007)
5. Levy, M., Conte, T.M.: Embedded Multicore Processors and Systems. IEEE Micro 29, 7–9 (2009)
6. Berg, T.B.: Maintaining I/O Data Coherence in Embedded Multicore Systems. IEEE Micro 29, 10–19 (2009)
7. Holt, J., Agarwal, A., Brehmer, S., Domeika, M., Griffin, P., Schirrmeister, F.: Software Standards for the Multicore Era. IEEE Micro 29, 40–51 (2009)
8. Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.H., Yeung, D., DSouza, G., Parkin, M.: Sparcle: An Evolutionary Processor Design for Multiprocessors. IEEE Micro 13, 48–61 (1993)

A Remote Mirroring Architecture with Adaptively Cooperative Pipelining*

Yongzhi Song, Zhenhai Zhao, Bing Liu, Tingting Qin, Gang Wang, and Xiaoguang Liu

Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University
94 Weijin Road, Tianjin 300071, China
syzcch@sina.com, zhaozhenhai1985@gmail.com, liubing87@126.com, paula_qin_1987@126.com, wgzwp@163.com, liuxg74@yahoo.com.cn

Abstract. In recent years, the remote mirroring technology has attracted increasing attention. In this paper, we present a novel adaptively cooperative pipelining model for remote mirroring systems. Unlike the traditional pipelining model, this new model takes the decentralization of processors into account and adopts an adaptive batching strategy to alleviate imbalanced pipeline stages caused by this property. To release the heavy load on CPU exerted by compression, encryption, TCP/IP protocol stack and so on, we design fine-grained pipelining, multi-threaded pipelining and hybrid pipelining. We implement a remote mirroring prototype based on Linux LVM2. The experimental results show that, the adaptively cooperative pipelining model balances the primary and the backup sites - the two stages of the pipeline effectively, and fine-grained pipelining, multi-threaded pipelining and hybrid pipelining improve the performance remarkably.

Keywords: remote mirroring, cooperative pipelining, adaptive batching, fine-grained, multi-threaded.

1 Introduction

Consistently, data protection is a hot topic in IT academia and industry. Especially in recent years, after several great disasters, some enterprises with perfect data protection resumed quickly, while many others went bankrupt because of data loss. So data protection technologies have attracted increasing attention. Remote mirroring is a popular data protection technology that tolerates local natural and human-made disasters by keeping a real-time mirror of the primary site in a geographically remote place. There are two typical remote mirroring strategies: synchronous and asynchronous [1]. The latter is preferable due to the former's heavy sensitivity to the Round Trip Time (RTT) [2].

* Supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), SRFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000).

In this paper, we present a new cooperative pipelining model to depict remote mirroring systems. By "cooperative" we mean that the pipeline is across the primary site, the network and the remote backup site. Since each subtask naturally has an "owner", so we cannot distribute them arbitrarily to balance the pipeline stages. For this, we present an adaptive batching algorithm. Write requests are propagated to the backup site in batches and the batch size (interval) is adjusted dynamically according to the processing speed of the primary and the backup sites. We implement data compression and encryption in our prototype to reduce network traffic and enhance security respectively. These operations put a lot of pressure on CPU. For this, we design three accelerating methods: fine-grained pipelining, multi-threaded pipelining and hybrid pipelining.

The rest of this paper is organized as follows: In Section 2, we focus on the related work in the recent years. In Section 3, we illustrate the basic architecture of our system and present the adaptive cooperative pipeline. In Section 4, we introduce the implementation and evaluate it from results of a quantity of experiments. Finally, the conclusions and future work is given in Section 5.

2 Related Work

EMC Symmetrix Remote Data Facility (SRDF) [3] is a synchronous block-level remote replication technique that will switch to semi-synchronous mode if the performance is below the threshold. Veritas Volume Replicator (VVR) [4] is a logical volume level remote replication technique. It supports multiple remote copies and performs asynchronous replication using a log and transaction mechanism. Dot Hill's batch remote replication service [5] schedules point-in-time snapshots of the local volume, then transfers the snapshot data changes to one or more remote systems.

Network Appliance's SnapMirror [1] uses snapshot to keep the backup volume up to date. The WAFL file system is used to keep track of the blocks that have been updated. Seneca [6] delays sending a batch of updates to the remote site, in the hope that write coalescing will occur. Writes is coalesced only within a batch, and batches must be committed atomically at the remote site to avoid inconsistency.

Our prototype is also implemented in the logical volume level like VVR and Dot Hill's remote replication service. Like Seneca, the updates are sent to the backup site in batches for performance reasons. Our prototype also adopts an adaptive mechanism. However, it is based on asynchronous mode and for pipeline stage balancing rather than network conditions adapting.

There are many other remote replication products, such as IBM's Extended Remote Copy (XRC) [7], HP Continuous Access Storage Appliance (CASA) [8] and so on. In recent academic studies, [9] presents a prototype in which the synchronous and asynchronous mode are specified by the upper level applications or the system. [2] uses Forward Error Correction (FEC) and "callback" mechanism for high reliability.

3 Adaptively Cooperative Pipelining

Fig. 1 shows the architecture of our prototype. It is an asynchronous remote mirroring system implemented in Linux LVM2 [10]. NBD (the Network Block Device) [11] is used for data transmission between the primary and the backup sites. When a write request arrives, it is duplicated, and then the original enters the local queue and the replica enters the remote queue. The requests in the remote queue are sent to the backup site in batches at intervals. Since NBD transfers messages using TCP, the consistency is guaranteed as long as each batch is committed atomically in the backup site. In order to reduce the network traffic, requests are compressed before they are being sent. They are also encrypted to provide good security. It is easy to see that the order of compressing before encrypting is superior to the reverse order in computational complexity. Therefore, in the backup site, the requests are decrypted and then decompressed before being committed.

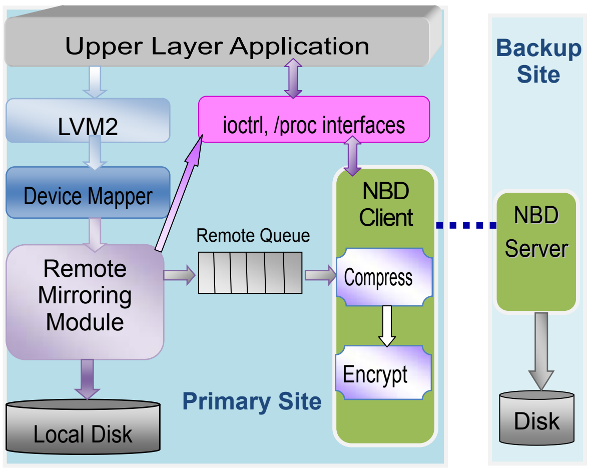


Fig. 1. Prototype architecture

3.1 Cooperative Pipelining

In order to maximize the throughput, an obvious way is to overlap the operations in the primary site and the operations in the backup site. That is, after a batch is sent, instead of waiting the reply, the primary site immediately throws itself into processing the next batch while the backup site processes the previous one. So we can describe the processing of the requests by a two-stage pipelining model. We call the two stages *the primary stage* and *the backup stage* respectively. This pipeline is "*cooperative*", that is, each batch is processed by the primary site and the backup site cooperatively. We know that, for a given task, the more the stages and the nearer the size of the stages, the higher the performance of the pipeline is. However, the cooperative pipelining model has some unique properties against this.

For traditional pipelining, to increase the speed of a single pipeline, one would break down the tasks into smaller and smaller units. For example, pipelines in modern CPUs typically have more than 20 stages. However, a cooperative pipeline is naturally composed of eight subtasks including batch assembling, compression, encryption, batch transmission, decryption, decompression, disk writing and reply. Since it is a software pipeline, further task decomposition will induce significant interaction overhead. Moreover, we can not accurately predict the execution time of some stages (primarily the disk operations and the network transmission whose performance depend on the current state of the system heavily). This is a serious obstacle to high efficient pipelining.

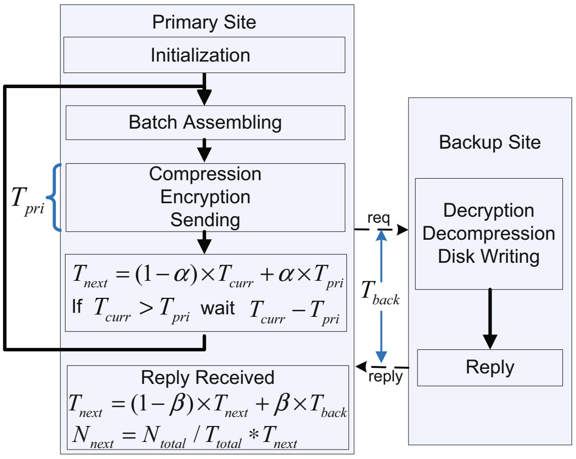
Typical distributed pipelining models, such as the pipelined gaussian elimination algorithm [12], break down the tasks into subtasks with the same size and assign them to proper processors. However, each subtask in a cooperative pipeline has a specific "owner" so that it can not be assigned to other processors. For example, the backup site can not perform data compression which must be done at the primary site. This inherently immutable task mapping contributes to the difficulty in load balancing. We must equal the speed of the primary stage and the backup stage for perfect load balance. Otherwise, processor will still be idle even if we break down the task into smaller subtasks with the same size. Moreover, if we want to improve performance by deepening the pipeline, we must further divide the primary stage and the backup stage identically.

3.2 Adaptive Batching

As mentioned above, the primary site can process the next batch immediately after the previous batch is sent. In a single-user system, this "best-effort" strategy guarantees the optimal performance. However, it has some drawbacks.

If the speed of the two stages are different, for instance, the primary stage is faster than the backup stage, the two sites are out of step under best-effort strategy. If the user application keeps up the pressure on the storage subsystem, the gap between the primary site and the backup site becomes wider and wider until the primary site exhausts system resource. Then the primary site will slow down to wait for the replies from the backup site to release enough resource. This brings unsteady user experience (response time of the primary site). Moreover, exhausting system resource by one process is not good for a multi-user system. This will impact the stability of the system and the performance of other processes seriously.

In a word, best-effort strategy does not coordinate the primary site and the backup site well. Or, it "coordinates" the two sites by exhausting system resource. So we introduce an *adaptive batching* algorithm for coordination. We set a *batch interval* when system initializing. This interval defines the request accumulating and the batch processing periods. That is, the requests accumulated in the previous period are processed in batches in the next period. Every time a batch finishes, we adjust the batch interval to approach the execution time of the primary stage and the execution time of the backup stage. Therefore, if the batch interval converges to a stable condition eventually, the primary stage and the backup stage will have the same execution time (both equal to the batch interval). Note that the execution time of disk operations and network transmission is not proportion to the batch size. So, although interval adjusts lengthens (or shortens) of both stages, the increment of the faster one (generally the primary stage) may be longer than that of the slower one (generally the backup site which contains disk operations and network transmission), therefore the batch interval converges.



Primary Site
Initialization
Batch Assembling
Compression
Encryption
Sending
T_{pri}
$T_{next} = (1-\alpha)\times T_{curr} + \alpha\times T_{pri}$
If $T_{curr} > T_{pri}$ wait $T_{curr} - T_{pri}$
Reply Received
$T_{next} = (1-\beta)\times T_{next} + \beta\times T_{back}$
$N_{next} = N_{total} / T_{total} * T_{next}$
req
reply
T_{back}
Backup Site
Decryption
Decompression
Disk Writing
Reply

Fig. 2. Adaptive batching algorithm

Unlike best-effort strategy, adaptive batching algorithm lets the faster site sleep for a while instead of processing the next batch immediately when the two sites are out of step, that is, slows down the faster site to force the two sites in step. The advantage of this strategy is obvious: the faster site will not exhaust system resource, therefore it does not impact other processes in the system. Another advantage of adaptive batching is the good adaptability to change of network conditions. If network fluctuates, by interval adjusting, the primary site and the backup site adapt to the speed of network automatically and timely. The following *adaptive formulas* are used to adjust the batch interval.

$$T_{next} = (1-\alpha) \times T_{curr} + \alpha \times T_{pri} \tag{1}$$

$$T_{next} = (1-\beta) \times T_{next} + \beta \times T_{back} \tag{2}$$

where T_{curr} and T_{next} denote the current and the next intervals respectively, and T_{pri} and T_{back} denote the execution time of the primary stage and the backup stage respectively. In our implementation, T_{pri} includes time spent in batch assembling, compression, encrypting and batch sending, and T_{back} includes time spent in batch receiving, decryption, decompression, disk write and reply. α and

β are *adjusting factors* which are real numbers between 0 and 1. They control how fast the batch interval approaches to real processing time and how fast the pipeline adapts to network change. To counter continuous heavy load, we use a *batch size threshold* to control the maximum number of requests in a batch:

$$N_{next} = N_{total}/T_{total} \times T_{next} \tag{3}$$

where N_{total} denotes the total number of requests that have been processed, and T_{total} denotes the total processing time. That is, the processing capacity is estimated by statistics and is used to set the next threshold N_{next}. Fig. 2 illustrates the adaptive batching algorithm. When the primary stage finishes, Formula 1 is used to adjust the batch interval and the primary site will sleep a while if the primary stage is shorter than the current interval. Formula 2 is used when the reply is received.

The adaptive batching algorithm has several variants for different purposes. For example, we can set lower and higher thresholds for batch interval. This implies the range of acceptable RPO (recovery point objective [13]). Moreover, the batch size threshold can be used to provide QoS. We can fix the ratio of threshold to interval which implies the fixed processing speed, therefore we fix the resource occupation.

3.3 Accelerating Techniques

Data compression/decompression and encryption/decryption put a lot of pressure on CPU. Considering the popularity of multi-core systems, accelerating adaptively cooperative pipelines using parallel techniques is a natural idea. So we design two accelerating approaches: *fine-grained pipelining* and *multi-threaded pipelining*. A combination of these two approaches called *hybrid pipelining* is also considered.

Fine-Grained Pipelining. A common way to accelerate a single pipeline is to deepen the pipeline, that is, breaking down the tasks into smaller units, thus lengthening the pipeline and increasing overlap in execution. However, as mentioned above, for a cooperative pipeline, we can not re-decompose the task arbitrarily. Instead, we must decompose the primary stage and the backup stage into the same number of smaller stages with the same size. It is difficult to decompose the two existing stages identically. We adopt two strategies for this:

- We decompose the two existing stages manually by experience. In our implementation, the primary stage is decomposed into two sub-stages: the *compression stage* containing batch assembling and data compression, and the *encryption stage* containing data encryption and batch sending. The backup stage certainly is also decomposed into two sub-stages: the *computation stage* containing batch receiving, data decryption and decompression, and the *I/O stage* containing disk write and reply. Both primary and backup sites invoke two threads. Each thread is responsible for a sub-stage.

- Obviously, experience only guarantees that the four stages are approximately equal. In order to make them nearer and counter their dynamic change, adaptive batching algorithm is used again. After each stage finishes, the corresponding adaptive formula is applied to adjust the batch interval.

Multi-threaded Pipelining. Another intuitive accelerating approach is to decompose the each existing stage into subtasks in parallel instead of smaller stages. Since each batch contains dozens to hundreds of requests, a simple and effective decomposition technique is data decomposition. In our implementation, each batch is decomposed into two sub-batches with the same size. Both primary and backup sites invoke two threads. Each thread is responsible for a sub-batch. They process the sub-batches in parallel, and then send them in serial for consistency reasons.

Like fine-grained pipelining, multi-threaded pipelining also faces the difficulty in load balancing. Fortunately, the problem is much easier in this method. Note that the computation time of a request is proportional to the number of data bytes in it. So, if the workload contains requests all of the same size (for example, the workload in our experiments generated by Iometer), load balance is guaranteed simply by partitioning sub-batches according to the number of requests. Otherwise, we can partition sub-batches according to the total number of bytes.

Serial network transmission seems to be a drawback of multi-threaded pipelining. However, time spent in this operation is only a small part of the total execution time, thus serial network transmission does not impact the overall performance much. Our experimental results verified this point.

In addition, multi-threaded pipelining is more flexible than fine-grained pipelining. It can even be used to deal with unequal cooperative stages. For example, if the backup stage is twice as long as the primary stage, the backup site can use twice the threads than primary site to equal the execution time of the two stages.

Hybrid Pipelining. Our experimental results showed that neither four-stage pipelining nor double-threaded pipelining fully occupies CPU. Theoretically, deepening the pipeline further or using more threads will make full use of CPU power. However, as mentioned above, very deep pipeline will introduce significant interaction overhead. For the latter strategy, decomposing a batch into too many sub-batches may induce load imbalance and too many threads may increase interaction overhead. So we combine these two techniques. Both primary and backup stages are decomposed into two sub-stages, and each sub-stage is accelerated further by multi-thread technique. We call this method *hybrid pipelining.*

In fact, fine-grained pipelining is an *inter-batch parallelization*, that is, each batch is processed by only one processor at a time, and several batches are processed by multiple processors simultaneously. Multi-threaded pipelining is an *inner-batch parallelization*, that is, batches are processed one-by-one, and each batch is processed by multiple processors simultaneously. Hybrid pipelining is a two-dimensional parallelization.

4 Experimental Evaluation

4.1 Prototype Implementation

We implemented adaptive batching algorithm as a "remote copy" module in Linux LVM2. Like snapshot module in LVM2, this module treats the remote mirror as an attached volume of the original volume. Three accelerating techniques were also implemented. The underlying OS was RedHat AS server 5 (kernel version 2.6.18-128.el5). LVM2 2.02.39, device mapper 1.02.28 and NBD 2.8.8 were used. LZW algorithm [14] was chosen as the compression/decompression algorithm, and AES algorithm [15] was chosen as the encryption/decryption algorithm. A log mechanism was implemented in backup site for batch automatic committing.

4.2 Experimental Setup

All experiments were performed on two single-core 2.66GHz Intel Xeon nodes. One acted as the primary site, and another acted as the backup site. Each machine has 2GB of memory and a hardware RAID-5 composed of six 37GB SAS disks. The two nodes were connected by a Gigabit Ethernet. In order to test three parallel models, we turned off log mechanism in backup site to put enough pressure on CPU. Both α and β were set to 0.5. The batch interval was initialized to 30ms. Iometer [16] was used to generate workload. Since write requests trigger remote mirroring module, we only test write workload. Unless otherwise expressly stated, sequential write workload was used. In order to eliminate the impact of asynchronous mode on performance, we recorded the experimental results after the performance curve reported by Iometer becomes stable over time. Each data point is the average of three samples.

4.3 Experimental Results

We first performed a baseline test. Fig.3 shows the result. "Pri" denotes the RAID-5 devices in the primary site and "back" denotes the RAID-5 devices in the backup site. "LVM" denotes the original volume in the primary site. "Async" denotes the asynchronous remote mirroring system without adaptive batching,

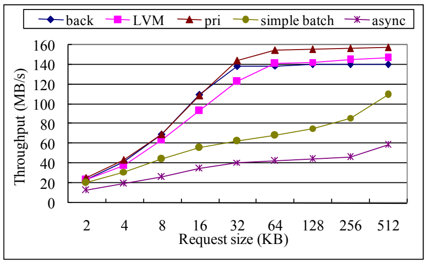


Fig. 3. Baseline test

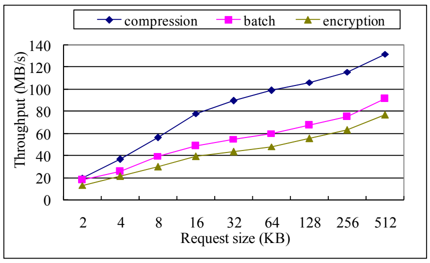


Fig. 4. Computation pressure

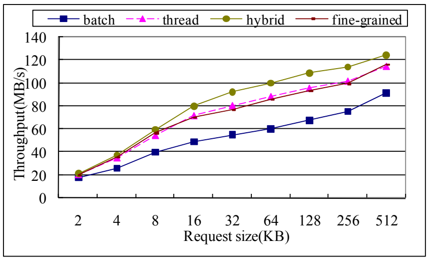


Fig. 5. Parallel models

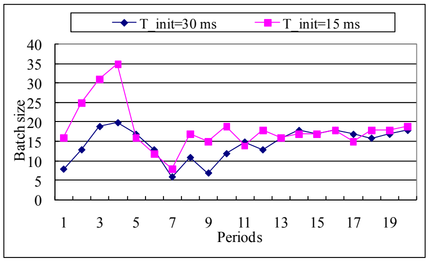


Fig. 6. Batch converging

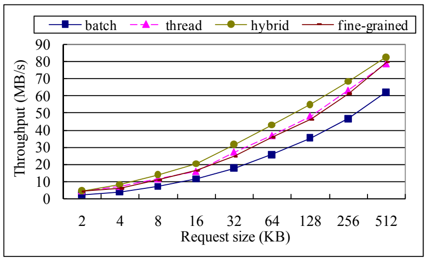


Fig. 7. 50% random write

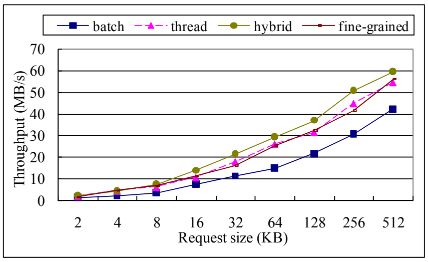


Fig. 8. 100% random write

"simple batch" denotes the one with adaptive batching but without data compression and encryption. We can see that adaptive batching algorithm improves performance greatly though it is far below raw devices yet.

Fig.4 shows the impact of data compression and encryption on performance. "Compression" means with data compression but without encryption, "encryption" means with encryption but without decompression, and "batch" means with both compression and decryption. As we expected, introducing compression improves performance significantly due to network traffic decreasing, and encryption impacts performance seriously due to high computational complexity. The complete version is between the other two versions.

Fig.5 shows how greatly the three parallel models improve performance. We can see that all three models accelerate the pipeline remarkably. Fine-grained pipelining and Multi-threaded pipelining exhibit almost the same performance. Hybrid pipelining improves the performance further because CPU is not overloaded - we observed an about 87% CPU occupation during hybrid pipelining test.

We also traced the batch interval (batch size). The results are showed in Fig.6. The batch interval was initialized to 30ms and 15ms respectively. The request size is 4KB. The batch sizes of the first 20 periods were recorded. We can see that the adaptive batching algorithm indeed coordinates the primary and the backup sites well. The batch interval converged to a stable condition quickly.

We also test random write performance. As Fig.7 and Fig.8 shows, compared with the result of sequential write test, the performance gap between the serial version and the parallel versions narrows. The reason is that the proportion of I/O part in the total running time increases and three parallel models accelerate only the computation part.

5 Conclusion and Future Work

In this paper, we presented a novel cooperative pipelining model for remote mirroring systems. Unlike traditional pipelining models, this new model considers the decentralization of processors. To solve the imbalance between the primary and the backup stages, we proposed an adaptive batching algorithm. To release the heavy load on CPU exerted by compression, encryption and TCP/IP protocol stack, we designed three parallel models: fine-grained pipelining, multi-threaded pipelining and hybrid pipelining. We implemented these techniques in our prototype. The experimental results showed that, the adaptively cooperative pipelining model balances the primary and the backup stages effectively, and the three parallel models improve performance remarkably.

All the experiments were performed in a LAN environment. Testing our prototype in the (emulated) WAN environment is an important future work. FEC (Forward Error Correcting) by using efficient erasure codes is also planned.

References

1. Patterson, R.H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S., Owara, S.: SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002, Monterey, California, USA, January 2002, pp. 117–129 (2002)
2. Weatherspoon, H., Ganesh, L., Marian, T., Balakrishnan, M., Birman, K.: Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance. In: Proceedings of the 7th USENIX Conference on File and Storage Technologies, FAST 2009, San Francisco, California, USA, February 2009, pp. 211–224 (2009)
3. EMC SRDF - Zero Data Loss Solutions for Extended Distance Replication. Technical Report P/N 300-006-714, EMC Corporation (April 2009)
4. VERITAS Volume Replicator (tm) 3.5 Administrator's Guide (Solaris). Technical Report 249505, Symantec Corporation, Mountain View, CA, USA (June 2002)
5. Secure Data Protection With Dot Hills Batch Remote Replication. White Paper, dot Hill Corporation (July 2009)
6. Ji, M., Veitch, A.C., Wilkes, J.: Seneca: Remote Mirroring Done Write. In: Proceedings of the General Track: 2003 USENIX Annual Technical Conference, San Antonio, Texas, USA, pp. 253–268 (June 2003)
7. DFSMS/MVS Version 1 Remote Copy Administrator's Guide and Reference 4th edition. Technical Report SC35-0169-03, IBM Corporation (December 1997)
8. HP OpenView continuous access storage appliance. White Paper, Hewlett-Packard Company (November 2002)
9. Liu, X., Niv, G., Shenoy, P.J., Ramakrishnan, K.K., van der Merwe, J.E.: The Case for Semantic Aware Remote Replication. In: Proceedings of the 2006 ACM Workshop on Storage Security and Survivability, StorageSS 2006, Alexandria, VA, USA, October 2006, pp. 79–84 (2006)

10. LVM, http://sources.redhat.com/lvm/
11. Breuer, P.T., Lopez, A.M., Ares, A.G.: The Network Block Device. Linux Journal 2000(73), 40 (2000)
12. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Essex (2003)
13. Keeton, K., Santos, C.A., Beyer, D., Chase, J.S., Wilkes, J.: Designing for Disasters. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, FAST 2004, San Francisco, California, USA, March 2004, pp. 59–72 (2004)
14. Welch, T.A.: A Technique for High-Performance Data Compression. IEEE Computer 17(6), 8–19 (1984)
15. NIST Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (2001)
16. Iometer, http://www.iometer.org/

SV: Enhancing SIMD Architectures via Combined SIMD-Vector Approach

Libo Huang and Zhiying Wang

School of Computer, National University of Defense Technology,
Changsha 410073, China
{libohuang,zywang}@nudt.edu.cn

Abstract. SIMD architectures are ubiquitous in general purpose and embedded processors to achieve future multimedia performance goals. However, limited to on chip resources and off-chip memory bandwidth, current SIMD extension only works on short sets of SIMD elements. This leads to large parallelization overhead for small loops in multimedia applications such as loop handling and address generation. This paper presents SIMD-Vector (SV) architecture to enhance SIMD parallelism exploration. It attempts to gain the benefits of both SIMD instructions and more traditional vector instructions which work on numerous values. Several instructions are extended that allows the programmer to work on large vectors of data and those large vectors are executed on a smaller SIMD hardware by a loop controller. To preserve the register file size for holding much longer vectors, we introduce a technique that the long vector references are performed on only one SIMD register in many iterations. We provide a detailed description of the SV architecture and its comparison with traditional vector architecture. We also present a quantitative analysis of the dynamic instruction size decrease and performance improvement of SV architecture.

Keywords: SIMD, vector, performance, loop.

1 Introduction

In the past few years, technology drivers changed significantly. Multimedia application has become the mainstream computing in driving microprocessor technology instead of scientific applications alone. In this situation, an increasingly popular way to provide more performance is through data level parallelism (DLP). This kind of parallelism performs operations in a single instruction multiple data (SIMD) fashion. The reason behind this is multimedia applications contain a lot of inherent parallelism which can easily be exploited by using SIMD instructions at low cost and energy overheads. Adopting it into general purpose processors results in SIMD extension architectures [1]. There are many examples of SIMD extension such as Intel's SSE, AMD's 3DNow!, IBM's VMX and Motorola's AltiVec.

Despite of their superior theoretic performance for multimedia applications, full potential of SIMD accelerator are not exploited effectively. Compared with conventional vector computer, one of the constraints for current SIMD architectures is that due to hardware cost, the supported vector are short, which is very small amount of datapath work to represent with each instruction. This forces the use of a complex superscalar instruction issue mechanism to keep an array of functional units busy. Furthermore, short vector implementations introduce large parallelization overhead such as loop handling and address generation which is not well supported by current SIMD architectures [2].

Therefore, we introduce hybrid SIMD-Vector (SV) architecture, which combines the main features of the two architectures, supporting vector processing with variable lengths. It uses shorter SIMD unit (i.e. 4-way SIMD) to do computations on longer vectors (i.e. 16-way SIMD) by loops. This variable length vector interface brings many advantages to the processor architecture: 1) each SIMD instruction can be assigned to do more computations which reduces the SIMD code size; 2) Conventional loop controlling instructions are transformed into the internal states of SIMD unit, thus eliminating large overhead instructions introduced by loop controlling. 3) using shorter SIMD unit allows the hardware cost to a feasible range as SIMD units consume much die area; 4) the memory bandwidth can be effectively utilized especially for emerging processor architecture with on chip memory [3]. The SV architecture also possesses many optimization opportunities due to its new vector interface. We employ a stream-like memory system which can overlap the computation and memory access leading to efficient utilization of SIMD unit.

Compared with assembly programming of wide out-of-order superscalar architectures, or tightly-coupled multiprocessors, or VLIW machines with exposed functional unit latencies, SV architecture offer a compact, predictable, single-threaded assembly programming model with loop unrolling performed automatically in hardware. Moreover, scheduled vector object code can directly benefit from new implementations with greater parallelism.

The main contribution of this paper is proposing and evaluating a new architecture for SIMD extension which brings new levels of performance and power efficiency. The remainder of this paper is organized as follows. In Section 2, the motivations of this work is introduced. Section 3 provides detailed description of the proposed SV architecture and its comparison with conventional vector architecture. Then the evaluation results are presented in Section 4. Finally, Section 5 gives the conclusion of the whole work.

2 Motivation

While the SIMD extension and vector architectures have many similarities, they differ principally in implementation of instruction control and communication between memory and the execution units. Figure 1 highlights the differences between these two DLP architectures. For SIMD extensions, vector length is short, instruction latency is roughly one cycle per instruction, strided memory accesses are not supported and memory accesses are usually required to be aligned. The vector processors can overlap load, computation, and store operations of vector elements by pipelining technology. So the vector length can be very long and variable but the instruction latency is roughly more longer with one cycle per vector element. This kind of parallelism is called *vertical parallelism*. On the contrary, the SIMD extension architecture use many duplicated function units to perform parallel operations on all the elements in vectors, which called *horizontal parallelism*. As the number of functional units can not be extensible due to the hardware cost, the vector length of SIMD extension is fixed and short.

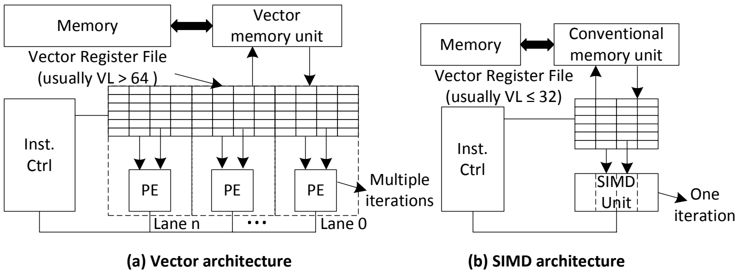


Fig. 1. Vector and SIMD architectures

Many advantages can be obtained from long vector. Firstly, it is very compact, which decrease the code size. Secondly, it is very expressive, which can specify a large number of operations and decrease the dynamic instruction count. Thirdly, it is scalable, which can benefit from the new hardware implementation and does not introduce binary compatibility problem. So adopting long vector into SIMD extension will result in an elegant solution. Figure 2 illustrates the advantage of long vector interface for the motivating example. It can be seen that when the supported vector length is longer than the number of iteration, the loop overhead instructions can be totally eliminated leading to high performance.

However, some barriers need to be conquered when introducing long vector processing into SIMD extension architecture: 1) the bandwidth of memory system in general purpose processor is limited for long vector operation; 2) due to

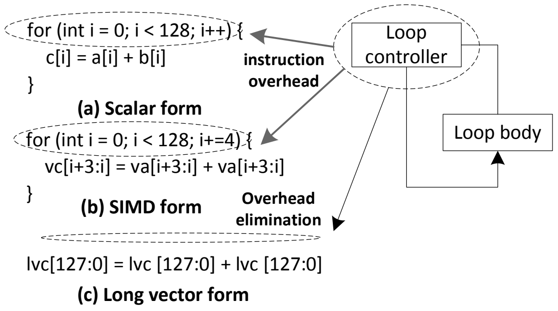

for (int i = 0; i < 128; i++) {
c[i] = a[i] + b[i]
}
(a) Scalar form
for (int i = 0; i < 128; i+=4) {
vc[i+3:i] = va[i+3:i] + va[i+3:i]
}
(b) SIMD form
lvc[127:0] = lvc [127:0] + lvc [127:0]
(c) Long vector form
instruction overhead
Overhead elimination
Loop controller
Loop body

Fig. 2. Motivation example: eliminating the loop control overhead

the die size, the vector length that vector register file can hold and SIMD unit can handle is short; 3) not all the multimedia application can benefit from long vector interface. This occurs when vectors in applications are shorter than the supported length. So how to take advantage of long vector interface is still an open issue.

This paper presents the SIMD-Vector architecture to address this challenge. It preserves the original datapath of SIMD architectures and adds efficient support for long vector memory accesses and computations without changing existing ISA. The SV architecture combines the advantages of SIMD extension and vector architectures, which can further exploit the data-level parallelism.

3 SIMD-Vector Architecture

In this section we describe SV architecture in detail, looking first at its architecture design, and then describing the new extended instructions and its programming, finally discussing the comparison with other DLP architectures.

3.1 Architecture Overview

Some parameters for describing SV architecture are listed in Table 1. Generally, the bit sizes of SIMD unit (S_S), VRF (S_R) and memory unit (S_M) are the same to obtain data bandwidth balance of the whole system. In the example, 128-bit are configured, which is popular in many SIMD extensions such as IBM's AltiVec ISA [4], ARM's Neon SIMD ISA [5] and Intel's SSE ISA. As the supported S_E is variable, so one vector register (VR) may have different register lengths. For example, the 128-bit VR can contains 4 32-bit word, 8 16-bit halfword, or 16 8-bit byte, so L_R could be 4, 8, or 16. The vector length L_V is also variable, which is specified by program.

Table 1. Architectural parameters

Factors	Description	Example
S_S	Bit size of SIMD unit	128
S_R	Bit size of vector register file	128
S_M	Bit size of load/store unit	128
S_E	Bit size of vector element	8,16,32
L_V	Vector length	variable
L_R	Register length (S_R/S_E)	4,8,16

Naive Structure. To support variable vector length, a straightforward approach is using iterative algorithm like in conventional vector architectures. Figure 3(a) shows the block diagram of naive SV structure. It preserves current memory system, and there is virtually no performance penalty for mingling scalar integer, FPU and SIMD operations. The SIMD unit is the functional unit to perform SIMD operations. It remains the same as conventional SIMD unit.

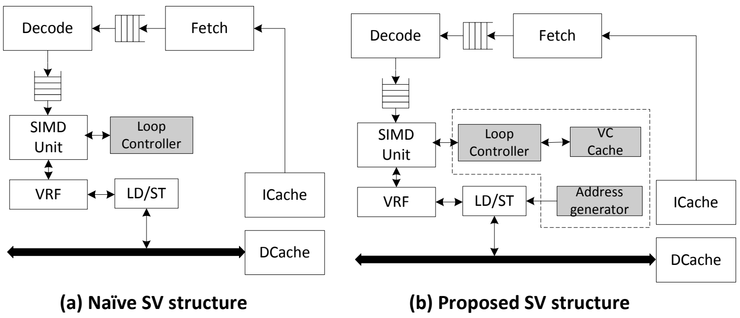


Fig. 3. SIMD-Vector Architecture

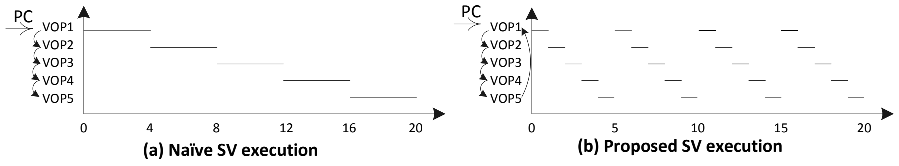


Fig. 4. Execution mechanisms for SV architecture

The modified part is presented in gray color. Only one component called loop controller is added. It generates control signals for loops to complete long vector operations. To eliminate the loop overhead, the configured LV should be equal to the numbers of iterations in the loop. When LR can not divide LV, then the SIMD execution times would be $\lceil LV/LR \rceil$.

Figure 4(a) shows the execution mechanism for naive SV structure. It completes operations for all the elements in long vector by loops before doing the next long vector operations. To store the long vector in VRF, a efficient way is to using several short VRs to represent long vectors. This implementation of long vector interface potentially gives many optimization opportunities. For example, we can select only the true operations as the SIMD element operations and skips over the false operations similar to vector architecture. This allows the SIMD unit to do more useful operations and save the unnecessary power consumption. Though the naive execution mechanism is simple to implement in existing SIMD architecture, it consumes large number of VRs and gives some limitations to the vector length of long vectors. Furthermore, it makes the long vector memory instructions can not be executed parallel with computation instruction which would degrades the performance.

Proposed Structure. To overlap the memory accesses with computations, a better method using vertical execution mechanism to support long vector operations as shown in Figure 4(b). For a given set of long vector operations, each time the SV architecture executes them with the length of SIMD unit vertically, and then execute another portion of long vector operations in a similar fashion.

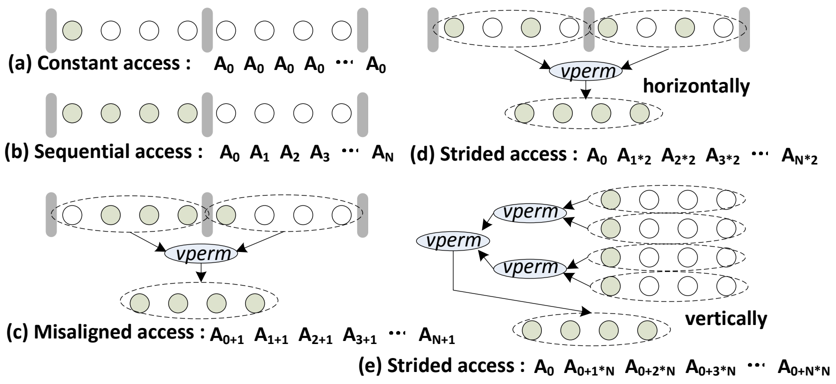

(a) Constant access : A0 A0 A0 A0 ··· A0
(b) Sequential access : A0 A1 A2 A3 ··· AN
(c) Misaligned access : A0+1 A1+1 A2+1 A3+1 ··· AN+1
(d) Strided access : A0 A1*2 A2*2 A3*2 ··· AN*2
(e) Strided access : A0 A0+1*N A0+2*N A0+3*N ··· A0+N*N
vperm
horizontally
vertically

Fig. 5. Memory access patterns

This type of execution mechanism offers a natural way to overlap the memory accesses with computations. Vertical execution also does not need extra VRs to represent long vector since each time the element values are loaded into the same VRs. So there is no length limitation for long vectors in vertical execution mechanism. In order to store the set of long vector operations, a small vector code cache (VCCache) is required. We restrict the capacity of VCCache to 1KB, which is 256 operations for 32-bit instruction encoding. This is fit for most of the key loops in multimedia applications.

The resulting block diagram of proposed SV structure is shown in Figure 3(b). The modified hardware consists of loop controller, vector code cache (VCCache) and address generator. The loop controller generates control signals for loops to complete long vector operations based on a loop granularity. The VCCache is used to store the instructions for vertical execution. Since it mostly used for inner loops, so the size could be not very large. The VRF has the same structure with conventional one, possibly with larger number of entries.

It is critical to provide adequate memory bandwidth for long vector operations. Since the long vector memory access is totally use original memory system, the conventional vector memory access can not be provided by the existing hardware. However, conventional SIMD address generation consumes many supporting instructions [6]. To decrease this kind of overhead instructions, automatic address generation for load and store operations are needed. This is achieved by the address generator connected to load/store unit. We define four types of memory access commonly used in multimedia applications as shown in Figure 5: constant access, sequential access, misaligned sequential access and strided access [6]. They are not exclusive and a loop may take accesses from all four categories. Supporting them using general-purpose instruction sets is not very efficient, as the available addressing modes are limited. Furthermore, there is not enough support for keeping track of multiple strides efficiently in general purpose processors. So in SV architecture, only sequential access is supported, but other three types of memory accesses can be efficiently accomplished by the

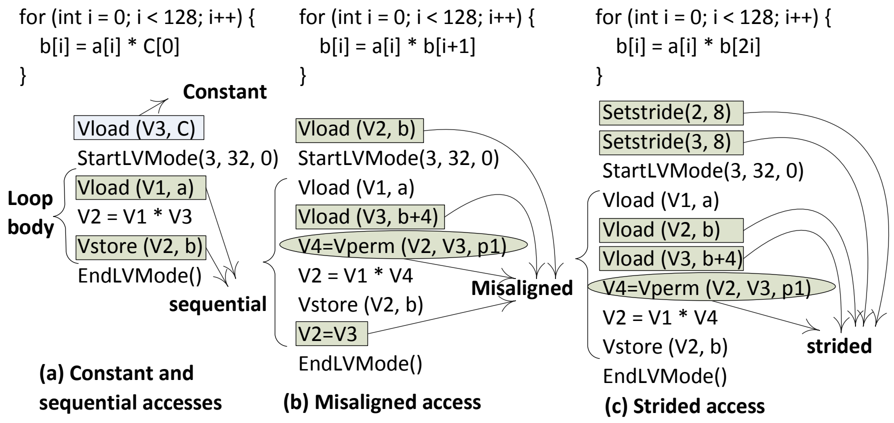
Table 2. Extended instructions

Instructions	Description
StartLVMode(mode, LV, VR)	Start SV execution with configured mode
EndLVMode()	Stop SV execution
SetStride(offset,stride)	Set stride of strided access

sequential access via permutation instructions exists in any SIMD ISA [1]. For example, *vperm* operation in AltiVec provides extremely flexible data permutation functionality, which can combine elements of two source registers in any permutation pattern [4].

3.2 Programming Issues

Extended Instructions. Provided the previous hardware support, Table 2 defines new SV interface instructions for configuration. Three instructions are defined. The *StartLVMode* instruction is used to set the internal execution states for loop controller. It has three source parameters: The 1-bit *mode* specifies the loop condition. When '1', then the vector length is used to guard the loop controller, which is specified by the second parameter, 16-bit *LV*. Otherwise, false value of register is used to stop the loop execution, which is specified by the third parameter *VR*. The *EndLVMode* instruction is used to end the SV execution. From the definition of extended instructions, we can see that SV execution needs configuration first, and later vector computations can be proceeded in the configured way. So any long vector instructions should be placed between the instruction pair *StartLVMode* and *EndLVMode*. The *SetStride* instruction is used to set the strided memory access. It has two source parameters: *offset* specifies the offset of instruction in vector code, *stride* specifies its stride. This instruction is very useful for strided memory access.



```
for (int i = 0; i < 128; i++) {
   b[i] = a[i] * C[0]
}
Vload (V3, C)
StartLVMode(3, 32, 0)
Vload (V1, a)
V2 = V1 * V3
Vstore (V2, b)
EndLVMode()
```

```
for (int i = 0; i < 128; i++) {
   b[i] = a[i] * b[i+1]
}
Vload (V2, b)
StartLVMode(3, 32, 0)
Vload (V1, a)
Vload (V3, b+4)
V4=Vperm (V2, V3, p1)
V2 = V1 * V4
Vstore (V2, b)
V2=V3
EndLVMode()
```

```
for (int i = 0; i < 128; i++) {
   b[i] = a[i] * b[2i]
}
Setstride(2, 8)
Setstride(3, 8)
StartLVMode(3, 32, 0)
Vload (V1, a)
Vload (V2, b)
Vload (V3, b+4)
V4=Vperm (V2, V3, p1)
V2 = V1 * V4
Vstore (V2, b)
EndLVMode()
```

Fig. 6. SIMD-Vector programming example

SIMD-Vector Programming. The main task of SV programming is to determine the loop body and its execution mode. Three examples are given to show the programming of SV architecture as shown in Figure 6. They represent different memory access pattern discussed previously. We assume that S_S=S_R=S_M=128 and S_E=32. Figure 6(a) shows the constant and sequential memory access. The constant access can be accomplished before loop which would stored as a register inside loop. The sequential memory access is the default memory access pattern. Only two instructions *StartLVMode* and *EndLVMode* are added to configure the execution mode. This leaves the SIMD unit to perform more actual computations. Figure 6(b) shows the misaligned memory access. Through *vperm* instruction, it can be well supported. Special case with strided access is shown in Figure 6(c), where *SetStride* should be used to configure the stride different from default sequential access.

3.3 Comparison with Other DLP Architectures

The SV architecture tries to combine the good features of SIMD and vector architectures. Table 3 shows the comparison result of these three architectures. We can see that compared to vector architecture, including the vector architecture implemented in some recent single chip processors [9, 10, 11], the width of SV VRF is much smaller. This makes the long vector support feasible. The SV architecture also has advantages over SIMD architecture, which support efficient long vector operation and SIMD address generation.

Table 3. Architecture comparison

Features	SV	SIMD extension	Vector
Vector length	any	$\leqslant 32$	> 64
Memory access	automatic address gen.	aligned access	strided access
Instruction latency	~1 cycle/SIMD elements	~1 cycle/instuction	~1 cycle/element
Data Parallelism	Combined	Vertical parallelism	Horizontal parallelism
VRF width	$= S_s$	$= S_s$	$= VL$

4 Evaluation

To test the SV architecture, we designed a multimedia processor based on the TTA (Transport Triggered Architecture) [7]. It has includes all the techniques discussed above. It has 128-bit wide SIMD datapath with four 32-bit vector elements. The TTA-based architecture makes the processor can integrate abundant arithmetic functional units in a very simple way, and gives flexible and fine-grained exploration of SIMD parallelism for compiler. It has four SIMD ALU, MAC and FPU units seperately. The processor is utilized for accelerating core processing of multimedia applications like data compression and graphics functions.

We study only kernels as opposed to entire applications. This is because they represent a major portion of many multimedia applications. All the benchmarks [8] are shown in Table 4. They ranging from fairly simple algorithms like

Table 4. Benchmark Description

Name	Description
FFT	16-bit fast fourier transform
MATMUL	16-bit matrix multiplication
MAXIDX	16-bit index of max value in array
IDCT	8-bit IDCT
FIR	16-bit complex finite impulse response filter
IIR	16-bit infinite impulse response filter

MAXIDX and *MATMUL* to more complex digital signal processing algorithms like *FFT*.

Figure 7 shows the dynamic instruction count comparison result. Three code versions are provided: scalar, conventional SIMD and proposed SV. We observe that, compared to conventional SIMD version, the proposed SV architecture can reduce the number of dynamic instructions by 16.8% on average. This is due to that SV architecture can overlap different iterations of one loop body and execute *SIZE-1* copies of the loop automatically, so *SIZE-1* copies of the loop handling and memory address generation overhead are eliminated, and only the start and the exist instructions of the loop body are retained.

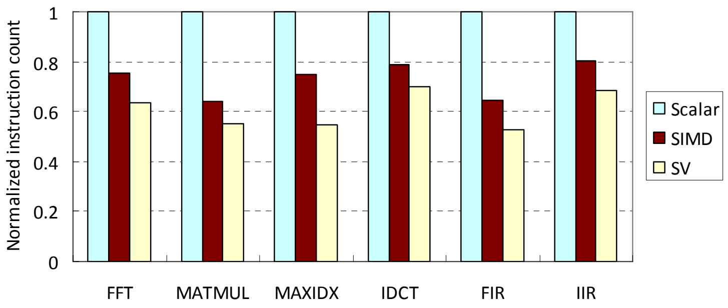


Fig. 7. Dynamic instruction count comparison

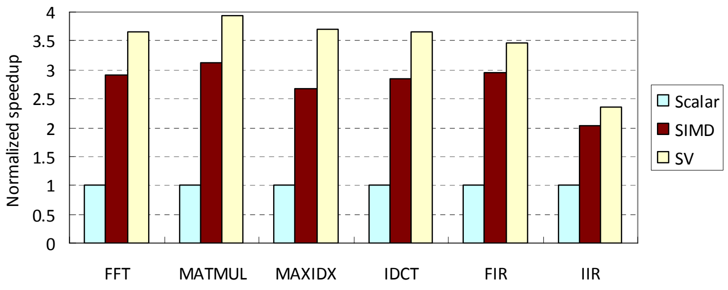


Fig. 8. Speedup over scalar code comparison

Figure 8 displays the obtained speedups for the different benchmarks. All results show SV architecture has a significant speedup over conventional SIMD which ranges from 1.15 to 1.38. Compared to scalar codes, most of these benchmarks saw a speedup of five to ten, with exceptional results for *IDCT*. This is because only 8-bit data are handled for *IDCT*. However, in all cases the theoretical limit is not reached, due to overhead introduced by packing and unpacking the data.

5 Conclusion and Future Work

This paper presents SV architecture for exploiting data-level parallelism. By leveraging variable vector lengths, much of the loop overhead is eliminated and developers can see dramatic acceleration in the benchmarks. For the future it might be interesting to automatically generate SIMD code in a compiler for SV architecture. Furthermore, it would be interesting to integrate hardware enhancement for packing and unpacking of data into SV architecture, which could improve performance.

Acknowledgements

We gratefully acknowledge the supports of the National Basic Research Program of China under the project number 2007CB310901 and the National Natural Science Foundation of China under Grant No. 60803041 and No. 60773024.

References

1. Lee, R.: Multimedia Extensions for General-purpose Processors. In: SIPS 1997, pp. 9–23 (1997)
2. Shin, J., Hall, M.W., Chame, J.: Superword-Level Parallelism in the Presence of Control Flow. In: CGO 2005, pp. 165–175 (2005)
3. Patterson, D., et al.: A Case for Intelligent RAM: IRAM. IEEE Micro 1997 17(2), 33–44 (1997)
4. Diefendorff, K., et al.: Altivec Extension to PowerPC Accelerates Media Processing. IEEE Micro 2000 20(2), 85–95 (2000)
5. Baron, M.: Cortex-A8: High speed, low power. Microprocessor Report 11(14), 1–6 (2005)
6. Talla, D.: Architectural techniques to accelerate multimedia applications on general-purpose processors, Ph.D. Thesis, The University of Texas at Austin (2001)
7. Zivkovic, V.A., et al.: Design and Test Space Exploration of Transport-Triggered Architectures, pp. 146–152 (2000)
8. TMS320C64x DSP Library Programmer's Reference, Texas Instruments Inc. (2002)
9. Corbal, J., Espasa, R., Valero, M.: Exploiting a New Level of DLP in Multimedia Applications. In: MICRO 1999 (1999)
10. Kozyrakis, C.E., Patterson, D.A.: Scalable Vector Processors for Embedded Systems. IEEE Micro 23(6), 36–45 (2003)
11. El-Mahdy, A., Watson, I.: A Two Dimensional Vector Architecture for Multimedia. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, p. 687. Springer, Heidelberg (2001)

A Correlation-Aware Prefetching Strategy for Object-Based File System[⋆]

Julei Sui, Jiancong Tong, Gang Wang, and Xiaoguang Liu

Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University
94 Weijin Road, Tianjin 300071, China
{nkujulei,lingfenghx}@gmail.com, wgzwp@163.com, liuxg74@yahoo.com.cn

Abstract. The prefetching strategies used in modern distributed storage systems generally are based on temporal and/or spatial locality of requests. Due to the special properties of object-based storage systems, however, the traditional tactics are almost incompetent for the job. This paper presents a new prefetching approach, which takes the correlationship among objects into account. Two orthogonal replica distribution algorithms are proposed to aggregate prefetching operations. A moving window mechanism is also developed to control prefetching. We implement these approaches in our object-based file system called NBJLOFS (abbreviated for Nankai-Baidu Joint Lab Object-based File System). The experimental results show that these approaches improves throughput by up to 80%.

Keywords: object-based storage, prefetching, object duplication, orthogonal layout.

1 Introduction

With the continuous growth of storage device capacity and the rapid development of applications, traditional block-based storage systems can no longer meets the demand. Object-based storage technology emerged and has attracted increasing attention. Generally, "object-based" means that the basic storage unit is object instead of block. An object is a combination of file data and a set of attributes [1]. In a distributed object-based file system, a file may be divided into many objects, which are distributed over several storage nodes.

In modern computer systems, I/O time often dominates the total running time. Cache and prefetching technologies are the standard way to alleviate the performance gap between disk and main memory/CPU. Traditional prefetching strategies typically read additional blocks physically adjacent to the current read request. That is, they assume that logically adjacent blocks are also

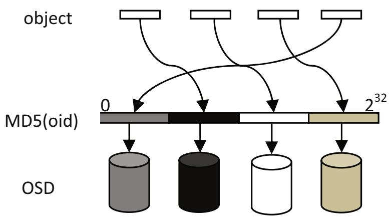
⋆ Supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), SRFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCY-BJC13000).

physically adjacent. However, these strategies do not work in distributed object-based storage systems because objects are distributed over many storage nodes. Moreover, these strategies can not deal with prefetching based on other relationship among objects. This paper puts forward an innovative prefetching strategy taking objects' correlation into account. It gains "prefetching aggregating" by an orthogonal object duplication layout. The experimental results show that this new strategy increases throughput and decreases network traffic.

The rest of this paper is organized as follows. In Section 2, we briefly describe NBJLOFS and namespace splitting algorithm. In Section 3, we introduce our prefetching approach together with details of our implementation. In section 4, we present the experimental results. Section 5 discusses related work and Section 6 concludes the paper.

2 NBJLOFS

NBJLOFS is a distributed object-based file system based on IP-SAN [2]. It employs FUSE [3] as filesystem interface and Berkeley DB [4] as storage infrastructure. Every file in NBJLOFS is split into objects. Each object is uniquely represented by a quintuple (fnso, fno, offset, type, flag) called object identifier (OID). NBJLOFS has no metadata servers. The clients provide a standard file system interface to the users, and the storage nodes (called Object-based Storage Devices, OSDs for short) are in charge of object storing and accessing. When a file is stored, the system splits it into fixed-size objects and distributes them over OSDs. When a read request arrives, the client node dispatches it to the proper OSD. The OSD will acknowledge the client with the required object and maintains a copy in its local cache. Objects are distributed according to a namespace splitting algorithm. As shown in Fig. 1, a 32 bits identifier space is divided into several segments. Each segment corresponds to a unique OSD. Every object is mapped to a value in the identifier space using MD5 algorithm (Message-digest Algorithm 5). Then the object is assigned to the corresponding OSD.



```
Procedure map_object(object_id , osd_num)
    x ← MD5(object_id) & 0xffffffff
    region ← ( uint32_t )( ~0 )
    interval_span ← region / osd_num
    position ← x / interval_span
    Distribute the object to osd[position]
end procedure
```

Fig. 1. Namespace splitting algorithm

3 Correlation-Aware Prefetching

In a Distributed Object-based File System, objects are dispersed. The objects stored on a single OSD may not be logically consecutive parts of a single file according to the object distributed algorithm mentioned above. So the traditional prefetching strategy that reads additional disk blocks adjacent to the current request no longer works. Instead, we should fetch logically related objects from different OSDs. Since objects are distributed over OSDs, inevitably, lots of prefetching requests are launched to many OSDs to prefetch a batch of logically adjacency objects. Therefore, as the system size increases, the network traffic will dramatically increases. In order to solve this problem, this paper presents a "file-centralized" duplication strategy. This strategy aggregates the replicas belonging to the same file, so only one request is need to be issued for a prefetching operation.

3.1 Object Duplication

We do not create mirror for each OSD like RAID-1 [5]. Instead, we adopt object-oriented duplication. Moreover, we aggregate replicas according to their correlationships. In this paper, we consider the "file-belonging" correlationship, that is, replicas with the same "fno" field in OID are aggregated. Note that we can easily aggregate replicas using other correlationships.

For each object, we make and maintain a replica for it and guarantee that these two copies of the object are stored on different OSDs. For clarity, we call them the *original object* and the *replica object* respectively. The former is assigned to a OSD using the namespace splitting algorithm mentioned in Section 2. The replica is assigned to the OSD determined by the MD5 digest of its file identifier ("fno"). As objects belonged to the same file share the same value in field "fno", the replica objects of a file aggregated in a specific OSD and the original objects are distributed over other OSDs. For a certain file, we call the specific OSD where all its replica objects aggregated as its *replica OSD* or *replica node*, and other OSDs as its *original OSDs* or *original nodes*.

Fig. 2 shows an example of orthogonal distribution. We use Xi denote the i-th original object of file X and Xi' denote the replica. In this example, there are four files A, B, C and D. Note that in NBJLOFS the objects are not really indexed in this way, the indices are just used to illustrate this problem simply. We can see that all replicas of file A are assigned to OSD1, and its original objects are distributed over other OSDs. From Fig. 2, we can find that the replica objects of a certain file didn't reside with any of its original objects in the same OSD.

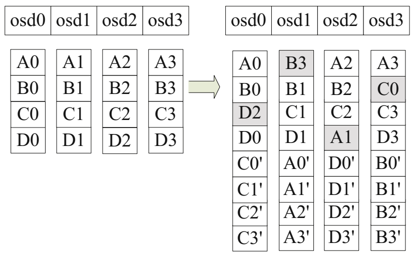

osd0
osd1
osd2
osd3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
A0 B0 D2 D0 C0' C1' C2' C3'
B3 B1 C1 D1 A0' A1' A2' A3'
A2 B2 C2 A1 D0' D1' D2' D3'
A3 C0 C3 D3 B0' B1' B2' B3'

Fig. 2. Orthogonal distribution

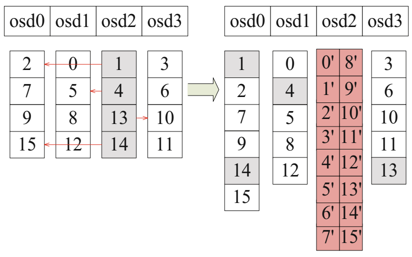


Fig. 3. Migration strategy

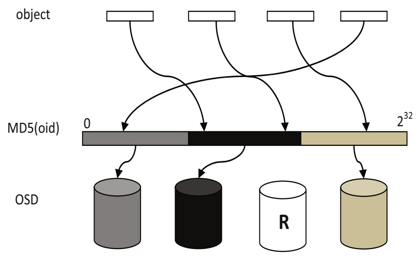


Fig. 4. Repartition namespace

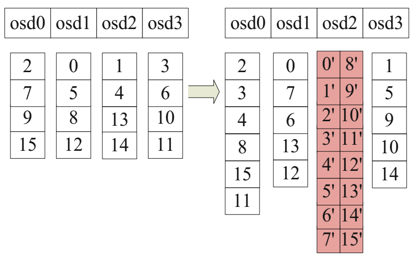


Fig. 5. Repartition strategy

We call this way of distribution as *orthogonal layout*. The original objects and replicas of other files are distributed in a similar way. Nevertheless, guaranteeing orthogonal distribution is not straightforward. We can see that, compared with the layout without duplication, some objects must be redistributed to avoid conflicting with their replicas. We extend namespace splitting algorithm in two ways for this purpose.

Migration Strategy. This strategy does not remap all original objects. Instead, it just redistributes those original objects on the replica node at the very start. Fig. 3 illustrates this strategy. The replica node is OSD2 in this example. We can see that the objects originally stored on ODS2, i.e., 1, 4, 13, 14 are migrated to other OSDs.

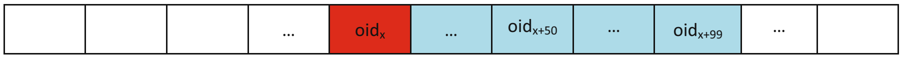
Repartition Strategy. This strategy repartitions the 32 bits identifier namespace into several segments. As shown in Fig. 4, the number of segments is just one less than the number of OSDs. For each file, each segment corresponds to one of its original nodes. By applying the namespace splitting algorithm, each original object is mapped to a unique segment. Then the original object is assigned to the corresponding original OSD. And all the replica objects are stored in its exclusive replica node(mark as R in Fig. 4). The replica node of the file is not involved in the process of namespace splitting. In other words, the replica OSD is simply omitted when we distribute the original objects. Therefore original-replica conflicts never occur. Fig. 5 illustrates this strategy.

At first glance, it seems that there is a serious load-imbalance among OSDs because of replica objects' aggregation and original objects' dispersion of a single file. However, from the perspective of the whole system, since there are huge number of files, the load is essentially balanced. For each OSD, it plays not only the role of the replica node of some files but also the role of a original node of many other files.

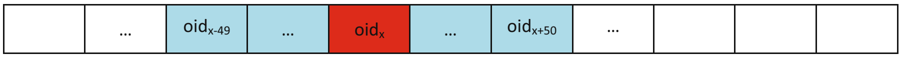
Our experimental results show that both repartition and migration strategy distributes original and replica objects over OSDs evenly. However, if we create a duplication for an existing single-copy system, the repartition strategy must redistribute all objects, while the migration strategy only redistributes the objects on their replica nodes. So, we select the latter as our distribution algorithm.

3.2 Moving Windows

Prefetching excessively or incorrectly certainly could be helpless, even harmful to system efficiency. In order to maximize the virtue of prefetching, we introduce the dynamic window mechanism. We treat the batch of to-be-prefetched objects as a window, and denote the number of objects in the window by *window_size*. This idea comes from the moving window concept in TCP/IP protocol. The window extent will be dynamically changed according to the on the fly object rather than keep stationary all the time. Here are also two alternative strategies: *forward window*, which only prefetches the objects that following the current request, and *wing window*, which prefetches both the previous and the following objects. These two strategies have the same window_size. Fig. 6(a) and Fig. 6(b) illustrate the two strategies respectively, where the dark areas are prefetching windows. Since spatial locality involves both forwards and backwards, we select the wing window strategy as the moving windows mechanism of NBJLOFS. Both "wings" have the same length, that called the *wing_length*. This implies that the wing_length is equal to half of the value of window_size.



(a) forward window



(b) wing window

Fig. 6. Two moving window strategies($window\_size = 100$)

In NBJLOFS, the replica node of a file maintains a unique wing window for each client that accesses this file, i.e., we have not implemented data sharing window. Each client maintains a wing window for every file it accesses. The windows in replica nodes and clients are always consistent. Any time a window in the OSD changed, it notifies the corresponding client to synchronize. Objects in the same file can be identified by their in-file offsets. We call the difference of two objects' offsets *distance*. For each window, we call the central object *pivot*. For example, the oid_x in Fig. 6(b) is the pivot.

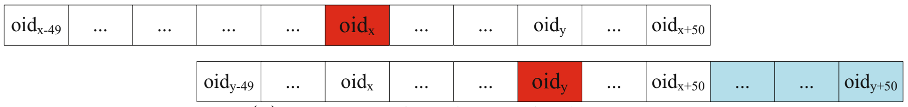
On the client side, whenever an object is accessed, NBJLOFS will determine whether it is within the wing window extent or not by comparing the wing_length with the distance between the pivot and the demanded object. If the latter is numerically smaller, which means that the object has been already prefetched from disk in the replica node, the client node will then send a request to the replica node, and correspondingly moves the window when reply is received. On the contrary, the client node will send a request to the original node.

Whenever an original OSD receive an access request, it implies that the required object has not been prefetched. The original OSD will read the required object from disk, send it back to the client and issue a prefetching request to the replica node.

On the replica node, if a request arrives, there are two cases.

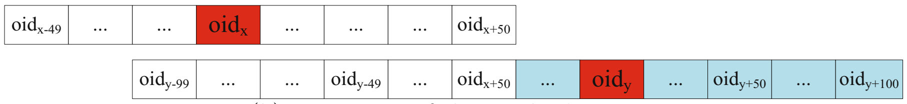
i) The request is received from a client node. This implies that the required object is within the window extent, which means that the previous prefetching operation works. So the window_size will remain unchanged. The required object will be sent to the client node. This object is chosen as the new pivot, and the difference between the new and the old windows, i.e., the objects within the new window but out of the old window, will be prefetched from disk. Fig. 7(a) shows the change of the window, where the dark area denotes the newly prefetched objects.

ii) Otherwise, the request is received from an original node. This implies that the distance is larger than the wing_length, showing that the current window is not wide enough. So the replica node doubles the window_size. In our implementation, the initial value of window_size is 1, and its maximum is limited to 40 objects (amount to 5MB data). Similar with case i, the required object is chosen as the new pivot and the difference of the new and the old window is prefetched from disk. Fig. 7(b) shows this case. In addition, the replica node will inform the related client to synchronize the window.



oid_{x-49} ... ... oid_x ... ... oid_y ... oid_{x+50}
oid_{y-49} ... oid_x ... ... oid_y ... oid_{x+50} ... ... oid_{y+50}

(a) case 1: within the window's range



oid_{x-49} ... ... oid_x ... ... oid_{x+50}
oid_{y-99} ... ... oid_{y-49} ... oid_{x+50} ... oid_y ... oid_{y+50} ... oid_{y+100}

(b) case 2: out of the window's range

Fig. 7. Two window changing cases

3.3 Analysis

In our prototype, a traditional spatial locality based prefetching approach was also implemented for comparison. It is very similar to the read-ahead strategy in Linux kernel. We call it SPA. Now we analyze this algorithm and our correlation-aware prefetching algorithm (COR for short).

Assume that the window size is W ($W \geq 1$) and the number of OSDs is N ($N > 1$). For simplicity, assume that $N > W$. When an OSD receives an access request from client, it should prefetch W objects. For SPA, since objects are not duplicated, the OSD has to issue W prefetching requests to other W OSDs assuming that the to-be-prefetched objects are evenly distributed. In contrast, COR issues only one prefetching request to the replica node. In a single-user system, COR may have no advantage over SPA if only a single file is accessed simultaneously. After all, SPA also offers timely prefetching. However, if the two algorithms are deployed in a multi-user system and many files are accessed simultaneously, COR will show its superiority. Since it induces lighter network traffic than SPA, there would be less occurrence of network saturation. Moreover, since COR stores all replicas of a file in a single OSD and just these replicas are prefetched, disk operations induced by prefetching requests are more likely to be large sequential read operations. In contrast, disk operations in SPA system are all small discontinuous (random) read operations. Our experimental results show that these two advantages of COR lead to significant performance advantage over SPA.

4 Experiment

4.1 Experimental Environment

All experiments were performed on a cluster of seven single-core 3.06GHz Intel Celeron nodes. Each machine has 512MB of memory and a 80GB hard disk. Each node runs Red Hat Enterprise Linux AS 4. All the nodes are connected by a Gigabit Ethernet. One of them acts as the client node, while other six nodes act as OSDs.

4.2 Performance Evaluation

We used a modified Bonnie to generate sequential and random workload. In each test, we first wrote five 100MB files, then tested sequential or random read performance. Each data point is the average of 5 runs. Each file was composed of 800 objects, that is, the object size was 128KB. The request generated by Bonnie is of size 16KB. However, each read/write operation in NBJLOFS deals with a complete object, namely is of size 128KB.

In pervious version of NBJLOFS, common requests and prefetching requests are processed in serial. This strategy apparently does not make full use of CPU. So we implemented a double-threaded version. One thread processes only common requests and another processes prefetching request. Fig. 8 shows the remarkable improvement of read performance. In this test, a single client read all five files in a round robin fashion.

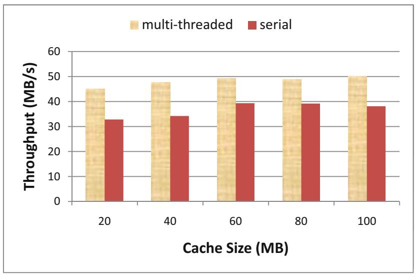


Fig. 8. Serial vs. Multi-threaded

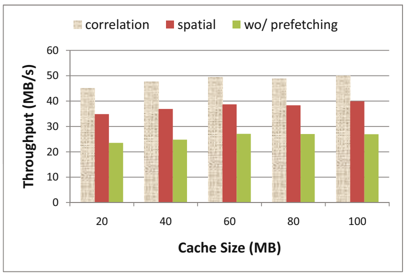


Fig. 9. The Impact of Cache Size

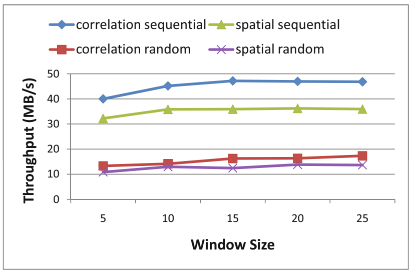


Fig. 10. The Impact of Window Size

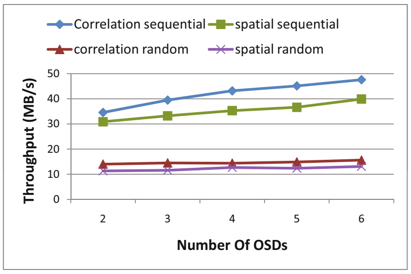
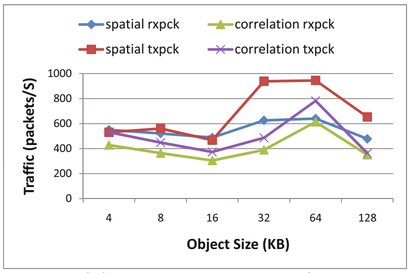
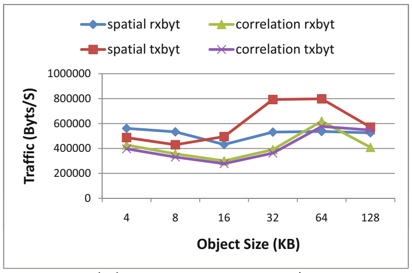


Fig. 11. Scalability



(a) traffic in packets/s



(b) traffic in bytes/s

Fig. 12. Network Traffic

Next, we tested the impact of the cache size on the read performance. We used a fixed window size 10 and files are still read in a round robin fashion. Fig. 9 shows that no matter how big the cache, correlation-aware prefetching outperform simple spatial locality based prefetching and NBJLOFS without prefetching. For a certain system, the minor performance difference is induced by cache replacement.

We tried to figure out how the window size will impact the overall system efficiency. Cache size was set to 40MB in this test (we just choose it randomly). The window size was set changeless in every single run and varied from 5 to 25 in different runs. Beside sequential read test mentioned above, we also tested completely random read. The 5 files were also read in a round robin fashion. As shown in Fig. 10, the throughput keeps increasing as the window size grows until reaches 15. Fig. 10 also shows that the random read benefits from big prefetching window too.

Scalability is important metrics for distributed systems. We tested different system sizes. As illustrated in Fig. 11, the correlation-aware prefetching always exhibits the best performance, regardless of the system size.

To prove that correlation-aware approach indeed decreases network traffic compared with traditional one, we traced real network traffic of a single OSD using *Sysstat* utilities. Fig. 12 shows the result. Since objects are distributed over OSDs evenly and prefetching and common requests are evenly distributed too in the sense of probability, we can conclude that correlation-aware approach decreases both the number of packets sent and the number of bytes sent effectively.

5 Related Work

Many literals about caching and prefetching have been published. Some of them focused on utilizing the existence of locality in disk access patterns, both spatial locality and temporal locality [6][7]. The modern operating system design concept attempts to prefetching consecutive blocks from disk to reduce the cost of on-demand I/Os [8][9]. However, when a file is not accessed sequentially or the whole data of a file are not stored consecutive, prefetching can probably result in extra I/Os. Thus numerous works have been done to find other prefetching strategies. By implementing history log and data mining technique, [10][11][12] detect access patterns which can be put to use in the future access. J.R.Cheng et al [13] has considered semantic links among objects to prefetching data in object-oriented DBMSs, while [1] tracking multiple per-object read-ahead contexts to performance prefetching.

Besides, existing studies have aimed at changing the prefetching extent dynamically and adaptively without manual intervention. A window covers all the objects has semantic link was proposed in [13]. Also, Linux kernel adopts read-ahead window to manage its associated prefetching [8].

Our approach differs from these works. In NBJLOFS, since a file may be scattered over many storage nodes, and the correlation among objects can be determined easily by the object attributes, so the correlation-aware prefetching with orthogonal layout is a natural solution. Our moving window mechanism is largely derived from these previous works.

6 Conclusions and Future Work

This paper proposed an innovative prefetching strategy for distributed object-based file system. The correlationship among objects was used to determine which objects should be prefetched. We designed an orthogonal replica layout, it reduces network traffic effectively compared with the scattered layout. We presented two distribution algorithms to guarantee this kind of layouts. We also designed two moving window strategies to adjust the size and the content of the prefetching window automatically. Compared with the traditional spatial based prefetching approach, our new approach decreases network traffic and may produce large sequential instead of small random disk operations. We admit, the introducing of the prefetching mechanism bring some negative impacts along with the dramatically benefits. The way object clustering only guarantees global load balance, while load imbalance will occur locally. However, the experiment results show that the benefits are far beyond the side effects. In the distributed object-based file system, our new prefetching approach exhibited much higher performance than the traditional spatial locality based approach.

This research can be extended in several directions. For example, in order to alleviate the workload of replica node, we are thinking about dividing duplication to several OSDs other than one single OSD. As a result, the migration strategy also needs to be redesigned. Moreover, by studying the access pattern, more works can be done to investigate other correlations among objects.

References

1. Tang, H., Gulbeden, A., Zhou, J., Strathearn, W., Yang, T., Chu, L.: The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, Pittsburgh, PA, USA, November 2004, pp. 53–62 (2004)
2. Wang, P., Gilligan, R.E., Green, H., Raubitschek, J.: IP SAN - From iSCSI to IP-Addressable Ethernet Disks. In: Proceedings of the 20th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies, San Diego, CA, USA, April 2003, pp. 189–193 (2003)
3. Lonczewski, F., Schreiber, S.: The FUSE-System:an Integrated User Interface Design Environment. In: Proceedings of Computer Aided Design of User Interfaces, Namur, Belgium (June 1996) 37–56
4. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley DB. In: Proceedings of the annual conference on USENIX Annual Technical Conference, Monterey, California, USA, June 1999, pp. 183–192 (1999)
5. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). In: Proceedings of the 1988 ACM SIGMOD international conference on Management of data, Chicago, Illinois, United States, June 1988, pp. 109–116 (1988)
6. Liu, H., Hu, W.: A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM. In: IEEE Proceedings of the 15th International Parallel and Distributed Processing Symposium, San Francisco, CA, USA, April 2001, pp. 62–67 (2001)
7. Jiang, S., Ding, X., Chen, F., Tan, E., Zhang, X.: DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality. In: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, December 2005, pp. 101–114 (2005)
8. Butt, A.R., Gniady, C., Hu, Y.C.: The performance impact of kernel prefetching on buffer cache replacement algorithms. In: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Banff, Alberta, Canada, June 2005, pp. 157–168 (2005)
9. Pai, R., Pulavarty, B., Cao, M.: Linux 2.6 Performance Improvement through Readahead Optimization. In: Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, July 2004, pp. 391–401 (2004)
10. Soundararajan, G., Mihailescu, M., Amza, C.: Context-aware prefetching at the storage server. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, Boston, Massachusetts, June 2008, pp. 377–390 (2008)
11. Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J.: Informed Prefetching and Caching. In: Proceedings of the fifteenth ACM symposium on Operating systems principles, Copper Mountain, Colorado, United States, December 1995, pp. 79–95 (1995)
12. Amer, A., Long, D.D.E., Burns, R.C.: Group-Based Management of Distributed File Caches. In: Proceedings of the 22 nd International Conference on Distributed Computing Systems, Vienna, Austria, July 2002, pp. 525–534 (2002)
13. Cheng, J.R., Hurson, A.R.: On The Performance Issues of Object-Based Buffering. In: Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, FL, USA, December 1991, pp. 30–37 (1991)

An Auxiliary Storage Subsystem to Distributed Computing Systems for External Storage Service

MinHwan Ok

Korea Railroad Research Institute,
Woulam, Uiwang, Gyeonggi, Korea
panflute@informatics.krri.re.kr

Abstract. Demands for efficient and effective utilization of computing resources have derived distributed computing systems of the large scaled such as Grid for scientific computing and Cloud for utility computing. In these distributed computing systems the total capacity of consolidated storages should expand as the amount of user data grows. The user's data are necessarily kept in the distributed computing system such as the Grid and the Cloud, which some users don't allow the system to. In this work, an auxiliary storage subsystem is proposed to provide external storages to the distributed computing system. The storage subsystem lets the iSCSI devices be connected to the distributed computing system as needed. As iSCSI is a recent, standard, and widely deployed protocol for storage networking, the subsystem is able to accommodate many types of storages outside of the distributed computing system. The proposed subsystem could transmit data by streams attaining high bit rate of the network capacity in the experiment.

Keywords: Distributed Computing System, Storage Subsystem, iSCSI, External Storage, Storage Cloud.

1 Introduction

Computing resources commonly comprises one for programs of their executing, and the other for data of their storage. These two sorts of resources are of course essential in distributed computing including Grid computing and Cloud computing. The resources for programs and those for data are closely coupled for performance considerations at the execution time. In the distributed computing systems such as Grid or Cloud these days, the user data are necessarily transmitted to the Grid system or the Cloud system in order to be processed in the system. In such systems there should be two fundamental pitfalls. The first one comes from the case the amount of user data is larger than the amount of available space left in the distributed computing system. This is a well-known problem in the areas of scientific computing[1,8]. One solution straightforward is consolidating the data center in which the user data is stored into the Grid or the Cloud. It seems a similar solution that *Open Cirrus* federates distributed data centers[2]. Open Cirrus covers locations of data centers distributed worldwide. The data center would be consolidated, after the data center has agreed on the consolidation. The second one comes from the case the owner of the user data never allow the data is dealt under the control of the Cloud or the Grid, although the data center has agreed, even though the amount of user data is small.

One solution would be that the data transmitted become transient, which the requested part of the data is available at the execution time only. The part of data is not valid after the execution. The whole data is not remained in the system without the user's approval. An auxiliary storage is required to the distributed computing systems, analogous to the removable devices of personal computers, in this manner. As it is an auxiliary storage, it is desirable to use a widely deployed protocol to be attached or detached to or from the distributed computing system. In this work, a storage subsystem is proposed to supply external storages to the distributed computing system. The storage transfers user data in the transient state with a widely deployed protocol, *iSCSI*, Internet SCSI. The subsystem could support even personal storages of private data. The details are continued in the next section.

Another solution to the first pitfall is a confederation of volunteer PCs to gain unlimited storage capacity, such as *Storage@home*[1]. The members of consolidation are PCs providing irresponsible storages and thus Storage@Home is a formation of the opposite property to Open Cirrus with its members providing responsible storages. It exploits aggregate spaces and bandwidths of volunteering PCs however it has an indigenous trustworthiness problem as its members are volunteering PCs. Further Storage@home isn't a solution to the second pitfall.

A storage cluster has focused on database transactions toward the storage Cloud, *DataLab*[3]. Although it does not provide external storages but provide fault tolerant storage, all the data could be open to the users only in the form of transactions for those recorded in the databases. DataLab supplies data transactions for distributed job executions in the distributed computing system, making use of a special hardware named *Active Storage Unit.*

2 Storage Subsystem for External Storage Service

Since the storages to be attached or detached are outside of the distributed computing system, only a registered user should be able to attach an external storage to the distributed computing system. An auxiliary storage subsystem would take the trustworthiness part on the side of the distributed computing system. The storage subsystem should also play a manager role on the external storages. The storage subsystem is depicted in Fig. 1.

The storage subsystem is an additional trustworthiness part and the owner should register an external storage in which the user data located, as a registrant of the distributed computing system. It has also a manager role on the external storage including connection, disconnection, and loading data, which is an additional one to the basic storage management of the distributed computing system. The procedure to access user data is shown in Fig. 2.

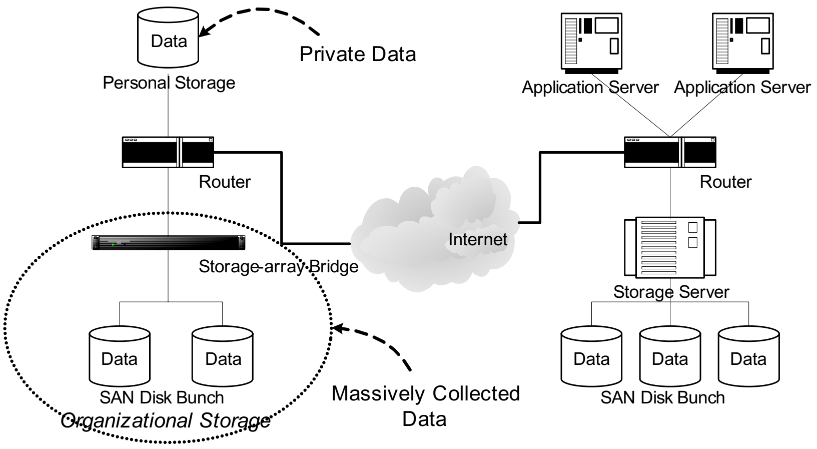


Fig. 1. The storage subsystem resides in the distributed computing system (right) and manages external storages to be attached or detached (left)

(1) Application Client	Login to the distributed computing system
(2) Application Client	Login to the auxiliary storage subsystem
(3) Storage Server or Application Server	Connect to the external storage
(4) Application Server	Load data from the external storage or preloaded data from the storage server
(5) Storage Server or Application Server	Disconnect from the external storage
(6) Application Client	Logout from the auxiliary storage subsystem
(7) Application Client	Logout from the distributed computing system

Fig. 2. The application client logs-in the auxiliary storage subsystem for the second

The storage subsystem supplies external storages including personal storages and organizational storages as depicted in Fig.1. For the personal storages containing private data, it is presumed in this work that the amount of user data is relatively small. Upon the user's application opens a file, requested blocks are transmitted from the personal storage by the application at the application server. Conventionally the file is not transmitted as a whole although it is dependent on the memory usage of the application. Note that the transmitted blocks do not constitute a regular file on the application server's local storage. The whole blocks could be transmitted to the application server and stored as a regular file for performance considerations in the case the user approved. Direct transmission from or to the personal storage is depicted in Fig. 3 and 4.

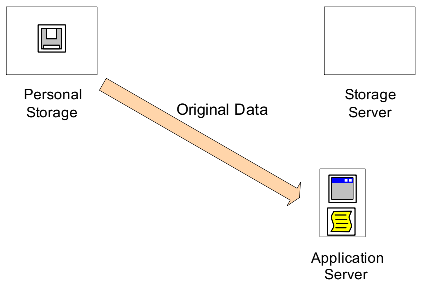


Fig. 3. Direct read from an external storage

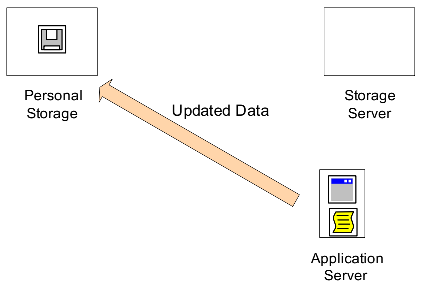


Fig. 4. Direct write to an external storage

For the organizational storages containing massively collected data, it is presumed in this work that the amount of user data is relatively large. Conventionally the file is not transmitted as a whole although it is dependent on the memory usage of the application. However as the amount of user data is massive, parts of all the blocks need be transmitted to the storage server and stored as regular files for performance considerations with the user's approval. In this case the user's application opens the files stored in the storage server. As those files are transmitted from the organizational storage to be stored, the process called *preloading* is performed[7]. Indirect transmission from or to the organizational storage is depicted in Fig. 5 and 6. When a file is to be written, the application server stores it in the storage server and then the storage server transmits the file to the organizational storage.

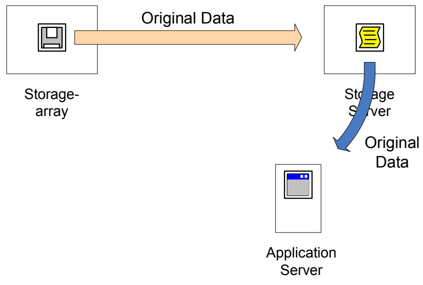


Fig. 5. Indirect read via the storage server

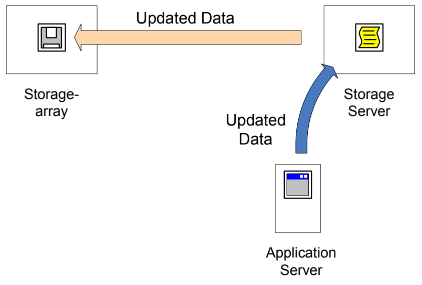


Fig. 6. Indirect write via the storage server

3 Storage Subsystem Architecture

Almost all the software support TCP/IP over Internet, and Gigabit Ethernet is a dominantly deployed LAN technology with the IP stack. SCSI, *Small Computer System Interface*, have been the basic disk interface for server systems. iSCSI is the protocol for transmissions of data on SCSI disks over Internet. iSCSI is a recent, standard, and widely deployed protocol for storage networking. The storage subsystem proposed is founded on the iSCSI technology. The storage space prepared by iSCSI is not adherent to a specific file system, but the owner of the storage space determines which file system at the time the storage space is prepared.

The iSCSI protocol can make clients access the SCSI I/O devices over IP network, and a client can use the remote storage transparently[4]. The owner's personal storage is recognized as a part of the storage server's local storage by a particular network device driver[5]. Once the software requests a file to the network device driver, *iSCSI Initiator*, it relays the request to the other network device driver, *iSCSI Target*, of the storage or the iSCSI device itself. The target storage starts to transfer the file to the storage server via the network device drivers.

When the target storage transfers data to the storage server, data blocks are delivered via iSCSI Target/Initiator or iSCSI device/iSCSI Initiator. For the performance reason, block I/O was adopted[6] that provides necessary blocks of an opened file to the storage server and updates corresponding blocks when modification occurs. Block I/O outperforms file I/O and does not adhere to a certain file system. The architecture of the storage subsystem is shown in Fig. 7.

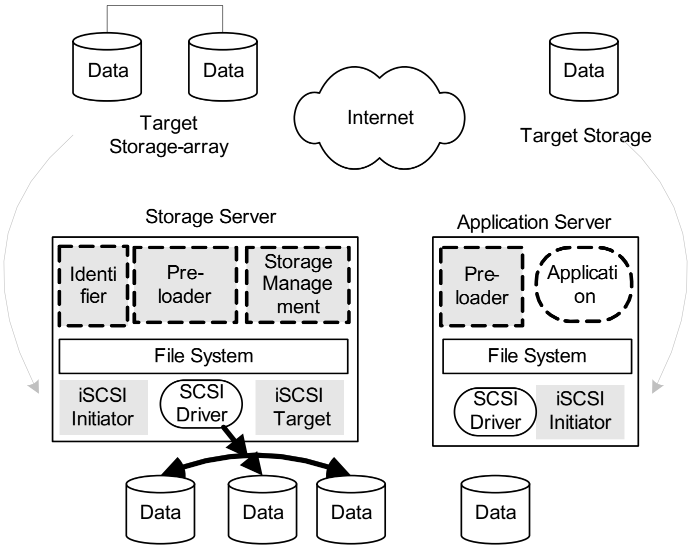


Fig. 7. The storage server or the application server connects to the external storages by iSCSI protocol

Storage Management module integrates its disk spaces and makes up volumes for preloading. The disk spaces can be interleaved to speed up the transfer, suchlike RAID(Redundant Array of Inexpensive Disks)s. *Identifier* module monitors connections with the client and maintains each connection, i.e. connection re-establishment and transfer recovery. It also manages user login or logout to or from the auxiliary storage subsystem from or to the distributed computing system. The user among the registrant of the distributed computing system is allowed to transmit data by the auxiliary storage subsystem. *Pre-loader* module caches the user data from the external storage for the application server. On arrival a new job invokes its application but waits for data to process, while other jobs are processing their data. Pre-loader loads data from an external storage and stores it in the storage server with the user's approval, once the job has started and becomes asleep waiting for the data to be loaded. Further on the Pre-loader is detailed in our previous work[7].

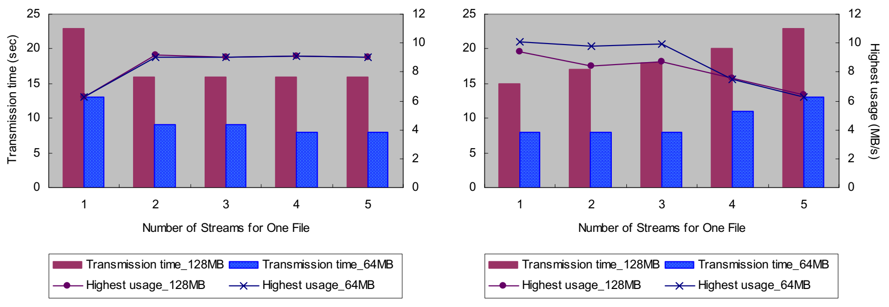
4 Subsystem Evaluation onto the External Storage

The storage service is composed of the storage server and clients, with iSCSI target and iSCSI initiator as respective network device drivers. The storages server is implemented and evaluated through Gigabit Ethernet concatenated to Fast Ethernet. For evaluation of the subsystem, only one storage server exists on Gigabit Ethernet and only one client exists on Fast Ethernet. After logged into the storage server, the client declares NTFS on a volume mounted from the external storage.

Table 1. Summary of Experimentation Environment

	Storage Server	Client
OS	Redhat-Linux kernel 2.6	MS-Windows 2000 Professional
CPU	Intel Xeon 2.8GHz	Intel Pentium M 1.5GHz
Memory	1GB	512MB
HDD	Ultra320 SCSI 10,000RPM	EIDE ATA-100 4,200RPM
NIC	Gigabit Ethernet	Fast Ethernet

Only one drive of the storage server is used in the experiment. A SCSI disk drive nowadays is capable of pumping up data, in a bulk transfer mode, sufficient to fill up the Gigabit Ethernet capacity. However multiple data streams would be required to fully utilize the available bandwidth on TCP/IP networks. The multiple transmission streams are generated by storing a file into multiples of a few allocation units. It is very similar to that of RAID. Two sizes of allocation units, 64kB and 1kB, are chosen in this experiment as shown in Fig. 8.



(a) Lavish allocation unit (64kB) (b) Compact allocation unit (1kB)

Fig. 8. Transmission time and the highest usage during transmission of one file

The transmission times extend by file sizes. A tendency is shown that the smaller file gets the higher bandwidth usage. The transmission times are reciprocally proportional to the bandwidth usages. The bandwidth of Fast Ethernet would dominate the transmission time despite the superior bandwidth of Gigabit Ethernet.

The size of an allocation unit is 64kB in the left of Fig. 8. The bandwidth usage is attained to 6.3MB/s for both transmissions of 128MB and 64MB. By multiplying the number of transmission streams the bandwidth usage is attained to nearly 9.1MB/s for both transmissions. Due to the limit in capacity of Fast Ethernet and the protocol interactions, no more bandwidth usage is attained after doubling the transmission stream. The size of an allocation unit is 1kB in the right of Fig. 8. The bandwidth usages are attained to 9.4MB/s and 10.1MB/s for transmissions of 128MB and 64MB, respectively. By multiplying the number of transmission streams the bandwidth usage is shrunk to nearly 6.4MB/s for both transmissions. Due to the limit in capability with a single drive and the overhead from the protocols, the attained bandwidths shrank by multiplying the number of the transmission streams.

In the case each storage server is equipped with very large main memory, i.e. 1TB, a file is preserved in the main memory without storing it on storage after transmitted as a whole. Such storage management originally aims at high performance[8], and it would be also positive to the second pitfall. Currently the auxiliary storage subsystem is not for data sharing between users, but there are an alternative for multiple users in the relative study[8]. Another issue to consider could be effective placements of the auxiliary storage subsystems. The broker intervenes in efficient allocations of the distributed storages[9], and this method would be helpful to the subsystem placements in future work.

5 Conclusion

As the amount of user data grows the total capacity of consolidated storages should expand in the distributed computing system. It is an unlikely surmise that all the computing sites or data centers would agree on consolidation into the Grid or the Cloud. Further all the data stored are not continuously used in the distributed computing system and in this sense the Grid or the Cloud is not efficient on storage. The storage subsystem proposed is founded on iSCSI, a recent, standard, and widely deployed protocol for storage networking. The iSCSI protocol can make clients access the SCSI I/O devices over IP network, and a client can use the remote storage transparently. The storage subsystem lets the iSCSI devices be connected to the distributed computing system as needed. In the experiment, the transmission streams gained about 80% of Fast Ethernet capacity. Since the result came from an experiment that only the storage server and the client is connected to Gigabit Ethernet and Fast Ethernet, respectively, the other traffic could make the attained usage lower when passing through Internet. The pre-loader is an important module on this respect. It exploits idle network bandwidth while other applications are processing data since their data are loaded.

All the users do not like to let their data under control of the Cloud or the Grid system. The concept of Cloud or Grid would not appeal to these users since possessing their data is not an acceptable premise. This is a big blockage in the developments of applications for the Grid and the Cloud. In the proposed subsystem the transmitted blocks do not constitute a regular file in the application server or the storage server without the user's approval. This feature is oriented to the users' view but not to the developers' view.

References

[1] Campbell, R., Gupta, I., Heath, M., Ko, S.Y., Kozuch, M., Kunze, M., Kwan, T., Lai, K., Lee, H.Y., Lyons, M., Milojicic, D., O'Hallaron, D., Soh, Y.C.: Open Cirrus^{TM} Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In: Workshop on Hot topics in Cloud computing, pp. 1–5. USENIX (2007)

[2] Beberg, A.L., Pande, V.S.: Storage@home: Petascale Distributed Storage. In: IEEE Int. Parallel and Distributed Processing Symposium, pp. 1–6 (2007)

[3] Rich, B., Thain, D.: DataLab: transactional data-parallel computing on an active storage cloud. In: Int. Symp. on High Performance Distributed Computing, pp. 233–234. ACM, New York (2008)

[4] Lu, Y., Du, D.H.C.: Performance study of iSCSI-based storage subsystems. Communication Magazine 41, 76–82 (2003)

[5] Ok, M., Kim, D., Park, M.-s.: UbiqStor: A remote storage service for mobile devices. In: Liew, K.-M., Shen, H., See, S., Cai, W. (eds.) PDCAT 2004. LNCS, vol. 3320, pp. 685–688. Springer, Heidelberg (2004)

[6] Block Device Driver Architecture, `http://msdn.microsoft.com/library/en-us/wceddk40/html/_wceddk_system_architecture_for_block_devices.asp`

[7] Ok, M.: A Sharable Storage Service for Distributed Computing Systems in Combination of Remote and Local Storage. In: Hua, A., Chang, S.-L. (eds.) ICA3PP 2009. LNCS, vol. 5574, pp. 545–556. Springer, Heidelberg (2009)

[8] Nicolae, B., Antoniu, G., Bouge, L.: Distributed Management of Massive Data: An Efficient Fine-Grain Data Access Scheme. In: Palma, J.M.L.M., Amestoy, P.R., Daydé, M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 532–543. Springer, Heidelberg (2008)

[9] Secretan, J., Lawson, M., Boloni, L.: Efficient allocation and composition of distributed storage. Jour. of Supercomputing 47(3), 286–310 (2009)

Checkpointing and Migration of Communication Channels in Heterogeneous Grid Environments

John Mehnert-Spahn and Michael Schoettner

Heinrich-Heine University, Duesseldorf, NRW 40225, Germany
John.Mehnert-Spahn@uni-duesseldorf.de,
Michael.Schoettner@uni-duesseldorf.de

Abstract. A grid checkpointing service providing migration and transparent fault tolerance is important for distributed and parallel applications executed in heterogeneous grids. In this paper we address the challenges of checkpointing and migrating communication channels of grid applications executed on nodes equipped with different checkpointer packages. We present a solution that is transparent for the applications and the underlying checkpointers. It also allows using single node checkpointers for distributed applications. The measurement numbers show only a small overhead especially with respect to large grid-applications where checkpointing may consume many minutes.

1 Introduction

Fault tolerance can be achieved for many distributed and parallel applications, particularly for scientific ones, using a rollback-recovery strategy [1]. Here, programs are periodically halted to take a checkpoint that can be used to restart the application in the event of a failure. As nodes may fail permanently the restart implementation must support restarting checkpoints on new healthy nodes. The latter is also important for application migration, e.g. to realize load balancing.

Checkpointing solutions have been available for many years, mainly used in high performance computing and batch processing systems. As grid applications are getting into mainstream there is an emerging need for a transparent checkpointing solution. Right now there is a great variety of different checkpointing packages such as single node checkpointers, e.g. BLCR (with MPI support) [8], OpenVZ [9]. Virtual machine technologies, e.g. VMware [4] and XEN [7] are also used as checkpointers. Furthermore, there are distributed checkpointers such as LinuxSSI [5], DCR [12] and DMTCP [2]. All these implementations come with different capabilities and there is no ultimate best checkpointer.

Therefore, we have designed XtreemGCP - a grid checkpointing service capable of checkpointing and restarting a grid job running on nodes equipped with different checkpointing packages [13]. Each grid job consists of one or multiple job units. Each job unit represents a set of processes of a job on one grid node. Checkpointing a distributed job does not only require to take a snapshot of all process states on all involved nodes but also to handle in-transit messages as well. Otherwise, orphan messages and lost messages can lead to inconsistent checkpoints. Orphan messages occur if the reception of a message is part of the receiver side checkpoint, however the message send event is not part of the sender side checkpoint. In case of a restart the sender will send this message again which may cause faults.

Lost messages occur if reception of a message is not part of the receiver side checkpoint, however the message send event is part of the sender side checkpoint. During restart this message will not be sent again and the receiver may block and run into a failure situation. Obviously, in-transit messages need to be handled by all involved checkpointers. As the latter have not been designed to cooperate with each other, e.g. BLCR does not cooperate with DMTCP, we need a subordinated service transparently flushing all communication channels at checkpoint time avoiding in-transit messages. And this service must also support channel reconnections in case of a job migration. The contributions of this paper are the concepts and implementation of a transparent grid channel checkpointing (GCC) facility for a heterogeneous setup with various existing checkpointer packages. We address TCP sockets used by a great range of distributed applications. As UDP communication inherently tolerates lost messages we do not take care of in-transit UDP messages.

The outline of this paper is as follows. In Section 2 we present an overview of grid channel checkpointing followed by Section 3 describing the architecture in detail. The different phases to checkpoint, restart and migrate channels are discussed in Section 4 followed by an evaluation. Related work is discussed in Section 6 followed by conclusions and future work.

XtreemGCP is implemented within XtreemOS - a Linux-based distributed OS for next generation grids [3]. This work is funded by the European Commision under FP6 (FP6-033576).

2 Grid Channel Checkpointing Overview

The main idea of the GCC approach is to flush all TCP channels of a distributed application before a checkpoint operation in order to avoid lost and orphan messages. Flushing of TCP channels is achieved by preventing all application threads from sending and receiving messages as well as from creating new channels during the checkpointing operation.

Concurrent GCC protocol execution is achieved by using separate threads for channel control and flushing. Once, appropriate controller threads have been installed at both peer processes of an application TCP channel, a marker message is sent through the channel, signaling the marker receiver that the channel is empty. Potential in-transit messages received by a controller thread in the context of channel flushing are stored in a so-called channel draining buffer at the receiver side.

No messages can get lost since all in-transit messages will be assigned to the current checkpoint on the receiver side. No orphan message can occur since sending of application messages is blocked until the checkpoint operation has finished and all received messages have been recognized as being sent on the sender side.

To support checkpointers incapable of saving/restoring socket descriptors, sockets may need to be closed before a checkpoint operation and recreated after a taken checkpoint. A recent version of LinuxSSI failed during restart, since sockets that were open during checkpointing time could not be reconstructed. But of course, whenever possible we leave sockets up and running for performance reasons.

Before an application resumes with its normal execution, potentially closed sockets will be recreated and reconnected. Furthermore, any blocked channel creation system calls and formerly blocked send and recv calls will be released. Messages formerly stored in the channel draining buffer get consumed by application threads before new messages can be received.

Finally, GCC also needs to handle changed IP-addresses caused by job-unit migration. To handle changing IP-addresses we introduce a GCC manager component that can be contacted by all involved GCC controller instances.

3 GCC Architecture

In the following text we describe the Grid Channel Checkpointing (GCC) architecture and its components in detail. Fig. 1 presents all GCC components.

GCC marks TCP sockets because UDP sockets are not relevant. It also determines the current socket usage mode, indicating whether being in receive or in send mode. The socket usage mode may change dynamically as TCP sockets can be used in a bidirectional fashion. If a process sends and receives messages over one TCP socket it is called to be in duplex mode.

For application-transparent channel checkpointing network system calls such as send, recv, connect, accept, listen, etc. must be controlled. Application threads must neither send or receive messages nor create new channels while checkpointing is in progress. However, GCC threads must be able to exchange GCC-messages on application channels (to be flushed) and on GCC control channels.

The library interposition technique is used, to achieve these features in an application-transparent way. Therefore, we initialize the environment variable LD_PRELOAD with the path to our interposition library. Thus, all network calls of the application end up in the interposition library. The contained network function wrappers pass the calls to the original library. Therewith we are able, e.g. to block a send call within the associated function wrapper.

As previously mentioned, sockets may have to be re-connected to new IP addresses in case of process migration. Sockets must also be closed and recreated on nodes whose checkpointers are incapable of handling open socket descriptors. Both tasks can be handled using the callback mechanism provided by XtreemGCP [13]. The latter explicitly executes registered callbacks before and after a checkpoint or after a restart.

In addition these callbacks are used to integrate the channel flushing protocol. GCC callbacks are registered in an application-transparent way, realized by a fork-wrapper included in the above mentioned interposition library.

Two control threads are created per process, the send-controller thread and the recv-controller thread, see Fig. 1. If a socket is shared by multiple processes

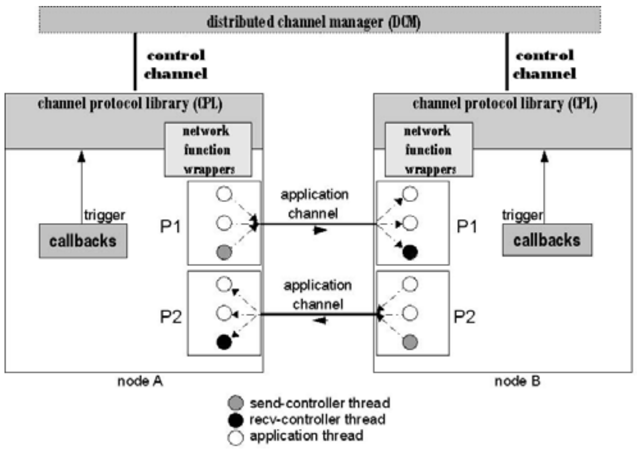


Fig. 1. GCC components

on one node, the send- or recv-controller thread of one process becomes channel leader of it. The channel leader exclusively supervises channel flushing and reconnection, but relies on cooperation with the remaining controller threads. More precisely, the send-controller takes control over sockets of TCP channels being in send mode. At checkpoint and restart time it is in active mode which means it initiates flushing and reconnection with the remote recv-controller. The recv-controller takes control over TCP channels being in receive mode. It is in passive mode and reacts on flushing and reconnection request from a send-controller.

Distinguishing send- and recv-controllers allows handling situations where two communicating processes are server and client for different channels at the same time. No deadlock can occur during restart e.g. if one server socket on each side, recreated by the relevant send-controller, waits for the opposite node to reconnect, since the relevant recv-controller is ready to handle a reconnection request.

Migrated sockets must be detected before a controller thread tries to initiate a socket reconnection. Thus, each socket creation, recreation and any other socket state changes are registered at the distributed channel manager (DCM). Therefore, all controller threads can use a TCP control channel to the channel manager. Currently, we have a central channel manager implementation (one manager for each grid job) that will be replaced by a distributed version for scalability reasons.

GCC execution is triggered by callbacks triggered before and after checkpointing and immediately after a restart.

3.1 Shared Sockets

As mentioned before a socket formerly shared by multiple processes must be recreated with special care, just one process must recreate it. Thus, in a multi-process

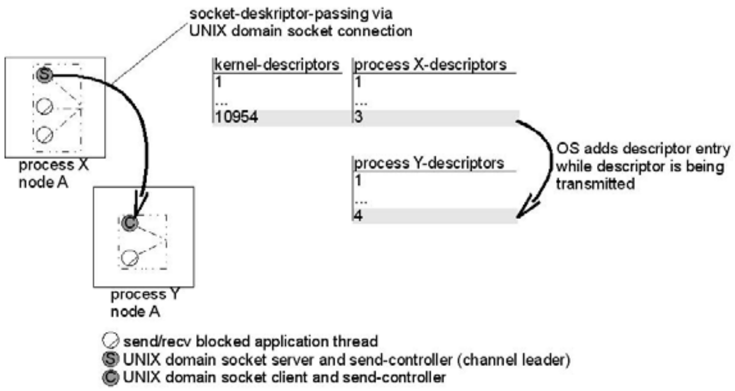

socket-deskriptor-passing via UNIX domain socket connection
kernel-descriptors
1
...
10954
process X-descriptors
1
...
3
OS adds descriptor entry while descriptor is being transmitted
process Y-descriptors
1
...
4
process X node A
process Y node A
send/recv blocked application thread
UNIX domain socket server and send-controller (channel leader)
UNIX domain socket client and send-controller

Fig. 2. Process-wide shared soclets and descriptor passing

application the channel leader exclusively recreates the socket in the kernel within the post-checkpoint or restart callback. For the remaining application processes to see the newly recreated socket we use the UNIX descriptor passing mechanism [10]. Using the latter allows the channel leader to pass recreated socket descriptors to controller threads of other processes using UNIX domain socket connections, see Fig. 2.

The channel manager assigns a unique key per channel to involved controller threads for a UNIX domain socket connection setup and descriptor exchange. The key remains the same also after a potential migration. While the descriptor is sent, a corresponding entry is made in the process-owned descriptor table. Process socket descriptors assigned to a logical channel must be matched with those being valid before checkpointing. Socket descriptors are assigned in an increasing order during normal runtime. If an intermediate descriptor gets closed, a gap exists, and the highest descriptor number is bigger than the total number of sockets currently used. During socket rebuild, descriptors will be assigned in an increasing number, without gaps by taking the association of channel and descriptor number into account saved during pre-checkpoint time and descriptors will be rearranged to the correct order using the dup2 system call.

Thus, this approach avoids false multiple recreations of a shared socket and the latter do not need to be reestablished exclusively via process forking and inheritance in user space. Additionally kernel checkpointer based process recreation, which excludes the calling of fork at user space, is supported as well.

4 GCC Phases

4.1 Pre-checkpoint Phase

Blocking channel creation: Since there is a short time gap between pre-checkpoint-callback execution and process synchronization in the checkpoint phase, the creation of new TCP channels must be blocked. This is realized by blocking the socket calls accept and connect until the checkpoint operation is finished.

Determining channel leaders: Sending on TCP channels must be prevented to drain channels in finite time. Two challenges need to be addressed in this context. First, if multiple processes use a shared socket, each of them may send a marker, many could receive one or multiple markers. Obviously, it is unclear when the last message has been received. Second, application threads could be blocking on send and recv calls which would prevent the protocol to start.

This first challenge is addressed by determining a so-called channel leader whose task is to exclusively treat sockets shared by multiple processes. More precisely, one send controller will be installed per process and out-going channel, one recv controller thread will be installed per process and in-coming channel. Each controller sends a channel-leader-request to the channel manager. Thus, e.g. a two-process application, whose child inherited socket descriptors of its parent, sends two requests to the channel manager. The channel manager selects one of the requesting controller threads as the channel leader and informs all about this decision.

The second challenge is solved by the controller threads sending a SIGALRM signal to applications threads blocking on send or recv calls. Application and controller threads can be distinguished such that just application threads will be put asleep in the signal handler routine while controller threads can start with channel draining. Finally, no application thread is able to send or receive messages along TCP channels anymore until the end of the post-checkpoint phase, see Section 4.2.

Before channel flushing can be initiated by a send controller, a recv controller must contact the DCM to learn on which socket it is supposed to listen for the marker. Both use a handshake to agree on which channel is to be flushed. Channel flushing: The send controller being the channel leader at the same time sends a marker. The marker is the last message received by the remote recv controller

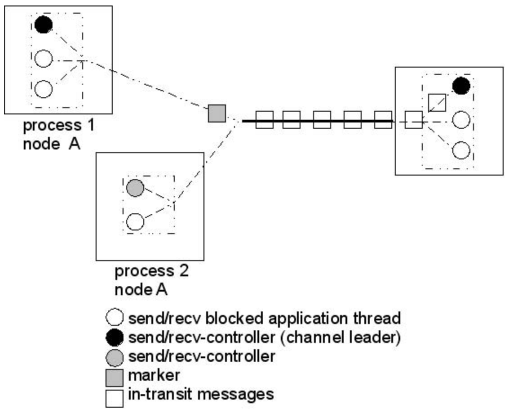


Fig. 3. Channel flushing with marker message

which is the channel leader on the peer node. Messages received before the marker are in-transit messages. Marker and in-transit content will be separated. The latter will be put in a channel draining buffer. The buffer data is assigned to the appropriate receiving application thread. Usually, just one thread waits for messages on a peer. However, multiple threads can do so as well. In the latter case the OS scheduler decides non-deterministically which thread receives the message. Thus, the channel checkpointing protocol copies received data into the receive buffer of the application thread that has been listening to this channel recently. This equals a possible state during fault-free execution.

Marker recognition can be achieved at different levels. Currently, a special marker message is sent along the application channel, see Fig. 3. Since the underlying TCP layer fragments application data independent of application semantics a marker message can overlap with two or multiple TCP packages. Thus, at each byte reception the recently received and buffered data must be matched backwards against the marker. In the future we plan to replace this first approach by another alternative, e.g. extending each application network packet by a GCC header which is removed at receiver side or using a socket shutdown to enforce the sending of a TCP FIN message to be detected in the kernel.

Furthermore, we optionally need to close open sockets for checkpointer packages that cannot handle them. This is no problem for the application threads as they are blocked and sockets will be re-opened in the post-checkpoint phase, see Section 4.2. The final step of the pre-checkpoint phase is saving the association of socket descriptor and channel key, needed during post-checkpoint and restart time.

4.2 Post-checkpoint Phase

Unblocking channel creation: This GCC phase aims at unblocking the network communication just after a checkpoint has been taken. At first, channel creation blocking is released, unblocking system calls such as connect and accept.

Recreating (shared) sockets: Sockets need to be recreated only if they had been closed before checkpointing or in case of a restart. There is no need to adapt server socket addresses, since no migration took place.

Socket recreation becomes more complex if sockets are shared by multiple processes, see Section 4.1.

Release send/recv barriers: The last step of this GCC phase is to unblock formerly blocked send and recv calls and to wake up application threads. Furthermore, any buffered in-transit messages need to be consumed before any new messages.

4.3 Migration and Restart Phase

The GCC restart is similar to the post-checkpoint phase but both differ from the location of execution and migration-specific requirements. The first step here includes the release of the channel creation blockade (unblocking connect and accept calls). Furthermore, we need to address different checkpointer (CP) capabilities:

1. CPs capable of saving, restoring and reconnecting sockets (for changed IP addresses),
2. CPs capable of saving and restoring sockets, but not supporting socket migration,
3. CPs being unable to handle sockets at all.

Obviously, we need to address cases 2 and 3, only. Here sockets are recreated, their descriptors are rearranged as described under Section 3.1. Before a client socket can reconnect to a server socket, we must check if a migration took place recently and if the server is already listening for reconnections. The latter is the task of the channel manager which receives any changed server addresses from the control threads, see Section 3. Thus, a client just queries the DCM to learn the reconnection state.

The last step in this phase includes releasing any formerly blocked send and recv calls and waking up application threads.

5 Evaluation

The GCC pre- and post-checkpoint phases have been measured using a synthetic distributed client server application running on nodes with heterogeneous and homogeneous checkpointer packages installed. The test application sends periodically 100 Byte packets in five second intervals. At client and server side each channel is handled by a separate thread.

In the first test case the server part is executed and checkpointed on one node part of a LinuxSSI cluster (v2.1), the client part on grid node with BLCR (v0.8.2) installed. The channel manager is executed on a separate node inside the LinuxSSI cluster. In the second test case client and server have been executed and checkpointed on nodes with BLCR installed.

The testbed consists of nodes with Intel Core 2 DUO E6850 processors (3 GHz) with 2 GB RAM interconnected with a Gigabit Ethernet network.

5.1 Test Case 1: Heterogeneous Checkpointers and GCC

Fig. 4 indicates the times taken at a client and a server for flushing, closing and reestablishing channels on top of LinuxSSI and BLCR checkpointers (no shared sockets have been used). The pre-checkpoint phase takes up to 4.25 seconds to handle 50 channels. The duration is mainly caused by the serial synchronization of the send and recv controller threads via the channel manager. An improved channel manager is on the way handling requests concurrently. Furthermore, the duration includes memory buffering and consumption of potential in-transit messages.

Fig. 4 also shows the time needed for the post-checkpoint phase taking about half of the time as the pre-checkpoint phase. This is due to less interaction with

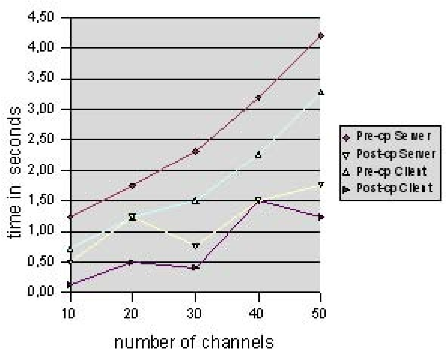


Fig. 4. GCC behavior on top of LinuxSSI and BLCR with closing and reestablishing channels

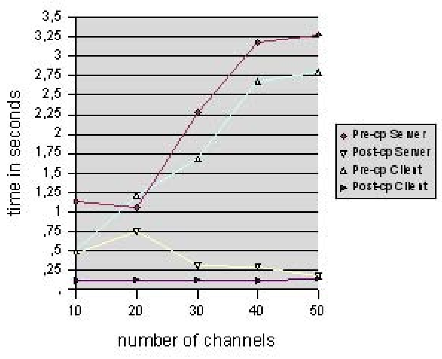


Fig. 5. Same scenario as in Fig. 5 but without closing and reestablishing channels

the channel manager and of course no channels need to be flushed. As expected if necessary, rebuilding and reconnecting of sockets is costly.

Fig. 5 indicates the times for the same scenario as shown in Fig. 4 but without closing and reestablishing channels. Here the pre-checkpoint phase takes less time (about 3.25 seconds to handle 50 channels). Furthermore, without socket rebuilding and reconnecting this post-checkpoint phase is also significantly shorter than the one from above just taking about 120 milliseconds for 50 channels.

Another aspect is that GCC is working on top of heterogeneous callback implementations without major performance drawbacks. While BLCR comes with its own callback implementation implicitly blocking applications threads, LinuxSSI does not. For the latter we have to use the generic callback implementation provided by XtreemGCP.

5.2 Test Case 2: Homogeneous Checkpointers and GCC

Fig. 6 indicates the times taken for the client and server to flush, close and reestablish channels based on the mere usage of BLCR checkpointers. The pre-checkpoint phase takes up to 2.75 seconds to handle 50 channels. It is faster than the pre-checkpoint phase of test case 1 because both grid nodes run native Linux there is no SSI-related overhead caused by LinuxSSI. The post-checkpoint phase is significantly shorter than the one of test case 1 because no native SSI structures must be updated when sockets are being recreated and reconnected.

Fig. 7 indicates the times taken for a client and server to flush open channels. It takes less time during the pre- and post-checkpoint phase compared to the previous setup of test case 2.

Overall we see that the current implementation of GCC consumes more time when checkpointing more channel connections per grid node. Although we do not expect thousands of connections per grid node as the typical case we plan to optimize the GCC solution to be more scalable.

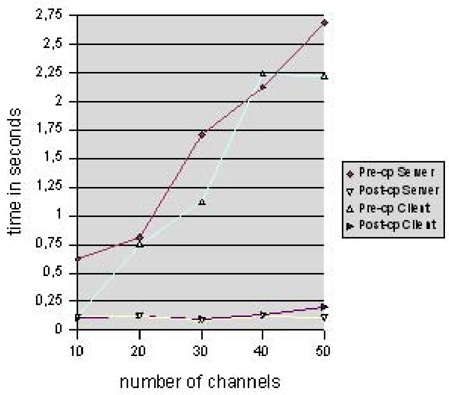


Fig. 6. GCC with BLCR only, with closing and reestablishing channels

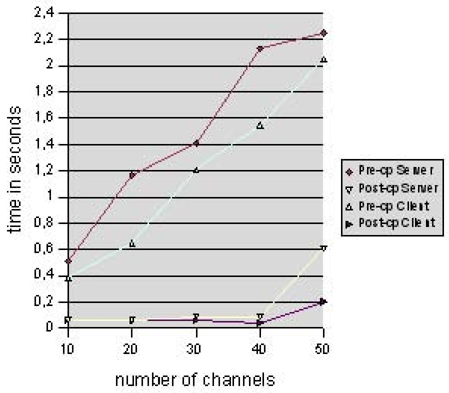


Fig. 7. Same scenario as in Fig. 6 but without closing and reestablished channels

Another aspect is that the amount of messages in-transit to be drained during checkpointing operation will also influence GCC times. But as the bandwidth of grid networks is typical several Mbit/s or even more, we do not expect an extensive time overhead here. Furthermore, though checkpointing communication channels may consume several seconds we think this is acceptable because checkpointing large grid applications may take many minutes to save the application state to disk.

6 Related Work

Overall there is only one other project working on checkpointing in heterogeneous grid environments while there are different projects implementing checkpointing for MPI applications [11]. However, there are many publications proposing sophisticated checkpointing protocols but that are not related to heterogeneity challenges addressed by this paper.

The CoreGRID grid checkpointing architecture (GCA) [14] proposes a similar architecture like XtreemGCP aiming to integrate low-level checkpointers. However, the current GCA implementation supports Virtual Machines (VMs), only, and does not support checkpointing communication channels of distributed applications.

The Open Grid Forum (OGF) GridCPR Working Group has published a design document for application-level checkpointing [6] that is not addressing transparent channel checkpointing.

DMTCP [2] is most close to the approach proposed in this paper. It is a distributed library checkpointer able to checkpoint and migrate communication channels. They also use a marker message to flush in-transit messages but the latter will be sent back to the original sender at checkpoint time and forth to the receiver at resume/restart time. In contrast we store in-transmit messages at the receiver side. Furthermore, DMTCP supports only one specific checkpointer whereas our approach is designed for heterogeneous grid environments. Finally, shared sockets are recreated during restart by a root process in user space which inherits them to children processes created later. In contrast to our approach processes with disturbed original parent-child relations cannot be recreated.

In [12] communication states are saved by modifying the kernel TCP protocol stack of the OS. The approach is MPI specific, does not support shared sockets, and is designed for one checkpointer (BLCR) and not for a heterogeneous setup.

7 Conclusions and Future Work

Transparent checkpointing and restarting distributed applications requires handling in-transit messages of communication channels. The approach we propose flushes communication channels at checkpoint time avoiding orphan and lost messages. The contribution of this paper is not a new checkpointing protocol but concepts and implementation aspects how to achieve channel flushing in a heterogeneous grid where nodes have different checkpointer packages installed.

The proposed solution is transparent for applications and existing checkpointer packages. It also allows to use single node checkpointers for distributed applications without modifications because GCC takes care of checkpointing communication channels.

GCC is a user mode implementation not requiring kernel modifications. It also offers transparent migration of communication channels and supports recreation of sockets shared by multiple threads of one or more processes. Our measurements show that the current implementation can handle dozens of connections in reasonable time, especially with respect to checkpointing times of huge applications which can be many minutes.

Future goals include implementation optimizations to improve scalability and channel flushing support for asynchronous sockets.

References

1. Elnozahy (Mootaz), E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3), 375–408 (2002)
2. Ansel, J., Arya, K., Cooperman, G.: DMTCP: Transparent checkpointing for cluster computations and the desktop. In: 23rd IEEE International Parallel and Distributed Processing Symposium, Rome, Italy (May 2009)
3. http://www.xtreemos.eu
4. Sugarman, J., Venkitachalam, G., Lim, B.H.: Virtualizing I/O devices on VMWare Workstations Hosted Virtual machine Monitor (2001)
5. Fertre, M., Morin, C.: Extending a cluster ssi os for transparently checkpointing message-passing parallel application. In: ISPAN, pp. 364–369 (2005)
6. Stone, N., Simmel, D., Kilemann, T., Merzky, A.: An architecture for grid checkpoint and recovery services. Technical report (2007)

7. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, J., Warfiled, A.: XEN and the art of virtualization (2003)
8. Duell, J.: The design and implementation of berkeley lab's linux checkpoint/restart (2003)
9. http://download.openvz.org/doc/openvz-intro.pdf
10. Stevens, W.R., Fenner, B., Rudoff, A.M.: UNIX Network programming The Sockets Networking API, vol. I. Addison-Wesley, Reading (2004)
11. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI checkpoint/restart framework: System-intiated checkpointing. International Journal of High Performance Computing Applications 19(4) (2005)
12. Ma, C., Huo, Z., Cai, J., Meng, D.: Dcr: A fully transparent checkpoint/restart framework for distributed systems (2009)
13. Mehnert-Spahn, J., Ropars, T., Schoettner, M., Morin, C.: The Architecture of the XtreemOS Grid Checkpointing Service Euro-Par. Delft, The Netherlands (2009)
14. Jankowski, G., Januszewski, R., Mikolajczak, R., Stroinski, M., Kovacs, J., Kertesz, A.: Grid checkpointing architecture - integration of low-level checkpointing capabilities with grid. Technical Report TR-0036, CoreGRID, May 22 (2007)

On-Line Task Granularity Adaptation for Dynamic Grid Applications*

Nithiapidary Muthuvelu[1], Ian Chai[1], Eswaran Chikkannan[1], and Rajkumar Buyya[2]

[1] Multimedia University, Persiaran Multimedia, 63100 Cyberjaya, Selangor, Malaysia
{nithiapidary,ianchai,eswaran}@mmu.edu.my
[2] Dept. of Computer Science and Software Engineering,
The University of Melbourne, 3053 Carlton, Victoria, Australia
raj@csse.unimelb.edu.au

Abstract. Deploying lightweight tasks on grid resources would let the communication overhead dominate the overall application processing time. Our aim is to increase the resulting computation-communication ratio by adjusting the task granularity at the grid scheduler. We propose an on-line scheduling algorithm which performs task grouping to support an unlimited number of user tasks, arriving at the scheduler at runtime. The algorithm decides the task granularity based on the dynamic nature of a grid environment: task processing requirements; resource-network utilisation constraints; and users QoS requirements. Simulation results reveal that our algorithm reduces the overall application processing time and communication overhead significantly while satisfying the runtime constraints set by the users and the resources.

1 Introduction

A grid application contains a large number of tasks [1] and a scheduler at the user site transmits each task file to a grid resource for further execution and retrieves the processed output file(s) [2][3]. A lightweight or fine-grain task requires minimal execution time (i.e. 15 seconds to one minute). Executing a computation-intensive application with a large number of lightweight tasks on a grid would result in a low computation-communication-ratio due to the overhead involved in handling each task [4]; the term *computation* refers to the task execution time, whereas *communication* refers to the user authentication, and task and output file transmission time. There are two issues involved in this matter:

1. The communication overhead increases proportionally with the number of tasks.
2. A resource's processing capability and the network capacity may not be optimally utilised when dealing with fine-grain tasks. For example:
   (a) Assume that a high-speed machine allows a user to use the CPU power for x seconds. Executing lightweight tasks one at a time would not utilise the full processing speed (i.e. $x\times$Million Instructions per Second) of the machine within x seconds due to the overhead involved in invoking and executing each task.

* This research is partially supported by e-ScienceFund, Ministry of Science, Technology and Innovation, Malaysia, and Endeavour Awards, Austraining International.

(b) Transmitting task/output files (of very minimal sizes) one by one between the user and the resources would underutilise the relevant achievable bandwidth.

In short, deploying lightweight tasks on grid would lead to inefficient resource-network utilisation, and unfavourable application throughput. This statement is proven with experiments in Sec. 5 of this paper. In our previous work [5], we showed that task grouping reduces the overall application processing time significantly. In our current work, we present an on-line scheduling algorithm for deciding the task granularity. The scheduler has no knowledge on the total number of tasks in the application as the tasks come on a real-time basis, arriving at the scheduler at runtime (e.g. processing data arriving from sensors).

Upon receiving the tasks, the scheduler selects and groups a number of tasks into a batch, and deploys the grouped task on a grid resource. The task granularity is determined as to maximise the resource-network utilisation and minimise the overall application processing time. Hence, the decision highly depends on the dynamic nature of a grid environment:

1. The processing requirements of each task in a grid application.
2. The utilisation constraints imposed by the resource providers to control the resource usage [6].
3. The varying bandwidths of the networks interconnecting the scheduler and the resources [7].
4. The quality of service (QoS) requirements for executing an application [8].

Our proposed scheduling algorithm focuses on computation-intensive, bag-of-tasks applications. It assumes that all the tasks in an application are independent and have similar compilation platform. The algorithm considers the latest information from the grid resources, decides the task granularity, and proceeds with task grouping and deployment. Our ultimate goal is to reduce the overall application processing time while maximising the usage of resource and network capacities.

The rest of the paper is organised as follows: Section 2 presents the related work. The factors and issues involved in task grouping in grid are described in Sec. 3. Section 4 explains the strategies and the process flow of the proposed scheduler system which is followed by the relevant performance analysis in Sec. 5. Finally, Sec. 6 concludes the paper by suggesting future work.

2 Related Work

Here, we focus on the work related to batch processing in distributed computing which involve task granularity adaptation. James et al [9] grouped and scheduled equal numbers of independent jobs using various scheduling algorithms to a cluster of nodes. This induced an overhead as the nodes were required to be synchronised after each job group execution iteration. Simulations were conducted to optimise the number of jobs in a batch for a parallel environment by Sodan et al [10]. The batch size is computed based on average runtime of the jobs, machine size, number of running jobs in the machine, and minimum/maximum node utilisation. However, these simulations did not consider the varying network usage or bottleneck, and it limits the flexibility of the job groups by fixing the upper and lower bounds of the number of jobs in the group.

Maghraoui et al [11] adapted the task granularity to support process migration (upon resource unavailability) by merging and splitting the atomic computational units of the application jobs. The jobs are created using a specific API; special constructs are used to indicate the atomic jobs in a job file which are used as the splitting or merging points during job migration.

A number of simulations had been conducted to prove that task grouping reduces the overall grid application processing time. The authors in [5][12] grouped the tasks based on resource's Million Instructions Per Second (MIPS) and task's Million Instructions (MI); e.g. for utilising a resource with 500 MIPS for 3 seconds, tasks were grouped into a single task file until the maximum MI of the file was 1500. MIPS or MI are not the preferred benchmark matrices as the execution times for two programs of similar MI but with different program locality (e.g. program compilation platform) can differ [13]. Moreover, a resource's full processing capacity may not be available all the time because of the I/O interrupt signals.

In our previous work [14], task grouping was simulated according to the parameters from users (budget and deadline), applications (estimated task CPU time and task file size), utilisation constraints of the resources (maximum allowed CPU and wall-clock time, and task processing cost per time unit), and transmission time tolerance (maximum allowed task file transmission time). The simulations show that the grouping algorithm performs better than the conventional task scheduling algorithm by 20.05% in terms of overall application throughput when processing 500 tasks. However, it was assumed that the task file size is similar to the task length which is an oversimplification as the tasks may contain massive computation loops.

In this paper, we treat the file size of a task separately from its length or processing needs. We also consider two additional constraints: space availability at the resource; and output file transmission time. In addition, the algorithm is designed to support an unlimited number of user tasks arriving at the scheduler at runtime.

3 Factors Influencing the Task Granularity

Figure 1 shows the implementation focus of our scheduling algorithm in a grid environment. The factors influencing the task granularity are passed to the scheduler from the (1) user application, (2) grid resources, and the (3) scheduler. The user application is a bag-of-tasks (BOT) with QoS requirements. Each task is associated with three requirement properties: size of the task file (TFSize); estimated size of the output file (OFSize); and the estimated CPU time (ETCPUTime). The QoS includes the budget (UBudget) and deadline (UDeadline) allocated for executing all the user tasks.

Each resource (R) from a set of participating grid resources (GR) provides its utilisation constraints to the scheduler:

1. Maximum CPU time (MaxCPUTime) allowed for executing a task.
2. Maximum wall-clock time (MaxWCTime) a task can spend at the resource. This encompasses the CPU time and the processing overhead (waiting time and task packing/unpacking overhead) at the resource.

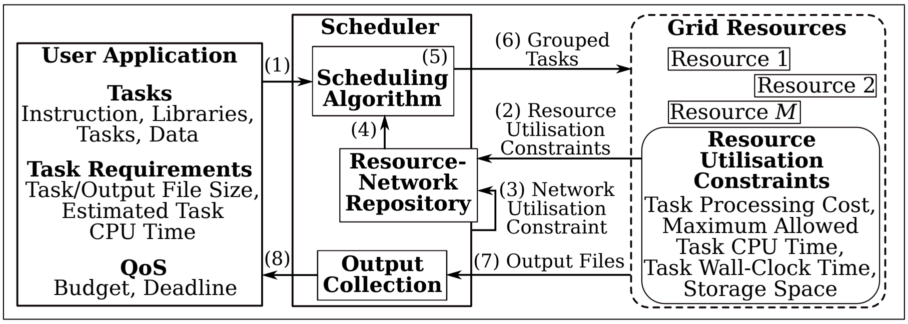

User Application
Tasks
Instruction, Libraries, Tasks, Data
Task Requirements
Task/Output File Size, Estimated Task CPU Time
QoS
Budget, Deadline
(1)
(8)
Scheduler
(5)
Scheduling Algorithm
(4)
Resource-Network Repository
Output Collection
(6) Grouped Tasks
(2) Resource Utilisation Constraints
(3) Network Utilisation Constraint
(7) Output Files
Grid Resources
Resource 1
Resource 2
Resource M
Resource Utilisation Constraints
Task Processing Cost, Maximum Allowed Task CPU Time, Task Wall-Clock Time, Storage Space

Fig. 1. Scheduler Components and their Information Flow

3. Maximum storage space (MaxSpace) that a task or a set of tasks (including the relevant output files) can occupy at a time.
4. Task processing cost (PCost) per unit time charged by a resource.

Finally, the network utilisation constraint is the maximum time that a scheduler can wait for the task/output files to be transmitted to/from the resources (MaxTransTime). It is the tolerance threshold that a scheduler can accept in terms of file transmission time.

Having these information, we derived the seven objective functions for determining the granularity of a task group, TG, for a resource, Ri, as follows:

Objective 1: TG CPU time $\leq MaxCPUTime_{Ri}$
Objective 2: TG wall-clock time $\leq MaxWCTime_{Ri}$
Objective 3: TG and output transmission time $\leq MaxTransTime_{Ri}$
Objective 4: TG and output file size $\leq MaxSpace_{Ri}$
Objective 5: TG turnaround time $\leq$ Remaining $UDeadline$
Objective 6: TG processing cost $\leq$ Remaining $UBudget$
Objective 7: Number of tasks in $TG \leq$ Remaining BOT_{TOTAL}
where, BOT_{TOTAL} = total number of tasks waiting at the scheduler.

However, there are three issues that affect the granularity according to these seven objective functions.

ISSUE I: A resource can be a single node or a cluster. The task wall-clock time is affected by the speed of the resource's local job scheduler and the current processing load. In order to obey the objectives 2 and 5, one should know the overheads of the resources' queuing systems in advance.

ISSUE II: The task CPU time differs according to the resources' processing capabilities. For example, a group of five tasks can be handled by Resource A smoothly, whereas it may exceed the maximum allowed CPU time or wall-clock time of Resource B, in spite of having a similar architecture as Resource A; the processing speed of a resource cannot be estimated in advance based on the hardware specification only. Moreover, the task CPU time highly depends on the programming model or compilation platform. Hence, we should learn the resource speed and the processing need of the tasks prior to the application deployment.

ISSUE III: Task grouping increases the resulting file size to be transmitted to a resource, leading to an overloaded network. Moreover, the achievable bandwidth and latency [7][15] of the interconnecting network are not static; e.g. the bandwidth at time t_x may support the transmission of a batch of seven tasks, however, at time t_y, this may result in a heavily-loaded network (where $x < y$). Hence, we should determine the appropriate batch size that can be transferred at a particular time.

4 Scheduler Implementation

In our scheduling algorithm, the issues mentioned in Sec. 3 are tackled using three approaches in the following order: Task Categorisation; Task Category-Resource Benchmarking; and Average Analysis. The following subsections explain the three approaches respectively and present the process flow of the entire scheduler system.

4.1 Task Categorisation

The tasks in a BOT vary in terms of TFSize (e.g. a non-parametric sweep application), ETCPUTime, and OFSize. When adding a task into a group, the resulting total TFSize, ETCPUTime, and OFSize of the group get accumulated. Hence, the scheduler should select the most appropriate tasks from the BOT (without significant delay) and ensure that the resulting group satisfies all the seven objective functions.

We suggest a task categorisation approach to arrange the tasks in a tree structure based on certain class interval thresholds applied to the TFSize, ETCPUTime, and OFSize. The tasks are divided into categories according to TFSize class interval ($TFSize_{CI}$), followed by ETCPUTime class interval ($ETCPUTime_{CI}$), and then OFSize class interval ($OFSize_{CI}$).

Algorithm 1 depicts the level 1 categorisation in which the tasks are divided into categories (TCat) based on TFSize and $TFSize_{CI}$. The range of a category is set according to $TFSize_{CI}$. For example, the range of:

$TCat_0$: 0 to ($TFSize_{CI} + TFSize_{CI}/2$)
$TCat_1$: ($TFSize_{CI} + TFSize_{CI}/2$) to ($2 \times TFSize_{CI} + TFSize_{CI}/2$)
$TCat_2$: ($2 \times TFSize_{CI} + TFSize_{CI}/2$) to ($3 \times TFSize_{CI} + TFSize_{CI}/2$)

The category ID (TCatID) of a task is 0 if its TFSize is less than the $TFSize_{CI}$ (line 2,3). Otherwise, the mod and base values (line 5,6) of the TFSize are computed to determine the suitable category range.

For example, when $TFSize_{CI} = 10$ size units, then a task with,
$TFSize = 12$ belongs to $TCat_0$ as $TCat_0$($0 < TFSize < 15$)
$TFSize = 15$ belongs to $TCat_1$ as $TCat_1$($15 \leq TFSize < 25$)
$TFSize = 30$ belongs to $TCat_2$ as $TCat_2$($25 \leq TFSize < 35$)

Algorithm 1. Level 1 Task Categorisation

```
Data: Requires $TFSize$ of each $T$ and $TFSize_{CI}$
1  for $i \leftarrow 0$ to $BOT_{TOTAL}$ do
2      if $T_{i-TFSize} < TFSize_{CI}$ then
3          $TCatID \leftarrow 0$
4      else
5          $ModValue \leftarrow T_{i-TFSize} \bmod TFSize_{CI}$
6          $BaseValue \leftarrow T_{i-TFSize} - ModValue$
7          if $ModValue < TFSize_{CI}/2$ then
8              $TCatID \leftarrow (BaseValue/TFSize_{CI}) - 1$
9          else
10             $TCatID \leftarrow ((BaseValue + TFSize_{CI})/TFSize_{CI}) - 1$
11     $T_i$ belongs to $TCat$ of ID $TCatID$
```

This is followed by level 2 categorisation; TCat(s) from level 1 is further divided into sub-categories according to ETCPUTime and $ETCPUTime_{CI}$. The similar categorisation algorithm is applied with ETCPUTime of each task and $ETCPUTime_{CI}$. Subsequently, level 3 categorisation divides the TCat(s) from level 2 into sub-categories based on OFSize and $OFSize_{CI}$.

Figure 2 shows an instance of categorisation with $TFSize_{CI} = 10$, $ETCPUTime_{CI} = 6, OFSize_{CI} = 10$. The category(s) at each level is created when there is at least one task belonging to the particular category. For each resulting TCat, the average requirements are computed: average TFSize (AvgTFSize); average ETCPUTime (AvgETCPUTime); and average OFSize (AvgOFSize).

When a new set of tasks arrives at the scheduler, each task is checked for its requirements and assigned to the appropriate TCat; new categories with certain ranges are created as needed. Having this organisation, the scheduler can easily locate the task files (for a group) that obey the utilisation constraints and QoS requirements.

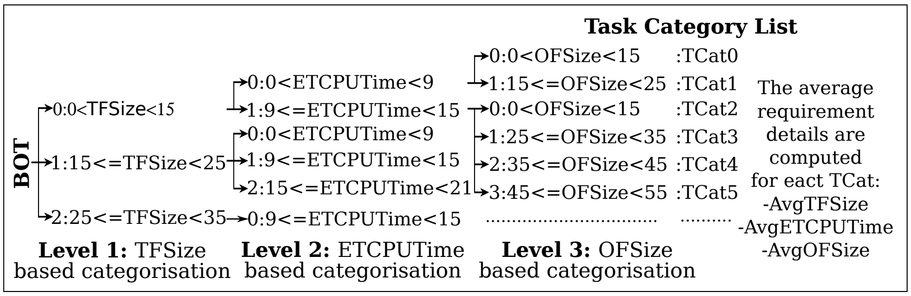


Fig. 2. Task Categorisation

4.2 Task Category-Resource Benchmarking

In this benchmark phase, the scheduler selects a few tasks from the categories for further deployment on the resources before scheduling the entire user application. This helps the scheduler to study the capacity, performance, and overhead of the resources and the interconnecting network over the user tasks. It selects p tasks from the first m dominating categories (based on the total number of tasks in each category) and sends to each resource. The total number of benchmark tasks, $BTasks_{TOTAL}$, can be expressed as:

$$BTasks_{TOTAL} = m \times p \times GR_{TOTAL} \tag{1}$$

Upon retrieving the processed output files, the remaining UBudget and UDeadline are updated accordingly, and the following seven actual deployment matrices of each task are computed:

> task file transmission time (scheduler-to-resource); CPU time; wall-clock time; processing cost; output file transmission time (resource-to-scheduler); processing overhead; and turnaround time.

Finally, the average of each deployment matrix is computed for each task category-resource pair. For a category k, the average deployment matrices on a resource j are expressed as average deployment matrices of $TCat_k - R_j$, which consist of:

> average task file transmission time ($AvgSTRTime_{k,j}$); average CPU time ($AvgCPUTime_{k,j}$); average wall-clock time ($AvgWCTime_{k,j}$); average processing cost ($AvgPCost_{k,j}$); average output file transmission time ($AvgRTSTime_{k,j}$); average processing overhead ($AvgOverhead_{k,j}$); and average turnaround time ($AvgTRTime_{k,j}$).

The average deployment matrices of those categories which did not participate in the benchmark phase are then updated based on the average ratio of the other categories. Assume that m categories have participated in the benchmark phase, then the average matrices of a category can be formulated in the following order:

$$AvgSTRTime_{i,j} = (\textstyle\sum_{k=0}^{m-1}(AvgTFSize_i \times AvgSTRTime_{k,j}/AvgTFSize_k))/m$$
$$AvgCPUTime_{i,j} = (\textstyle\sum_{k=0}^{m-1}(AvgETCPUTime_i \times AvgCPUTime_{k,j}/AvgETCPUTime_k))/m$$
$$AvgRTSTime_{i,j} = (\textstyle\sum_{k=0}^{m-1}(AvgOFSize_i \times AvgRTSTime_{k,j}/AvgOFSize_k))/m$$
$$AvgPCost_{i,j} = (\textstyle\sum_{k=0}^{m-1}(AvgCPUTime_{i,j} \times AvgPCost_{k,j}/AvgCPUTime_{k,j}))/m$$
$$AvgOverhead_{i,j} = (\textstyle\sum_{k=0}^{m-1} AvgOverhead_{k,j})/m$$
$$AvgWCTime_{i,j} = AvgCPUTime_{i,j} + AvgOverhead_{i,j}$$
$$AvgTRTime_{i,j} = AvgWCTime_{i,j} + AvgSTRTime_{i,j} + AvgRTSTime_{i,j}$$

where,
k = 0,1,2,...,$TCat_{TOTAL} - 1$; TCat ID participated in benchmark.
j = 0,1,2,...,$GR_{TOTAL} - 1$; grid resource ID.
i = 0,1,2,...,$TCat_{TOTAL} - 1$; TCat ID did not participate in benchmark.
m = Total categories participated in benchmark.

In short, the benchmark phase studies the response and performance of the resources and the interconnecting network on each category.

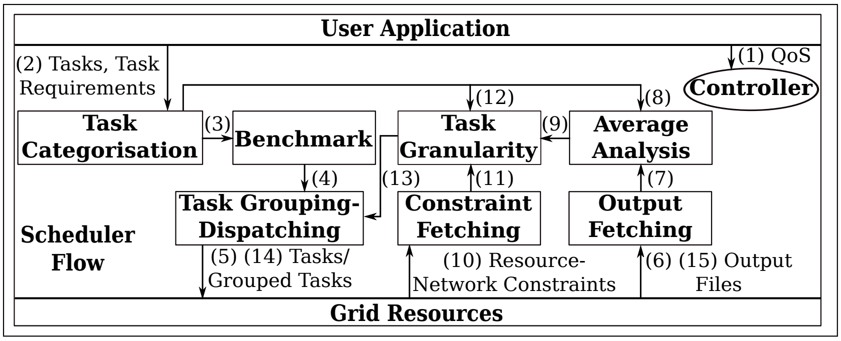


Fig. 3. Process Flow of the Scheduler System

4.3 Average Analysis

Knowing the behaviour of the resources and network, we can group the tasks according to the seven objective functions of task granularity. However, as grid resides in a dynamic environment, the deployment matrices of the categories may not reflect the latest grid status after a time period. Therefore, the scheduler should update the deployment matrices of each $TCat_k - R_j$ pair periodically based on the latest arrived processed task groups.

First, it gets the 'actual' deployment matrices of the latest arrived processed groups. Using the previous $TCat_k - R_j$ average matrices, it computes the deployment matrices that each task group 'supposed' to get. Then, the ratio 'supposed':'actual' of each deployment matrix is computed to estimate and update the $TCat_k - R_j$ average matrices. For those categories which did not participate in the latest task groups, their $TCat_k - R_j$ average matrices get updated based on the ratio of the other categories as explained in Sec. 4.2.

4.4 Scheduler Process Flow

Figure 3 presents the process flow of the entire scheduler system. (1) The *Controller* manages the scheduler activity in terms of the flow and periodic average analysis. It ensures that the QoS requirements are satisfied at runtime. (2) The *Task Categorisation* categorises the user tasks. (3) It then invokes the *Benchmark* which selects $BTasks_{TOTAL}$ from the categorised BOT for (4,5) further task deployment on the grid resources. (6) The *Output Fetching* collects the processed benchmark tasks and (7,8) the *Average Analysis* module studies the task category-resource or $TCat_k - R_j$ average deployment matrices. (10) The *Constraint Fetching* retrieves the resource-network utilisation constraints periodically to set the task granularity objective functions. (9,11,12) Having the categorised tasks, $TCat_k - R_j$ average deployment matrices, and the resource-network utilisation constraints, the *Task Granularity* determines the number of tasks from various categories that can be grouped for a particular resource.

When selecting a task from a category, the expected deployment matrices of the resulting group are accumulated from the average task deployment matrices of the particular category. The final granularity must satisfy all the seven objective functions mentioned in Sec. 3 of this paper. The task categorisation process derives the need for enhancing objective 7 to control the total number of tasks that can be selected from a category k:

Objective 7: Total tasks in TG from a $TCat_k \leq size\_of(TCat_k)$

(13,14) Upon setting the granularity, the *Task Grouping-Dispatching* selects and groups the tasks, and transfers the batch to the designated resource. (15) The processed task groups are then collected by the *Output Fetching*; the remaining UBudget and UDeadline are updated accordingly. The cycle (10-15) continues for a certain time period and then the *Controller* signals the *Average Analysis* to update the average deployment matrices of each $TCat_k - R_j$ to be used by the subsequent task group scheduling and deployment iterations.

5 Performance Analysis

The scheduling algorithm is simulated using the GridSim [16]. There are 400-2500 tasks involved in this performance analysis with TFSize (6-40 size units), ETCPUTime (70-130 time units), and OFSize (6-40 size units). The $TFSize_{CI}$, $ECPUTime_{CI}$, $OFSize_{CI}$ are of 10 size units each. The QoS constraints are UDeadline (200K-600K time units) and UBudget (6K-8K cost units).

The grid is configured with eight cluster-based resources, each with three processing elements. The processing capacity of a cluster is 200-800 MIPS and the associated utilisation constraints: MaxCPUTime (30-40 time units), MaxWCTime (400-700 time units), MaxSpace (1K-5K size units), MaxTransTime (8K-9K time units), and PCost (3-10 cost units per a time unit). For benchmarking, two tasks are selected from the first four dominating categories. The user submits 400 tasks to the scheduler at start-up time and periodically submits 200 tasks at intervals set by Poisson distribution with λ=1.0 time unit. Figure 4 depicts the performance table/charts of the scheduler from the following experiments.

EXPERIMENT I: First, we trace the performance of the scheduler with three resources (R_0-R_2), UBudget=6000 cost units, and UDeadline=200K time units. Table 1 depicts the number of remaining tasks in each TCat during the deployment iterations. Initially, 13 categories are created as indicated in Column I. Column II shows the tasks upon the benchmark phase ($BTasks_{TOTAL} = 24$) with remaining UDeadline=190K time units and UBudget=5815 cost units.

After the benchmark, the task granularity is computed for each resource based on $TCat_k - R_j$ average deployment matrices. The resulting task groups for the three resources:

$R_0 : TCat_0(24), R_1 : TCat_0(13) + Tcat_1(56), R_2 : TCat_1(4) + TCat_2(20)$
e.g. $TCat_0(24)$ indicates 24 tasks from $TCat_0$

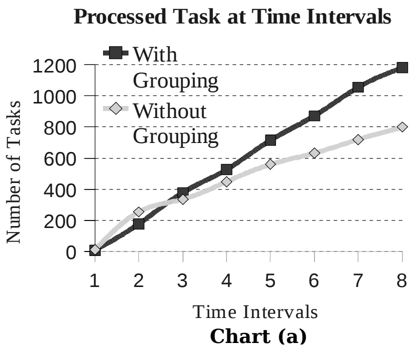
Table 2 shows how the estimated granularities for R_0 and R_1 adhere to the constraints in the objective functions. The actual deployment matrices of the relevant processed

Table 1. Remaining Category Tasks

TCat	0	1	2	3	4	5	6	7	8	9	10	11	12	Total
I	37	66	80	67	2	62	16	35	29	1	3	1	1	400
II	37	60	74	61	2	56	16	35	29	1	3	1	1	376
III	0	0	54	61	2	56	16	35	29	1	3	1	1	259
IV	0	0	20	9	2	56	16	35	29	1	3	1	1	173
V	15	45	63	38	3	82	23	51	44	2	4	2	1	373
VI	0	0	0	0	0	31	11	83	74	4	6	4	1	214

Table 2. The Validation of Task Granularity

R	Estimated Average Matrices according to the Task Granularity *vs* Objective Functions	Actual Deployment Matrices of the Processed Task Groups (Proposed Scheduler)	Average Deployment Matrices of the Processed Individual Tasks (Conventional Scheduler)
R_0	AvgCPUTime:29 *vs* MaxCPUTime:30 AvgWCTime:31 *vs* MaxWCTime:700 AvgTrantTime:1600 *vs* MaxTransTime:9500 AvgSpace:336 *vs* MaxSpace:5000 AvgPCost:88 *vs* UBudget:5815 AvgTRTime:965 *vs* UDeadline:190K	CPU time:22 Wall-clock time:25 Transmission time:1712 Space:355 Processing cost:66	CPU time:1.6 Wall-clock time:2 Transmission time:634 Space:20 Processing cost:4.8
R_1	AvgCPUTime:49 *vs* MaxCPUTime:50 AvgWCTime:53 *vs* MaxWCTime:700 AvgTrantTime:7310 *vs* MaxTransTime:8000 AvgSpace:1274 *vs* MaxSpace:10000 AvgPCost:299 *vs* UBudget:5727 AvgTRTime:7365 *vs* UDeadline:190K	CPU time:34 Wall-clock time:38 Transmission time:4900 Space:1113 Processing cost:204	CPU time:1.1 Wall-clock time:2 Transmission time:564 Space:20 Processing cost:6.6



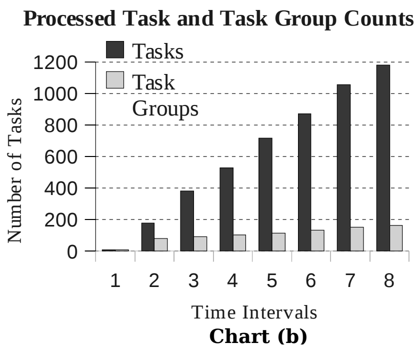


Fig. 4. Performance Tables and Charts of the Proposed Scheduler

task groups prove that the scheduling algorithm fulfills all the seven objective functions for deploying 117 tasks (Table 1, Column III) in batches. The next iteration uses the same average deployment matrices, resulting in groups with $R_0 : TCat_1(17), R_1 : TCat_2(17) + Tcat_3(19), R_2 : TCat_3(33)$; Column IV indicates the remaining 173 tasks.

After this point, a new set of 200 tasks arrived at the scheduler (Column V). The subsequent iteration is guided by the average analysis and task groups are formed based on the latest grid status; $R_0 : TCat_0(15) + TCat_1(14), R_1 : TCat_1(31) + TCat_2(19), R_2 : TCat_2(44) + TCat_3(1)$. The scheduler flow continues with average analysis, task granularity, grouping, deployment and new task arrival. At the end, the scheduler managed to complete 786 tasks out of 1000 within the UDeadline (Column VI). For comparison purpose, a similar experiment was conducted with conventional task scheduling (deploying tasks one-by-one). The scheduler deployed only 500 tasks out of 1000 within the UDeadline.

An instance of average deployment matrices of the conventional scheduler is shown in Table 2. R_0 manage to process 24 tasks (in a group) in 1759 time units which can be averaged as 73.29 time units per task. However, the conventional scheduler spent 637.6 time units to process one task; 99.4% of the deployment time is used for file transmission purpose. This indicates that a grid environment is not suitable for lightweight tasks. Hence, there is a strong need for the proposed scheduler which can adaptively resize the batch size for efficient grid utilisation.

EXPERIMENT II: Here, we conduct the simulation in an environment of eight resources with UDeadline=600K time units and UBudget=8000 cost units. The charts in Fig. 4 show the performance based on the observations at eight time intervals upon scheduler start-up. After the second interval, the scheduler produced better outcome throughout the application deployment in terms of total processed tasks as shown in Chart (a). For example, our scheduler successfully executed 1181 tasks by interval 8, whereas the conventional scheduler executed only 800 tasks, resulting in a performance improvement of 47.63%. Chart (b) depicts the task and task group counts processed by the proposed scheduler. For example, 716 tasks are successfully processed by our scheduler at interval 5 (Chart (a)). Interval 5 on Chart (b) indicates that there are only 113 file transmissions needed to process the 716 tasks (24 benchmark tasks and 89 groups). However, the conventional scheduler had 560 file transmissions by this interval (Chart (a)), an additional communication overhead of 20.18%.

6 Conclusion

The proposed scheduling algorithm uses simple statistical computations to decide on the task granularity that satisfies the current resource-network utilisation constraints and user's QoS requirements. The experiments prove that the scheduler leads towards an economic and efficient usage of grid resources and network utilities. The scheduler is currently being implemented for real grid applications. In future, the algorithm will be adapted to support work-flow application models. The scheduler will be improved to deal with unforeseen circumstances such as task failure and migration as well.

References

1. Berman, F., Fox, G.C., Hey, A.J.G. (eds.): Grid Computing - Making the Global Infrastructure a Reality. Wiley and Sons, Chichester (2003)
2. Baker, M., Buyya, R., Laforenza, D.: Grids and grid technologies for wide-area distributed computing. Softw. Pract. Exper. 32, 1437–1466 (2002)
3. Jacob, B., Brown, M., Fukui, K., Trivedi, N.: Introduction to Grid Computing. IBM Publication (2005)
4. Buyya, R., Date, S., Mizuno-Matsumoto, Y., Venugopal, S., Abramson, D.: Neuroscience instrumentation and distributed analysis of brain activity data: a case for escience on global grids: Research articles. Concurrency and Computation: Practice and Experience (CCPE) 17, 1783–1798 (2005)
5. Muthuvelu, N., Liu, J., Soe, N.L., Venugopal, S., Sulistio, A., Buyya, R.: A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids. In: Proceedings of the 2005 Australasian workshop on Grid computing and e-research, pp. 41–48. Australian Computer Society, Inc. (2005)
6. Feng, J., Wasson, G., Humphrey, M.: Resource usage policy expression and enforcement in grid computing. In: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, Washington, DC, USA, pp. 66–73. IEEE Computer Society, Los Alamitos (2007)
7. Arnon, R.G.O.: Fallacies of distributed computing explained (2007), http://www.webperformancematters.com/
8. Ranaldo, N., Zimeo, E.: A framework for qos-based resource brokering in grid computing. In: Proceedings of the 5th IEEE European Conference on Web Services, the 2nd Workshop on Emerging Web Services Technology, Halle, Germany, vol. 313, pp. 159–170. Birkhauser, Basel (2007)
9. James, H., Hawick, K., Coddington, P.: Scheduling independent tasks on metacomputing systems. In: Proceedings of Parallel and Distributed Computing Systems, Fort Lauderdale, US, pp. 156–162 (1999)
10. Sodan, A.C., Kanavallil, A., Esbaugh, B.: Group-based optimizaton for parallel job scheduling with scojo-pect-o. In: Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications, Washington, DC, USA, pp. 102–109. IEEE Computer Society, Los Alamitos (2008)
11. Maghraoui, K.E., Desell, T.J., Szymanski, B.K., Varela, C.A.: The internet operating system: Middleware for adaptive distributed computing. International Journal of High Performance Computing Applications 20, 467–480 (2006)
12. Ng, W.K., Ang, T., Ling, T., Liew, C.: Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing. Malaysian Journal of Computer Science 19, 117–126 (2006)
13. Stokes, J.H.: Behind the benchmarks: Spec, gflops, mips et al (2000), http://arstechnica.com/cpu/2q99/benchmarking-2.html
14. Muthuvelu, N., Chai, I., Chikkannan, E.: An adaptive and parameterized job grouping algorithm for scheduling grid jobs. In: Proceedings of the 10th International Conference on Advanced Communication Technology, vol. 2, pp. 975–980 (2008)
15. Lowekamp, B., Tierney, B., Cottrell, L., Jones, R.H., Kielmann, T., Swany, M.: A Hierarchy of Network Performance Characteristics for Grid Applications and Services (2003)
16. Buyya, R., Murshed, M.M.: Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. Concurrency and Computation: Practice and Experience (CCPE) 14 (2002)

Message Clustering Technique towards Efficient Irregular Data Redistribution in Clusters and Grids

Shih-Chang Chen[1], Tai-Lung Chen[1,*], and Ching-Hsien Hsu[2]

[1] College of Engineering
Fax: +886-5186416
[2] Department of Computer Science and Information Engineering
Chung Hua University, Hsinchu, Taiwan 300, R.O.C.
{scc,tai,robert}@grid.chu.edu.tw

Abstract. Efficient scheduling algorithms are essential to irregular data redistribution in cluster grid. Cluster grid is an environment with heterogeneous computing nodes and complex network. It is important for schedulers to keep an eye on load balance and low communication cost while distributing different size of data segment on various processors. High Performance Fortran Version 2 (HPF2) provides GEN_BLOCK distribution format which facilitates generalized block distributions. In this paper, we present a message clustering technique to derive low communication cost when performing such operation in cluster grids. The main idea of the proposed technique is to cluster three kinds of messages and normalize the cost. The performance evaluation is given and show the proposed method successfully adapts to heterogeneous environment.

1 Introduction

Researches on data redistribution can be classified into regular and irregular. Both fields required efficient data redistribution scheduling algorithms to shorten the communication cost. Before performing scheduling algorithms, array needs to be specified by BLOCK, CYCLIC and BLOCK-CYCLIC(*c*) for regular distribution, and user-defined function for irregular distribution such as GEN_BLOCK.

Cluster grids provide various processors and heterogeneous network environment to execute scientific applications. While performing irregular data redistribution with different ability of processors, different number of jobs and quantity of data should be distributed accordingly. Message generators refer to distribution schemes given by different computing phases of scientific applications to generate messages. Then, schedulers give schedules with low communication cost to redistribute data for status changed from current phase to next one.

[*] Corresponding author.

```
      PARAMETER (previous = /23, 8, 8, 17, 21, 8/)
!HPF$ PROCESSORS P(6)
      REAL A(85), P (6)
!HPF$ DISTRIBUTE A (GEN_BLOCK(previous)) onto P
!HPF$ DYNAMIC
      next = /12, 6, 27, 8, 19, 13/
!HPF$ REDISTRIBUTE A (GEN_BLOCK(next))
```

A code segment of High Performance Fortran version 2 (HPF2) is given above which provides GEN_BLOCK distribution format to perform generalized block distribution. Two parameters for two phases of a scientific application are given such as ***previous*** and ***next***. The parameters represent the uneven data size on array. According to the parameters, array is firstly distributed on six processors and then redistributed. Both operations are performed by GEN_BLOCK function. To redistribute data with low communication cost, efficient scheduling algorithms are essential.

Schedulers such as *Two-Phase Degree Reduction* algorithm (*TPDR*) [7] is an efficient algorithm. *TPDR* provides two phases to achieve good performance. This algorithm transforms communication relation between processors into a graph. The vertexes represent the processors while the edges represent the communication in the graph. According to color mechanism, the degree of in-fan or out-fan edges determines the number of communication steps, which is also the number of scheduling steps given by *TPDR*. The idea of *TPDR* is to reduce a degree to derive a scheduling step. The first phase of *TPDR* is a degree reduction phase, which reduce the degree of the graph and derive a communication step recursively while the current degree is larger than 2. The second phase is an adjustable coloring mechanism to derive the last two steps for *TPDR*.

Local message reduction (*LMR*) [3] is an optimization technique which provides better load balance during data redistribution. Authors of *LMR* observed that the cost of messages is also data size generated by message generator. In fact, messages are only transmitted in local memory or between two processors in one cluster. The transmitting rate in local memory should be different from that between two processors. Then, messages are clustered in two sorts: one is local data access, which is transmitted in local memory. The relative transmitting rate is defined as local access time (*LAT*). The other one is remote data access, which is transmitted between two different processors. Remote access time (*RAT*) is defined for the relative transmitting rate. *LMR* defines *RLR* as *RAT* divided by *LAT* to normalize the cost of messages.

Both *TPDR* and *LMR* are discussed in single cluster. With the advancement of network and parallel computational architecture, GEN_BLOCK array redistribution is expected to be performed in grids. *TPDR* is doubted to adapt the topology and provide good schedules. In this paper, a *message clustering technique* (*MCT*) is proposed for such environment. The idea of proposed method is to cluster three kinds of messages and normalized cost of each message. Then different kinds of messages are scheduled in different steps to derive a low cost schedule.

The rest of this paper is organized as follows. Section 2 presents a brief survey of related work. Definitions and an example of schedule are given in Section 3. In Section 4, the *message clustering technique* (*MCT*) for multiple clusters in grids is introduced to reduce the cost of GEN_BLOCK redistribution. The example in Section 3 is used to explain the *MCT*. In Section 5, the simulation results and performance analysis are given to weigh the pros and cons. Finally, the conclusions and future works are presented in Section 6.

2 Related Work

Three kinds of researches are focused on regular array redistribution including the communications set identification; message packing and unpacking techniques; communication optimizations. The communication sets identification techniques were proposed including the *ScaLAPACK* in [13], and *CFS* and *ED* in [12]. The message packing and unpacking techniques including the *ECC* method [1], which were proposed by Bai *et al.* to pack/unpack array elements efficiently. The communication optimization techniques including the communication scheduling approaches proposed by Desprez *et al.* [4] for avoiding node contention. Hsu *et al.* [6] proposed the *Generalized Basic-Cycle Calculation* method to minimize the transmission overhead. Huang *et al.* [8] proposed a flexible processor mapping method to improve data locality. An efficient communication scheduling method [9] was proposed for processor mapping technique. To support compiled communication, an MPI prototype, *CC-MPI* [10], was proposed to allow user to manage network resources. Lim *et al.* [11] presented a general framework and discussed the direct, indirect, and hybird communication schedules for developing array redistributions. Sundarsan *et al.* [14] introduced a framework and derived an algorithm to redistribute two-dimensional block-cyclic arrays onto 2-D processor grids. With restrictions, contention free is guaranteed.

Two kinds of researches are focused on irregular array redistribution including the message generation and communication scheduling algorithms. Guo *et al.* [5] proposed communications optimization techniques for nested loops and introduced symbolic analysis algorithms. *HCS* and *HRS* [2] were proposed to improve data access on data grids. The communication optimization technique, called *local message reduction* (*LMR*) [3], was proposed to normalize the cost of communications and improve the scheduling results. The *two-phase degree reduction* (*TPDR*) was proposed by Hsu *et al.* [7] to minimize the communication for irregular array redistribution. *TPDR* employs two phases, the first phase is a degree reduction method and the second phase is a adjustable coloring mechanism. Both phases can successfully derive communication steps with the number of minimal steps. Based on previous work, Wang *et al.* [15] improved divide-and-conquer scheduling algorithm.

3 Preliminary

In Section 1, the code segment briefly introduces the processes of GEN_BLOCK redistribution. Processors represent the senders and receivers while changing phases of scientific applications. In such operation, communication scheduling algorithms are required for efficient GEN_BLOCK redistribution.

For scheduling algorithms, two policies are required to avoid contentions: First, a sender sends a message in one step. Second, a receiver receives a message in one step. The length[1] of each step is dominated by the message with largest data size. The length of total steps[2] represents the communication cost of the derived schedule for GEN_BLOCK redistribution.

To simplify the presentation, definitions are given as follows.

Definition 1: N is the number of processors. P_i are given to represent index of processors, where $0 \le i < N$.

Definition 2: The index of messages are given as m_j, where $0 \le j < 2N$-1.

Fig. 1 illustrates communication patterns on six processors on array for two phases according the code segment in Section 1. Numbers in parentheses beside $P_{0\sim5}$ are required data segment size in both phases. The arrows are the messages, $m_{1\sim11}$, represent the communication between each pair of nodes. Numbers in parentheses beside messages are the size of data segment needed to be transmitted. Scheduling algorithms must derive a schedule with at least four steps to perform GEN_BLOCK redistribution since P_2 has four messages to receive. The schedule with four steps is given in Fig. 2. The messages are arranged in steps with data size given in parentheses and

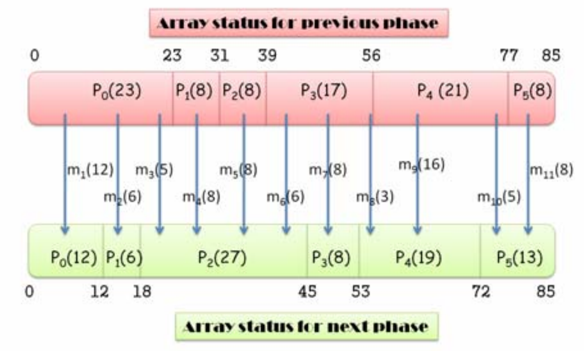


Fig. 1. Communication patterns on six processors on array for two phases

A four-step schedule		
No. of step	**No. of message**	**Cost of step**
Step 1	$m_1(12)$, $m_4(8)$, $m_8(3)$	**12**
Step 2	$m_2(6)$, $m_5(8)$, $m_{10}(5)$	**8**
Step 3	$m_3(5)$, $m_7(8)$, $m_9(16)$	**16**
Step 4	$m_6(6)$, $m_{11}(8)$	**8**
Total cost		**44**

Fig. 2. A schedule with four steps

[1] Length of a step is equal to the maximal data size of messages in this step.
[2] Length of total steps is the sum of length of all steps.

the cost of each step is easy to find. For example, m_1, m_4 and m_8 are scheduled in step 1 with length which is dominated by m_1. The length of total steps is 44 which is the sum of costs of each step.

4 The Proposed Method

Grids, which are connected by internet, have a variation of message categories. As described in introduction section, there are two clusters of messages, which are local data access and remote data access. The local data access represents the transmissions in memory while remote data access represents the transmissions between processors in single cluster. Grid, which inherit characteristics of single cluster, also classify messages into local data access and remote data access. In addition, the distant data access is defined to represents the transmissions between clusters. Therefore, distant access time (*DAT*) is defined for distant data access and the ratio of distant to remote access time (*DRR*) is defined as *DAT* divide by *RAT*.

Assume the number of nodes that are employed in each cluster is the same and the following definition is given.

Definition 3: Performing GEN_BLOCK redistribution on N processors between clusters in grids. The value NC represents the number of nodes in every C_i while C_i represents the ID of clusters, where $0 \le i \le \lfloor N/NC \rfloor$. $SpeedC_{i,j}$ represents the transmitting rate from C_i to C_j.

With above description, *DRR*, the ratio of distant to remote access time, is formulated as following equation:

$$DRR = \frac{DAT}{RAT} = \frac{SpeedC_{i,j}}{SpeedC_{i,i}} \tag{1}$$

Definition 4: Given communication patterns on processors on array for two phases; sender P_i and receiver $P_j \in C_r$, where $0 \le r \le \lfloor N/NC \rfloor$; message m_k is given, where $1 \le k \le 2N-1$. While sender P_i sends a message m_k to receiver P_j between clusters C_r, m_k represents the messages of local data access if $i = j$; m_k represents the messages of distant data access if P_i and P_j belong to different C_r; otherwise m_k represents remote data access.

The proposed *message clustering technique* (*MCT*) for multiple clusters in grids employs three scheduling levels which are defined as follows:

Process-1: The process which processes the arrangement of local data access.

Process-2: The process which is responsible for the arrangement of distant data access.

Process-3: The process which arranges remote data access.

Fig. 3 shows the effect of *DRR* by giving a schedule which is based on Fig. 2. The *NC* is assumed three, and then $P_{0\sim2}$ and $P_{3\sim5}$ are members of C_0 and C_1, respectively. The difference between two schedules in Fig. 2 and 3 is that m_6 is multiplied by *DRR* which is assumed five, and m_1, m_5, m_7, m_9 and m_{11} is divided by *LAT* which is assumed eight in Fig. 3. Two observations are found from the difference: (1) Dominators of all steps are changed. Effect of original dominators no longer exists. For example, m_4 replace m_1 in step 1; m_2 replace m_5 in step 2; m_3 replaces m_9 in step 3; m_{11} is replaced by m_6 in step 4. (2) Cost of m_6 is larger than half length of total steps. The above observations show the possibility that the hidden cost of messages can influence the quality of scheduling results.

A four-step schedule		
No. of step	**No. of message**	**Cost of step**
Step 1	m_1(1.5), m_4(8), m_8(3)	**8**
Step 2	m_2(6), m_5(1), m_{10}(5)	**6**
Step 3	m_3(5), m_7(1), m_9(2)	**5**
Step 4	m_6(30), m_{11}(1)	**30**
Total cost		**49**

Fig. 3. A schedule based on Fig. 2, the costs of m_4, m_5, m_6 and m_7 are changed

Fig. 4 is a schedule given by *MCT*. The result shows that scheduling local data access, remote data access and distant data access separately is helpful to arrange positions for messages and reduce total length of a schedule. Following the processes of *MCT*, the messages of local data access are selected first. The candidates are m_1, m_5, m_7, m_9 and m_{11}, and are arranged in step 4 for process-1. Then process-2 arranges m_6 in step 3. Other messages are scheduled by process-3 in step 1 and 2. The length is 45 and is better than the result in Fig. 3.

A schedule of MCT		
No. of step	**No. of message**	**Cost of step**
Step 1	m_2(6), m_4(8), m_8(3), m_{10}(5)	**8**
Step 2	m_3(5),	**5**
Step 3	m_6(30)	**30**
Step 4	m_1(1.5), m_5(1), m_7(1), m_9(2), m_{11}(1)	**2**
Total cost		**45**

Fig. 4. A schedule which is given by *MCT*

5 Performance Evaluation

To evaluate the performance of proposed methods, *MCT* were implemented along with *TPDR*. *NC* was given from 2~6 to evaluate the performance of *MCT* and *TPDR* on 32 nodes. There were 1,000 cases in each comparison; array size is 10,000 in each GEN_BLOCK distribution scheme; α and β represent two ranges of node size, and are shown in Fig. 5. The *Avg* represents the value of array size divided by *N*.

Size range of each node		
Symbol of ranges	**Lower bound**	**Upper bound**
α	0.5* *Avg*	**2 * *Avg***
β	**1**	**8 * *Avg***

Fig. 5. Lower bounds and upper bounds of two size ranges for each node

Fig. 6 shows the results of *MCT* and *TPDR* with α while *NC* was given from 2~6. Three kinds of plots are given in this section to weigh the pros and cons of both methods. The plots show *MCT* gives good performance in most cases with α. The results show that the *MCT* performs better while *NC* being smaller. The reason is that the smaller *NC* represents the larger number of distant data access which influences the estimation of *MCT* and *TPDR*. Unlike *TPDR*, the scheduling processes of *MCT* augment the effect of *DRR* for real situation. *MCT* also utilizes the idea of message clustering to arrange each kinds of message in specific level to decrease the length of schedules. Another phenomenon is observed that the performance of *MCT* is decreased while *NC* becoming larger due to less advantages for *MCT*.

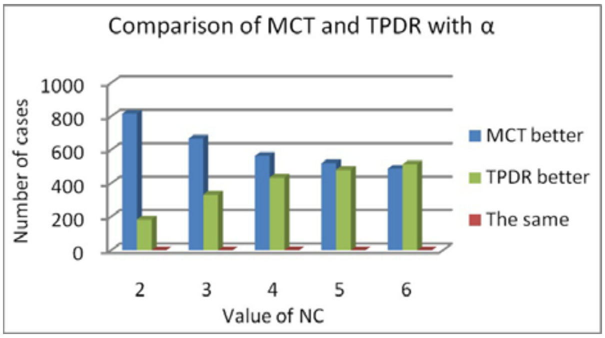


Fig. 6. The results of comparisons with α on various *NC*

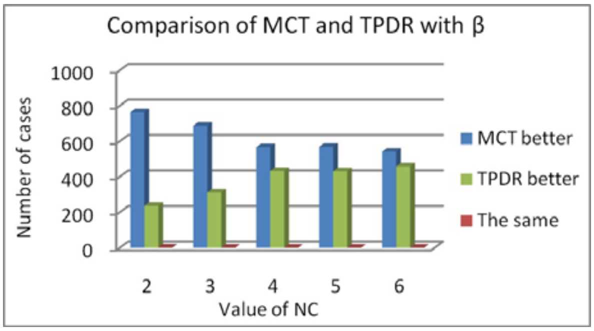


Fig. 7. The results of comparisons with β on various *NC*

The symbol β is expected to provide larger variation and more distant data access, which are advantageous to *MCT*. The effect of *DRR* is augmented by the variation of β and is utilized by *MCT* to derive better schedules for messages. With above advantages, Fig. 7 shows greater improvement in 56.7% to 76.3% cases by *MCT*.

6 Conclusions and Future Works

In this paper, we have presented *message clustering technique* (*MCT*) in grids to minimize the communication cost in `GEN_BLOCK` redistribution. The proposed technique adapts different data transmitting rate by applying *RLR* and *DRR*. Then, *MCT* schedules three clusters of messages in process-1, process-2 and process-3, respectively. The performance analyses show *MCT* performs better in most cases and adapts to high variation of `GEN_BLOCK` redistribution between clusters in grids. Future works of this technique are to develop improved algorithms for advanced computing, propose power saving technique and design better optimization technique.

References

1. Bai, S.-W., Yang, C.-S.: Essential Cycle Calculation Method for Irregular Array Redistribution. IEICE Transactions on Information and Systems E89-D(2), 789–797 (2006)
2. Chang, R.-S., Chang, J.-S., Lin, S.-Y.: Job scheduling and data replication on data grids. Future Generation Computer Systems 23(7), 846–860 (2007)
3. Chen, S.-C., Hsu, C.-H.: ISO: Comprehensive Techniques Toward Efficient GEN_BLOCK Redistribution with Multidimensional Arrays. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 507–515. Springer, Heidelberg (2007)
4. Desprez, F., Dongarra, J., Petitet, A.: Scheduling Block-Cyclic Data redistribution. IEEE Transactions on Parallel and Distributed Systems 9(2), 192–205 (1998)
5. Guo, M., Pan, Y., Liu, Z.: Symbolic Communication Set Generation for Irregular Parallel Applications. The Journal of Supercomputing 25(3), 199–214 (2003)
6. Hsu, C.-H., Bai, S.-W., Chung, Y.-C., Yang, C.-S.: A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution. IEEE Transactions on Parallel and Distributed Systems 11(12), 1201–1216 (2000)
7. Hsu, C.-H., Chen, S.-C., Lan, C.-Y.: Scheduling Contention-Free Irregular Redistribution in Parallelizing Compilers. The Journal of Supercomputing 40(3), 229–247 (2007)
8. Huang, J.-W., Chu, C.-P.: A flexible processor mapping technique toward data localization for block-cyclic data redistribution. The Journal of Supercomputing 45(2), 151–172 (2008)
9. Huang, J.-W., Chu, C.-P.: An Efficient Communication Scheduling Method for the Processor Mapping Technique Applied Data Redistribution. The Journal of Supercomputing 37(3), 297–318 (2006)
10. Karwande, A., Yuan, X., Lowenthal, D.K.: An MPI prototype for compiled communication on ethernet switched clusters. Journal of Parallel and Distributed Computing 65(10), 1123–1133 (2005)
11. Lim, Y.W., Bhat, P.B., Prasanna, V.K.: Efficient Algorithms for Block-Cyclic Redistribution of Arrays. Algorithmica 24(3-4), 298–330 (1999)
12. Lin, C.-Y., Chung, Y.-C.: Data distribution schemes of sparse arrays on distributed memory multicomputers. The Journal of Supercomputing 41(1), 63–87 (2007)

13. Prylli, L., Touranchean, B.: Fast runtime block cyclic data redistribution on multiprocessors. Journal of Parallel and Distributed Computing 45(1), 63–72 (1997)
14. Sudarsan, R., Ribbens, C.J.: Efficient Multidimensional Data Redistribution for Resizable Parallel Computations. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 182–194. Springer, Heidelberg (2007)
15. Wang, H., Guo, M., Wei, D.: Message Scheduling for Irregular Data Redistribution in Parallelizing Compilers. IEICE Transactions on Information and Sysmtes E89-D(2), 418–424 (2006)

Multithreading of Kostka Numbers Computation for the BonjourGrid Meta-desktop Grid Middleware

Heithem Abbes[1,2], Franck Butelle[2], and Christophe Cérin[2]

[1] Unité de recherche UTIC / ESSTT
5, Av. Taha Hussein, B.P. 56, Bab Mnara, Tunis, Tunisia
heithem.abbes@esstt.rnu.tn
[2] LIPN, UMR 7030, CNRS, Université Paris-Nord
99, avenue J.B Clément, 93430 Villetaneuse, France
franck.butelle@lipn.univ-paris13.fr,
christophe.cerin@lipn.univ-paris13.fr

Abstract. The aim of this paper is to show how to multithread a compute intensive application in mathematics (Group Theory) for an institutional Desktop Grid platform coordinated by a meta-grid middleware named BonjourGrid which is a fully decentralized Desktop Grid middleware. The paper is twofold: first of all, it shows how to multithread a sequential program for a multicore CPU which participates in the computation of some parameters and second it demonstrates the effort for coordinating multiple instances of the BonjourGrid middleware. The main results of the paper are: a) we develop an efficient multi-threaded version of a sequential program to compute Kostka numbers, namely the Multi-kostka program and b) a proof of concept is given, centered on user needs, for the incorporation into the BonjourGrid middleware of Multi-kostka program.[1]

Keywords: High-performance Scientific Computing, Parallel Algorithms, Parallel Scientific Application, Desktop Grids Case Study, Incorporation of a Threaded Application with Desktop Grid Infrastructures.

1 Introduction and Motivations

Desktop Grids have been successfully used to address large applications with significant computational requirements, including search for extraterrestrial intelligence (SETI@Home [20]), global climate predication (Climatprediction.net [19]),

[1] This work is supported by a grant of Regional Council of Ile-de-France under the SETCI program. We also thank Arthur Guittet for his participation in coding and testing the multithreaded approach. Experiments were conducted partly with the Gaia cluster at Paris 13 and partly with the Grid'5000 testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr).

and cosmic rays study (XtremWeb [9][14]). While the success of these applications demonstrates the potential of Desktop Grid, existing systems are often centralized and suffer from relying on an administrative staff who guarantees the execution with no faults for the master node. Moreover, although, in practice, the crash of the master is not frequent and replication techniques can resolve this problem when it occurs, we still believe in the need of decentralized approaches.

In this context, we have proposed a novel approach, called BonjourGrid [7,8], which allow us to establish a specific execution environment for each user. Indeed, BonjourGrid builds dynamically and in a decentralized way, a Computing Element (CE – a CE is a set of workers managed by one master or an instance of a local Desktop Grid middleware) when a user needs to run an application. BonjourGrid orchestrates multiple instances of CEs in a decentralized manner. Our approach does not rely on a unique static central element in the whole system, since we dedicate a temporary central element for each running application, in a dynamic way.

Furthermore, it is important to note that BonjourGrid is not only a decentralized approach to orchestrate and coordinate local DG (Desktop Grids) but it is also a system which is able, contrarily to classical DG, to build specific execution environment on-demand (based on any combination of XtremWeb, Boinc, Condor middlewares). This is the novelty of the BonjourGrid system. We consider that it is a step forward regarding Desktop Grid systems. As referenced in many other works [11,12], this kind of environment is called Institutional Desktop Grid or Enterprise Desktop Grid (e.g in the same institution).

The application that we investigate in this paper to realize a proof of concept based on a real application of our middleware is a compute intensive application in the field of Group Theory. To our knowledge, there has been no attempt in the past to derive a multi-threaded solution for the computation of Kostka numbers on Desktop Grids, especially for Desktop Grid based on multi-core processors. We derive an original and efficient solution to this difficult problem because it is hard to predict on the fly the space where the solution is located in, as we will see later in the paper.

Therefore, the paper is mainly organized according to two central discussions: first of all the multithreading of the computation of Kostka numbers (section 2) and second the integration of the solution into BonjourGrid (section 3). A related work section follows as well as a conclusion section. The typical use case underlying our work is the following: a mathematician needs to guess the property of some numbers. Usually he uses the PCs in his institution to 'compute' the properties of the objects with the sequential implementation of Schur[2] while colleagues work on others problems. He would like to go further with the properties he guesses, so he realizes that he is able to use the grid power. He also requires that it could be done with a minimal effort for him (minimum of deployment, minimum of code lines to launch the application) because he is not an expert in grid middleware. This user belongs to the community of Boinc users, so he wants to use this middleware whereas one of his colleague belongs to the Condor community.

[2] See http://www.sourceforge.net/projects/schur/

Our solution allows any user to run the multi-threaded release of Kostkas numbers computation that we introduce in this paper on its favorite middleware (BOINC, CONDOR, XtremWeb), concurrently with other colleagues working on the same problem i.e. with the same multithreaded code but with a different Desktop Grid Middleware.

This idea, while closely related to the concept of "Infrastructure on demand" concept, which is central to the Cloud Computing paradigm and thus not completely new, may help in widening the spread of the Desktop Grid approach. This article strives to demonstrate the feasibility of the solution using an application inherited from mathematics namely the computation of Kostka numbers because we still need to validate BonjourGrid through 'real applications' and not emulation as we have already done. We choose the Kostka application because it is used by people in our laboratory, and because it is a central piece in the Schur package that is a tool to 'guess', for instance, properties of symmetric functions. Note also that the problem is not a data intensive problem since we need to pass only few numbers to the main procedure. It is only a compute intensive problem.

Again, the main result of this paper is not about a new Desktop Grid middleware because in this case we need also to address, for the sake of completeness, the problems of volatility (host churn) and heterogeneity but it is about a multithreaded code for the computation of Kostka numbers. The ultimate goal is to demonstrate that BonjourGrid is able to coordinate multiple instances of the same application on multiple instances of grid middlewares deployed over an institution and also that a non specialist user may use this new environment. The paper is devoted to show how it might be simple to deploy applications with BonjourGrid and that the experiments could be efficient if we have at our disposal a multi-treaded code.

2 A Multithreaded Computation for Kostka Numbers

The aim of this section is to explain the different ways we compute the Kostka numbers in parallel. Two kinds of parallelization is under concern in this article which are the multithreading on a single multicore CPU and the parallelization at the Desktop Grid level. The goal is to replace the current sequential procedure included in Schur[3] by a multi-threaded one able to be integrated into a DG middleware. Schur is recognized to be a powerful and efficient tool for calculating properties of Lie groups and symmetric functions. Developing a parallel version of Schur is challenging and we restrict our study to the computation of Kostka numbers in parallel.

The mathematical problem may be reduced (thanks to the "hives model", see [3]) to counting the number of integer points inside a polytope. The problem is also related to the *partition* of an integer. A *partition* of a positive integer N is a way of writing N as a sum of non increasing strictly positive integers. For example $\lambda = (4, 2, 2, 1)$ and $\mu = (2, 1)$ are partitions of $n = 9$ and $n' = 3$ respectively. We write $\lambda \vdash n$ and $\mu \vdash n'$ or $|\lambda| = n$ and $|\mu| = n'$.

[3] See http://www.sourceforge.net/projects/schur/

Littlewood-Richardson coefficients $c^{\nu}_{\lambda\mu}$, which are closely related to Kostka numbers, have a polynomial growth with respect to the dilatation factor $N \in \mathbb{N}$:

$$c^{N\nu}_{N\lambda,N\mu} = P^{\nu}_{\lambda\mu}(N) \quad ; \quad P^{\nu}_{\lambda\mu}(0) = 1 \tag{1}$$

where $P^{\nu}_{\lambda\mu}$ is a polynomial in N with non negative rational coefficients depending on λ, μ and ν. This was first conjectured in [3]. Polynomials are obtained considering a model known as the hive model. An n-**integer-hive** is a triangular array of non negative integer variables a^i_j with $0 \leq i, j \leq n$ where neighboring entries have to respect inequality constraints (known as the "three rhombus").

The Littlewood-Richardson coefficient $c^{\nu}_{\lambda\mu}$ is the number of solutions for these variables a^i_j with border labeled as above by λ, μ and ν and according to the inequalities constraints. Then, Kostka numbers (a particular case of Littlewood-Richardson coefficients) are obtained by an enumerative process based on the building of hives (each obtained from λ, μ, ν and N parameters) until we converge to a 'stabilization' property.

2.1 The Multithreading Approach

The way we are computing the Kostka numbers allows to split the enumerative process of the feasible space in several parts. To do this, we use the Pthread API, and divide the enumerative process in two pieces:

1. We assume that only n threads are allowed for the process. The first step of the enumeration is to divide the feasible space in n parts. Since the enumeration space is made of a product of intervals, we divide the first interval in $min(nb\_thread, interval\_size)$ of the same size, and for each part we attach a thread. Note that we have also tested others splitting, but the experimental results were not so good.
2. Each thread uses an optimized method to explore the feasible space and returns the number of solutions. The method is an heuristics based on taking the first interval not reduced to a single value. Once the enumeration is done for all the threads, the main thread sends the result to the user.

2.2 Parallelization Using a Desktop Grid

The computation of Kostka polynomials can be achieved using multiple machines on a local network. The idea is to compute the Littlewood-Richardson coefficients on various machines for a set of dilatation coefficients and to collect the results to build the interpolating polynomial. The time needed to compute one Kostka number depends on the initial partitions given to the program and on the dilatation coefficient.

The four input parameters (λ, μ, ν, N) are given to the program, where $N \in \mathbb{N}$ is the dilatation coefficient. For each computer on the network, a unique dilatation coefficient is given. The result of each computation is the Littlewood-Richardson coefficient $c^{N\nu}_{N\lambda,N\mu}$. Thus many interpolation points $(N, c^{N\nu}_{N\lambda,N\mu})$ are computed. Each time a new interpolation point is produced, a new interpolating polynomial is computed using all the available values. Thus, a set of polynomials is built of the form:

$$(P_{i}{}^{\nu}_{\lambda\mu})_{i\in\mathbb{N}} = P_{1}{}^{\nu}_{\lambda\mu}(N),\ P_{2}{}^{\nu}_{\lambda\mu}(N),\ \cdots,\ P_{k}{}^{\nu}_{\lambda\mu}(N),\cdots \tag{2}$$

The computation is stopped when the set becomes stationary. That means that the equality $P_{i}{}^{\nu}_{\lambda\mu} = P_{(i+1)}{}^{\nu}_{\lambda\mu}$ becomes true, and it holds for any j greater than i. The main limit of the parallel computation for Kostka numbers is found here. In fact, the complexity of the algorithm used to compute each Littlewood-Richardson coefficient has the following writing:

$$c(N) = \left(\prod_{k=1}^{m} |I_k|\right) N^m \tag{3}$$

where m is the number of intervals in the hive (number of variables), $|I_k|$ is the number of elements in the interval I_k (the interval of values that the corresponding variable may have) and N is the dilatation coefficient. Thus, the time needed to compute a Littlewood-Richardson coefficient has a polynomial growth of order m (see figure 1). This means that for high values of m, the time needed to compute the coefficient $N+1$ will be greater than the time needed to compute all the coefficients up to N. In other words, for such values of m, the time needed for the whole computation (all the coefficients) is more or less the time needed to compute the last coefficient. This explain why we need a finer grain.

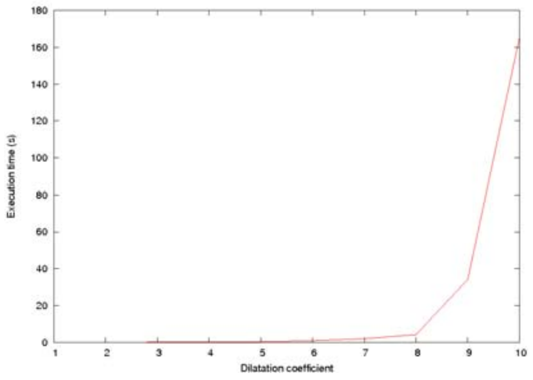


Fig. 1. Polynomial growth of the computation time with respect to the dilatation coefficient. The partitions are $\lambda = (5,3,2,2,1)$, $\mu = (4,3,2,2,1)$ and $\nu = (7,5,4,4,3,2)$.

2.3 Experimental Results

Performance gain on a single machine. The use of threads to compute the Littlewood-Richardson coefficients leads to an improvement in the execution time. To compare the gain in performance, we use a multi-core processor. In the remainder of the section, computation are done on a Bi-AMD Opteron dual core at 2.8GHz, which means that 4 cores were available for the computation. The computation of Littlewood-Richardson coefficients was done for various partitions and variable maximum number of threads. The first remark is that the gain in time depends on the first interval (which is distributed among the threads) and on the symmetry of the problem. For example if the first interval is a single value, only one thread is launched, and the time needed for the computation is the same than without threads. So we use the first interval of size more than one (the corresponding variable is not yet fixed). The symmetry of the feasible space is also important for the enumeration. To understand this, the algorithm has to be explained again. As said before, the enumeration method used by each thread is optimized, so that the whole feasible space is not searched. Each step of the algorithm corresponds to one point of the hive. For each feasible value of a point (a variable is fixed), the feasible space is updated (reduced by constraints) so that useless values are not tried. Therefore, the number of feasible values tried by each thread depends on the interval it received from the first step, and some values inside these interval will introduced more or less a reduction of the feasible space in the next steps. Consequently, some threads will take some more time to achieve their work than others. This phenomenon can be observed on Figure 2, which shows the results for the partitions $\lambda = (5, 5, 3, 2, 1, 1)$, $\mu = (6, 6, 4, 2, 1)$ and $\nu = (6, 6, 6, 5, 5, 3, 2, 2, 1)$. The computation time for the dilatation coefficient $N = 7$ does not decrease when the number of threads increases.

However, the majority of the practical cases shows that threading the program is useful. On the average, the computation time is divided by a factor of 3.5, and nearly four in the best cases. Increasing the number of threads has a limited interest: we can see on the left diagram of figure 2 that the time needed for the computation time is stationary for a number of threads greater than 10. This is explained by the fact that one of the threads is slower than the others because the number of values it deals with is bigger than for the other threads.

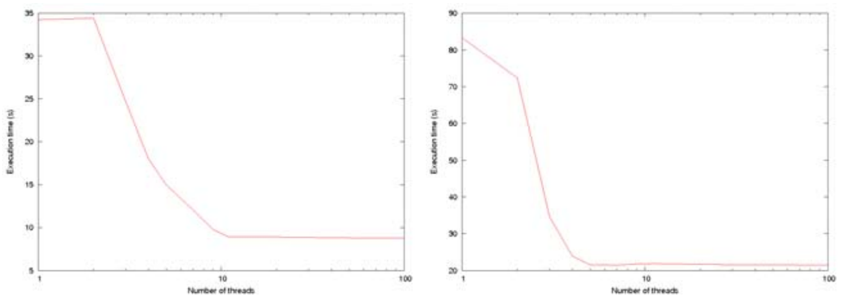


Fig. 2. Evolution of the computation time needed with the number of threads allowed for partitions $(\lambda = (5, 3, 2, 2, 1), \mu = (4, 3, 2, 2, 1), \nu = (7, 5, 4, 4, 3, 2))$ and $(\lambda = (7, 6, 5, 4), \mu = (7, 7, 7, 4), \nu = (12, 8, 8, 7, 6, 4, 2))$

For a very large number of threads, the performance mainly depends on the operating system's scheduler. We are using the Linux scheduler, and even with a great number of threads (over 200 threads), the results are impressive. Note also that our application is not a data intensive one since it requires only few integer parameters as input.

3 Porting the Parallel Code on a Desktop Grid Platform

BonjourGrid [7,8] is a meta Desktop Grid middleware developed in our research team that it is able to instantiate multiple Desktop Grid middleware in the same infrastructure. The principle of the proposed approach is to create, dynamically and in a decentralized way, a specific execution environment for each user to execute any type of applications without any system administrator intervention. An environment do not affect another one if it fails.

In this paper, we choose to use XW (XtremWeb) as the middleware for the Computing Element. To run or deactivate an XW service (coordinator or worker), the environment should be already installed. The procedure to install a XW-Coordinator, in particular, is not currently simple enough, so we have improved it. We would not want the user spends time configuring and installing files and modules necessary to the XW installation. An installation consists in installing a MySQL server, Java Development Kit, creating a specific database for XW, making several directories and configuring system files. Consequently, we set up an automatic installation of all the necessary packages. Such facilities were not included in the current distribution of XW.

3.1 Experiments and Validation of BonjourGrid

From a user point of view, deploying an application on BonjourGrid starts with preparing the executable code and data that are archived in a compressed file. BonjourGrid enables the execution of applications with precedences between tasks. Indeed, the user can describe the precedences constraints of a data flow graph using an XML description; this is not trivial, especially, for complex applications.

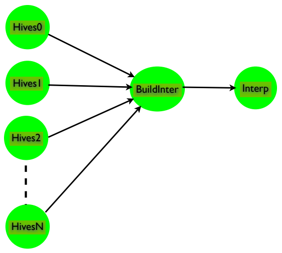


Fig. 3. Description of the data flow graph of Hives composed with 3 modules; Hives, BuildInter and Interp

```
<Deployment>
    <Application ApplicationDescription="monapp" Client="heithem">
         <Module ModuleDescription="hives" ModuleDirIn="/bin/">
             <Binary BinaryCpuType="amd64" BinaryExecutable="hives"
                     BinaryOsName="Linux" BinarySubFolder="linux"/>
         </Module>
         <Module ModuleDescription="buildinter" ModuleDirIn="/bin/">
             <Binary BinaryCpuType="amd64" BinaryExecutable="buildinter"
                     BinaryOsName="Linux" BinarySubFolder="linux"/>
         </Module>
         <Module ModuleDescription="interp" ModuleDirIn="/bin/">
             <Binary BinaryCpuType="amd64" BinaryExecutable="interp"
                     BinaryOsName="Linux" BinarySubFolder="linux"/>
         </Module>
    </Application>
```

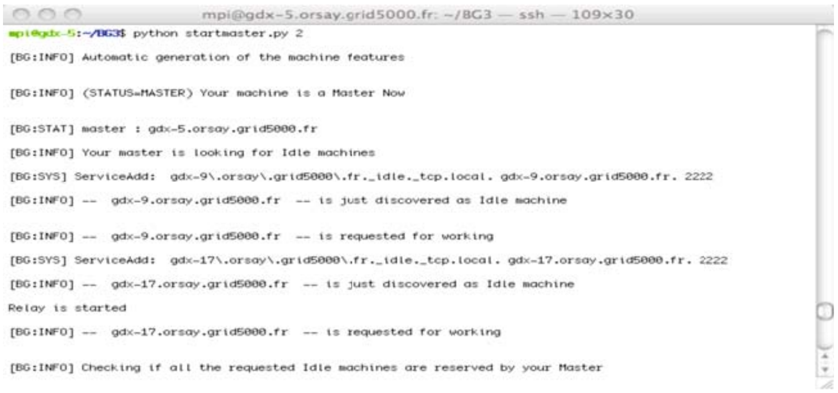
Fig. 4. Description of the flow data graph of the application Hive

SDAD is the system of deployment that we have introduced, it helps the user in describing the data flow graph of his application according to an XML syntax and using a graphical interface. It also helps user to put the different files of his applications (i.e binary and data files) at the right places. SDAD generates a compressed package ready to be deployed on BonjourGrid system. Now, the user can submit his application to BonjourGrid. He can specify the size of the computing element (CE) to run the application. BonjourGrid, then, will construct the CE according to the criteria mentioned in the XML file.

To summarize, using the SDAD tool that we have developed, the user can draw the task graph and put binary and data files in the suitable path in the application tree. Thereafter, SDAD will generate the XML description of the application which is used by XW and BonjourGrid.

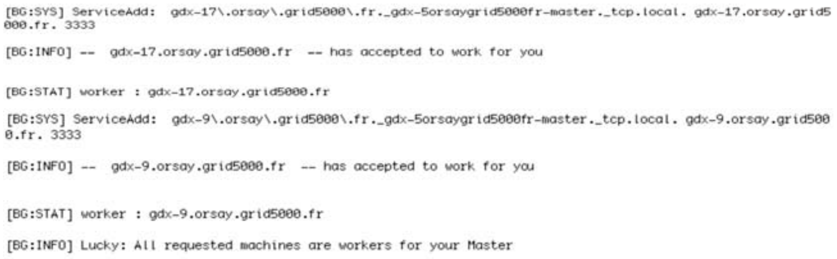
As described in subsection 2, the first module of the application, Hives, can be divided in several parallel tasks. The outputs of these tasks are forwarded to the so called BuildInter module. Finally, the module Interp gives the interpolation. Figure 4 shows the first part of this file which illustrates the description of the three modules of the application (i.e, OS architecture, OS type, location of binary files. . .). Here, the user can provide several binary files for different architectures and OS types.

Now, Hives is ready to be submitted to the BonjourGrid system. In the following, we illustrate snapshots picked out from a Hives execution using the Orsay node of Grid5000 [18]. Specifically for this example, we are going to dissociate the CE building phase from the effective submission to illustrate the different steps. First, we initiate machines in idle state. We launch the coordinator, on any machine, to start the building phase of a suitable CE. Figure 5 shows the outputs of a construction of a CE with 2 workers (just for the sake of clarity of reading). Indeed, the coordinator is started on the `gdx-5` node, requiring two workers as shown on Figure 5. For that, the coordinator gdx-5 searches for idle machines matching the tasks requirements. Figure 5 shows that the coordinator gdx-5 discovers, in this test, two idle machines `gdx-9` and `gdx-17` and asks them to accept to work for it. Thereafter, the `gdx-5` coordinator receives two confirmations from `gdx-17` and `gdx-9` as depicted in figure 6. On the coordinator machine `gdx-5`, we effectively submit Hives application as shown in figure 7. It is possible to control the execution of the different tasks of Hives application as



```
mpi@gdx-5.orsay.grid5000.fr: ~/BG3 — ssh — 109×30
mpi@gdx-5:~/BG3$ python startmaster.py 2
[BG:INFO] Automatic generation of the machine features

[BG:INFO] (STATUS=MASTER) Your machine is a Master Now

[BG:STAT] master : gdx-5.orsay.grid5000.fr
[BG:INFO] Your master is looking for Idle machines
[BG:SYS] ServiceAdd:  gdx-9\.orsay\.grid5000\.fr._idle._tcp.local. gdx-9.orsay.grid5000.fr. 2222
[BG:INFO] --  gdx-9.orsay.grid5000.fr  -- is just discovered as Idle machine

[BG:INFO] --  gdx-9.orsay.grid5000.fr  -- is requested for working
[BG:SYS] ServiceAdd:  gdx-17\.orsay\.grid5000\.fr._idle._tcp.local. gdx-17.orsay.grid5000.fr. 2222
[BG:INFO] --  gdx-17.orsay.grid5000.fr  -- is just discovered as Idle machine
Relay is started
[BG:INFO] --  gdx-17.orsay.grid5000.fr  -- is requested for working

[BG:INFO] Checking if all the requested Idle machines are reserved by your Master
```

Fig. 5. Construction of a new CE with 2 workers



```
[BG:SYS] ServiceAdd:  gdx-17\.orsay\.grid5000\.fr._gdx-5orsaygrid5000fr-master._tcp.local. gdx-17.orsay.grid5
000.fr. 3333
[BG:INFO] --  gdx-17.orsay.grid5000.fr  -- has accepted to work for you

[BG:STAT] worker : gdx-17.orsay.grid5000.fr
[BG:SYS] ServiceAdd:  gdx-9\.orsay\.grid5000\.fr._gdx-5orsaygrid5000fr-master._tcp.local. gdx-9.orsay.grid500
0.fr. 3333
[BG:INFO] --  gdx-9.orsay.grid5000.fr  -- has accepted to work for you


[BG:STAT] worker : gdx-9.orsay.grid5000.fr
[BG:INFO] Lucky: All requested machines are workers for your Master
```

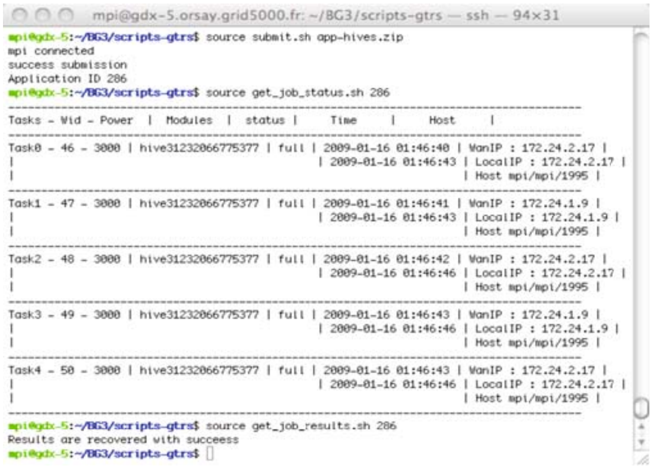
Fig. 6. Confirmation of two idle machines to work for the gdx-5 master

shown in figure 7. When the execution completes, we can invoke BonjourGrid to download the results.

To summarize, the objective of this experiment was to present a use case of BonjourGrid using a real application and from the user point of view. We have already done experiments to analyze the performance of BonjourGrid in [7,8] and [5] but the scope of this section is to demonstrate how BonjourGrid can help users to construct, dynamically and without any intervention of a system administrator, their own environments to deploy and running a parallel application.

4 Related Work on Advanced Desktop Grid Architectures

Before concluding, we compare BonjourGrid with others systems. OurGrid [13] system avoids the centralized server by creating the notion of the home machine from which applications are submitted; the existence of several home machines reduces the impact of failures at the same time. Moreover, OurGrid provides an accounting model to assure a fair resources sharing in order to attract nodes to join the system. However, the originality of BonjourGrid comparing to OurGrid



```
mpi@gdx-5.orsay.grid5000.fr: ~/BG3/scripts-gtrs — ssh — 94×31
mpi@gdx-5:~/BG3/scripts-gtrs$ source submit.sh app-hives.zip
mpi connected
success submission
Application ID 286
mpi@gdx-5:~/BG3/scripts-gtrs$ source get_job_status.sh 286
--------------------------------------------------------------------------------------
Tasks - Wid - Power  |  Modules  |  status |      Time       |      Host      |
--------------------------------------------------------------------------------------
Task0 - 46 - 3000 | hive31232066775377 | full | 2009-01-16 01:46:40 | WanIP : 172.24.2.17 |
|                                              | 2009-01-16 01:46:43 | LocalIP : 172.24.2.17 |
|                                                                     | Host mpi/mpi/1995 |
--------------------------------------------------------------------------------------
Task1 - 47 - 3000 | hive31232066775377 | full | 2009-01-16 01:46:41 | WanIP : 172.24.1.9 |
|                                              | 2009-01-16 01:46:43 | LocalIP : 172.24.1.9 |
|                                                                     | Host mpi/mpi/1995 |
--------------------------------------------------------------------------------------
Task2 - 48 - 3000 | hive31232066775377 | full | 2009-01-16 01:46:42 | WanIP : 172.24.2.17 |
|                                              | 2009-01-16 01:46:46 | LocalIP : 172.24.2.17 |
|                                                                     | Host mpi/mpi/1995 |
--------------------------------------------------------------------------------------
Task3 - 49 - 3000 | hive31232066775377 | full | 2009-01-16 01:46:43 | WanIP : 172.24.1.9 |
|                                              | 2009-01-16 01:46:46 | LocalIP : 172.24.1.9 |
|                                                                     | Host mpi/mpi/1995 |
--------------------------------------------------------------------------------------
Task4 - 50 - 3000 | hive31232066775377 | full | 2009-01-16 01:46:43 | WanIP : 172.24.2.17 |
|                                              | 2009-01-16 01:46:46 | LocalIP : 172.24.2.17 |
|                                                                     | Host mpi/mpi/1995 |
--------------------------------------------------------------------------------------
mpi@gdx-5:~/BG3/scripts-gtrs$ source get_job_results.sh 286
Results are recovered with succeess
mpi@gdx-5:~/BG3/scripts-gtrs$
```

Fig. 7. Submission of the application "app-hives" using the new CE

is that it supports distributed applications with precedence between tasks, while OurGrid supports only Bag-of-Tasks (BOT) applications (BOT applications are independent divisible tasks). WaveGrid [6] is a P2P middleware which uses a time-zone-aware overlay network to indicate when hosts have a large block of idle time. This system reinforces the idea of BonjourGrid concept since changing from a set of workers to another one depending on the time zone (Wave Grid) is analogous to the principle of creating a CE from an application to another one in BonjourGrid, and depending on users requirements.

Approaches based on publish/subscribe systems to coordinate or decentralize Desktop Grid infrastructures are not very numerous according to our knowledge. A similar project to ours is the Xgrid project [17]. In Xgrid system, each agent (or worker) makes itself available to a single controller. It receives computational tasks and returns the results of these computations to the controller. Hence, the whole architecture relies on a single and static component which is the controller. Moreover, Xgrid runs only on MacOS systems. In contrast with Xgrid, in BonjourGrid, the coordinator is not static and is created in a dynamic way. Furthermore, BonjourGrid is more generic since it is possible to "plug" inside it any computing system (XW, Boinc, Condor) while Xgrid has its own computing system. The key advantage of BonjourGrid is that the user is free to use his favorite desktop grid middleware.

5 Conclusion and Future Works

In this work, we have proposed a novel algorithm to compute Kostka numbers (by the Hive method) in parallel and we have shown how to run the code on top of BonjourGrid. The aim of BonjourGrid is to orchestrate multiple instances of computing elements, in a decentralized manner. BonjourGrid creates, dynamically, a specific environment for the user to run his application. There is no need for a system administrator. Indeed, BonjourGrid is fully autonomous and decentralized. We have conducted several experimentations to show that BonjourGrid operates well with a real world application. At this occasion, BonjourGrid demonstrate its usefulness. Moreover, the deployment of the Hive code on BonjourGrid, demonstrates that BonjourGrid may help users to create several independent environments. Then, mathematicians, for instance, can create their Desktop Grid easily to run their parallel applications.

So, we demonstrated that the approach for developping scientific applications on such infrastructure is, in fine, as simple as we introduced it in the paper and general enough to be applied in other contexts.

Several issues must be taken into account in our future work about meta-Desktop Grid middlewares. The first issue is to build a fault-tolerant system for the coordinators. In fact, it is important to continue the execution of the application when the coordinator (user machine) fails (it is disconnected for instance). This issue has been solved and it is currently tested over Grid'5000 and at a large scale. Condor, Boinc plugins have also been built and are currently tested with the Hive code, especially but not only. The second issue is the reservation of participants. In the current version, BonjourGrid allocates available resources for a user without any reservation rules. Thus, BonjourGrid may allocate to a single user all the available resources. The third issue is to go to a wide area network. The current version works only in a local network infrastructure, and it is important to bypass this constraint. The new release of Bonjour, (Wide Area Bonjour from Apple), seems to be a good solution to solve this problem.

We hope that this work helps to understand what is a possible future for Desktop Grid middleware as well as the new efforts that users should make to use such systems.

References

1. Rassart, E.: A polynomiality property for Littlewood-Richardson coefficients. J. Combinatorial Theory, Ser. A 107, 161–179 (2004)
2. MacDonald, I.G.: Symmetric functions and Hall Polynomials, 2nd edn. Clarendon Press/Oxford Science Publication (1995)
3. King, R.C., Tollu, C., Toumazet, F.: Stretched Littlewood-Richardson and Kostka coefficients. In: CRM Proceedings and Lecture Notes, vol. 34, pp. 99–112. Amer. Math. Soc., Providence (2004)
4. Steinberg, D., Cheshire, S.: Zero Configuration Networking: The Definitive Guide. O'Reilly Media, Inc., Sebastopol (2005)
5. Abbes, H., Dubacq, J.-C.: Analysis of Peer-to-Peer Protocols Performance for Establishing a Decentralized Desktop Grid Middleware. In: César, E., et al. (eds.) EuroPar 2008/SGS Workshop. LNCS, vol. 5415, pp. 235–246. Springer, Heidelberg (2009)
6. Zhou, D., Lo, V.: WaveGrid: a scalable fast-turnaround heterogeneous peer-based Desktop Grid system. In: IPDPS'20. IEEE Computer Society, Los Alamitos (2006)

7. Abbes, H., Cérin, C., Jemni, M.: BonjourGrid as a Decentralised Job Scheduler. In: APSCC 2008: Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference. IEEE Computer Society, Los Alamitos (2008)
8. Abbes, H., Cérin, C., Jemni, M.: BonjourGrid: Orchestration of Multi-instances of Grid Middlewares on Institutional Desktop Grids. In: 3rd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2009), in conjunction with IPDPS 2009, Roma, Italy, May 29 (2009)
9. Fedak, G., Germain, C., Néri, V., Cappello, F.: Xtremweb: A generic global computing system. In: Proceedings of IEEE Int. Symp. on Cluster Computing and the Grid (2001)
10. Thain, D., Livny, M.: Condor and the grid. In: Berman, F., Hey, A.J.G., Fox, G. (eds.) Grid Computing: Making The Global Infrastructure a Reality. John Wiley, Chichester (2003)
11. Domingues, P., Andrzejak, A., Silva, L.M.: Using Checkpointing to Enhance Turnaround Time on Institutional Desktop Grids. In: e-Science, p. 73 (2006)
12. Kondo, D., Chien, A.A., Casanova, H.: Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. In: SC 2004, p. 17. IEEE Computer Society, Los Alamitos (2004)
13. Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., Mowbray, M.: Labs of the World, Unite!!! Journal of Grid Computing, 225–246 (2006)
14. Cappello, F., Djilali, S., Fedak, G., Herault, T., Magniette, F., Néri, V., Lodygensky, O.: Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. FGCS 21(3), 417–437 (2005)
15. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
16. Bonjour protocol, `http://developer.apple.com/networking/bonjour`
17. Xgrid, `http://gcd.udl.cat/upload/recerca/`
18. Grid'5000, `http://www.grid5000.fr`
19. BOINC, `http://boinc.berkeley.edu`
20. Seti@home, `http://setiathome.ssl.berkeley.edu`

Adaptable Scheduling Algorithm for Grids with Resource Redeployment Capability*

Cho-Chin Lin and Chih-Hsuan Hsu

Department of Electronic Engineering
National Ilan University
Yilan 260, Taiwan
cclin@niu.edu.tw

Abstract. Two distinct characteristics of grid computing systems are resource heterogeneity and availability variation. There are many well-designed scheduling algorithms proposed for heterogeneous computing systems. However, the availability variation is seldom considered in developing scheduling ongoing applications on a grid. In this paper, a scheduling algorithm called AROF is proposed. It considers availability variation as well as resource heterogeneity while scheduling applications to the grids. An experiment has been conducted to demonstrate that AROF algorithm outperforms the well-known scheduling algorithms Modified HEFT and GS in most of the cases.

1 Introduction

Many well-designed scheduling algorithms [1,2,3,5,6,7,8,9,10,11,12,13,14] have been proposed for distributed systems for achieving high performance computing. The computing resources in a grid are generally heterogeneous and, thus, most of the scheduling algorithms have taken resource heterogeneity into consideration. When an application is submitted to a grid, the application starts its work as soon as the requested resources are available. After it has completed its work, the requested resources will be released. Thus, the number of idle computing nodes which can be deployed to an ongoing application varies with the time. We use the term *availability variation* to indicate this phenomenon. To the best of our knowledge, most of the scheduling algorithms do not consider the availability variation in developing their scheduling strategies. The scheduling algorithms which are sensitive to the variation in computing nodes is named as adaptable scheduling (AS) algorithm. An AS algorithm needs to take resource heterogeneity and availability variation into consideration while scheduling the tasks of an ongoing application to computing nodes. A traditional scheduler schedules all the tasks of an application to computing nodes before any computation can be performed. Thus, the application cannot employ the computing nodes which are recently released from other complete applications even though

* This research is supported by National Science Council under the grant 98-2221-E-197-014.

it still remains lots of computations to be performed. Contrary to the traditional schedulers, an AS algorithm schedules an application in several iterations. Each iteration maps the tasks of an ongoing application to the available computing nodes based on a predefined criterion. By this way, an AS algorithm responds to the availability variation and the work quality needed by the application can be satisfied.

Except the case of releasing computing nodes voluntarily, availability variation can happen when a grid redeploys computing nodes for a set of ongoing applications in response to an urgent event. The grid which employs AS algorithms to *reinforce* (*revoke*) computing nodes to (from) an ongoing application is called R^2 grid. Scheduling an application to computing nodes of a R^2 grid takes several iterations. Each iteration consists of three steps: selection step, ranking step and mapping step. The tasks which are qualified for mapping are collected in the selection step. Then, the selected tasks are assigned priorities in the ranking step. Finally, the mapper maps the selected tasks one by one to available computing nodes based on the assigned priorities. This paper studies the performance which can be achieved by our AS algorithm by comparing its performance with those of other scheduling algorithms. The major contributions of this paper are twofold: a framework of AS algorithms is developed and the task selection strategies employed by various scheduling algorithms are assessed.

This paper is organized as follows. In Sect. 2, the models of workload and R^2 grids are defined. In Sect. 3, related work is discussed. In Sect. 4, a framework of AS algorithms is developed. An AS algorithm called AROF is proposed in Sect. 5. In Sect. 6, the usefulness of AROF is demonstrated by comparing its performance with those of the well-known scheduling algorithms GS and modified version of HEFT. Finally, concluding remarks are given in Sect. 7.

2 Models

In this paper, an application is characterized by a workflow and the grid allows computing nodes to be reinforced (revoked) to (from) an ongoing application. We present our workload and grid models in this section.

2.1 Workload Model

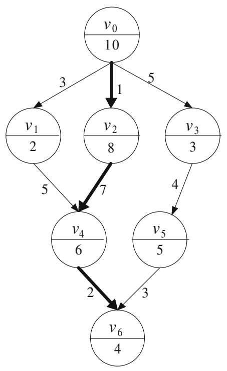
A workflow application can be modeled using a diagraph $G = (V, E)$, where V is a vertex set and E is an edger set. Each vertex in $V = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$ represents a task in the application. The number of tasks in V is denoted as $|V|$. The unique entry task and exit task of an application is vertex v_0 and vertex v_{n-1} in the diagraph, respectively. The weight of task v_i is denoted as $w(v_i)$. Edge $(v_i, v_j) \in E$ if there is a dependency between tasks v_i and v_j. In this case, v_i is a predecessor of v_j and v_j is a successor of v_i. The weight of edge (v_i, v_j) is the number of data units sent from v_i to its successor v_j. Denote $w((v_i, v_j))$ as the weight of the edge (v_i, v_j). Task v_i, $i \neq 0$, gets inputs from its predecessors and stores its output at the computing node where it is mapped. Our model assumes that a task needs to receive inputs from all of its predecessors before it proceeds to perform computations. This paper focuses on the applications with one entry (exit) task. However, any application with more than one entry (exit) task can be modified by adding a *virtual* task above (below) the actual entry (exit) tasks with zero weight edges connecting the virtual task to the entry (exit) tasks. Denote the start time and complete time of task v_i as t_i^s and t_i^f, respectively. The makespan of an application is given by $t_{n-1}^f - t_0^s$,

2.2 R^2 Grid Model

The gird model captures two significant properties: resource heterogeneity and availability variation. Resource heterogeneity means that some computing nodes have special hardware for accelerating the computations of special types. Availability variation means that the set of computing nodes which are available to an ongoing application can be variant. Let τ_{ij}^{cmp} be the cost of executing task v_i on computing node p_j. Then, $\tau_{ij}^{cmp} = w(v_i)/r(p_j)$, where $r(p_j)$ is the processing speed of computing node p_j. Regarding the availability variation, the available computing nodes may vary while an application continues its work. Let $\mathcal{P}$ be a sequence of variation events $(P_0, P_1, P_2, \ldots, P_{s-1})$, where the ith event P_i is the set containing the nodes available at time interval $[t_i, t_{i+1})$. Variation event P_i provides an opportunity for an AS algorithm to reinforce (revoke) computing nodes to (from) an ongoing application at time interval $[t_i, t_{i+1})$. In the sequence, $P_i \neq P_{i+1}$. This study assumes that either $P_i \subset P_{i+1}$ or $P_{i+1} \subset P_i$ for $0 \leq i < s-1$. The scheduler in our model also decides the execution order of the tasks on a computing node. When a computing node is revoked from an application, all the tasks in the computing node migrate to other available computing nodes except the one which is currently performing computations. Nowadays, many techniques for end-to-end QoS guarantee to an individual network have been proposed [4,15]. Our R^2 grid model characterizes that a pair of communicating nodes can reserve the required network bandwidth before they start to communicate. Based on this, it is assumed that a computing node can send and receive data concurrently, and it also can send (receive) data to (from) multiple computing nodes simultaneously. Similar communication assumptions are also used in [1,6,7,8,9,11,12]. Let the bandwidth between computing nodes p_i and p_j as b_{ij}. Define $\tau_{ij}^{cmm} = w((v_i, v_j))/b_{ij}$ as the communication cost of tasks v_i and v_j.

3 Related Works

Many algorithms for scheduling workflow applications adopt the list-scheduling technique [2,7,11,12,14]. In the technique, task mapping is preceded by task ranking. The ranking step assigns a priority to each task using a well designed ranking function. The mapping step maps the ranked tasks to the computing nodes one by one based on an appropriate mapping strategy. The upward rank $rank_u(v_i)$ and downward rank $rank_d(v_i)$ are the attributes used in many list-based scheduling algorithms for assigning priorities to tasks. Attribute $rank_u(v_i)$



(a) Diagraph of an application

tasks	Rank$_u$	Rank$_d$	Rank$_u$+Rank$_d$
v_0*	38	0	38
v_1	19	13	32
v_2*	27	11	38
v_3	19	15	34
v_4*	12	26	38
v_5	12	22	34
v_6*	4	34	38

(b) Attributes for task ranking

Fig. 1. Task ranking attributes of an application

is the length of the longest path from task v_i (including v_i) to the exit task and attribute $rank_d(v_i)$ is the length of the longest path from the entry task to task v_i (excluding v_i). They are defined using the following formulae:

$$rank_u(v_i) = \overline{\tau}_i^{cmp} + \max_{v_j \in succ(v_i)} \{rank_u(v_j) + \overline{\tau}_{ij}^{cmm}\} \tag{1}$$

$$rank_d(v_i) = \max_{v_j \in pred(v_i)} \{rank_d(v_j) + \overline{\tau}_j^{cmp} + \overline{\tau}_{ij}^{cmm}\} \tag{2}$$

where $\overline{\tau}_i^{cmp}$ is the average computation cost of task v_i and $\overline{\tau}_{ij}^{cmm}$ is the average communication cost for sending data from task v_i to task v_j. A critical path of a diagraph is the longest path from the entry task to the exit task. The length of a critical path is $rank_u(v_i) + rank_d(v_i)$ which is the sum on the weights of the vertices and edges constituting the path [14]. A critical task is the task on a critical path. Let v_i be a critical task. Figure 1 shows the upward ranks, downward ranks and a critical path in a diagraph. In the figure, the critical path is illustrated by bold lines. The $rank_u + rank_u$ values of critical tasks v_0, v_2, v_4 and v_6 are 38.

Heterogeneous Earliest Finish Time (HEFT) [12] has two major steps: ranking step and mapping step. In the ranking step, the $rank_u$ of each task is computed. In the mapping step, HEFT maps the tasks in the order of decreasing $rank_u$ values to their most suitable computing nodes. It also employs insertion-based strategy [12] to improve the mapping solution. Critical-Path-on-a-Processor (CPOP) [12] also has two major steps: ranking step and mapping step. The rank of task v_i is the sum of $rank_u(v_i)$ and $rank_d(v_i)$. In the mapping step, if the task in consideration is a critical task, then CPOP maps it to the critical-path processor. A critical-path processor is the computing node which can complete the critical tasks earlier than the others. Otherwise, CPOP maps it to the most suitable computing node which can achieve the earliest finish time. Hybrid Heuristic [11] consists of three major steps: task ranking, task grouping and group mapping. It is designed to be insensitive to ranking functions. The task ranking step computes $rank_u$ for each task. The task grouping step partitions the tasks into groups based on the ranking values and the dependence between the tasks. The group mapping step maps the tasks group by group using any appropriate mapping algorithm. DCP [7] is proposed for homogeneous computing environments with unbounded computing nodes. DCP defines attribute *mobility* for each task. The mobility of task v_i is $|P_{ct}| - (rank_u(v_i) + rank_d(v_i))$, where $|P_{ct}|$ is the length of a critical path. The mobility of a critical task is zero [14]. The task with the lowest mobility is mapped first. DCP uses looking-ahead strategy in selecting appropriate computing nodes and adopts different mapping policies for critical tasks and non-critical tasks. MCP [14] is designed for homogeneous computing environment with bounded computing nodes. At first, MCP computes the value of $|P_{ct}| - rank_u(v_i)$ for each task and then sorts the tasks in the order of increasing computed values. MCP maps tasks with the smallest value to the computing node which can achieve the earliest start time. Insertion-based policy [12] is also employed to improve the mapping solution. DAGMap [2] consists of three major phases: task ranking, task grouping and task scheduling. DAGMap computes the $rank_d$ and $rank_u$ for all the tasks to determine a critical path. In the algorithm, $rank_u(v_i)$ is used as the rank value of task v_i. DAGMap uses two auxiliary scheduling algorithms Minmin and Maxmin [5]. The heterogeneity factor (HF) is used to determine which auxiliary scheduling algorithms should be employed. If the HF value of a group is large, then DAGMap adopts Maxmin scheduling algorithm; otherwise, DAGMap adopts Minmin scheduling algorithm.

Some algorithms [3,8] schedule an ongoing application stage by stage; in each stage, eligible tasks are selected for mapping. Generational scheduling (GS) algorithm [3] is one of the stage scheduling algorithms proposed for heterogenous computing environments. The strategy employed by GS is very simple. A task is eligible for scheduling after all of its predecessors, if any, have completed their works. The characteristics of GS algorithm is that it transforms the problem of scheduling a workflow into many scheduling subproblems which can be solved without considering the precedence constraints. GSTR [8] is an extended version of the GS algorithm with two major modifications: (1) it does not map all tasks at the beginning of each iteration as GS does and (2) a task will be replicated to another computing node if the task has not completed its work after a long time period.

4 A Framework for Adaptable Scheduling Algorithms

A framework for AS algorithms takes several iterations to schedule all the tasks of an ongoing application to available computing nodes incrementally. Each iteration consists of three major steps: selection step, ranking step and mapping step. Figure 2 shows the details of the framework. The input is the diagraph of an application and the mapping solution is given by assignment matrix A which is updated in each iteration. Entry $A_{ij} = t$ indicates task v_i which has been mapped to node p_j can start its work at time t. In addition to scheduling the tasks at a variation event (from step 20 to step 31), the framework migrates

```
   Input  : A diagraph $G$ of an application
   Output: Assignment matrix $A$
 1 $P_{new} \leftarrow$ ExtractAvailResource
 2 $v_{finish} \leftarrow nil$
 3 while $v_{finish} \neq v_{exit}$ do
 4     if ResourceChangeEvent() $= true$ then
 5         $P_{old} \leftarrow P_{new}$
 6         $P_{new} \leftarrow$ ExtractAvailResource
 7         $P_{revoked} \leftarrow (P_{old} - P_{new})$
 8         if $P_{revoked} \neq \emptyset$ then
 9             $L \leftarrow$ CollectScheduledTask($P_{revoked}$)
10             $L \leftarrow$ Ranker($L$)
11             repeat
12                 $max \leftarrow$ ExtractMaxPriorityTask($L$)
13                 $revoked \leftarrow$ GetOriginalP($v_{max}, A$)
14                 $A_{max,revoked} \leftarrow -1$
15                 $best \leftarrow$ Mapper($max, P_{new}, A$)
16                 $A_{max,best} \leftarrow$ GetStartTime($v_{max}, p_{best}, A$)
17             until $L = \emptyset$
18         end
19     end
20     if any task $v_i$ has finished or starts to receive input data then
21         if task $v_i$ has finished then
22             $v_{finish} \leftarrow v_i$
23         end
24         $L \leftarrow$ Selector($G, A$)
25         $L \leftarrow$ Ranker($L$)
26         repeat
27             $v_{max} \leftarrow$ ExtractMaxPriorityTask($L$)
28             $p_{best} \leftarrow$ Mapper($max, P_{new}, A$)
29             $A_{max,best} \leftarrow$ GetStartTime($v_{max}, p_{best}, A$)
30         until $L = \emptyset$
31     end
32 end
```

Fig. 2. A framework for AS algorithms

the scheduled tasks to other nodes (from step 4 to step 19) if their original nodes are revoked.

The information regarding the available computing nodes for application G is acquired using `ExtractAvailResource` at step 1. Variable v_{finish} stores the most recently complete task and its initial value is set to nil at step 2. `Resource ChangeEvent` returns true if a variation in the available computing nodes occurs. In this case, the set of revoked computing nodes $P_{revoked}$ is determined at steps 5, 6 and 7. If $P_{revoked}$ is not empty, `CollectScheduledTask` collects the tasks which are previously scheduled to the revoked computing nodes, then inserts the tasks into list L at step 9. At step 10, `Ranker` computes the priorities again for the tasks in L. The task v_{max} with the highest priority is extracted using `ExtractMaxPriorityTask` at step 12. The revoked computing node to which the task is originally mapped is found at step 13. The previous mapping solution regarding task v_{max} is discarded at step 14. At step 15, `Mapper` determines the most suitable computing node p_{best} for task v_{max}. The new mapping solution for task v_{max} is recorded at step 16. The steps from 12 to 16 repeat until all the tasks in L have been considered.

The process of scheduling tasks to computing nodes is triggered if a task v_i has completed its work or starts to receive input data as shown at step 20. In this case, variable v_{finish} is set to v_i at step 22. At step 24, the `Selector` selects the tasks based on a specified selection strategy, then inserts the selected tasks into list L. At step 25, `Ranker` computes priority for each selected task. At step 27, `ExtractMaxPriorityTask` extracts task v_{max} with the highest priority from L. At step 28, `Mapper` schedules task v_{max} to computing node p_{best} which achieves the minimal completion time for the task. The mapping solution regarding task v_{max} is recorded at step 29. The process of mapping the selected tasks repeats until all of the tasks in L have been considered. Scheduling an application is done when $v_{finish} = v_{exit}$.

5 AS Algorithm: AROF

If the selector selects tasks based on an aggressive strategy then the tasks may miss the most suitable computing nodes which are available in the near feature. However, if the selector selects the tasks based on a conservative strategy then the newly mapped tasks may have delayed their works due to waiting for a severe condition to match. Either of these cases degenerates the performance of a R^2 grid. The timing for task v_i to be selected can be defined by the states of v_i's predecessors. A selector $\mathcal{S}$ uses two attributes α and β to select a task. Let r_i^{pre} denote the number that v_i's predecessors have received all or part of their inputs and c_i^{pre} denote the minimum number that v_i's predecessors have completed their works. Selector $\mathcal{S}(\alpha, \beta)$ selects a task whose predecessors satisfy the following criterion: $r_i^{pre} = \alpha$ and $c_i^{pre} = \beta$. For example, a task will be selected by $\mathcal{S}(4, 2)$ if four of its predecessors have received all or part of their inputs and at least two of them have completed their works. The selector of our AS algorithm is $\mathcal{S}(|pre(v_i)|, 1)$, where $|pre(v_i)|$ denotes the number of all the task v_i's predecessors. It indicates that a task is selected if all of its predecessors have received all or part of their inputs, and at least one of them has finished its work. Our AS algorithm is named AROF which means $\mathcal{A}$ll $\mathcal{R}$eceive $\mathcal{O}$ne $\mathcal{F}$inish. Furthermore, the selectors of HEFT algorithm and GS algorithm are $\mathcal{S}(0, 0)$ and $\mathcal{S}(|pre(v_i)|, |pre(v_i)|)$, respectively.

The ranking step of a scheduling algorithm computes the priority for each selected task. Based on their priorities, the tasks are mapped to the available computing nodes one by one in the mapping step. The ranking functions used in many scheduling algorithms are either too simple or too complex. For example, GS [3] ranks the selected tasks randomly and DCP [7] ranks the selected tasks using a complex strategy. HEFT [12] is a low complexity and high performance scheduler. However, the availability variation and known execution costs of the scheduled tasks have not be used to precisely compute the upward rank values for the newly selected tasks. Thus, a revised ranking function $rerank_u(v_i)$ is proposed. It takes the availability variation and known costs of the scheduled tasks into consideration. Function $rerank_u$ is defined as follows: $rerank_u(v_i) = \mathcal{X} + \max_{v_j \in succ(v_i)}\{rerank_u(v_j) + \mathcal{Y}\}$, where

$$\mathcal{X} = \begin{cases} \tau_{ik}^{cmp} & \text{if task } v_i \text{ has been mapped to computing node } p_k \\ \overline{\tau}_i^{cmp} & \text{otherwise} \end{cases} \tag{3}$$

$$\mathcal{Y} = \begin{cases} \tau_{ij}^{cmm} & \text{if both task } v_i \text{ and task } v_j \text{ have been mapped} \\ \overline{\tau}_i^{cmm} & \text{otherwise} \end{cases} \tag{4}$$

There are two major differences between the ranking step of AROF and that of HEFT. Firstly, HEFT computes the upward rank values of all the tasks at the beginning for determining the mapping priorities of the tasks. Since the available computing nodes are variant, the value of $rerank(v_i)$ is not computed until task v_i has been selected. Thus, the upward rank values of the selected tasks can be computed using the most recently information. Secondly, some tasks assigned to the revoked computing nodes need to migrate to other available computing nodes. In order to keep the upward rank values precise for scheduled tasks, the upward rank values of the migrant tasks and their predecessors are modified using $rerank_u(v_i)$. The mapping step maps tasks to computing nodes such that the completion times of the tasks are minimized. The insertion-based technique [12] is also adopted to improve the mapping solution.

6 Experimental Results and Analysis

Our AROF scheduling algorithm is compared with HEFT [12] and GS [3] to demonstrate its effectiveness. In this section, the parameter settings used in the simulations are given and the results are presented and analyzed.

6.1 The Workload Parameters

The random graph generator adopted in [12] is used to create diagraphs of different types. The shape of a diagraph is defined by the number of task layers (height) and the average number of independent tasks in one layer. Attributes used to generate the diagraphs are given as follows.

In our simulations, an application consists of $|V|$ tasks, where $|V|$ =50, 100 or 150. The entry and exit tasks are included in V. The height of a diagraph is $\lfloor \sqrt{|V|}/shape \rfloor$. Except for the first (last) layer which only has one task, the widths for the other layers are random numbers generated from a uniform distribution with a mean value $\lfloor shape\sqrt{|V|} \rfloor$, where $shape = 0.5, 1.0,$ or 2.0. If the number of total tasks is less or more than $|V|$ then tasks are added to or deleted from randomly selected layers subject to that the number of tasks in any layer cannot be less than one. Each task has three successors which are randomly selected from the next layer if the layer consists of at least three tasks. Otherwise, all the tasks in the next layer are the successors of those in the previous layer. If any task in a layer has not been assigned as a successor then a task in the previous layer is randomly selected as its predecessor. Communication to computation ratio (CCR) is the ratio of average task communication cost to average task computation cost. In our simulations, CCR =0.5, 1.0 or 2.0. Denote $\overline{b}$ as the average bandwidth among all the pairs of computing nodes. The number of data units for task v_i to communicate with any of its successors is a random number within the range between 1 and $2\overline{b}\overline{\tau}_i^{cmp}CCR$. Note that $\overline{\tau}_i^{cmp}$ is the average computation cost of task v_i. The values of $\overline{\tau}_i^{cmp}$ and $\overline{b}$ depend on the parameter settings for the grid model. Thus, a diagraph is generated layer by layer after the parameter settings for the gird model have been known.

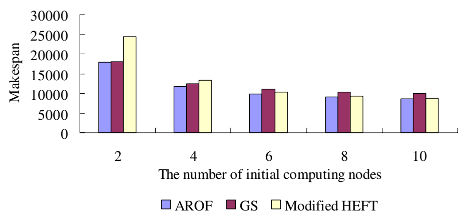
6.2 The R^2 Grid Parameters

Heterogeneity factor HF , where $HF < 1$, defines the variance in the execution costs of a task on different computing nodes. If HF is large, the variance in the execution costs is high. The execution cost of task v_i on computing node p_j is denoted as τ_{ij}^{cmp}. For a given i, $\overline{\tau}_i^{Ecmp}$ is a random number selected from the range between 1 and 1000. For a selected $\overline{\tau}_i^{Ecmp}$, τ_{ij}^{cmp} for each computing node p_j is a number randomly selected from the range between $\overline{\tau}_i^{Ecmp}(1-HF)$ and $\overline{\tau}_i^{Ecmp}(1+HF)$. In our simulations, we have $HF = 0.5$. The reasons are given as follows. Firstly, the computing node with extremely slow speed should not be dispatched by the R^2 grid as a reinforcing node. Secondly, the performances achieved by scheduling algorithms cannot be effectively evaluated if extremely fast computing nodes are available. The reserved bandwidth for a pair of computing nodes is randomly selected from the range between 10 and 50.

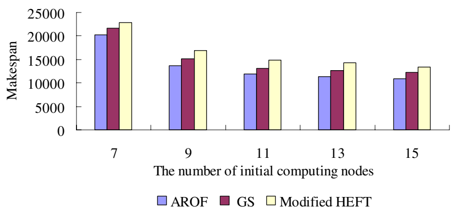
In the simulations, the number of available computing nodes alters once while an application is performing computations. Thus, the sequence of variation events is given as (P_0, P_1), where P_0 is the initial set of available computing nodes and P_1 is the set of available computing nodes after reinforcement or revocation. In the reinforcement case, $|P_0| = 2$, 4, 6, 8 or 10 and $|P_1| = |P_0| + 5$. In the revocation case, $|P_0| = 7$, 9, 11, 13 or 15 and $|P_1| = |P_0| - 5$. The time duration that an algorithm schedules the tasks of application $\mathcal{A}$ to the computing nodes in P_0 is denoted as $\delta_{\mathcal{A}}$. In our experiment, we set $\delta_{\mathcal{A}}$ as the interval of time for application $\mathcal{A}$ to complete forty percent of its tasks under HEFT algorithm. For comparison purpose, we measure the performances of AROF and GS under the same variation event sequence (P_0, P_1) and the same time duration $\delta_{\mathcal{A}}$ for switching variation event P_0 to variation event P_1.

6.3 Experimental Results

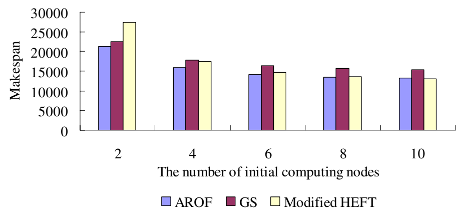
In this section, AROF algorithm is compared with the well-known scheduling algorithms HEFT [12] and GS [3]. The traditional HEFT performs very well on the platform where there is no variation in computing nodes. However, the design of the traditional HEFT does not consider the necessity of migrating the scheduled tasks previously mapped on the revoked nodes. Thus, a modification to the traditional HEFT is needed for an application to complete its work in case a revocation of computing nodes can happen. The modified version is called Modified HEFT. The strategy employed by the modified HEFT to migrate tasks is very simple. It just randomly maps the scheduled tasks on revoked computing nodes to the available computing nodes. The strategy employed by the traditional HEFT does not suspend an ongoing application in the case of reinforcing



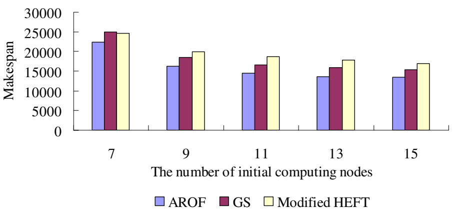
(a) $(100, 1, 0.5, \text{increase})$



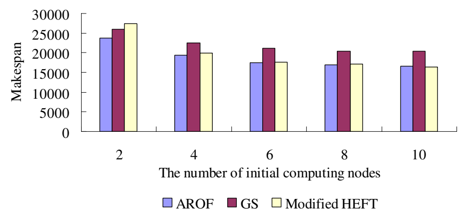
(b) $(100, 1, 0.5, \text{decrease})$



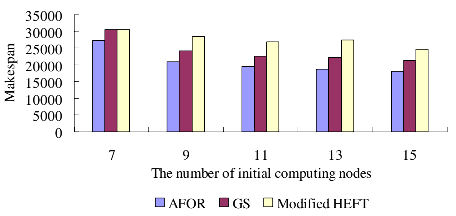
(c) $(100, 1, 1, \text{increase})$



(d) $(100, 1, 1, \text{decrease})$



(e) $(100, 1, 2, \text{increase})$



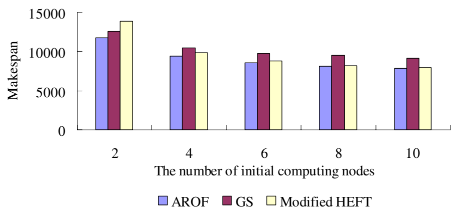
(f) $(100, 1, 2, \text{decrease})$

Fig. 3. The makespans under various CCR values

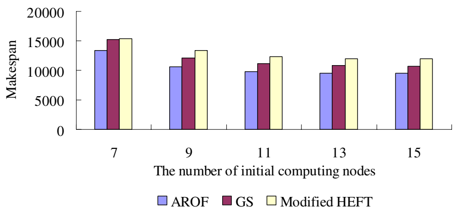
computing nodes. Thus, under our Modified HEFT, the newly jointed computing nodes will not be used by an ongoing application, either. For comparison purpose, the steps for ranking and mapping used by AROF are also employed by GS algorithm. In our simulations, a 4-tuple $(|V|, shape, CCR, variation)$ is used to indicate a combination of parameter settings, where $variation$ can be either increase or decrease. The result of each parameter combination presented in the figures is given by averaging 30 simulation outcomes.

Figure 3 illustrates the makespans of the applications scheduled under AROF, GS and Modified HEFT with CCR =0.5, 1 and 2. In general, the makespan of an application with a high CCR value is longer than that with a low CCR value. The reason is that an application with a high CCR value takes more time to perform communications. In the case of reinforcement, the performance gap between AROF and Modified HEFT is wide for a small initial set such as $|P_0| = 2$. The reason is that the the total computing power provided by the computing nodes in P_1 has been significantly enhanced compared to that provided only by the computing nodes in P_0. The performance gap becomes narrow when CCR is high. An obvious instance can be observed in the case of $|P_0|$ =4 in which the gap for $CCR = 2$ is obviously narrow than that for $CCR = 0.5$ and 1. The reason is that an application is communication-intensive if its CCR is high. Thus, the advantage of assigning more computing nodes to an application can be offset by the increasing communication costs among the nodes in P_1. Their performance gap becomes narrow when the number of initial computing nodes in P_0 is large. The major reason is that the ratio of $|P_1|$ to $|P_0|$ becomes small when $|P_0|$ is large. In this case, the computing power provided by the newly jointed computing nodes does not increase significantly compared with that provided by the initial computing nodes in P_0. Another reason is that our grid model is defined for heterogeneous systems. Thus, the most suitable computing node for each of the remaining tasks may not be found in the newly jointed computing nodes. GS performs worst compared with the others in most cases because of its conservative task selection strategy. The conservation selection strategy of GS prohibits any complete predecessor of a task from sending data to the task before all of its predecessors have completed their work. When the CCR value increases, the performance gap between GS and any other scheduling algorithm widens. The reason is that, under GS algorithm, many tasks may have suffered from a long waiting time before they can be selected for mapping. In the case of revocation, the performance achieved by Modified HEFT is worst for $|P_0| \geq 9$. The reason is that the scheduled tasks on the revoked computing nodes are randomly mapped to the computing nodes in P_1. Thus, the tasks may not migrate to their most suitable computing nodes. On the contrary, AROF performs well because its mapper schedules the tasks previously mapped on the revoked computing nodes to the available computing nodes which minimize the completion time of the tasks.

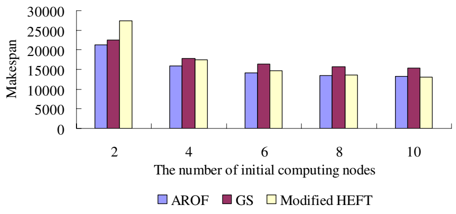
Figure 4 illustrates the makespans of the applications scheduled under AROF, GS and Modified HEFT with $|V| = 50, 100$ and 150. We observe that the makespan of the application consisting of more tasks is longer than that consisting of less tasks. The attained performances as shown in Fig. 4 have the similar trends as those in Fig. 3. In the case of reinforcement, the performance gap between AROF and HEFT is significantly wide for $|P_0| \leq 4$. We also observe that the gap increases in direct proportion to the number of tasks in an application. The reason is that the more the tasks constitute an application, the more the tasks remain unscheduled as the R^2 grid switches from event P_0 to event P_1. In this case, the newly jointed computing nodes can be fully utilized by the application scheduled by AROF algorithm. The performance achieved by GS is worst in most cases except $|P_0| = 2$. For $|P_0| = 2$, the computing power provided by the computing nodes in P_1 overwhelms the adverse effect of conservative selection strategy employed by GS. In the case of revocation, AROF outperforms the modified HEFT and GS algorithms because the appropriate selection strategy is employed by AROF. GS outperforms Modified HEFT except for the instance $|P_0| = 7$ at which they have the similar performance. The reason is that the strategy for GS to map tasks to computing nodes cannot maintain its advantage when the number of available computing nodes in P_1 is small.



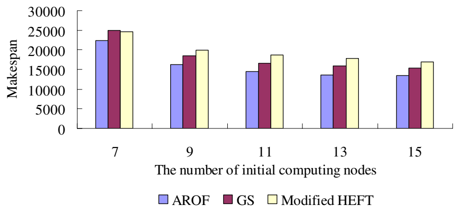
(a) $(50, 1, 1, \text{increase})$



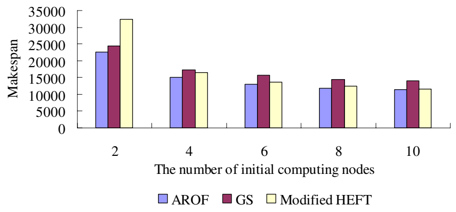
(b) $(50, 1, 1, \text{decrease})$



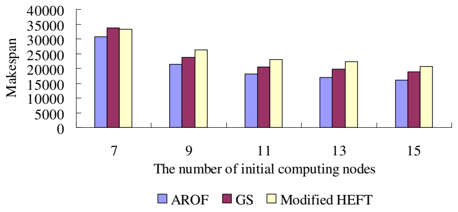
(c) $(100, 1, 1, \text{increase})$



(d) $(100, 1, 1, \text{decrease})$



(e) $(150, 1, 1, \text{increase})$



(f) $(150, 1, 1, \text{decrease})$

Fig. 4. The makespans under various task numbers

Figure 5 illustrates the makespans of the applications scheduled under AROF, GS and Modified HEFT with *shape* =0.5, 1 and 2. The figure shows that the makespan of an application with a small *shape* value is longer than that with large *shape* value. Note that the height of a diagraph equals $\lfloor \sqrt{|V|}/shape \rfloor$. Thus, the total communication cost of an application increases as the application is defined by a high diagraph. The attained performances as shown in Fig.4 have the similar trends as those in Fig.3. In the case of reinforcement, the performance gaps between AROF and HEFT are wide for $|P_0| = 2$. GS performs worst compared with the others in most cases because of its conservative task selection strategy. However, when $shape = 2$ and $|P_0| = 2$, GS performs best compared with the others. The reason is that the adverse effect of the conservative selection strategy employed by GS is offset by the large number of tasks in a layer. Another reason is that the tasks mapped to a small number of computing nodes can reduce the communication costs. In the case of revocation, AROF still outperforms the other algorithms. Modified HEFT performs worst in most cases. The performance gaps between AROF and GS become narrow when *shape*

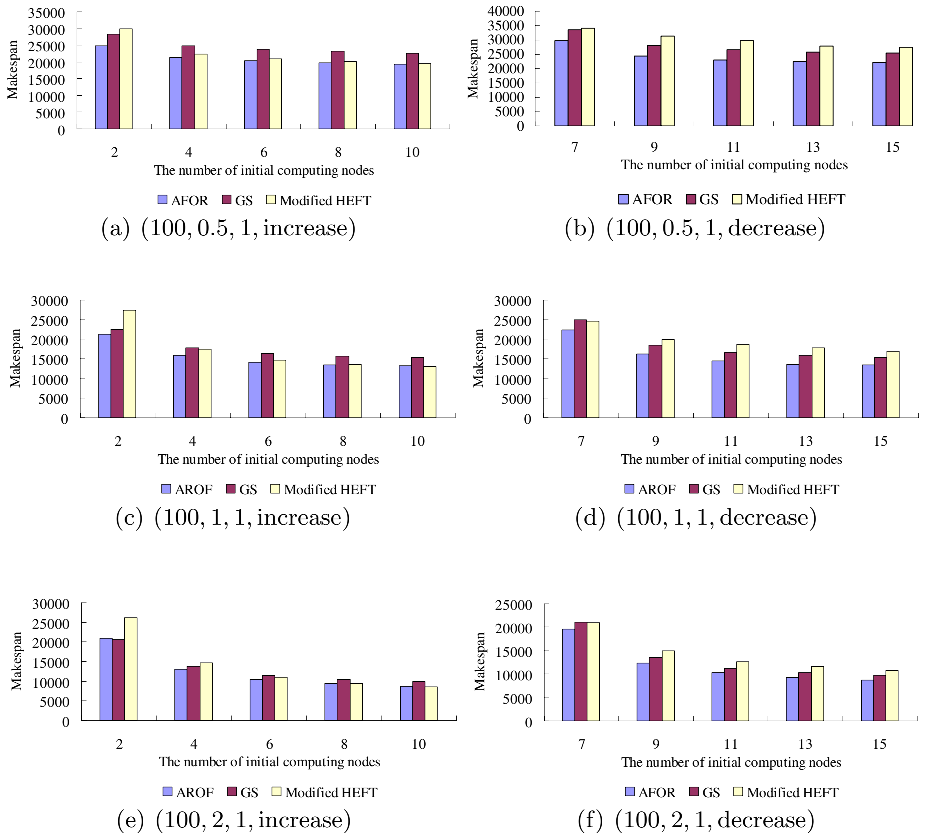


Fig. 5. The makespans under various *shape* values

value increases. The reason is that the communication cost decreases when an application is modeled by a short diagraph.

According to the experimental results, it leads to a conclusion as follows. Modified HEFT still can achieve high performance when the ratio of $|P_1|$ to $|P_0|$ is small even though it cannot employ the newly jointed computing nodes in the case of reinforcement. However, it cannot perform well in the case of revocation. GS usually performs worst in the case of reinforcement except for $|P_0| = 2$ with which the adverse effect of its conservative task selection strategy is not obvious. AROF algorithm achieves best performance in most of the cases under the computing environment in which the set of available computing nodes can be variant.

7 Conclusions and Future Work

In this paper, we have demonstrated that AROF outperforms the well-known scheduling Modified HEFT and GS algorithms in most cases. In the future, we will study the optimal number of computing nodes assigned to an application by AS algorithms for achieving maximum performance under difference sequences of variation events in the R^2 grid.

References

1. Baskiyar, S., SaiRanga, P.C.: Scheduling Directed A-cyclic Task Graphs on Heterogeneous Network of Workstations to Minimize Schedule Length. In: Proc. The Int'l Conf. Parallel Processing Workshops, pp. 97–103 (2003)
2. Cao, H., et al.: DAGMap: Efficient Scheduling for DAG Grid Workflow Job. In: The 9th IEEE/ACM Int'l Conf. Grid Computing, pp. 17–24 (2008)
3. Carter, B.R., et al.: Generational Scheduling for Dynamic Task Management in Heterogeneous Computing Systems. J. Information Sciences series 106, 219–236 (1998)
4. Chang, J.-Y., Chen, H.-L.: Dynamic-Grouping Bandwidth Reservation Scheme for Multimedia Wireless Networks. IEEE J. Selected Areas in Communications 21, 1566–1574 (2003)
5. Freund, R.F., et al.: Scheduling Resources in Multi-user, Heterogeneous, Computing Environments with SmartNet. In: Proc. Heterogeneous Computing Workshop, pp. 184–199 (1998)
6. Ilavarasan, E., Thambidurai, P., Mahilmannan, R.: Performance Effective Task Scheduling Algorithm for Heterogeneous Computing System. In: Proc. The 4th Int'l Symp. Parallel and Distributed Computing, pp. 28–38 (2005)
7. Kwok, Y.-K., Ahmad, I.: Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. IEEE Trans. Parallel and Distributed Systems series 7, 506–521 (1996)
8. de O. Lucchese, F., et al.: An Adaptive Scheduler for Grids. J. Grid Computing series 4, 1–17 (2006)
9. Maheswaran, M., et al.: Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing System. J. Parallel and Distributed Computing 59, 107–131 (1999)
10. Rauber, T., Rünger, G.: Anticipated Distributed Task Scheduling for Grid Environments. In: Proc. The 20th Int'l Parallel and Distributed Processing Symposium (2006)
11. Sakellariou, R., Zhao, H.: A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In: Proc. The 18th Int'l Parallel and Distributed Processing Symposium (2004)
12. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Trans. Parallel and Distributed Systems series 13, 260–274 (2003)
13. Wang, L., et al.: Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. J. Parallel and Distributed Computing series 47, 8–22 (1997)
14. Wu, M.-Y., Gajski, D.D.: Hypertool: A Programming Aid for Message-Passing Systems. IEEE Trans. Parallel and Distributed Systems 1, 330–343 (1990)
15. Yang, M., et al.: An End-to-End QoS Framework with On-Demand Bandwidth Reconfiguration. Computer Communications series 28, 2034–2046 (2005)

Using MPI on PC Cluster to Compute Eigenvalues of Hermitian Toeplitz Matrices

Fazal Noor and Syed Misbahuddin

College of Computer Science and Engineering,
Computer Science and Software Engineering Department,
University of Hail, Hail, Saudi Arabia
`{fnoor,smisbah}@uoh.edu.sa`

Abstract. In this paper MPI is used on PC Cluster to compute all the eigenvalues of Hermitian Toeplitz Matrices. The parallel algorithms presented were implemented in C++ with MPI functions inserted and run on a cluster of Lenovo ThinkCentre machines running RedHat Linux. The two methods, MAHT-P one embarrassingly parallel and the other MPEAHT using master/ slave scheme are compared for performance and results presented. It is seen that computation time is reduced and speedup factor increases with the number of computers used for the two parallel schemes presented. Load balancing becomes an issue as number of computers in a cluster are increased. A solution is provided to overcome such a case.

Keywords: Parallel processing, load-balancing, MPI, Speedup, PC cluster, Hermitian Toeplitz Matrices, Eigenvalues.

1 Introduction

The Message Passing Interface Standard (MPI) is a message passing library. MPI is widely used in writing programs in which distributed computing is necessary. Nowadays with affordable personal computers a cluster of PCs can be setup with LINUX or MS-Windows operating system. A cluster of PCs can be used to run a program in parallel to solve computationally intensive complex problems. Such a problem arises in array signal processing where a process is stationary and the covariance matrix is Hermitian Toeplitz. In this case matrices formed are of large order and problem reduces to compute eigenvalues. In this paper we have implemented an algorithm using MPI to run in parallel on a cluster of Pentium machines to compute all the eigenvalues of a given Hermitian Toeplitz matrices. In 1989, Trench[1] presented a numerical solution of eigenvalue problem for Hermitian Toeplitz matrices and later a modified version of Trench's method with Hu's method[2][5] was presented in [3].

The idea of parallel eigenvalue computation for real Toeplitz matrices was first presented by Hu[5]. Badia and Vidal[8] presented parallel methods for the case of real symmetric Toeplitz matrices using Parallel Virtual Machine (PVM). In this work, we deal with Hermitian Toeplitz matrices and use MPI. In Section 2, we present the mathematical background and in Section 3 we present the results.

2 Mathematical Development

Given a Hermitian Toeplitz matrix C_n of order n,

$$C_n = \begin{bmatrix} c_0 & c_1^* & \cdots & c_{n-1}^* \\ c_1 & c_0 & \cdots & c_{n-2}^* \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ c_{n-1} & c_{n-2} & \cdots & c_0 \end{bmatrix} \tag{1}$$

where c_0 is real and c_1, c_2, …, c_{n-1} are complex, the problem is to find the complete eigenspectrum. Since C_n is Hermitian, $c_{-i}^* = c_i, i = 0,1,...,n-1$. The principal submatrix of C_n of order k is defined as $C_k = [c_{i-j} : 0 \le i, j \le k-1]$, $k = 1,2,...,n$.

Consider the following linear system involving the shifted Hermitian Toeplitz coefficient matrix $C_n - \lambda I_n$:

$$(C_n - \lambda I_n)\begin{pmatrix} 1 \\ -\Phi_{n-1}(\lambda) \end{pmatrix} = \left[E_n(\lambda)\ 0 \cdots 0\right]^T \tag{2}$$

where λ is treated as a continuous real variable. We assume all principal submatrices $C_k - \lambda I_k$, $k = 1,2,\cdots,n,$ are nonsingular. This implies that the Levinson-Durbin(L-D) recursion can be applied to $(C_{n-1} - \lambda I)\Phi_{n-1} = \left[c_0 c_1^* \cdots c_{n-1}^*\right]^T$.

In order to find the elements $\phi_{n-1,i}$, $i = 1,2,\cdots,n-1,$ of $\Phi_{n-1}(\lambda)$ in a recursive fashion we use the following L-D algorithm:

Initialization:
$E_0 = c_0 - \lambda$,

For $1 \le k \le n-1$.
Compute:

$$\rho_k = \frac{1}{E_{k-1}}\left[c_k - \sum_{i=1}^{k-1} \phi_{k-1,i} c_{k-i}\right]$$
$$\phi_{kk} = \rho_k$$
$$\phi_{kk} = \phi_{k-1,i} - \rho_k \phi_{k-1,k-i}^*, \qquad 1 \le i \le k-1$$
$$E_k = E_{k-1}(1 - |\rho_k|^2). \tag{3}$$

The parameters ρ_k and E_k are known as the reflection coefficient and prediction error, respectively, at the *k*th recursive step. Note that all the quantities in equation (3) depend on the parameter λ through the initialization E_0. The sequential algorithm is summarized below.

Modified-Algorithm-Hermitian Toeplitz Matrices *[3]*

Step 1-Select: *Find the eigenvalues* $\lambda_p, \lambda_{p+1}, \cdots, \lambda_q$, $1 \le p < q \le n$. *Using trial and error, select an interval* (a,b) *by bisection such that* $Neg_n(a) \le p-1$, $Neg_n(b) \ge q$. *Note: Neg (λ) is the number of negative elements* $E_i(\lambda)$ *equals the number of eigenvalues* λ_i *of C that are less than λ .*

For $i = p$ *To* $q-1$.

Step 2-Search: *Search for the endpoint* $U\xi_i$ *not captured by trial and error such that* $(L\xi_i, U\xi_i)$ *contains* λ_i. *This is done by bisection and by keeping count of the negative signs of* $\{E_1(U\xi_i), E_2(U\xi_i), \cdots, E_n(U\xi_i)\}$. *During this search process, keep tightening, capturing, and storing the locations of other desired eigenvalues, while also retaining the values* $E_n(L\xi_i)$, $E_n(U\xi_i)$, *and* $E_n(L\xi_{i+1})$.

Step 3-Refine: *Once all the intervals* $L\xi_i < \lambda_i < U\xi_i$, $p \le i \le q$, *are obtained:*

For j=p To q

(a) Set $\alpha = L\xi_j$, $E_\alpha = E_n(L\xi_j)$ *and* $\beta = U\xi_j$, $E_\beta = E_n(U\xi_j)$.

(b) For multiple eigenvalues, set the matrix order n to n-m+1 and work with the submatrix C_{n-m+1}. *By trial and error, refine the interval* (α,β) *to* (α',β') *by bisection such that the following conditions hold:*

i. $Neg_n(\alpha') = j-1$ *and* $Neg_n(\beta') = j$

ii. $E_n(\alpha') > 0$ *and* $E_n(\beta') < 0$.

(c) Switch to Modified Rayleigh Quotient Iteration (MRQI) or PEGASUS method to find λ_j.

Next j

End.

Parallelism in the above algorithm can be accomplished several ways. Some of the methods are as follows:

Method 1). The above algorithm *MAHTM* is executed in parallel on each computer (node) in the cluster for different range of eigenvalues [5][8]. So in parallel each node finds for p=my_rank*n/psize +1 to q=(my_rank +1) * n/psize range of eigenvalues, where my_rank is the rank of each computer in the group, psize is the number of computers or nodes in a group, and n is the size of the matrix. We will call this method *MAHT-P*. This method has almost no communication overhead and falls in the category of embarrassingly parallel algorithms [10].

Method 2). Master Computer: Performs Step1-Select and Step2-Coarse Search then once all intervals are obtained then in parallel these intervals may be sent to the slaves to extract the eigenvalues**.** Slave computers: Each $slave_x$ in parallel receives an interval and uses either MRQI or Pegaus method to find λ_j . In other words, parallelism is done at Step 3 of MAHTM and slaves are idle the whole time Master Computer performs step 1 and 2.

Method 3). Master and Slave Computers: All computers master and slave participate in obtaining the coarse intervals and pass the intervals to the master which acts as a coordinator. The master then sends one interval to each slave and waits for the result. Once the result is received it sends another interval to the same slave. The process is repeated until there are no more intervals to send.

Method 3 is summarized as follows:

MPI-Parallel-Eigen Algorithm Hermitian Toeplitz (MPEAHT) Matrices

Step 1: *In parallel each computer (master and slaves) finds the range of intervals from p to q each containing an eigenvalue according to their rank . Each has range from p= my_rank*n/psize + 1 to q=(my_rank + 1) * n / psize intervals to find, where my_rank is rank of each computer in the group, psize is the number of computers in the group, and n is the size of the matrix.*

Step 2: *Intervals found are sent to master computer and then master computer acting as a coordinator sends an interval to a slave to find* λ_j *. Each slave receives a single interval at a time. The slave computes an eigenvalue and sends back a signal to master to send another single interval containing an eigenvalue. In this scenario, there is communication between master and slave but slaves are load balanced. Time spent by slave to compute an eigenvalue exceeds the time taken to communicate with the master.*

In MPI-Parallel-Eigen (MPEAHT) the master and slave computers perform the coarse search in which non-contiguous intervals, in general, are obtained containing an eigenvalue per interval. In the Appendix a skeleton of the MPI program used with the above algorithm is provided.

In MPEAHT the master dispatches intervals among the slaves and waits to receive result and then sends another interval. In this case load is balanced in time among the slaves, therefore some slaves may compute more eigenvalues than others.

We implemented MPEAHT in C++ with MPI functions and ran it on our cluster under LAM environment. With LAM, a LINUX cluster acts as one parallel computer solving one compute-intensive problem as in our application.

3 Experimental Results

We implemented the above algorithms MAHT-P (Method 1) and MPEAHT (Method 3) in C++ with MPI functions and ran it on a 30 node cluster. Speedup is defined as

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{T_{computation} + T_{Communication}} \tag{4}$$

where T_s is the sequential execution time on one computer, and T_p is the execution time on *P* computers consisting of computation and communication time. Note communication time is negligible for MAHP-T algorithm compared to computation time. In case of MPEAHT algorithm the effect of communication time diminishes as number of computers increase in the cluster. Communication time in our sample runs was around 10^{-5} seconds. Computation time complexity is on order of $O(kMt_i)$ where *k* is number of eigenvalues searched per computer, *M* is average number of iterations required of Levinson-Durbin (L-D) algorithm per eigenvalue, and t_i is time taken by one iteration of L-D algorithm. For MAHT-P algorithm *k* is known and is defined as *k*=n/p (number of eigenvalues to be searched per computer) with n being the order of the matrix and p being the number of computers but for MPEAHT algorithm *k* is not known beforehand. For MPEAHT communication time would be approximately *k* times 10^{-5} seconds. We formed matrices of order 500 in which the off diagonal elements were randomly chosen having uniform distribution. Ideal speedup that can be achieved with ***p*** identical computers working is at most ***p*** times faster

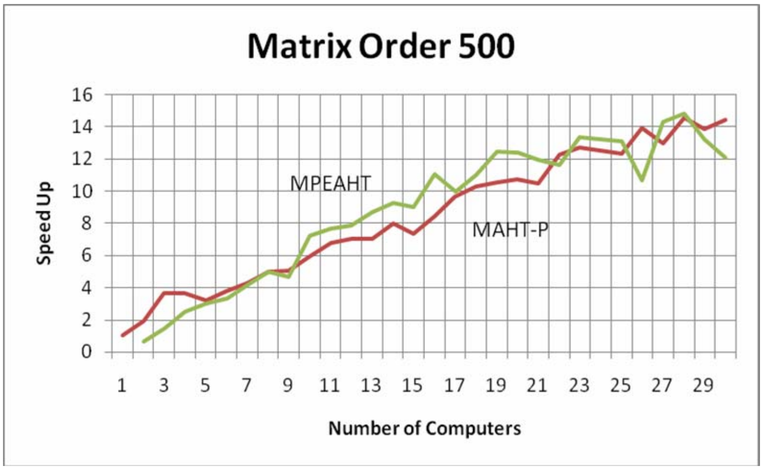


Fig. 1. *Speedup* vs *number of computers* in a cluster for MAHT-P and MPEAHT algorithms

Table 1a. Shows a sample run of algorithm MPEAHT in a cluster of size 6

Machine	No. Roots	No. Iter	Seconds
P1	62	287	5.72
P2	86	410	5.5
P3	96	419	5.34
P4	93	431	5.52
P5	82	356	5.25
P6	81	406	5.7
	500	2309	5.505

Table 1b. Shows another sample run of algorithm MPEAHT in a cluster of size 6

Machine	No. Roots	No. Iter	Seconds
P1	60	293	5.85
P2	96	443	5.56
P3	60	285	5.66
P4	96	438	5.51
P5	100	450	5.7
P6	88	400	5.35
	500	2309	5.605

than a single computer [10]. From Figure 1 with 30 computers the ideal speedup would be 30 times that of sequential time, however, it is approximately 12 to 14 range which is less than half the ideal. Also from Figure 1, the MAHT-P speedup graph shows more linearity than that of MPEAHT.

Tables 1a and 1b, below show two sample runs of algorthim MPEAHT on six slaves for a matrix of size 500. Note, same computer may have different number of intervals at different runs resulting in imbalance in number of roots. Also note some computers may be slower than others, for example, P1 although has less number of iterations than P2, P1 takes more time than P2. Similar conclusions can be made for other computers.

Figure 2 shows average time in seconds taken by number of computers in a cluster for matrix of size 500. As seen for number of computers less than 4 in a cluster MPEAHT took longer time than MAHT-P and as number of computers in a cluster increases time decreases and the difference between the two diminish.

Figures 3 and 4 show a sample run on a cluster of 30 computers. The horizontal axis shows the *i*-th computer in a cluster. Figure 3 shows the number of iterations taken for each computer and Figure 4 shows the corresponding average time taken for each computer. Figure 4 shows MAHT-P more stable in number of *iterations* than MPEAHT. However in Figure 4 MPEAHT shows more stability in *time* than MAHT-P. In our sample runs and the type of matrices used, MAHT-P performed a little better than MPEAHT.

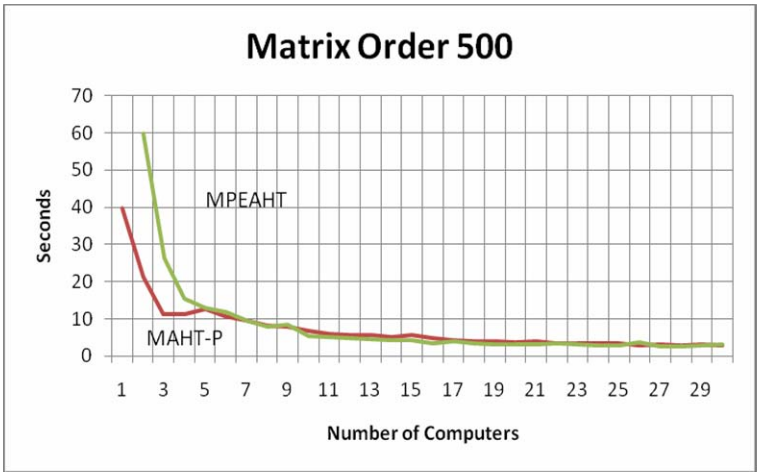


Fig. 2. Average time vs. number of computers in a cluster for MAHT-P and MPEAHT algorithms

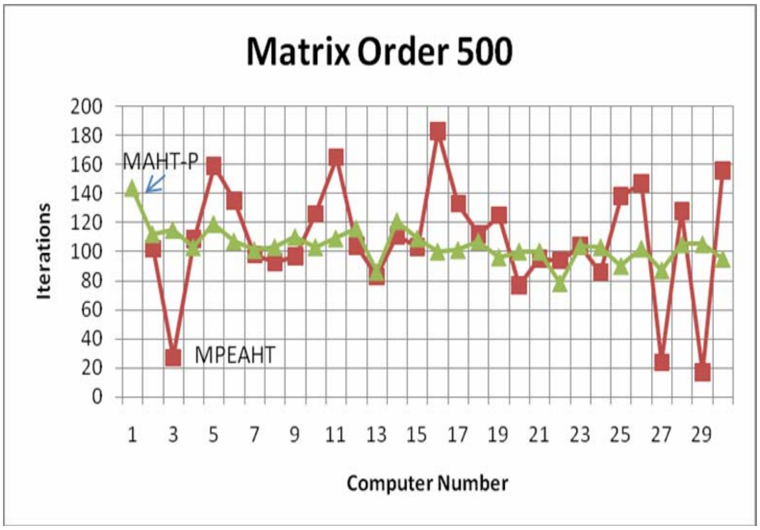


Fig. 3. Iterations on the *i*-th computer node of MAHT-P and MPEAHT algorithms

We also noted that in the case of MPEAHT (master/slave) *fair* load balancing degraded as number of computers in a cluster increased beyond 24. In practice *fair* load balance becomes poor because multiple slaves when finish their work contend to communicate at the same time with master for more work and it may happen a few slaves get fewer work than others. This is due to Carrier Sense Multiple Access with Collision Detection (CSMA-CD) technology being used on Ethernet networks. A solution to this problem would be to provide multiple communication channels to the master computer. Providing such communication channels would eliminate the contention problem.

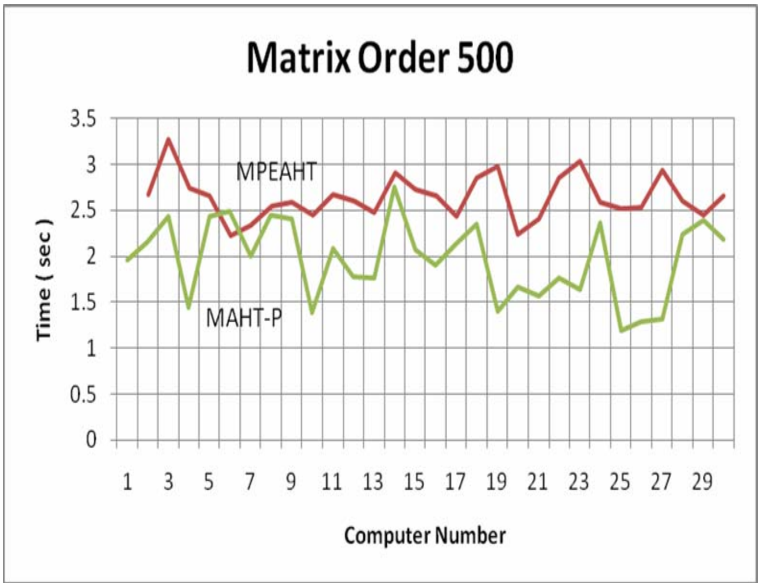


Fig. 4. Average time on *i*-th computer node of MAHT-P and MPEAHT algorithms

3 Conclusion

We have presented two methods for parallel computation of eigenvalues of Hermitian Toeplitz Matrices, namely MAHT-P and MPEAHT. MAHT-P and MPEAHT both use non-contiguous intervals but MAHT-P distributes equal amount of intervals to the workers whereas MPEAHT uses a master-slave scheme to distribute the intervals. It is seen that the results depend on the distribution of eigenvalues in an interval and MAHT-P gave better results in terms of time than the master-slave method of MPEAHT. Further we noted MPEAHT needs better ways to have *fair* load balancing specially as number of computers participating in a cluster increase. A solution to this problem is providing multiple hardware interfaces for handling multiple contentions.

Acknowledgments. The authors would like to thank the University of Hail research center and the reviewers for their helpful comments.

References

1. Trench, W.F.: Numerical Solution of the Eigenvalue Problem for Hermitian Toeplitz Matrices. SIAM J. Matrix Anal. Appl. 10(2), 135–146 (1989)
2. Hu, Y.H., Kung, S.Y.: Toeplitz Eigensystem Solver. IEEE Trans. Acoust. Speech, Signal Processing ASSP-33, 1264–1271 (1985)
3. Noor, F., Morgera, S.D.: Recursive and Iterative Algorithms for Computing Eigenvalues of Hermitian Teoplitz Matrices. IEEE Transcations on Signal Processing 41(3), 1272–1279 (1993)

4. Noor, F., Morgera, S.D.: Construction of a Hermitian Toeplitz Matrix from an Arbitrary Set of Eigenvalues. IEEE Trans. Siganl Processing 40(8), 2093–2094 (1992)
5. Hu, Y.H.: Parallel Eigenvalue Eecomposition for Toeplitz and related Matrices. In: Proc. ICASSP 1989, Glasgow, Scotland, pp. 1107–1110 (1989)
6. Cybenko, G., Van Loan, C.: Computing the Minimum Eigenvalue of a Symmetric Positive Definite Teoplitz Matrix. SIAM J. Sci. Stat. Comput. 7, 123–131 (1986)
7. Beex, A.A., Fargues, M.P.: Highly Parallel Recursive Iterative Toeplitz Eigenspace Decomposition. IEEE Trans. Acoust. Speech, Signal Processing 37(11), 1765–1768 (1989)
8. Badia, J.M., Vidal, A.M.: Parallel Algorithms to Compute the Eigenvalues and Eigenvectors of Symmetric Toeplitz Matrices. Parallel Algorithms and Applications 13, 75–93 (2000)
9. Golub, E.H., Van Loan, C.: Matrix Computations, pp. 305–312. John Hopkins University Press, Baltimore (1983)
10. Wilkinson, B., Allen, M.: Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd edn. Pearson Education Inc., London (2005)

Appendix

Below is the skeleton of the MPI program used with the above algorithm.

```
void intervals( int p, int q );
void roots( int kkk );

int main(int argc, char** argv)
{
    int WORKTAG = 1;
    int EXITTAG = 2;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &psize);
for ( rank = 0;  rank < psize; ++rank)
   { if ( my_rank == rank )
         { intervals ( p, q );
           if ( my_rank != 0 )
           { for ( kkk=p; kkk<=q; ++kkk)
                {
// Each slave P_i have found intervals will send from p
to q intervals to Master
// Sending intervals back to Master with rank = 0
MPI_Send(&work, 5, MPI_DOUBLE, 0, 22, MPI_COMM_WORLD);
                } /* end-for kkk */
              } /* end if rank not 0 */
   } /* if my rank == rank */
   } /* for loop rank = 0 */
```

```
    if ( my_rank == 0 )
    { n1=nn/psize + 1;
        for (  l=n1; l<=nn; ++l )
      {
MPI_Recv(&work,5,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,MP
I_COMM_WORLD, &status);
       } /* end for l=n1 */
   } /* end-if my rank == 0 */
//Range of eigenvalues to search
          p = 1;  q = nn;
//Master is finding the coarse intervals containing an
eigenvalue
   if ( my_rank == 0 )
     {   intervals ( p, q );
//Sending intervals to processors with rank 1 to psize -1
     for(rank=1; rank<psize; ++rank)
{MPI_Send(&work,5,MPI_DOUBLE,rank,WORKTAG,PI_COMM_WORLD);
     }
//  While  there  are  intervals  send  them  to  find
eigenvalues in them
    i=psize;
    while ( i != nn+1 )
{MPI_Recv(&result,4,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG
,MPI_COMM_WORLD, &status);

MPI_Send(&work,5,MPI_DOUBLE,status.MPI_SOURCE,WORKTAG,
MPI_COMM_WORLD);
     } /* end-while */
// No More Intervals left so recieve all that is still
being worked upon
   for    (    rank=1;    rank    <    psize;    ++    rank)
{MPI_Recv(&result,4,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG
,MPI_COMM_WORLD, &status);
      }
// Broadcast signal to slaves to indicate no more jobs
therefore exit
     for ( rank =1; rank < psize; ++rank)
    {MPI_Send(0,0,MPI_INT,rank,EXITTAG, MPI_COMM_WORLD);
    }
   } /* end-if my rank equals 0 */
  else {
//   Slaves do the work; Processor i receives the lower
and upper intervals
//   slave   returns   with   result   array   containing,
eigenvalue, no of iterations, time elapsed in seconds.
     while ( 1 ) {
```

```
/*Receive      information      from      Master      */
MPI_Recv(&work,5,MPI_DOUBLE,
0,MPI_ANY_TAG,MPI_COMM_WORLD,  &status);
  /* Check the tag of the received message */
       if ( status.MPI_TAG == EXITTAG )
        {   MPI_Finalize( );
             return 0;
        }
  /* Do the work */
      roots( kkk );
      MPI_Send(&result,4,MPI_DOUBLE,0,0, MPI_COMM_WORLD);
      } /* end while ( 1 ) */
   } /* end-if my rank equals 0  else */
    MPI_Finalize( );
    return ( 0 );
  }
```

idsocket: API for Inter-domain Communications Base on Xen

Liang Zhang, Yuein Bai, and Cheng Luo

School of Computer Science, Beihang University,
100191 Beijing, China
l_zhang@cse.buaa.edu.cn

Abstract. In virtualization environment, all physical hardware resource is maintained by virtual machine monitor (VMM). And as more and more applications and software deployed on virtual machine are communication intensive, they have a demand for communicate with each other. In-built communication ability between virtual machines is a necessary component of a mature virtual environment and is often easy to use, however, the performance of native in-built inter virtual machine communication is often not so good. At the same time, in virtual environment such as Xen hypervisor, a delicate approach, shared memory and event channel mechanism is provided. And this mechanism can be utilized for inter domain communication but not so easy to use. And to get a balance between performance and convenience, in this paper, we design and implement idsocket, an API suite for inter domain communication based on Xen using shared memory and event channel mechanism, bypassing the traditional front-back driver model. Benchmark evaluations and program tests have demonstrated that it has a better performance and a lower resource cost than the in-built front-back driver model for inter domain communication in Xen. Meanwhile, the rule and philosophy behind the design and implementation of idsocket is almost universal in any type of virtual machine.

1 Introduction

Recently, with the rapid development of hardware and software, virtualization technology is becoming mainstreamed. In virtualization environment, all physical hardware resource is maintained by virtual machine monitor (VMM), and above it, several virtual machines residing on the same physical server are logically isolated. This type of design makes the virtual environment more robust and secure [7][10]. However, as more and more communications intensive applications and software, such as web service, database centre, and gateway or name server, deployed in virtualized environment, communication mechanism of high performance must be offered by the VMM to make them run properly. In most VMM, this may be a very resource-consuming process because the barrier of isolation between virtual machines and the performance of communication between virtual machines is usually poor [1] [11], comparing with that in a physical environment.

In this paper, we propose and design an API suite for inter domain communication based on Xen hypervisor, named idsocket, which is short for *i*nter *d*omain *socket*, and it is easy to use and has an performance enhancement comparing to the in-built communication mechanism of Xen. Meanwhile, the rule and philosophy behind the design and implementation of idsocket is almost universal in any type of virtual machine.

The rest part of this paper is organized as follows. Section 2 states the motivation of our work. And in Section 3, a briefly introduction to the relative structure of the network architecture of Xen is presented. Section 4 discusses the design and implementation of idsocket. Performance evaluation is presented in Section 5. And relative works are mentioned in Section 6. Finally, Section 7 introduces future works and concludes.

2 Motivation

In virtual environment, communication request between applications and software is increasingly more; however, as a part of virtualized I/O, the in-built communication mechanism of most virtual machine monitor is seriously poor. Take Xen as an example. In the environment of Xen, there are two ways to do inter domain communication, the first way is that each virtual machine can be assigned an IP address and communicates with each other as they are in a same LAN. Although it is easy to use, due to the internal principle of Xen, it can be time-and-resource-consuming and inefficient. The poor performance makes the virtualized network communication of Xen a bottleneck when deploying network-sensitive application. And the second way is to use share memory and event channel mechanism. Shared memory is used to copy and read data while event channel is used to pass signals to notify domains that something has happened (such as data copy has ended, data is ready to be read and so on). This mechanism is more efficient and delicate but has a short coming that it is not so easy to use by user space program. All the operation is done through *hypercall*s which are kernel-level interface and any illegal use of them may cause fatal system error or failure.

And now, our research mainly focus on developing a new type of inter domain communication mechanism which provides high performance and is easy-to-use. And with this mechanism, inter domain communication is achieved through direct memory copy and it provides API in user space to make it easy to invoke.

3 Background

3.1 Architecture of Xen about Inter Domain Communication

Xen hypervisor is a virtual machine monitor of the powerful open source industry standard for virtualization [4]. Xen is a software layer which resides directly on the underlying hardware. There are guest domains above Xen. Domain 0 (Dom0), also known as privileged domain, is the most important domain, providing device drivers, user interface and tools, while other domains are unprivileged Domain U (DomU). Xen provides an environment that allowing multiple operating systems run concurrently as DomUs in an unprivileged way. All the privileged requests from DomU(s) are transformed to and proceed by Dom0 [8].

In the socket communication, Xen provides the front-back spilt driver model. In this model, DomU hosts the front-end and Dom0 hosts the back-end. All the device drivers are installed in Dom0, and DomU just has an abstract interface. Packages for inter domain communication travel from source DomU to Dom0 in the path of front-end to back-end. And then travel from Dom0 to destination DomU in the path of back-end to front-end [8]. Fig 1 shows the architecture of front-back driver model in Xen.

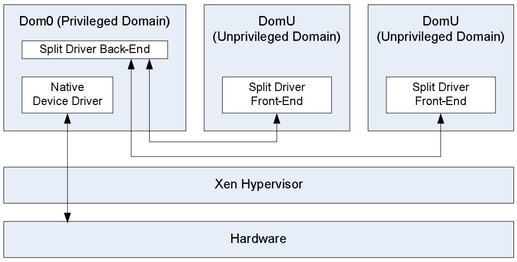


Fig. 1. Front-Back Driver Model Architecture in Xen Hypervisor

In the shared memory and event channel mechanism communication, Xen provides *hypercalls* for this function [8]. Shared memory is measured in page, whose size is 4KB, and can be created and granted by one domain and then be mapped by another domain. Shared memory is just lockless main memory space without any synchronization, and the task is for event channel. Event channel is a type of software interrupt, and it is a one-bit notification indicating something has happened, such as there is waiting data or memory space is ready to be rewritten.

3.2 Data Sharing and Security Issues in a Virtual Machine Environment

All the virtual machines reside on a same physical server and share the memory space of the physical machine. Every virtual machine has its own memory address space virtualized by the virtual machine monitor, and from the view of virtual machine, it monopolizes all the memory assigned to it. As one of the issues about isolation, the in-memory data must be protected by some security rules to make it safe and only accessible to the virtual machine owing it. Also, during the communication process between two virtual machines based on the way of shared memory, memory pages used for transferring data should be only written by the sender and then granted to the particular virtual machines, and also, these memory pages are only allowed to be mapped and read by the legal receiver. It is illegal and prohibited that other virtual machines either map or read to these memory regions without any grants.

As a mature virtual machine monitor, it must be designed to address these problems. “Physical” memory address of any virtual machine, which is actually virtual address, is maintained by the virtual machine monitor. And in the inter-domain communication of shared memory based way, the virtual machine monitor also should provide some functions for virtual machines to control memory access privilege, in form of interfaces or system calls and so on to guarantee the in-memory data isolation and security [14].

4 Design and Implementation

4.1 General Architecture of idsocket

idsocket is resided both on Dom0 and DomUs as kernel modules and therefore, idsocket can run without any modification of domains' kernel. In Dom0, there is a module for discovering DomUs and dispatching configuration messages for communication preparations between DomUs, named *dis2-module*. And in each DomU, there is a configuration module, named *conf-module* and a communication module, named *com-module*, respectively meet the requests of configuration negotiation and data transportation. *dis2-module* and *conf-module* contact with *Xenstore*, which is a small database with a tree-like structure storing configuration of all living domains [5]. DomU has a capacity of reading and writing their own Xenstore content while Dom0 can fully control Xenstore of all domains. And *com-module* in DomU exchanges data and notification with shared memory and event channel. The architecture of idsocket is illustrated in Fig 2.

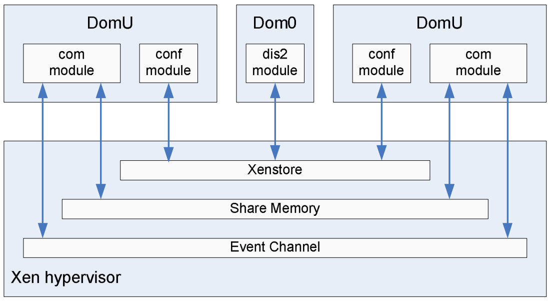


Fig. 2. General Architecture of idsocket

4.2 Discovering and Dispatching - Module in Dom0

4.2.1 Discovering DomUs

One DomU wants to communicate with other DomU, but the lack of the information of other DomUs makes it difficult to start a communication. And we design a module for discovering all DomUs for any DomU choosing from them. Meanwhile, communicating DomUs must share some configuration information to negotiation connection and they must be transport automatically between DomUs. These tasks are the responsibility of *dis2-module*.

When *dis2-module* begins to work, it firstly creates an entry "idsocket" with three keys "`domlist`", "`tohv`" and "`fromhv`" in every DomU's Xenstore. Then, it cyclically scans Xenstore to acquire *domid* and *name* of all DomUs and collect them in a string with `[domid, name]` pairs and then write it to the key "`domlist`" of each DomU.

4.2.2 Dispatching Function, Ways to Forward Configuration Message

Also, dis2-module cyclically scans the "`tohv`" key of each DomU, and then parses the content as a configuration message and forwards it to the destination DomU by writing the message to the key "`fromhv`" in its Xenstore. The scanning frequency can be configured as desired, and now we set it to one scanning per second.

With *dis2-module*, all DomUs can get the latest status about other DomUs which are potential communication targets, and any message between two domains in the communication process can be parsed and forwarded in time. As a result, one DomU is no longer bothered by the barrier of lack of information about other DomUs and configuration messages can be easily exchanged.

Configuration negotiation between DomUs with the assistance of Dom0 uses message of types listed in Table 1. The messages ending with "FWD" stands for "forward" which implies that the message is forwarded by Dom0. The whole process of configuration negotiation is illustrated in Fig 3.

Table 1. Message Type in Communication Negotiation

IDS_MSG_BAD_UNO	Unknown message, error occurred	IDS_MSG_SND_FWD	Send request message forward
IDS_MSG_RCV_REQ	Receive request message	IDS_MSG_RCV_RSP	Receive response message
IDS_MSG_RCV_FWD	Receive request message forward	IDS_MSG_RCV_ACK	Receive response message forward
IDS_MSG_SND_REQ	Send request message	IDS_MSG_SND_CLR	Send finish message

4.3 Configuration and Transportation - Module in DomU

4.3.1 Configuration Negotiation, Preparing or Destroying the Data Channel

Configuration negotiation is the first step of communication. In this step, sender DomU allocates shared memory pages and grants them, getting an integer value *gntref* as an identifier. After that, sender DomU allocates an unbound event channel and an integer *portnum* is returned as an indicator. And then, sender DomU writes the two integers to key "`tohv`" in Xenstore, and they are forwarded to receiver DomU. Receiver DomU reads them from key "`fromhv`", and shared memory is mapped with *gntref*, and event channel is bound with *portnum*. This process is much like a simple

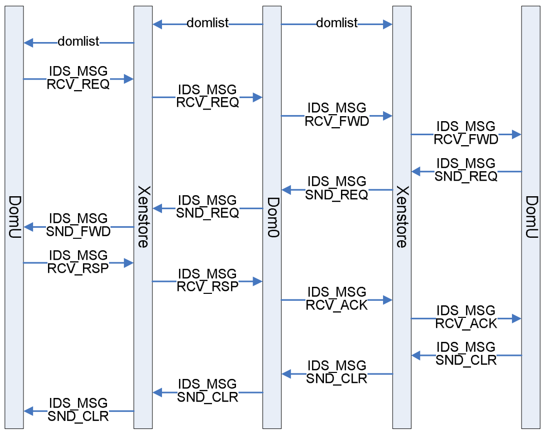


Fig. 3. Configuration Negotiation between DomUs

hand-shaking protocol, and through this process, the sender and receiver DomUs build up a data channel and get ready to data transportation.

And if there is no longer any data to transport, sender DomU also writes a request for removing the data channel into Xenstore. And once this message is forwarded to receiver DomU, receiver DomU tries to unbind from event channel and unmap the shared memory region. Then receiver DomU writes a response message which is to be forwarded to sender DomU into Xenstore. After this message reaches sender DomU, it frees the shared memory pages and withdraws the unbound event channel.

4.3.2 Data Transportation, together with Notification

Data is transported by copying into and reading from shared memory regions. Shared memory is memory pages linking one by one in a cycling-queue with a buffer head and two pointers which represent the sender position and the receiver position, as shown in Fig 4. At first, shared memory is empty so receiver DomU blocks waiting for a notification from sender DomU. Sender DomU writes data to shared memory if only there is free space, that is, that the sender pointer does not reach the buffer head again. When shared memory is full, sender DomU blocks on a wait queue and notifies receiver DomU that shared memory is ready to be read. Then after being notified, the blocked receiver DomU wakes up and takes the data away from shared memory and notifies sender DomU and blocks itself again. If at one time, sender DomU cannot fill all the shared memory pages, it also notifies receiver DomU to wake it up and receiver DomU just copy data from the buffer head to the current sender pointer position.

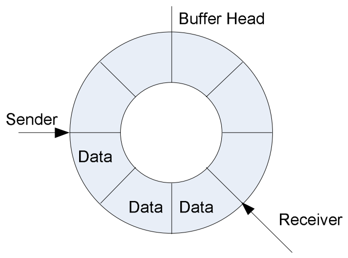


Fig. 4. Buffer for Data Transportation with Sender and Receiver Points

Until now, in our design, the data transportation is unidirectional but not bi-directional. As our main target is to prove that our memory-copying pattern for inter domain communication is effective and to offer a suite of convenient API for user space and our in our future work we will improve the underlying data transportation mechanism to be two-way mode.

4.4 Porting API Suite into User Space

As we mentioned above, idsocket is an API suite for the demand of inter domain communication based on Xen, so all these function must be ported into and accessible in user space. Also, inter domain communication can be treated as one form of network transportation, so we choose the way of registering a new network protocol family in Linux kernel to make the interfaces to shared memory and event channel available from user space. At the same time, we also choose `/proc` file system as the inter-medium to fill the gap between user space and kernel space in configuration negotiation message exchange process.

Table 2. API List in idsocket

Categories	**Sender APIs Pack**	**Receiver APIs Pack**
Configuration Negotiation	get_domlist()	send_conf()
	send_conf()	get_conf()
	get_conf()	
Data Transportation	new_socket()	new_socket()
	ids_wait()	ids_conn()
	ids_send()	ids_receive()
	ids_shut()	ids_shut()

The API suite consists of two parts, the sender pack and the receiver pack. And some APIs for configuration negotiation are shared by both sides, and the API list is shown below in Table 2.

And the hierarchy between user space API and kernel space is illustrated in Fig 5.

It's easy to use idsocket API suite as it's similar to common BSD style socket interfaces except that some interfaces about configuration negotiation is added in.

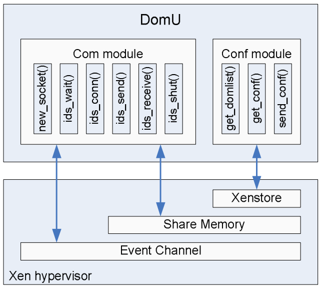


Fig. 5. Hierarchy between User Space API and Xen Component

5 Performance Evaluation

5.1 Test Environment

Performance Evaluation is carried out on a Dell® Optiplex® 760MT Workstation with Intel® Core™2 Quad Q9400 four-core processor, and 2GB main memory. We deployed Xen-3.2.0 and installed CentOS 5.3 as Dom0 and DomUs. And in every DomU, we assign one virtual CPU and 256MB memory, also, DomUs work in text mode and any unnecessary service is stop or removed.

5.2 Benchmarks and Tests

In our experiments, we mainly measure the data transportation rate between DomUs and the processors utilization rate of Dom0.

To test the data transportation rate, we use ttcp as our main benchmark. Ttcp is a "quick-hit" benchmark which is simple tests to measure a certain aspect of performance, but usually do not give a larger perceptive of system performance [6]. “Quick-hit” benchmarks are more suitable for our test in that as an API suite, we believe that idsocket is to be used in “real-world” style data traffic rather than measuring the maximum capacity of transportation most of time.

We measure inter domain communication using idsocket and then compare it with in-built front-back driver inter domain communication in Xen.

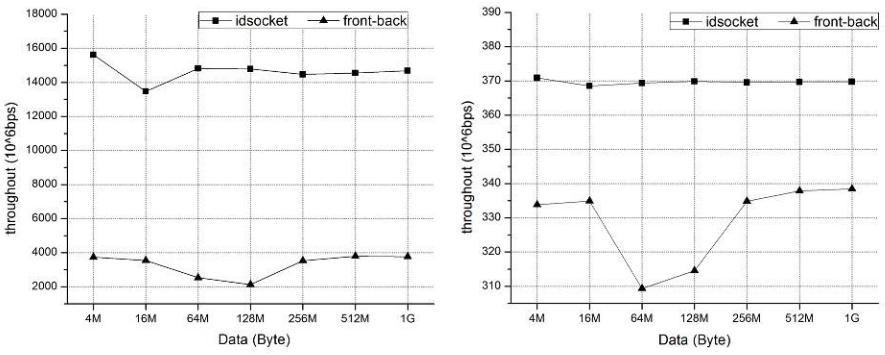
By shared memory and event channel mechanism, data sent from one DomU to another does not have to go through the front-back driver model and be transited by Dom0. So we want to prove that this will greatly save the computing resource of Dom0 by observing the processor utilization rate of Dom0. The processors utilization information is reported by `xm top` command.

All the experiments are taken at least five times and the average results are reported.

5.3 Data Transportation Performance

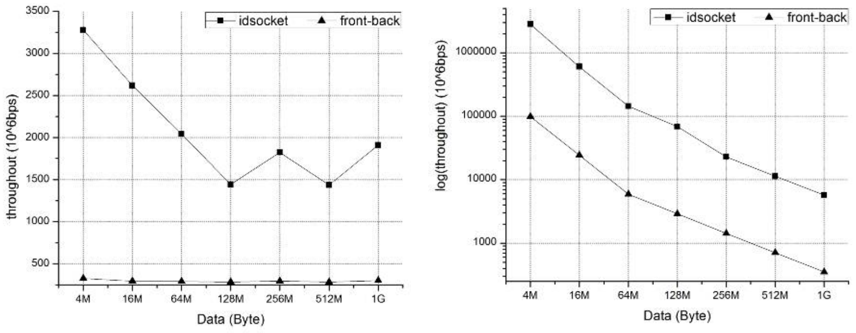
To make a comparison, we run unmodified ttcp and then modify it to use our APIs for data transportation, and the buffer size of a socket in idsocket is one page (4KB), so we configure ttcp also use a socket buffer of same size with `-b 4096` switch. The shared memory of idsocket is set to 32 pages (128KB) and total data amount ranges from 4MB to 1GB.

Fig 6 shows the test result of data transportation rate evaluation. idsocket archived a speed of around 15000*10^6 bits/s (1788MBps) in data sending while in-built front-back driver model only has a sending speed less than 5000*10^6 bits/s (596MBps). And in the receiver side, idsocket has a speed about 370*10^6 bits/s (44MBps), which



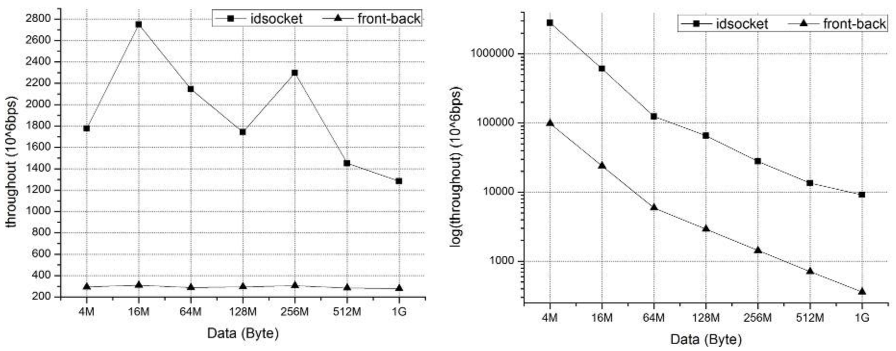
(a) Data Send Rate (b) Data Receive Rate

Fig. 6. Data Transportation Rate Evaluation (128KB Shared Memory)



(a) Data Send Rate (b) Data Receive Rate

Fig. 7. Data Transportation Rate Evaluation (256KB Shared Memory)



(a) Data Send Rate (b) Data Receive Rate

Fig. 8. Data Transportation Rate Evaluation (512KB Shared Memory)

is higher than the receive speed less than 340*10^6 bits/s (40MBps) in front-back model. The efficiency of shared memory and event channel mechanism can be proved.

Then, we modify the shared memory size from 32 pages (128KB) to 64 pages (256KB) and 128 pages (512KB), and perform the same test. The results are illustrated in Fig 7 and Fig 8. And at sending side, idsocket at least has about five times higher than traditional front-back mode. Also, at receiving side, idsocket has an obvious better performance when data size is less than 128MB. From the evaluation test results, we can see that idsocket can send data at a faster speed and in less time, also, idsoket can receive data more quickly when shared memory size is larger than 64KB. But memory resource is rare in kernel mode, so it is recommended that shared memory size is no more than 256KB.

It is obviously seen that, idsocket outperformance in inter-domain communication in throughout, especially in sending data than the front-back network model. This is mainly because the design mechanism which idsocket applies. With UNIX-styled socket, data of inter-domain communication in sending DomU must travel through the protocol stack to Dom0, in which it is forwarded to receiving DomU. During this procedure, serious long time is cost due to data packaging and depacketization plus privileged/unprivileged operation alternation, such as page flip, privileged I/O requests, between Dom0 and DomU [12]. Every bit must follow this way and extra time is added, resulting in the prolonged total transportation time [8]. In contrast, Socket created with interface `new_socket()` is a socket which bypass the traditional Linux network stack, and consequently, the send interface `ids_send()` and the receive interface ids`_receive()` have nothing to do with network stack either. They just operate directly on shared memory, whose reading and writing speed is absolutely fast comparing to the time-consuming protocol stack traveling. As a result, no extra time as what UNIX-style socket to do is attached to, the total transportation time is short enough to make a much higher throughout in the way of shared memory-based idsocket.

5.4 Processor Utilization Evaluation

After that, we continue our test focusing on the processor utilization of Dom0 to validate whether the shared memory and event channel mechanism saves processor cost in Dom0 when data transportation is ongoing. This time, we just write two simple client-server programs with traditional BSD socket using Xen's front-back driver model and idsocket respectively. The programs run in DomU with `xm top` keeping observing the information about processor utilization. The `xm top` command is a part of Xen management tool `xm` written in Python Language. We design our test programs to pause before send/receive function is called. Tests are performed in two scenes, and in the first scene, a pair of DomUs is communicating, and in the second, two pairs of DomUs are communicating respectively.

In the first scene, with front-back driver model, CPU cost of Dom0 is less than 10% only when total data to transport is 4MB, and when the total data size is 512MB, CPU cost raise to about 50%. Fig 9 illustrates this result. This is a serious problem that almost half of the CPU resource is used for inter domain communication and CPU resource left for other task is less and the overall performance of system inevitably decrease. Meanwhile, with idsocket, even if 1GB data is transporting between DomUs, CPU cost in Dom0 keeps steady at about 1%. And in the second scene, which is illustrated in Fig 10, the CPU utilization is no less than 10% in all data size, and when data size is 1GB, the cost raise to about 55%, meaning that more than half of the CPU resource is cost. But with idsocket, CPU utilization also keeps steady at about 3%. Although this is higher than that in the first scene, this CPU cost is still quite small and acceptable.

Consequently, this is also a result of bypassing network protocol stack in virtualized environment. The high resource cost of traditional BSD socket communication in virtualized environment is that virtualized I/O system is implemented in the front-back model, and that is to say that I/O requests from DomUs must be ported to Dom0 and I/O operation, certainly including network communication, is finished with the assistance from Dom0. In Dom0, many data packages come one by one, each of them causing a software interruption, and these interruptions are dealt by the physical processor [12]. This is very resource-consuming since operations like privileged/

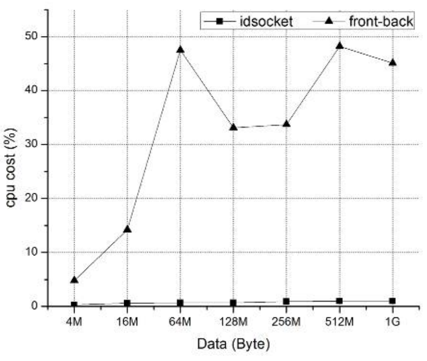


Fig. 9. CPU Utilization Rate Evaluation I

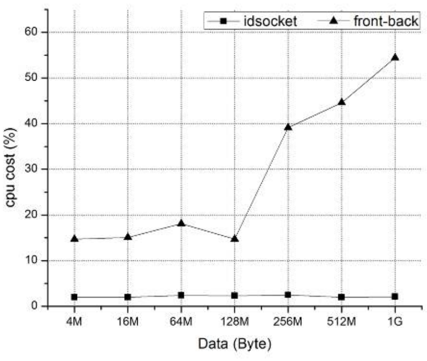


Fig. 10. CPU Utilization Rate Evaluation II

unprivileged commands alternation is triggered and executed for every single data pack, and lots of processor resource is wasted to handle interruption, leaving only a small percentage of free processor resource for other works [8]. As a result, the virtual machine monitor, together with domains, has to endure communication of low throughout and inefficient resource utilization. Meanwhile, as a comparison, the situation in communication between domains with idsocket API suite is quite different. This is because in the shared memory based communication mechanism, Dom0 is not involved in the communication procedure like the in-built communication mechanism does in Xen. Data only transport between DomUs, that is, no network I/O request is generated in the bottom of protocol stack and ported to Dom0. Hence, Dom0 does have to handle any interruption, of type of virtualized network I/O request generated in DomUs, which requires a great deal of processor resource. Not only the processor utilization ratio can keep low in Dom0 and resource in Dom0 is saved for more other works, but also that Dom0, together with DomUs, can get a better running environment.

6 Related Works

In the past few years, new progress and trend on virtualize inter-domain communication appeared. The mostly applied approach, which is also applied by our work, is based on the direct memory copy to bypass the cost made by push data to and pull data from network protocol stack. Recently, there have been some other works and research about inter domain communication, such as XenLoop [2], XWay [3], XenSocket [13], and IVC [9]. XenLoop and XWay provide transparent mechanism for inter domain communication, and XenLoop works as Linux kernel whereas Xway needs to patch Xen source code and rebuild. Both of them have a great performance increase in maximum capacity test, but in "real-world" workload, they may not have such an obvious performance enhancement. Xensocket is an API style implementation for inter domain communication which is much like our works; however, it is a little simple and does not consider anything about configuration negotiation and even requires manually passing parameters to programs. Certainly, it is not convenient for use. IVC provides shared memory communication library for HPC applications in virtual machines residing on a same physical machine. It provides socket-style APIs which an IVC-aware application can invoke.

7 Future Works and Conclusion

As virtualization technology becomes more and more mainstream and popular, it is nowadays deployed widely. Operating systems offering particular function such as web services, data base service and gateway or name server can reside on a same physical machine of high performance but be made isolated running in respectively in different virtual machines. In this way, isolation and security issue is easily resolved than when no virtualization is introduced. However, virtualized I/O mechanism has been proved producing a poor performance. When processes in different virtual machines need to communication with each other, the low band width and throughout make them cannot enjoy the communicate rate in traditional no-virtualized environment.

To resolve this critical problem, in the paper, we present the design and implementation of idsocket, which is an API suite for inter domain communication based on Xen. And the benchmark evaluation and program tests have demonstrated that it has a better performance and a lower resource cost than the in-built front-back driver model for inter domain communication mechanism in Xen. The design principle behind idsocket, memory-sharing and notification mechanism can also be applied to any virtualization environment other than Xen. Prior researches focus on either performance increase or convenience for programmer. But the two sides are contradictory, high performance is easy to archived in kernel level but not in user space yet convenience in user space inevitably has an impact on high performance. Our work has just only begun, and in the future, we will continue investigating on a balance between high performance and convenience, and enhance the implementation of idsocket to make it more efficient in transportation and more convenient to use.

Acknowledgment

This work is supported by the National High Technology Development 863 Program of China under Grant No.2007AA01Z118.

References

1. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: USENIX Annual Technical Conference, Boston, Massachusetts (2006)
2. Wang, J., Wright, K.-L., Gopalan, K.: XenLoop: A Transparent High Performance Inter-VM Network Loopback. In: Proceedings of the 17th International Symposium High Performance Distributed Computing (June 2008)
3. Kim, K., Kim, C., Jung, S.-I., Shin, H., Kim, J.-S.: Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In: Virtual Execution Environments, VE 2008 (2008)
4. Xen-3.2.0, http://www.xen.org

5. Xenbus, http://wiki.xensource.com/xenwiki/XenBus
6. Ttcp, http://www.pcausa.com/Utilities/pcattcp.htm
7. Rosenblum, M., Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends. Computer 38, 39–47 (2005)
8. Chisnall, D.: The Definitive Guide to the Xen Hypervisor, 2nd edn. Prentice Hall, Englewood Cliffs (2007)
9. Huang, W., Koop, M., Gao, Q., Panda, D.K.: Virtual machine aware communication libraries for high performance computing. In: Proc. of SuperComputing (SC 2007), Reno, NV (November 2007)
10. Kourai, K., Chiba, S.: HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. In: Proc. of Virtual Execution Environments (2005)
11. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing performance overheads in the xen virtual machine environment. In: Proc. of Virtual Execution Environments (2005)
12. Muir, S., Peterson, L., Fiuczynski, M., Cappos, J., Hartman, J.: Proper: Privileged Operations in a Virtualised System Environment. In: USENIX Annual Technical Conference (2005)
13. Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J.L.: Xensocket: A high-throughput interdomain transport for virtual machines. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 184–203. Springer, Heidelberg (2007)
14. Cheng, P., Rohatgi, P., Keser, C., Karger, P.A., Wagner, G.M., Reninger, A.S.: Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. Technical Report RC24190, IBM Research, Yorktown Heights, NY, USA (February 2007)

Strategy-Proof Dynamic Resource Pricing of Multiple Resource Types on Federated Clouds

Marian Mihailescu and Yong Meng Teo

Department of Computer Science,
National University of Singapore,
Computing 1, 13 Computing Drive, Singapore 117417
{marianmi,teoym}@comp.nus.edu.sg

Abstract. There is growing interest in large-scale resource sharing with emerging architectures such as cloud computing, where globally distributed and commoditized resources can be shared and traded. Federated clouds, a topic of recent interest, aims to integrate different types of cloud resources from different providers, to increase scalability and reliability. In federated clouds, users are rational and maximize their own interest when consuming and contributing shared resources, while globally distributed resource supply and demand changes as users join and leave the cloud dynamically over time. In this paper, we propose a dynamic pricing scheme for multiple types of shared resources in federated clouds and evaluate its performance. Fixed pricing, currently used by cloud providers, does not reflect the dynamic resource price due to the changes in supply and demand. Using simulations, we compare the economic and computational efficiencies of our proposed dynamic pricing scheme with fixed pricing. We show that the user utility is increased, while the percentage of successful buyer requests and the percentage of allocated seller resources is higher with dynamic pricing.

1 Introduction

Currently, several technologies such as grid computing and cloud computing among others, are converging towards federated sharing of computing resources [11]. In these distributed systems, resources are commodities and users can both consume and contribute with shared resources. In cloud computing [12], resources are provided over the Internet on-demand, as a service, without the user having knowledge of the underlying infrastructure. *Public clouds* are available to all users, while *private clouds* use similar infrastructure to provide services for users within an organization. At the present, several companies such as Amazon [1], Rackspace Cloud [5], and Nirvanix [3], provide computing and storage services, using pay-per-use fixed pricing, and new capabilities, such as .NET and database services [2] are expected in the near future. Cloud computing usage is increasing both in breadth, such as the number of resource types offered, and in depth, such as the number of resource providers. Thus, with an increasing number of cloud users, it is expected that more providers will offer similar services. Furthermore, with interoperability between different providers [7], users will able to use the same service across clouds to improve scalability and reliability. In this context, the aim of *federated clouds*, a topic of recent interest, is to integrate resources from different providers such that access is transparent to the user.

A fundamental problem in any federated system is the allocation of shared resources. Recent work in distributed systems acknowledges that users sharing resources are self-interested parties with their own goals and objectives [28, 20, 16]. Usually, these parties can exercise their partial or complete autonomy to achieve their objectives and to maximize their benefit. They can devise strategies and manipulate the system to their advantage, even if by doing so they break the rules of the system. To manage rational users, economics [30] and mechanism design [24] offer market-based approaches for pricing and allocation of shared resources. Although we cannot assume rational users are trusted to follow the algorithm or protocols designed and deployed, we can assume that they participate in sharing in order to maximize their personal gain, such that incentives may be used to induce the desired behaviour. Mechanism design studies how to structure incentives such that users behave according to protocols. Thus, recent work in peer-to-peer networking [28, 15], grid or cluster computing [20], Internet routing [18], general graph algorithms [17], and resource allocation [10, 31], use a form of incentives to manage rational users.

In this paper we discuss a dynamic pricing scheme suitable for allocating resources on federated clouds, where pricing is used to manage rational users. A rational user may represent either an individual user, a group, or an organization, depending on the application context. In federated clouds, users request more than one type of resources from different providers. In contrast to fixed pricing, where users have to manually aggregate resources from different providers, our pricing scheme is designed to *allocate a request for multiple resource types*. Moreover, in a federated cloud, resource demand and supply fluctuate as users join and leave the system. We show using simulations that using the proposed *dynamic* scheme, the user welfare, the percentage of successful requests, and the percentage of allocated resources is higher than using fixed pricing.

The remainder of this paper is structured as follows. Section 2 presents related works from grid computing and distributed systems. We discuss dynamic pricing for cloud computing and federated clouds in Section 3. Our auction framework is introduced in Section 4, while in Section 5 we evaluate the economic efficiency, the individual user welfare, the impact of multiple resource types and computational efficiency, measured by the computational time incurred by the proposed algorithm. Finally, Section 6 contains our conclusions and discusses our future work.

2 Related Works

Resource markets have been previously proposed for sharing computational resources in the presence of rational users [32, 30, 31, 14, 26, 19]. A *resource market* consists of the environment, rules and mechanisms where resources are exchanged. In this context, related works have used either bartering or pricing to exchange resources. In bartering, resources are exchanged directly, without using any form of currency. For example, in BOINC [9], users donate their CPU cycles by running a software client which polls a server for new jobs. In BitTorrent [15], rational users that behave selfishly and do not cooperate in sharing files are punished by other users. In contrast, in OurGrid [10], each user keeps track of other users that provide resources for their jobs, and prioritize their requests when their own resources are idle. Bartering is simple to implement and allows several types of incentives for rational users: moral incentives (volunteer computing) or coercive incentives (tit-for-tat, network of favors). However, bartering allows exchanges of a single resource type. For example, BitTorrent exchanges blocks from the same file, OurGrid is used for CPU cycles, etc. In order to exchange different types of resources, pricing and a common currency is used to express the value of each resource type.

Pricing is the process of computing the exchange value of resources relative to a common form of currency. Economic models for the allocation of shared resources may use fixed or dynamic pricing. When using fixed pricing, each resource type has a predefined price, set by the seller. For example, Amazon provides disk space for $0.15/GB. In contrast, when using dynamic pricing, the resource price is computed for each request according to the pricing mechanism used. More specifically, a resource type can have the same price for all resource providers (non-discriminated pricing), or payment is computed differently for each resource provider (discriminated pricing). Pricing schemes use financial incentives in addition to payments to motivate rational users to be truthful.

Several market-based allocation systems for grids, such as Sorma [22] and Nimrod/G [13], use bargaining or negotiation to determine the resource price. The advantage of this approach is that sellers and buyers communicate directly, without a third party mediating an allocation. The seller attempts to maximize the resource price, while the buyer strives to minimize it. However, communication constitutes the main disadvantage of bargaining: in a large dynamic market, each buyer has to negotiate with all sellers of a resource type in order to maximize his utility. The communication costs grow further when a buyer requires more than one resource types. Thus, scalability becomes a major issue when increasing the number of users or resource types in a request.

In contrast to resource sharing systems used in research and academic communities or for personal benefit, cloud computing has been put into commercial use and its economic model is based on pricing. Previous unsuccessful cloud computing attempts, such as Intel Computing Services, required users to negotiate written contract and pricing. However, current online banking and currency transfer technologies allow cloud providers to use fixed pricing, with buyer payments made online using a credit-card. Federated clouds can be formed by combining private clouds to provide users with resizeable and elastic capacities [11]. Currently, companies such as Amazon operate as standalone clouds service providers. However, in a federated cloud, any globally distributed user can both offer and use cloud services. A user is either an individual, a group, or an organization, depending on the application context.

3 Market-Based Pricing Mechanisms

Market-based resource allocation mechanisms based on pricing introduce several economic and computational challenges. From a computational perspective, a mechanism must compute in polynomial time the allocation of multiple resource types while maximizing the number of allocated resources and satisfied requests. However, an optimal allocation mechanism for multiple resource types such as combinatorial auctions requires a NP-complete algorithm [23]. Accordingly, many systems share only one resource type, such as CPU cycles in volunteer computing, and file blocks in file-sharing.

From an economic perspective, the desirable properties for resource allocation are: *individual rationality*, *incentive compatibility*, *budget balance* and *Pareto efficiency*

[21]. In an individual rational allocation mechanism, rational participants gain higher utility by participating in resource sharing than from avoiding it. Incentive compatibility ensures that the dominant strategy for each participant is truth-telling. Budget-balance verifies that the sum of all payments made by buyers equal the total payments received by the sellers. *Pareto efficiency*, the highest economic efficiency, is achieved when, given an allocation, no improvement can be made that makes at least one participant better off, without making any other participant worse off. However, according to the Myerson-Sattherwithe impossibility theorem [21], no mechanism can achieve all four properties together. Accordingly, related works have traded incentive compatibility [14,30], economic efficiency [19] or budget-balance [23].

Our approach is designed to achieve individual rationality, incentive compatibility and budget balance using a computationally efficient algorithm that can allocate buyer requests for multiple resource types.

In a resource market, with a large number of providers (sellers) and users (buyers), fixed pricing does not reflect the current market price resource price due to the changing demand and supply. This leads to lower user welfare and to imbalanced markets, e.g. under-demand. Figure 1 shows the welfare lost by a seller that uses fixed pricing. In the case of under-demand, the fixed price tends to be higher than the market price and buyers may look for alternative resources. In the case of over-demand, the fixed price limits the seller welfare, which could be increased by using a higher resource price.

In a federated clouds market, *dynamic pricing* sets resource payments according to the forces of demand and supply. Moreover, the use of dynamic pricing facilitates sellers to provide multiple resource types. Early cloud services such as Sun Grid Compute Utility were restricted to one resource type, e.g. CPU time [8]. More recent services, such as Amazon S3 and EC2, introduced more resource types, i.e. storage and bandwidth. Currently, Amazon has expanded its offer to 10 different virtual machine instance configurations, with different prices for each configuration, and practice tiered pricing for storage and bandwidth [1]. We see this as the first step towards dynamic pricing, where users can request for custom configurations with multiple resource types.

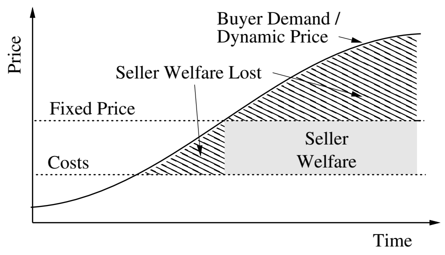


Fig. 1. Fixed Pricing Limits Seller Welfare

Resource_Type = *Description*
Publish = *Seller_Address*, *Resource_Type*, *Items*, *Cost*
Request = *Buyer_Address*, (*Resource_Type*, *Items*)+, *Price*

Fig. 2. Simplified Model for Multiple Resource Types Buyer Request

The resource market in federated clouds consists of many *resource types*. Figure 2 shows a simplified resource model where a buyer request can consist of many resource types and many resource items for each type. A resource type is loosely defined, and can be a hardware resource, a service, or a combination. We consider the example of a New York Times employee that used 100 EC2 instances to convert 4 TB of TIFF files to the PDF format [6]. To complete his job, the user required multiple resource types (storage from Amazon S3 and computational power from Amazon EC2), and multiple items (100 Amazon EC2 instances) to complete his task. In this example, we assume Amazon EC2 provides ten resource types, called *instances*. A *small instance* consists of 1 EC2 compute unit (approx. 1 GHz CPU), 1.7 GB memory and 160 GB storage, and is priced at \$0.10/hour, while an *extra-large instance* consists of 8 EC2 compute units, 15 GB memory and 1.6 TB storage, and is priced at \$1.00/hour.

4 Achieving Strategy-Proof Resource Pricing

In the context of federated clouds, we propose a strategy-proof dynamic pricing mechanism for allocating shared resources with multiple resource types. We assume a federated cloud resource market where rational users can both provide (sellers) and utilize resources (buyers). Rational users represent either an individual or an organization. Interoperability provides the buyers with uniformity and elasticity. Thus, a buyer request for a large number of resources can be met by more than one seller.

In a previous paper, we propose a mechanism design problem which describes a resource sharing system where rational users can be both buyers and sellers of resources [29]. Given a set of alternative choices, a rational user selects the alternative that maximizes the expected value of his utility function. In our mechanism design problem, the utility functions are determined by the seller costs and the buyer budget, respectively.

Definition (The Market-based Resource Allocation Problem). *Given a market containing requests submitted by buyers and resources offered by sellers, each participant is modeled as a rational user i with private information t_i. A seller has private information t_s^r, the underlying costs for the available resource r, such as power consumption, bandwidth costs, etc. The buyer's private information is t_b^R, the maximum price the buyer is willing to pay such that resources are allocated to satisfy its request R. Seller's i valuation is t_i^r if the resource r is allocated, and 0 if not. Similarly, buyer's i valuation is t_i^R of the request R is allocated, and 0 if not. For a particular request R, the goal is to allocate resources such that the underlying costs are minimized.*

To address the market-based resource allocation mechanism design problem in which sellers and buyers are rational participants, we propose a reverse auction-based mechanism, which we prove formally to be individual rational, incentive compatible and budget-balanced. A mechanism that is both individual rational and incentive compatible is known as *strategy-proof*.

Auctions are usually carried out by a third party, called the *market-maker*, which collects the bids, selects the winners and computes the payments. Since this paper focuses on the economical and computational advantages of dynamic pricing, we consider for

```
Allocate()
  while request = request_queue.dequeue()
    // determine winners
    DeterminWinners(request, resource_list,
                    &winners, &$c_{M|s}$)
    foreach seller in winners
      // determine $c_{M|s=\infty}$
      resource_list = resource_list − seller.resources
      DetermineWinners(request, resource_list,
                       nil, &$c_{M|s=\infty}$)
      // determine $c_{M|s=0}$
      $c_{M|s=0} = c_{M|s}$
      foreach Resource Type rt in request
        $c_{M|s=0} = c_{M|s=0}$ − seller.rt.items ∗ seller.rt.price
      end foreach
      payment = $c_{M|s=\infty} - c_{M|s=0}$
      payments.add(seller, payment)
    end foreach
    // if the request is not allocated
    // put it back in queue
    if (request.price > payments.total)
      then request_queue.enqueue(request)
    end if
  end while
end Allocate
```

```
DetermineWinners(request, resource_list,
                 &winners, &payments)
  foreach Resource Type rt in request as subreqest
    resources = filter (resource_list, Resource Type rt)
    priceSort (resources)
    while subreqest.items > 0 do
      // determine sellers
      seller = resources[rt].head().owner
      if seller.rt.items ≥ subreqest.items
        then itemsno = subrequest.items
        else itemsno = seller.rt.items
      end if
      seller.rt.items = seller.rt.items − itemsno
      subreqest.items = subrequest.items − itemsno
      winners.add(seller, itemsno)
      payments.add(seller, itemsno ∗ seller.rt.price)
    end while
  end foreach
end DetermineWinners
```

Fig. 3. Dynamic Resource Pricing Algorithm

simplicity a centralized market-maker, to which sellers publish resources, and buyers send requests. In order to improve scalability, Section 6 shows our insights into distributed auctions, where more than one market-makers are able to auction at the same time. Given that buyers and sellers are globally distributed, it is practical to adopt a peer-to-peer approach, where, after pricing and allocation, buyers connect to sellers to use the resources paid for.

The reverse auction contains two steps, *winner determination* and *payment computation*. Winner determination decides which sellers are selected for allocation, based on the published price, such that the underlying resource costs are minimized. However, to achieve strategy proof using financial incentives, the actual payments for the winning sellers are determined in the second step, based on the market supply for each resource type.

The payment for a seller is determined for each resource type using a VCG-based [17] function, which verifies the incentive-compatibility property for sellers:

$$p_s = \begin{cases} 0, & \text{if seller } s \text{ does not contribute with} \\ & \text{resources to satisfy the request} \\ c_{M|s=\infty} - c_{M|s=0} & \\ & \text{if seller } s \text{ contributes with} \\ & \text{resources to satisfy the request} \end{cases} \tag{1}$$

where:
$c_{M|s=\infty}$ is the lowest cost to satisfy the request without the resources from seller s;
$c_{M|s=0}$ is the lowest cost to satisfy the request when the cost of resources from seller s resources is 0.

To achieve the incentive-compatibility property for buyers, we select the requests using the first-come-first-serve strategy [29]. To obtain budget-balance, the buyer payment function is the sum of all seller payments:

$$p_b = -\sum_{s \in S} p_s \tag{2}$$

where S is the set of winner sellers.

Figure 3 shows the auction algorithm implemented by the centralized auctioneer. Requests are sorted in a queue according to their arrival times and are processed according to the first-come-first-serve policy (line 2). Next, the market-maker solves the winner determination problem (line 4). Payments are computed for each winner (line 6) by determining $c_{M|s=\infty}$ (line 9) and $c_{M|s=0}$ (line 11). Finally, allocation may take place if the buyer price is higher than the buyer payment, which is the sum of seller payments (line 29). Winner determination (line 26) finds the best sellers for all resource types (line 28) based on the published resource item price.

5 Impact of Dynamic Pricing

We evaluate the proposed pricing mechanism both for economic and computational efficiency. Using simulation, we compare our dynamic pricing scheme with fixed pricing, currently used by many cloud providers.

We implement our framework as an application built on top of FreePastry [27], an open-source DHT overlay network environment. FreePastry provides efficient lookup, i.e. in $O(\log N)$ steps, where N is the number of nodes in the overlay. In addition, FreePastry offers a discrete-event simulator which is able to execute applications without modification of the source code. This allows us both to simulate large systems[1], and to validate the results in a deployment over PlanetLab [4].

For simplicity, we use a centralized market-maker to compare the economic and computational advantages of dynamic pricing. A centralized implementation has the advantage of allowing the measurement of economic and computational efficiency with a simple setup for a large simulated network. Moreover, the use of a peer-to-peer substrate such as FreePastry allows us to address the scalability issue in our future work. Thus, our simulated environment contains one market-maker and 10,000 nodes, where each node can be seller and buyer. Publish and request messages are sent to the market-maker node using the FreePastry routing process, which then performs the reverse auctions using the first-come-first-serve policy and computes the payments using the algorithm in Figure 3.

5.1 Economic Efficiency

Traditionally, efficiency in computer science is measured using system-centric performance metrics such as the number of completed jobs, average system utilization, etc. All user applications are equally important and optimizations ignore the user's valuation for resources. Thus, resource allocation is unlikely to deliver the greatest value to

[1] In our experiments, we were able to use up to 35,000 peers in the FreePastry simulator.

the users, especially when having a limited amount of resources. In contrast, economic systems measure efficiency with respect to user's valuations for resources (utility). Consequently, in a Pareto efficient system, where economic efficiency is maximized, a user's utility cannot improve without decreasing the utility of another user.

Economic efficiency is a global measure and represents the *total* buyer and seller welfare. More specifically, there are two factors that affect the economic efficiency: *i)* average user welfare; and *ii)* number of successful requests, for buyers, and number of allocated resources, for sellers.

Using fixed pricing, the average user welfare is constant, since the user utility is also constant. In contrast, when using dynamic pricing, the average user welfare fluctuates with the computed payments, according to the resource demand. Moreover, a dynamic pricing scheme is able to balance the number of successful requests and the number of allocated resources depending on the market condition. For example, resource contention in the case of over-demand is balanced by increasing the resource price. Similarly, buyers are incentivized by a lower price when the market condition is under-demand. Overall, dynamic pricing achieves better economic efficiency both with higher average user welfare, and a higher number of successful buyer requests and allocated seller resources.

5.2 User Welfare

The *user welfare* is determined by the difference between the user utility and payment. In our proposed framework, the user utility is the same as the published price, since both buyers and sellers are truthful, according to the incentive compatibility property of our pricing algorithm. In the case of fixed pricing, we also consider a truthful buyer, i.e. the published request price represents the buyer's utility. However, we do not make the same assumption about sellers, which have a fixed resource price that may differ from the seller's utility. Thus, in our experiment, we compare only the average buyer welfare when using fixed and dynamic pricing, respectively.

For this experiment, we consider a balanced market, where supply and demand are equal. Thus, we assume that the market-maker receives events with an interarrival time of $1s$, where an event has equal probability of being a buyer request or a seller resource publish. Events are uniformly distributed between 10,000 FreePastry nodes, and contain of a number of resource types uniformly distributed between 1 and 3, chosen randomly from a total of 5 resource types. The number of items for each resource type is generated

Table 1. Dynamic Pricing Increases Buyer Welfare

Metric	%Price Variation	Pricing Scheme	
		Fixed	*Dynamic*
avg buyer welfare	10	3.5	4.6
	20	7.4	9.3
	50	18.8	23.3
%succ buyer	10	47.7	62.5
	20	48.8	62.2
	50	49.5	62.1

according to an exponential distribution with mean 10. For sellers, we assume 100 as the fixed price, while in the case of dynamic pricing we vary the price by 10%, 20% and 50%, i.e. the price is generated according to a uniform distribution between 90 and 110, 80 and 120, and 50 and 150, respectively. Buyer price is varied according to the same percentage, shown in Table 1 as *%Price Variation*. The simulation runs for 600,000 events, which, for an arrival rate of $1s$, give a total simulation time of approximately seven days. To reduce sampling error, we run our experiments three times and compute the average.

The results in Table 1 show that dynamic pricing increases the buyer welfare and the percentage of successful buyer requests (*%succ buyer*). Given that the mean buyer utility is 100, and a *theoretical*(the actual maximum welfare can be computed using a NP-complete algorithm, and is smaller than the *theoretical* welfare.) maximum welfare for an item is achieved when having the minimum payment, i.e. $100-$*Price Variation*, we derive that the maximum welfare equals the price variation. Thus, using the proposed dynamic pricing mechanism increases buyer welfare by approximately 10%, when compared to fixed pricing.

5.3 Multiple Resource Types in Different Market Conditions

In contrast to fixed pricing, where users have to manually aggregate resources, the proposed dynamic pricing scheme can allocate buyer requests for multiple resource types. However, with the increase in the number of resource types in the request, it is reasonable to assume that the overall user welfare will decrease. Another factor that influences the user welfare is the market condition. Thus, when there is under-demand, the buyer welfare should increase. Similarly, the seller welfare should increase when there is over-demand. Next, we study the influence of multiple resource types and different market conditions for the proposed dynamic scheme and compare to fixed pricing.

We vary the number of resource types in a request to 5, 10, and 20, while the price variation is set to 20%. We consider 3 market conditions: *Under-Demand*, when supply

Table 2. Dynamic Pricing Increases Efficiency For Multiple Resource Types

Resource Types	**%succ buyer**		**%succ seller**		**avg seller welfare**		**avg buyer welfare**	
	fixed	*dynamic*	*fixed*	*dynamic*	*fixed*	*dynamic*	*fixed*	*dynamic*
Under-Demand								
5	48.4	82.3	24.1	41.8	N/A	2.9	6.2	10.9
10	47.4	86.3	23.4	44.1	N/A	3.0	4.7	9.4
20	46.5	89.7	22.1	46.4	N/A	3.2	3.3	8.1
Balanced Market								
5	48.2	62.4	47.5	61.0	N/A	4.5	6.2	7.9
10	47.1	62.9	46.5	62.5	N/A	4.6	4.7	6.3
20	46.2	63.3	46.0	64.0	N/A	4.7	3.4	4.9
Over-Demand								
5	48.2	42.1	95.4	75.5	N/A	5.9	6.2	6.1
10	47.4	41.4	93.1	74.2	N/A	5.8	4.7	4.8
20	46.2	40.4	91.7	73.0	N/A	5.6	3.4	3.7

is greater than demand, *Balanced Market*, when supply equals demand, and *Over-Demand*, when supply is less than demand. To simulate different market conditions, we vary the probability of a request event. Thus, in the case of a balanced market, the probability is set to 50%, while for under-demand is 33%, and for over-demand is 66%. We measure economic efficiency, i.e. *avg seller welfare* and *avg buyer welfare*, and pricing scheme performance, i.e. *%succ buyer* and *%succ seller*. Table 2 presents our results.

In the case of fixed pricing, the percentage of successful buyer request is close to 50% for all market conditions, since the buyer item price is uniformly distributed with the mean equal to the seller item price. However, the percentage of successful buyer requests decreases when the number of resource types increases since the number of sellers that are allocated to satisfy a request also increases.

In contrast, when using dynamic pricing, the percentage of successful buyer requests varies under different market conditions, according to the forces of supply and demand. Thus, when supply is greater than demand, the percentage of successful buyer requests is higher than in the case of a balanced market, while for over-demand the percentage decreases further. Using the proposed auction mechanism achieves a higher percentage of successful buyer requests and seller allocated resources, in the case of under-demand and balanced market. When demand is higher than supply and the number of resource types in a request increases, there is premise for monopolistic sellers [25], i.e. there are not enough sellers in the market to compute payments using a VCG-based payment function such as the seller payment used by our auction framework.

Similarly, using fixed pricing results in the same mean buyer welfare when varying market conditions, and welfare decreases when increasing the number of resource types. For the proposed pricing mechanism, the buyer welfare is higher when compared to fixed pricing, and varies according to supply and demand: buyer welfare increases when supply is greater than demand, and decreases when demand is higher.

5.4 Cost of Dynamic Pricing

Computational efficiency is a major design criteria in the allocation of shared resources. Optimal mechanisms such as combinatorial auctions [23] are not feasible since the winner determination algorithm is NP-complete. Fixed pricing has the advantage of eliminating the payment computation. To determine the time cost of the algorithm used by the proposed mechanism, we analyze the run-time complexity of the winner determination and the buyer and seller payment functions. Without considering queuing time, the total time incurred by our mechanism is then:

$$T = T_{wd} + T_p$$

where T_{wd} is the time taken to determine the winners, and T_p is the time for the payment computation. We consider the following inputs: RT, the number of resource types in a request; I_{RT_k}, the number of items from the resource type RT_k in a request; S_{RT_k}, the number of sellers with resource type RT_k. We use $\sum_k S_{RT_k}$ to represent the total number of published resources.

The winner determination algorithm in Figure 3 contains 2 loops (lines 28 and 31) for the number of resource types in the request, and the number of items of each resource type, while the inner code (lines 32–41) takes a constant amount of time. Finding all resources with the same type (line 29) depends on $\sum_k S_{RT_k}$, while sorting resources according to their price takes $O(S_{RT_k} \log S_{RT_k})$. Thus, in the worst case, the complexity of the winner determination algorithm is:

$$T_{wd} = O(RT \times (\sum_k S_{RT_k} + S_{RT_k} \log S_{RT_k} + I_{RT_k})) \tag{3}$$

Similarly, we compute the complexity of the payment algorithm (lines 6–18), which, in the worst case scenario, is: $T_p = O(I_{RT_k} \times T_{wd})$, when each winner seller provides one item. Thus, the total time taken by the proposed allocation algorithm is:

$$\begin{aligned} T = T_{wd} + O(I_{RT_k} \times T_{wd}) = O(I_{RT_k} \times T_{wd}) = \\ = O(I_{RT_k} \times RT \times (\sum_k S_{RT_k} + S_{RT_k} \log S_{RT_k} + I_{RT_k})) \end{aligned} \tag{4}$$

In conclusion, the complexity of the algorithm used by the proposed framework is a polynomial function of the number of resource types in a request, RT; the number of items requested for each resource type, I_{RT_k}; the total number of published resources, $\sum_k S_{RT_k}$; and the number of sellers with resource type k, S_{RT_k}.

Figure 4 shows the mean request allocation time obtained in our simulations. We run the simulator on a quad-core Intel Xeon 1.83 GHz CPU with 4 GB of RAM. The figure shows the increase in allocation time due to the increase in number of resource types, and the different market scenarios. In the case of over-demand, the number of sellers is smaller and, consequently, the allocation time is smaller. Similarly, in the case of under-demand, the allocation time increases.

While the allocation time for a small number of resource types is under 1 second, a large number of resource types in the system leads to increased allocation times. Thus,

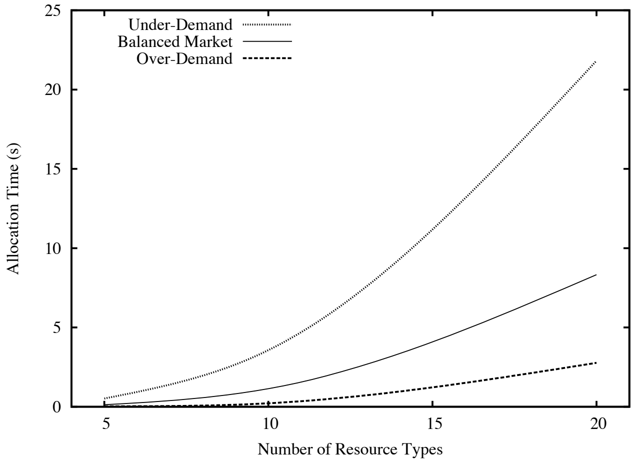


Fig. 4. Allocation Time When Varying Resource Types Under Different Market Conditions

scalability becomes an issue both when the number of resource types, and the number of sellers increases. To improve vertical scalability, i.e. when the number of resource types increases, we are currently developing a distributed framework, where a peer-to-peer network of nodes perform reverse auctions for different resource types at the same time. Furthermore, to address horizontal scalability, i.e. when the number of users increases, different peers maintain separate resource lists and request queues, such that allocation of requests for different resource types is parallelized.

6 Conclusions and Future Work

In this paper, we discuss current resource allocation models for cloud computing and federated clouds. We argue that dynamic pricing is more suitable for federated sharing of computing resources, where rational users may both provide and use resources. To this extent, we present an auction framework that uses dynamic pricing to allocate shared resources. We define a model in which rational users, classified as buyers and sellers, trade resources in a resource market. Our previous paper shows that the payment mechanism used by the proposed auction framework is individual rational, incentive compatible and budget balanced. In this paper we use the defined model to study both the economic and computational efficiency of dynamic pricing, in the context of federated clouds. Our focus is a dynamic pricing scheme where a buyer request consists of multiple resource types. We implement our framework in FreePastry, a peer-to-peer overlay, and use the FreePastry simulator to compare our dynamic pricing mechanism with fixed pricing, currently used by many cloud providers. We show that dynamic pricing increases the buyer welfare, a measure of economic efficiency, while performing a higher number of allocations, measured by the percentage of successful buyer requests and allocated seller resources.

Even though the auction algorithm is polynomial, scalability becomes an issue as the number of resource types in a request increases. We are currently implementing a scheme that uses distributed auctions, where multiple auctioneers can allocate different resource types at the same time. Specifically, by taking advantage of distributed hash tables, we aim to create an overlay peer-to-peer network which supports resource discovery and allocation using the proposed dynamic pricing mechanism.

Acknowledgments

This work is supported by the National University of Singapore under grant number R-252-000-339-112.

References

1. Amazon Web Services (2009), http://aws.amazon.com
2. Microsoft Azure Services Platform (2009), http://www.microsoft.com/azure
3. Nirvanix Storage Delivery Network (2009), http://nirvanix.com
4. An open platform for developing planetary-scale services (2009), http://planetlab.org

5. The Rackspace Cloud (2009), http://www.rackspacecloud.com
6. Self-service, prorated super computing fun (2009), http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun
7. Sun Cloud Computing Initiative (2009), http://www.sun.com/solutions/cloudcomputing
8. Sun Grid Compute Utility (2008), http://www.network.com
9. Anderson, D.P.: BOINC: A System for Public-Resource Computing and Storage. In: 5th IEEE/ACM Intl. Workshop on Grid Computing, Pittsburgh, USA, pp. 4–10 (2004)
10. Andrade, N., Cirne, W., Brasileiro, F.V., Roisenberg, P.: OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 61–86. Springer, Heidelberg (2003)
11. 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 UCB/EECS-2009-28, EECS Department, University of California, Berkeley, USA (2009)
12. Buyya, R., Yeo, C.S., Venugopal, S.: Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In: Proc. of the 10th IEEE Intl. Conf. on High Performance Computing and Communications, Dalian, China, pp. 5–13 (2008)
13. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. In: Proc. of the 4th Intl. Conference on High Performance Computing in Asia-Pacific Region, Beijing, China, pp. 283–289 (2000)
14. Chun, B.N., Culler, D.E.: Market-based Proportional Resource Sharing for Clusters. Technical Report UCB/CSD-00-1092, EECS Department, University of California, Berkeley, USA (2000)
15. Cohen, B.: Incentives Build Robustness in BitTorrent. In: Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, USA (2003)
16. Dani, A.R., Pujari, A.K., Gulati, V.P.: Strategy Proof Electronic Markets. In: Proc. of the 9th Intl. Conference on Electronic Commerce, Minneapolis, USA, pp. 45–54 (2007)
17. Elkind, E.: True Costs of Cheap Labor Are Hard to Measure: Edge Deletion and VCG Payments in Graphs. In: Proc. of the 7th ACM Conference on Electronic Commerce, Vancouver, Canada, pp. 108–116 (2005)
18. Feigenbaum, J., Papadimitriou, C.H., Shenker, S.: Sharing the Cost of Multicast Transmissions. Journal of Computer and System Sciences 63, 21–41 (2001)
19. Lai, K., Huberman, B.A., Fine, L.R.: Tycoon: A Distributed Market-based Resource Allocation System. Technical Report cs.DC/0404013, HP Labs, Palo Alto, USA (2004)
20. Lin, L., Zhang, Y., Huai, J.: Sustaining Incentive in Grid Resource Allocation: A Reinforcement Learning Approach. In: Proc. of the IEEE Intl. Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brazil, pp. 145–154 (2007)
21. Myerson, R.B., Satterthwaite, M.A.: Efficient Mechanisms for Bilateral Trading. Journal of Economic Theory 29(2), 265–281 (1983)
22. Nimis, J., Anandasivam, A., Borissov, N., Smith, G., Neumann, D., Wirstrm, N., Rosenberg, E., Villa, M.: SORMA - Business Cases for an Open Grid Market: Concept and Implementation. In: Altmann, J., Neumann, D., Fahringer, T. (eds.) GECON 2008. LNCS, vol. 5206, pp. 173–184. Springer, Heidelberg (2008)
23. Nisan, N.: Bidding and Allocation in Combinatorial Auctions. In: Proc. of the 2nd ACM Conference on Electronic Commerce, Minneapolis, USA, pp. 1–12 (2000)
24. Nisan, N., Ronen, A.: Algorithmic Mechanism Design (extended abstract). In: Proc. of the 31st Annual ACM Symposium on Theory of Computing, Atlanta, USA, pp. 129–140 (1999)

25. Pham, H.N., Teo, Y.M., Thoai, N., Nguyen, T.A.: An Approach to Vickrey-based Resource Allocation in the Presence of Monopolistic Sellers. In: Proc. of the 7th Australasian Symposium on Grid Computing and e-Research (AusGrid 2009), Wellington, New Zealand, pp. 77–83 (2009)
26. Regev, O., Nisan, N.: The Popcorn Market: An Online Market for Computational Resources. In: Proc. of the 1st Intl. Conference on Information and Computation Economies, Charleston, USA, pp. 148–157 (1998)
27. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object address, and Routing for Large-Scale Peer-to-Peer Systems. In: Proc. of the IFIP/ACM Intl. Conference on Distributed Systems Platforms, Heidelberg, Germany, pp. 329–350 (2001)
28. Shneidman, J., Parkes, D.C.: Rationality and Self-Interest in Peer to Peer Networks. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 139–148. Springer, Heidelberg (2003)
29. Teo, Y.M., Mihailescu, M.: A Strategy-proof Pricing Scheme for Multiple Resource Type Allocations. In: Proc. of the 38th Intl. Conference on Parallel Processing, Vienna, Austria, pp. 172–179 (2009)
30. Wolski, R., Plank, J.S., Brevik, J., Bryan, T.: Analyzing Market-Based Resource Allocation Strategies for the Computational Grid. International Journal of High Performance Computing Applications 15(3), 258–281 (2001)
31. Wolski, R., Plank, J.S., Brevik, J., Bryan, T.: G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid. In: Proc. of the 15th Intl. Parallel and Distributed Processing Symposium, San Francisco, USA, pp. 46–54 (2001)
32. Yeo, C.S., Buyya, R.: A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. Software: Practice and Experience 36, 1381–1419 (2006)

Adapting Market-Oriented Scheduling Policies for Cloud Computing

Mohsen Amini Salehi and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
{mohsena,raj}@csse.unimelb.edu.au

Abstract. Provisioning extra resources is necessary when the local resources are not sufficient to meet the user requirements. Commercial Cloud providers offer the extra resources to users in an on demand manner and in exchange of a fee. Therefore, scheduling policies are required that consider resources' prices as well as user's available budget and deadline. Such scheduling policies are known as market-oriented scheduling policies. However, existing market-oriented scheduling policies cannot be applied for Cloud providers because of the difference in the way Cloud providers charge users. In this work, we propose two market-oriented scheduling policies that aim at satisfying the application deadline by extending the computational capacity of local resources via hiring resource from Cloud providers. The policies do not have any prior knowledge about the application execution time. The proposed policies are implemented in Gridbus broker as a user-level broker. Results of the experiments achieved in real environments prove the usefulness of the proposed policies.

1 Introduction

In High Performance Computing (HPC), providing adequate resources for user applications is crucial. For instance, a computing center that a user has access to cannot handle the user applications with short deadlines due to limited computing infrastructure in the center [2]. Therefore, to get the application completed by the deadline, users usually try to get access to several computing centers (resources). However, managing several resources, potentially with different architectures, is difficult for users. Another difficulty is optimally scheduling applications in such environment.

User-level brokers work on behalf of users and provide access to diverse resources with different interfaces. Additionally, existing brokers such as Gridway [6] and Gridbus broker [11] optimally schedule user application on the available resources. User-level brokers consider user constraints (such as deadline and budget) and user preferences (such as minimizing time or cost) in their scheduling [11].

Recently, commercial Cloud providers offer computational power to users in an on-demand manner and in exchange of a fee. These Cloud providers are also known as Infrastructure as a Service (IaaS) providers and charge users in a pay-as-you-go fashion. For instance, in Amazon Elastic Compute Cloud (Amazon EC2) [1], which is a popular IaaS provider, users are charged in an hourly basis for computational resources. In this paper, we term this charging period as "charging cycle".

The computational power offered by IaaS providers can compensate for the limited computational capacity of non-commercial local resources when they are not enough to meet the user deadline. However, as mentioned earlier, getting access to this extra computational power incurs cost for the user. In fact, there is a trade-off between spending budget to get resources from IaaS providers and running the application on local resources.

Therefore, the problem we are dealing with is how scheduling policies inside the broker can benefit from resources supplied by the IaaS providers in addition to the local resources to get the user application completed by the requested deadline and provided budget. Furthermore, we assume that the end user does not have any knowledge about the application execution time. The problem is more complicated when we consider the user preference in terms of time minimization or cost minimization in addition to the budget and deadline limitations. Such scheduling policies are broadly termed market-oriented scheduling policies [3].

Buyya et al. [3] propose scheduling policies to address the time minimization and cost minimization problem in the context of Grid computing. They term their proposed policies DBC (Deadline Budget Constraint) scheduling policies and define them as follows:

– Time Optimization: minimizing time, within time and budget constraints.
– Cost Optimization: minimizing cost, within time and budget constraints.

However, Buyya et al. do not consider the mixture of non-commercial and commercial resources. Moreover, there are some differences in hiring resources from IaaS providers and assumptions in mentioned DBC policies. One difference is that in the policies proposed by Buyya et al., the user is charged when a job is submitted to a resource. Nevertheless, IaaS providers charge users as soon as a resource is allocated to the user regardless of being deployed by the user or not [10]. Another difference is that current IaaS providers charge users in an hourly basis, whereas in these policies [3] user is charged in cpu-per-second basis. These differences raise the need to propose new DBC scheduling policies to meet the user constrains by hiring resources from IaaS providers.

In this work, we propose two scheduling policies (namely Cost Optimization and Time Optimization) to increase the computational power of the local resources and complete the application by the given deadline and budget.

In summary, our work makes the following contributions:

– We propose the Cost Optimization and the Time Optimization scheduling policies that increase the computational capacity of the local resources by hiring resources from IaaS providers to meet the application deadline within a budget.

– We extend Gridbus broker (as a user-level broker) to hire resources from Amazon EC2 (as an IaaS provider).
– We evaluate the proposed policies in a real environment by considering different performance metrics.

The rest of this paper is organized as follows. In Section 2, related works in the area are introduced. Proposed scheduling policies are described in Section 3. Details of implementation are described in Section 4. Then, in Section 5, we describe the experiments for evaluating the efficiency of the new policies. Finally, conclusion and future works are provided in Section 6.

2 Related Work

A number of research projects have been undertaken over the last few years on provisioning resources based on IaaS providers. The approach taken in these research projects typically consists of deploying resources offered by IaaS providers in two levels. One approach is using resources offered by IaaS providers at resource provisioning level, the other approach deploys resources offered by the IaaS provider at broker (user) level. In this section, a review of these works is provided.

2.1 Deploying Cloud Resources at Resource Provisioning Level

The common feature of these systems is that they do not consider user constrains such as deadline or budget in hiring resources from IaaS providers. In other words, in these works resources offered by IaaS providers are used in order to satisfy system level criteria such as handling peak load.

OpenNebula [5] is a system that can manage several virtualization platforms, such as Xen, inside a cluster. OpenNebula is able to hire resources from IaaS providers, such as Amazon EC2, in an on-demand manner. In OpenNebula hiring resources from Amazon happens when the capacity of local resources is being saturated. Llorente et al. [7] extend OpenNebula to provision excess resources for high performance applications in a cluster or grid to handle peak load.

Table 1. Comparing different aspects of resource provisioning mechanisms from IaaS providers

Proposed Policy	**Use Non-Cloud Resources**	**User constrains**	**User Transparency**	**Scheduling Level**	**Goal**
OpenNebula [5]	Local	No	Yes	System-level	Handling peak load
Llorenete et al. [7]	Local and grid(Globus enabled)	No	Yes	System-level	Provision extra resources to handle peak load
Assunção et al. [4]	Local	Yes (Budget)	Yes	System level	Handling peak load
Vazquez et al. [2]	Local and grid(Globus enabled)	No	No	User level	Federating several providers from Grid and Cloud
Silva et al. [9]	No	Yes (Budget)	No	User level	Run Bag-Of-Task application on Cloud
Time and Cost Optimization (this paper)	Local	Yes (Budget and Deadline)	Yes	User level	Minimizing completion time and incurred cost within a deadline and budget

Assunção et al. [4] evaluate the cost-benefit of deploying different scheduling policies, such as conservative backfilling and aggressive backfilling for an organization to extend its computing infrastructure by allocating resources from an IaaS provider. However, our work proposes cost and deadline aware scheduling policies for user application.

2.2 Deploying Cloud Resources at Broker (User) Level

Vazquez et al. [2] propose dynamic provisioning mechanism by federating grid resources from different domains with different interfaces along with resources from IaaS providers. The federation is achieved through GridWay [6].

In this solution all the resources have to support Globus Toolkit (GT). Even in the case of resources from IaaS providers, Gridway can just awake resources with the Globus adapter. Since this work takes advantage of resources from IaaS providers in the user-level broker, it is similar to our work. However, GridWay neither considers the user constraint in terms of budget nor the user preferences in terms of time or cost minimization. In the mentioned work [2], it is stated that investigating cost-aware scheduling policies for resources from an IaaS provider is required and, in fact, our work addresses this requirement.

Silva et al. [9] propose a mechanism for creating optimal number of resources on Cloud based on the trade-off between budget and speedup. This work considers Bag-of-Tasks applications where the run times are not known in advance. In fact, heuristics proposed by Silva et al. [9] focus on predicting the workload. Nonetheless, our work focuses on providing scheduling policies to satisfy user preferences and we do not deal with workload prediction issues.

In Table 1 different systems that provision resources from IaaS providers are compared from different aspects. This table also reveals the differences of our proposed policies with similar works in the area.

3 Proposed Policy

Scheduling applications is complex when a user places constraints such as execution time and computation cost. Satisfying such requirements is challenging specifically when the local resources are limited in computational capacity and the execution time of the application is not known in advance. In this situation, scheduling policies need to adapt to the changing load in order to meet the deadline and cost constraints. Moreover, the scheduling policy should consider the user preference in terms of time or cost minimization.

To cope with the challenge, our solution relies on supplying more resources from IaaS providers. Therefore, we propose two scheduling policies namely, Time Optimization and Cost Optimization. In this section, details of these policies are described.

3.1 Time Optimization Policy

The Time Optimization policy, as mentioned before, aims at completing the application as quickly as possible, within the available budget. Therefore, according to the pseudo code presented in Algorithm 1, the scheduler spends the whole available budget for hiring resources from the IaaS provider (steps 1 to 3). *Available budget* is defined according to equation (1) and indicates the amount of money the scheduler can spend per hour. However, the number of hired resources should not be more than the number of remaining tasks.

There is a delay between the time the request is sent to the IaaS provider and the time resources become accessible[1]. After getting accessible, hired resources are added to the list of available servers (step 4) and the scheduler can dispatch tasks to them (steps 6, 7). *AddAsServer()* is a thread that keeps track of the request sent to the IaaS provider to get accessible. To attain the minimum execution time, hired resources are kept up to the end of execution. At the end of execution, all resources from the IaaS provider are terminated (step 8).

$$BudgetPerHour = \frac{TotalBudget}{\lceil Deadline - CurrenrtTime \rceil} \tag{1}$$

Algorithm 1. Time Optimization Scheduling Policy.

```
  Input: deadline, totalBudget,resourceCost
1 budgetPerHour =totalBudget / (deadline −currentTime);
2 reqCounter =budgetPerHour / resourceCost;
3 RequestResource(reqCounter);
4 availResources += AddAsServer();
5 DoAccounting();
6 while TaskRemained() = True do
7     SubmitTask(availResources);
8 Terminate(reqCounter);
```

3.2 Cost Optimization Policy

The Cost Optimization policy completes the application as economically as possible within the deadline. According to the pseudo code presented in Algorithm 2, In each scheduling iteration, a set of tasks are submitted to available resources (step 4). Available resources refer to local resources plus resources hired from IaaS provider so far. Then, in step 5 the scheduler estimates the completion time of the remaining tasks based on the time taken for the tasks that have got completed so far.

However, since we are not dealing with the workload prediction issues in this work, we assume that all tasks of the high performance application have the same execution time. Therefore, *EstimateCompletionTime()* in step 5 is a function that estimates the completion time based on equation (2).

$$Estimation = TasksRemained * TaskRunTime \tag{2}$$

[1] The delay is actually because of the time taken to make (boot) a virtual machine from machine image. For more details see [10].

If there is not any available resource (step 6), then a default initial estimation is assumed (step 7) to make the policy hire one resource from the IaaS provider.

In each scheduling iteration, if it is realized that the deadline could not be met and there is enough budget available (steps 9, 10), then just one resource is hired from the IaaS provider. *AddAsServer()* add the hired resource to available resources as soon as it becomes accessible.

Thus, in the Cost Optimization policy resources are requested from the IaaS provider over time. This results in spending more time to get access to hired resources rather than the Time Optimization policy. We investigate the impact of this issue in the experiments in more details.

Termination of the hired resources happens when the estimated completion time is smaller enough than the deadline (steps 15, 16). In fact, to maximize the chance that the deadline can still be met after terminating one resource, termination is only done if the estimated completion time is smaller than a fraction of the deadline ($estimation < (deadline * \alpha)$ where $\alpha < 1$). In the current implementation of Cost Optimization policy, we consider α as a constant coefficient (0.7 in our experiments). However, as a future work, we plan to investigate an adaptive value for α.

DoAccounting(), in both Time Optimization and Cost Optimization policies, takes care of budgeting for hired resources and decreases the available budget base on the price of the hired resources per hour. If there is not enough budget, then *DoAccounting()* terminates each hired resource before it starts a new charging cycle.

Note that the implementation of the above policies contains extra steps to keep track of ordered resources to get accessible, accounting, and terminating hired resources. All of these processes are done in separate threads to have the minimum impact on the scheduling performance.

Time Optimization and Cost optimization policies are implemented in Gridbus broker. Moreover, Gridbus broker is extended to be able to interact with Amazon EC2 as an IaaS provider. In the next section, details of extending the broker to Amazon EC2 are discussed.

4 System Implementation

Gridbus broker mediates access to distributed resources running diverse middleware. The broker is able to interface with various middleware services, such as Condor, and SGE [11]. In this work, we extend Gridbus broker to interact with Amazon EC2 as an IaaS provider. Then, we incorporate the aforementioned scheduling policies into the broker.

In our implemented architecturen (Fig. 1), *EC2ComputesManager* has a key role in managing resources from Amazon EC2. *EC2ComputesManager* initiates a thread that keeps track of resources requested by scheduling policy on Amazon EC2. When a resource gets accessible, the resource is added to the available resources as an *EC2ComputeServer* object and scheduler can submit task to it. *EC2ComputeServer* also deals with the pricing model of the Amazon EC2 by

Algorithm 2. Cost Optimization Scheduling Policy.

```
   Input: deadline, totalBudget,resourceCost
1  SetAvailBudget(totalBudget);
2  while TaskRemained() = True do
3      if availResources > 0 then
4          SubmitTask(availResources);
5          estimation = EstimateCompletionTime();
6      else
7          estimation = deadline + 1;
8      if estimation > deadline then
9          availBudget = GetAvailBudget();
10         if availBudget ≥ resourceCost then
11             RequestResource(1);
12             availResources + = AddAsServer();
13             DoAccounting();
14     else
15         if estimation < (deadline ∗ α) then
16             Terminate(1);
```

overriding the payment method in *isPaid()* method. Finally, *EC2ComputeServer* terminates resources from IaaS provider when *terminate()* method is invoked by the scheduler. *EC2Instance* contains all related attributes and relevant methods for managing resources from the IaaS provider. Particularly, *isReadytoUse()* method determines if a requested resource from the Amazon EC2 is accessible or not.

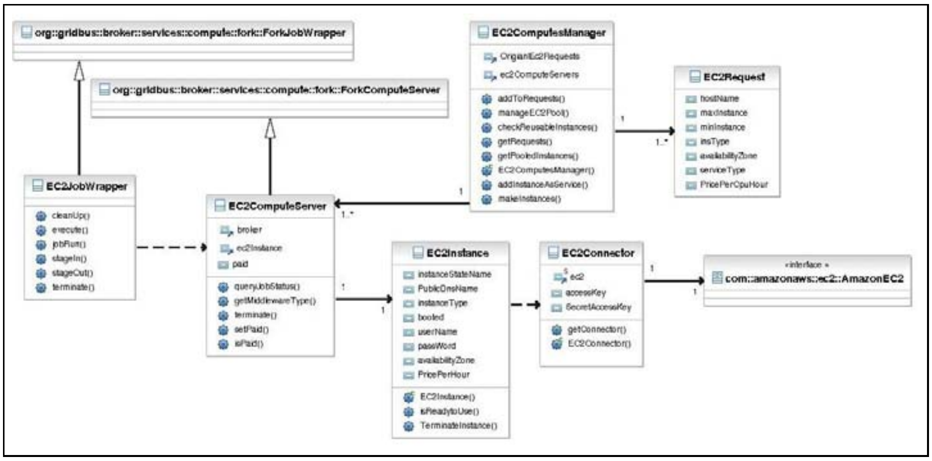


Fig. 1. Class diagram describing the implementation details of extending Gridbus roker to Amazon EC2 as an IaaS provider

Results of our evaluation on the Time Optimization and the Cost optimization policies in Gridbus broker are presented in the next section.

5 Performance Evaluation

5.1 Experiment Setup

The specification of the resources and applications used in the experiments are described in this section. We use a cluster (Snowball.cs.gsu.edu) as the local resource. The cluster has 8 Pentium III (XEON 1.9 GHZ) CPU, 1GB RAM, and Linux operating system. We also use Amazon EC2 as the IaaS provider.

Amazon EC2 offers resources with different computational power. In the experiments we use the cheapest resource type which is known as *small* computational unit (we call it small instance). Another reason for using small instances is that, in terms of hardware specification, small instances are the most similar resource types to our local resources in the cluster. Each small instance is equivalent to 1.2 GHZ XEON CPU, 1.7GB RAM, Linux operating system, and costs 10 cents per hour.

We use a Parameter Sweep Application (PSA) in the experiments. A PSA typically is a parameterized application which has to be executed independently with different parameter values (each one is called a task). Pov-Ray [8] is a popular parameter sweep application in image rendering and it is widely used in testing distributed systems. In the experiments, we configure Pov-Ray to render images with the same size. Therefore, we ensure that the execution time is almost the same for all the tasks.

5.2 Experiment Results

The Impact of Budget Spent on Completion Time. In the first experiment, we measure how the application completion time is affected based on the different budget allocated by the user.

For this purpose, we consider the situation that the user wants to run Pov-Ray to render 144 images in two hours (120 minutes) as the deadline. The overall execution time when just Snowball.cs.gsu.edu cluster is used is 138 minutes and since no cost is assigned to the cluster, the overall execution time does not vary by increasing budgets (Baseline in Figure 2). Then, two proposed policies (Time Optimization and Cost Optimization) are tested in the same situation.

As it can be understood from Figure 2, in the Time Optimization policy the application completion time decreases by available budget almost linearly. However, in the Cost Optimization the completion time does not improve anymore after a certain budget (100 cents in this case). In fact, this is the point that the policy does not spend any more money to request more resources from the IaaS provider even if there is some budget available. Moreover, for the budgets less than 100 cents, Cost Optimization policy takes more time to complete rather than Time Optimization with the same budget allocated. This is mainly because the Cost Optimization policy does not spend all of the allocated budget. This

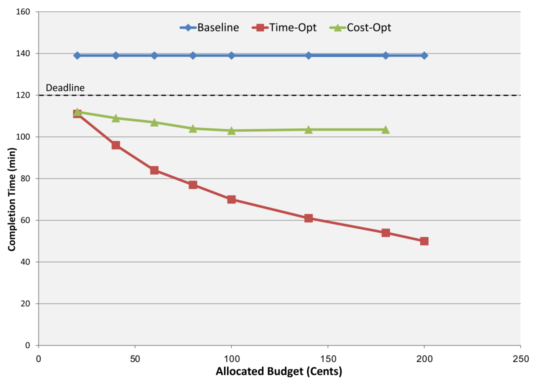


Fig. 2. The impact of allocating more budget on the application completion time without resources from the IaaS provider (Baseline), with Time optimization policy (Time-Opt) and with Cost Optimization policy (Cost-Opt)

issue is discussed more in the next experiment in which the efficiency of the two policies is illustrated. Another reason is that, resources in the Cost Optimization are added over the time and terminated as soon as the scheduler realizes that the deadline can be met. However, in the Time Optimization all resources are requested in the very first moments and kept up to the end of execution.

Efficiency of the Time Optimization and Cost Optimization Policy. In this experiment, the efficiency of the Time Optimization and the Cost Optimization scheduling policies are measured for different amount of allocated budget. We define the efficiency as follows:

$$efficiency = \frac{(\alpha - \beta)}{\delta} \tag{3}$$

Where α is the time taken to complete the application just by deploying Snowball.cs.gsu.edu cluster. β is the completion time by using both Snowball.cs.gsu.edu cluster and resources from the IaaS provider. Finally, δ indicates the budget spent to hire resources from IaaS providers. Other experiment parameters are the same as experiment 5.2.

According to Figure 3, in the Time Optimization policy, the decrease in efficiency (from 1.4 to 0.77 and from 1.38 to 0.89) happens because of the rapid increase in spent budget. However, there is a sharp rise in efficiency (from 0.77 to 1.38) when the allocated budget increases to 100. This is mainly because of the decrease in spent budget. In other words, by hiring five resources from the IaaS provider, the application gets completed before another charging cycle for resources from IaaS provider starts. Therefore, the spent budget decreases sharply (from 80 to 50 cents) and the efficiency increases. We expect more similar sudden rises in the Time Optimization policy, specifically when the deadline is long (several hours or days).

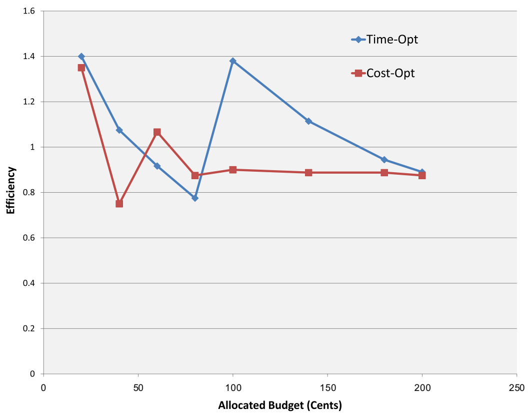


Fig. 3. Efficiency of Time Optimization and Cost Optimization policies with different budget allocated

Similar behavior happens in Cost Optimization, when the allocated budget is increased from 40 to 60 cents. In this case, again more resources (three resources for one hour) are kept for less time instead of fewer resources for more time (two resources for two hours when allocated budget is 40).

The Impact of the Time Optimization and Cost Optimization Scheduling Policies on the Completion Time of Different Workload Types. In this experiment, we investigate how the Time Optimization and the Cost Optimization policies behave for different workload types. Doing this experiment, we consider five workload types. According to Table 2, the number of tasks increase exponentially whereas the run time for each task decreases at the same rate from type 1 to type 5. All of these workloads have the same completion time (150 minutes) when just Snowball.cs.gsu.edu cluster is used. For all of these workloads, we use the same deadline and budget (120 minutes and 100 cents respectively) during the experiment.

This experiment demonstrates the applicability of the proposed policies for different kinds of workloads. As it is illustrated in Figure 4, both policies can get the application completed before the deadline. The only deadline violation is in Cost Optimization policy for workload type 1. The reason is that there is not enough scheduling iteration in which Cost Optimization policy can request more resources from the IaaS provider, thus that workload get completed just by two resources ordered from the IaaS provider.

The minimum difference in completion time between Time Optimization and Cost Optimization is in the workload type 5. The reason is that the execution time for each task is short (2.34 minutes according to Table 2). This results in more frequent scheduling iterations. Hence, extra resources from the IaaS provider are requested in the very first scheduling iteration and these additional resources can contribute more for running tasks. However, the reason for difference in

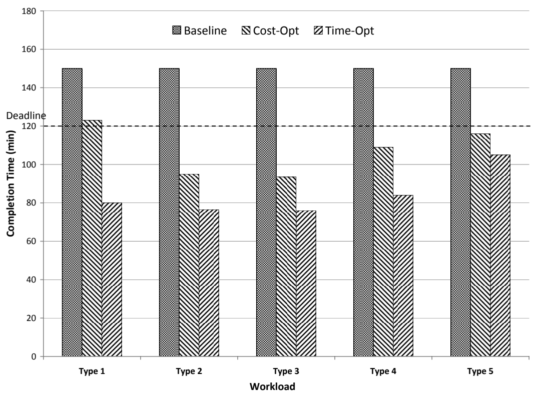


Fig. 4. Application completion time for different workload types with the cluster as local resource (Baseline) and with the resources from the IaaS provider in Time Optimization (Time-opt) and Cost Optimization (Cost-opt) policies

Table 2. Different workload types used in the experiment 5.2

Workload	No of Tasks	Task Time (minutes)
Type 1	32	38
Type 2	64	18.75
Type 3	128	9.37
Type 4	256	4.65
Type 5	512	2.34

completion time is that in the Cost Optimization resources are not retained up to the end of execution. Another reason for difference in completion time is the time taken by the IaaS provider to make the resources accessible. Since in the Cost Optimization policy resources are requested over time, the overhead related to preparing resource by the IaaS provider is longer than the Time Optimization policy in which all resources from the IaaS provider are requested at the same time.

6 Conclusion and Future Work

In this paper, two market-oriented scheduling policies are proposed to increase the computational capacity of the local resources by hiring resources from an IaaS provider. Both policies consider user provided deadline and budget in their scheduling. Time Optimization scheduling policy minimizes the application completion time. On the other hand, Cost Optimization scheduling policy minimizes the cost incurred for running the application. We evaluate these policies in real environment using Gridbus broker as a user-level broker. We observed that in the Time Optimization policy, completion time reduces almost linearly by increasing the budget. However, in the Cost Optimization the completion time does not improve after a certain budget (100 cents in our experiments). We can also conclude that the efficiency of the Time Optimization and Cost Optimization policies can potentially increase by increasing the budget. Finally, we observed that different workload types can get completed before the deadline and within the budget using the proposed policies.

As a future work we plan to extend the current work to a situation that there are several IaaS providers with different prices for their resources.

References

1. Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2
2. Blanco, C.V., Huedo, E., Montero, R.S., Llorente, I.M.: Dynamic provision of computing resources from grid infrastructures and cloud providers. In: Grid and Pervasive Computing Conference, pp. 113–120 (2009)
3. Buyya, R., Murshed, M.M., Abramson, D., Venugopal, S.: Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. Softw. Pract. Exper. 35(5), 491–512 (2005)
4. de Assunção, 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, pp. 141–150. ACM, New York (2009)
5. Fontán, J., Vázquez, T., Gonzalez, L., Montero, R.S., Llorente, I.M.: OpenNEbula: The open source virtual machine manager for cluster computing. In: Open Source Grid and Cluster Software Conference (2008)
6. Huedo, E., Montero, R.S., Llorente, I.M.: A framework for adaptive execution in grids. Softw. Pract. Exper. 34(7), 631–651 (2004)
7. Llorente, I., Moreno-Vozmediano, R., Montero, R.: Cloud computing for on-demand grid resource provisioning. Advances in Parallel Computing (2009)
8. The Persistence of Vision Raytracer, http://www.povray.org
9. Silva, J.N., Veiga, L., Ferreira, P.: Heuristic for resources allocation on utility computing infrastructures. In: MGC, p. 9 (2008)
10. Sotomayor, B., Keahey, K., Foster, I.T.: Combining batch execution and leasing using virtual machines. In: HPDC, pp. 87–96 (2008)
11. Venugopal, S., Buyya, R., Winton, L.: A grid service broker for scheduling e-science applications on global data grids, Citeseer, vol. 18, pp. 685–699 (2006)

A High Performance Inter-VM Network Communication Mechanism

Yuebin Bai, Cheng Luo, Cong Xu, Liang Zhang, and Huiyong Zhang

School of Computer Science, Beihang University, Beijing 100191, China
byb@buaa.edu.cn

Abstract. In virtualization technology domain, researches mainly focus on strengthening the isolation barrier between virtual machines (VMs) that are co-resident within a single physical machine. At the same time, there are many kinds of communication intensive distributed applications such as web services, transaction processing, graphics rendering and high performance grid applications, which need to communicate with each other on the co-resident VMs. Current inter-VM communication mechanisms can't adequately satisfy the requirement of such applications. In this paper, we present the design and implementation of a high performance inter-VM communication mechanism called IVCOM in Xen virtual machine environment. We propose IVCOM in para-virtualization and also extend for full-virtualization. As a result of our survey, in Para-virtualization, there are mainly three kinds of overheads that contribute to the poor performance: the TCP/IP processing cost in each domain, page flipping overhead and long communication path between both sides of the socket. IVCOM achieves high performance by bypassing protocol stacks, shunning page flipping and providing a direct and high performance communication path between VMs residing with the same physical machine. And in Full-virtualization, frequent mode tuning between root mode and non-root mode import too much overhead. IVCOM applies a direct communication channel between domain 0 and hardware virtual VM (HVM) and can greatly reduce the VM entry/exit operations which can improve the HVM performance. In our evaluation, we observe that IVOCM can reduce the inter-VM round trip latency by up to 3 times and increase throughput by up to 3 times which prove the efficiency of IVCOM in para-virtualized environment. In full-virtualized environment, IVCOM can greatly reduce mode tuning times in the communication between domain 0 and HVM.

1 Introduction

VM technologies were first introduced in the 1960s, and reached prominence in the early 1970s. They can create virtual machines which can provide functional and performance isolation across applications and services that share a common hardware platform. At the same time, VMs can improve the system-wide utilization efficiency and provide lower overall operational cost of the system. With the advent of low-cost minicomputers and personal computers, the need for virtualization declined [1]. As growing interests in improving the utilization of computing resources through server consolidation, VM technologies are getting into the spotlight again and are wildly used in many fields. Now, there are many virtual machine monitors (VMMs) such as VMware [2, 3], Virtual PC [4], UML [5], KVM [22], and Xen [6]. Xen system develops one technique known as para-virtualization [7] which offers virtualization with low overhead and has attracted a lot of attention from both the academic VM and the enterprise market. However para-virtualized approach has its intrinsic shortcomings, because it has to modify the OS kernel to shut down the processor's virtualization holes. To implement full virtualization [23] on x86 platform, some processor manufacturers propose hardware assisted technologies to support full virtualization such as Intel's VT technology, and AMD's Pacifica technology.

In spite of the recent advance in the VM technologies, virtual network performance remains a major challenge [8]. Some researches done by Menon et al [9] show that Linux guest domain has far lower network performance than native Linux in the scenarios of inter-VM communication. The communication performance between two processes in their own VMs on the same physical machine is even worse than we expected which is mainly due to the virtualization technologies' main characteristic of isolation. While enforcing isolation is an important requirement from the viewpoint of security of individual software components, it also can result in significant communication overheads as different software components may need to communicate across this isolation barrier to achieve application objectives in specific scenarios. For example, a distributed HPC application may have two processes running in different VMs that need to communicate using messages over MPI libraries. Another example is network transaction. In order to satisfy a client transaction request, a web service running in one VM may need to communicate with a database server which is running in another VM. Even routine inter-VM communication, such as file transfers may need to frequently cross this isolation barrier. In these examples, it is not necessary to use the traditional protocols for inter-VM communication as they are originally developed to transfer data over unreliable WAN. When the VMs reside on the same physical machine, we would like to use a direct and high performance communication mechanism that can minimize the communication latency and maximize the throughput.

In this paper, basing on the research about virtualization technologies and multi-core technologies, we combine both technologies and propose a high performance inter-VM communication mechanism. The rest of this paper is organized as follow: Section 2 gives a brief view of Xen network background. Section 3 presents the design and implementation of IVCOM. Section 4 discusses the overhead and the detailed performance evaluation of IVCOM. Section 5 presents the related work. Section 6 draws the conclusion.

2 Xen Network Background

Xen is an open source hypervisor running between hardware and operating systems (OS). It virtualizes all resources over the hardware and provides the virtualized resources to OS running on Xen. Each OS is called guest domain or domain U and one privileged domain for hosting the application level management software is called domain 0. Xen provides two virtualization ways: full-virtualization and para-virtualization. Xen can provide close to native machine performance by using para-virtualization. In para-virtualization, Xen exports virtualized network devices to each domain U with the actual network drivers that interact with the real network card within domain 0. Domain 0 communicates with Domain U by means of a split network driver architecture shown in Figure 1.

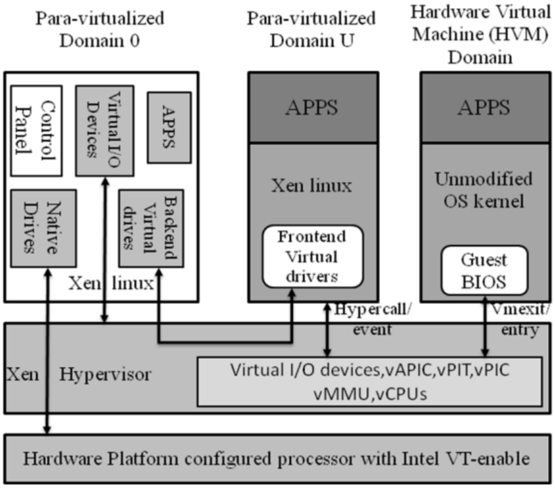


Fig. 1. Xen architecture with hardware-assisted virtual machine support

The domain 0 hosts the backend of the split network driver called netback, and the domain U hosts the frontend called netfront. They interact by using high level network device abstraction rather than low level network hardware specific mechanisms. That means, domain U only knows that it is using a network device, but doesn't care about what type the network card is. The split drivers communicate with each other through using two producer- consumer ring buffers. The ring buffers are a standard lockless shared memory data structure built on grant table and event channels which are two primitives in Xen architecture.

The grant table can be used to share pages between domain U and domain 0. As the frontend of the split driver in domain U can notify Xen hypervisor, by using gnttabl_grant_foreign_access hypercall, that a memory page can be shared with domain 0. Domain U then passes a grant table reference through the event channel to domain 0 that copies data to or from the memory page of domain U. Once complete the page access, domain U removes the grant reference. Page sharing is useful for synchronous I/O operations such as sending packets by a network device. Meanwhile, domain 0 may not know the destination domain for an incoming packet until the entire packet has been received. In this situation, domains 0 will first DMAs the packet into its own memory page. Then, domain 0 can choose to copy the entire packet to the domain U' memory. If the packet is large, domain 0 will notify Xen hypervisor that the page can be transferred to the target domain U. The domain U then initiates a transfer of the received page and returns a free page back to hypervisor. By detailing the network communication process between VMs, we can find there are too much switching of a CPU between domains which can negatively impact the performance due to increase in TLB and cache miss. At the same time, the frequent hypercalls, equal to system calls for hypervisor, also increases the overhead greatly.

In full-virtualization, Intel VT technology defines two modes: root mode and non-root mode for virtual machines. In root mode, virtual machine has whole privilege and has full control of the processor(s) and other platform hardware. In non-root mode, virtual machine can only operate with limited privilege. Corresponding to two modes, VT provides a new form of processor operation called VMX (virtual machine extension) operation. There are two kinds of VMX operation: VMX root operation and VMX nonroot operation. In general, a VMM will run in VMX root operation and guest software will run in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions (also called mode tuning). There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits. In Xen, domain 0 runs in root mode and HVM runs in non-root mode. When applications in HVM need to access hardware resources such as IO, mode tuning will occur. First of all, the current running scene will be saved into a virtual machine control structure (VMCS), and the root mode scene will be load from it. By this way, the HVM domain is scheduled out and domain 0 is scheduled in. Then it is time for handling the real IO request by device module. After that, domain 0 is scheduled out and the domain switches back to HVM. After analysing the process of mode tuning, we can find the actual IO handle cost only take little part in the total overhead in one switch. As all privileged access such as IO access or interrupt will be handled by this way, the performance of HVM certainly degrades.

3 Design and Implementation of IVCOM

3.1 IVCOM Architecture

In IVCOM architecture, there is a discovery module within domain 0 which is responsible for collecting VM information and managing resources such as event channel table and shared memory. In each domain U, there are manage module which is used for the management of communication channels, communication module which is used for the communication between VMs and an IVCOM switch that decides whether uses IVCOM or not in specific scenarios. The details are illustrated in Figure 2.

As we have discussed in section 2, the traditional communication way requires the involvement of domain 0 and results in lower performance. With IVCOM, we can establish high performance communication channel between VMs with little maintenance done by domain 0. The discovery module within domain 0 maintains an event channel table which records necessary information for inter-VM communication. When a new VM is created, first of all, it will send a registration message to discovery module. This message includes the new VM's IP and domain ID which is used to identify each VM within the host OS. After receives this registration message, discovery module will set up a record with several properties for the new VM in the event channel table. Then the manage module in the new VM begins the channel bootstrap process with each existing virtual machine by accessing the event channel table. The process includes the allocation of event channel and the establishment of

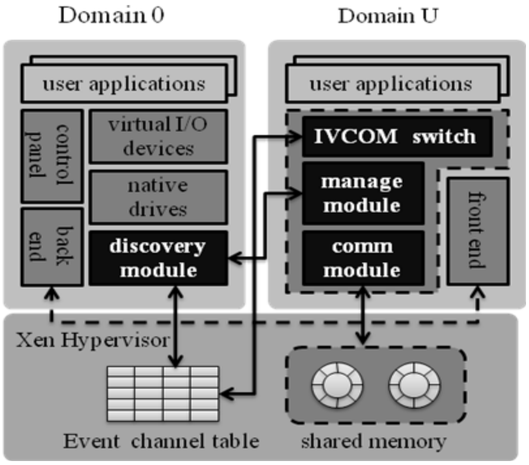


Fig. 2. IVCOM architecture

shared memory. After this, the new VM can exchange with any VMs residing within the same physical computer and achieve high performance inter-VM communication through the data structure of circular queue in the shared memory space.

3.2 Event Channel Table Design

In IVCOM, we use event channel [10] to deliver notifications between VMs. In Xen architecture, events are the standard mechanism for delivering notifications from the hypervisor to guests, or between guests. They are similar to UNIX signals, and can be used for the inter-VM communication. Here we use event channels as part of the communication channels between VMs, and set an event channel between each pair of VMs.

As each VM has to connect with all other VMs residing within the host OS, there are many event channels in each VM. VM needs to know which channel connects with the communication target VM. Therefore, we set up a global event channels table in XenStore to record the necessary information of VM as illustrated in Figure 3. XenStore is a hierarchical namespace which is shared between VMs. It can be used to store information about the VMs during their execution and as a mechanism of creating and controlling VM devices. There are several properties in the event channel table as follow:

- IP: VM's network address
- VM_ID: VM's domain ID
- Port: event channel port num
- Status: VM's status including running(r), pause(p), shutdown(s)

VMs can access "IP" in the event channels table to know the existence of other VMs that reside in the same physical computer. "VM_ID" is used to identify each VM, and "port" is used to identify each event channel in a VM. The terms "event channel" and "port" are used almost interchangeably in Xen. Each event channel contains two ports.

One is in the VM that allocates the event channel and the other is used for remote VM to bind. From the perspective of either VM, the local port is the channel; it has no other means of identifying it. So the VM can access the event channels table to know which port connects the target VM. "Status" shows the current status of VM. Only the VM is in the running status that the corresponding event channel can be used.

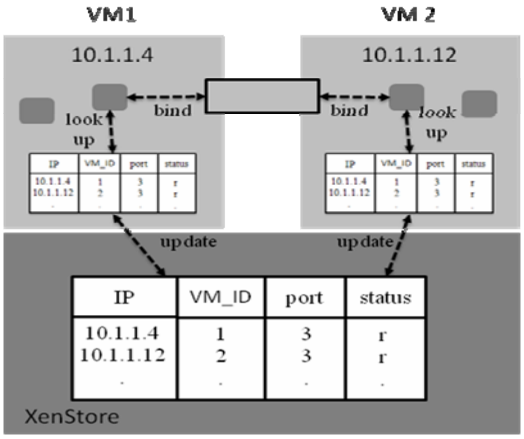


Fig. 3. Event channels table design

To guarantee the validity of the information in the event table, the discovery module will periodically collect the information of VMs to update the event channels table. We set the discovery module to update every 5 minutes which can be adjusted to keep the validity of records. Due to the event channels table being within hypervisor, VMs need to communicate with hypervisor to access it which leads to too much overhead and makes the table become the performance bottleneck. To solve this problem and gain high performance access to the event channels table, we copy the table in each VM. When the VM is created, domain 0 will send it a copy of the table and will update the table periodically to keep the validity of the data. In this way, when a VM has a requirement of communication, it can look up its local event channels table to establish communication path which improves performance greatly.

3.3 Shared Memory Design

Xen's grant tables provide a generic mechanism for memory sharing between VMs which allows shared memory communications between unprivileged VMs. The data transmission between inter-VMs is possible with data copy through the shared memory space. We use the data structure of circular queue to store data in the shared memory space as illustrated in Figure 4. The circular queue is a producer-consumer circular buffer that avoids the need for explicit synchronization between the producer and the consumer endpoints. The circular queue resides in the shared memory between the participating VMs. Both front and back are atomically incremented by the consumer and producer respectively, as they insert or delete data packets into or from the circular queue. When multiple producer threads might concurrently access the front of the queue, we can use producer spinlocks to guarantee mutually exclusive access which still do not require any cross-domain synchronization. The multiple consumers concurrently access the back of the queue also can be solved by this way.

When a new VM is created, the manage module will set up a pair of virtual circular queue for sending and receiving data (SQ and RQ) with each of the existing VM within the host OS. At the same time, the existing VMs also set up a pair of virtual circular queue for data exchange with the new VM. These virtual circular queues in each VM do not have shared memory space. Actually, they are mapped to the physical circular queues which are set up by the discovery module in the shared memory space between the VMs as illustrated in Figure 4. Consequently, a queue in the shared memory space is mapped to the SQ in guest VM1 and also mapped to the RQ in the VM2. In this way, the operation of putting the data into the SQ of VM1 is equal to the operation of putting the data into the RQ of VM2 which achieve the transmission between VMs by one time memory copy [11]. Once the event channel and the circular queues are set up, they won't be destroyed until one of the VMs that the communication channel connects is destroyed.

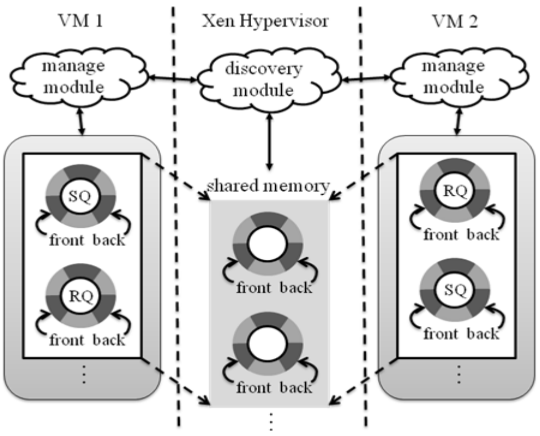


Fig. 4. Shared memory design

3.4 Establish Communication Channel and Data Transmission

When a new VM is created, its manage module will send a register message to the discovery module in domain 0 and receive a copy of event channel table. After that, it begins to establish communication channel including one event channel and two circular queues with each existing VMs. The establishing process is similar to the "client-server" connection setup.

As illustrated in Figure 5, during the channel establishing process, we designate the guest VM with the smaller guest ID as the role of "server", whereas the other VM as the role of "client". First of all, the manage module of the server side send a connect request message to hypervisor, and the hypervisor will send the domain id of server

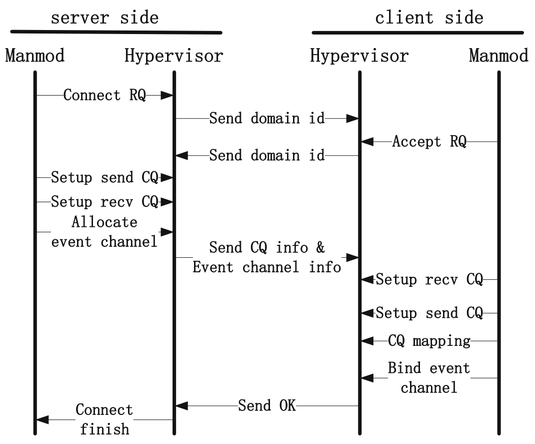


Fig. 5. Channel establishing process

side to the manage module of client side. Client side accepts the request and returns its domain id to the server side. Then the manage module of server side sets up send circular queue and receive circular queue, and allocates an event channel. After this, it will send a message containing the information of the circular queues and the event channel. Receiving the message, the manage module of client side will set up receive circular queue and send circular queue, and map them to the circular queues of server side by the mapping method described in the shared memory design section. The client side also needs to bind to the event channel allocated by the server side, and sends an ack message to the server side. Since then, the channel establishing process is complete. To protect against the loss of either message, the server side will time out if the return ack does not arrive as expected and resend the create channel message 3 times before giving up.

When application in the sender VM has a communication requirement, the IVCOM switch layer will look up the local event channel table to find whether the communication target is within the same physical computer. If the target does not reside within the host OS or the status of the target is not running, IVCOM switch will let the front driver handle the communication requirement. Otherwise, IVCOM will take over to handle the requirement.

First of all, the communication module will access the event channel table to get the right event channel port and the VM's status that connects the target VM. Then the VM will choose a virtual CPU to bind to the port. As only one core can bind to the port at one time, so the virtual CPUs in the same VM need to negotiate to achieve dynamically binding. The asynchronous communication between virtual CPUs in the same VM is implemented basing on APIC (Advanced Programmable Interrupt Controller) [12] which is developed by Intel and used in the communication in multi-core platform. APIC is also adopted by Xen and used in the communication between virtual CPUs. We define two functions that one is used for releasing the event channel called release_evtch and another is used for virtual CPUs to bind to the event channel called bind_evtch. Therefore, a core that wants to use the event channel can send an inter-processor interrupt provided by APIC to the core that is binding to the event channel with release_evtch which will make the event channel available, and then bind itself by using bind_evtch. By this way, the control of the event channel can be transferred within virtual CPUs in a VM.

Once the virtual CPU gains the event channel, the direct data exchange can be available between VMs. The sender VM copies its data packet into the SQ, and then it signals the target VM with the event channel. The target VM intercepts the signal and copies the data packet from the RQ, then frees up the space for future data and returns an ack signal through the event channel.

3.5 Hardware Virtual Machine Extension

As we have discussed in section 2, if the applications in HVM try to access privileged resources such as IO, the HVM will execute VM exit and the domain will switch to root mode to handle the requests. After handling the request, the domain will execute VM entry and it will switch back to HVM. The overhead of frequent transitions between root mode and non-root mode is considerable. According to analyzing the reason of mode tuning, we know that HVM has some tasked done by domain 0 and they need to communicate with each other. Then we are eager to find out if IVCOM is suitable in HVM and the result proves the hypothesis. Although IVCOM cannot put an end to tuning as using hypercalls also will cause mode tuning, it can reduce the tuning time greately.

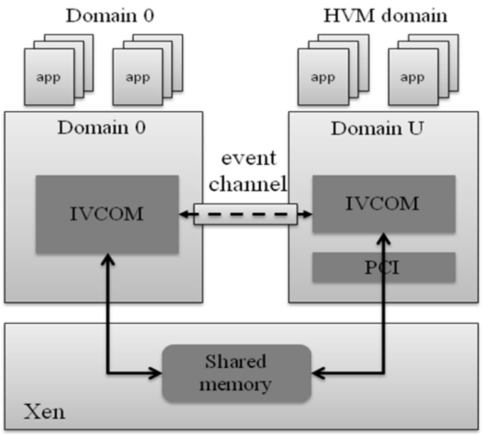


Fig. 6. IVCOM in HVM

The IVCOM cannot be available directly in HVM as it lacks under layer Xen support. Due to un-modified guest, HVM cannot use hypercalls applied by Xen, such as event channel and grant tables, to send notification between guest domains. Therefore, we need to insert a PCI module into the Linux kernel which can enable us to access limited hypercalls. The PCI module enables HVM to access some hypercalls as para-virtualized domain U by mapping its memory page to the hypercall page. After that, HVM can access the memory address in the mapping page to get the entry address of the hypercall.

After inserting PCI module into the Linux kernel as illustrated in Figure 6, we can use IVCOM in HVM as what we do in para-virtualized domain. In HVM, IVCOM can establish the direct communication channel between HVM and domain 0 to achieve high performance communication. When applications in HVM have to access privileged resources, it can use IVCOM to exchange information with domain 0 and let domain 0 done the request instead of mode tuning which degrades the HVM performance a lot.

4 Experiments in Para-Virtualization

In this section, we show the performance evaluation of IVCOM in para-virtualization environment. We perform the experiments on a test machine with Intel Q9300 2.5GHz four core processor, 2 MB cache and 4GB main memory. We use Xen 3.2.0 for the hypervisor and para-virtualized Linux 2.6.18.8 for the guest OS. We configure two guest VMs on the test machine with 512MB memory allocation each for inter-VM communication experiments. We compare the following three scenes:

- IVCOM: Guest to guest communication through IVCOM inter-VM communication mechanism
- Split driver: Guest to guest communication through the split driver
- Host: Network communication between two processes within the Host OS. This experiment works as a standard comparison for other scenes.

To carry out the experiments, we use two benchmarks: netperf [13], lmbench [14] and ping. Netperf is a benchmark that can be used to measure the performance of many different types of networking. It provides tests for both unidirectional throughput, and end-to-end latency. The environments currently measureable by netperf include TCP and UDP via BSD Sockets for IPv4 and IPv6, DLPI, Unix Domain Sockets and SCTP for both IPv4 and IPv6. Lmbench is a set of utilities to test the performance of a Unix system producing detailed results as well as providing tools to process them. It includes a series of micro benchmarks that measure some basic operating system and hardware metrics.

4.1 The Impact of Message Size on Performance

We measure the throughput in three scenes by using netperf's UDP_STREAM test with the sending message size increases, the results are shown in Figure 7. Throughput increases in all three communication scenes along with the increase of message size. This is because a large number of system calls are used to send the same number of bytes with smaller message size, which results in more crossings between user and kernel. When the message size is larger than 512 bytes, IVCOM achieves higher throughput than split driver. The split driver throughput increases slowly and reaches its peak when the message size is 4k bytes. As we have discussed in section 2, split driver uses share memory page to exchange data between VMs. Hence, the maximum data size it can exchange for one time is one page which is 4k bytes. In IVCOM, we

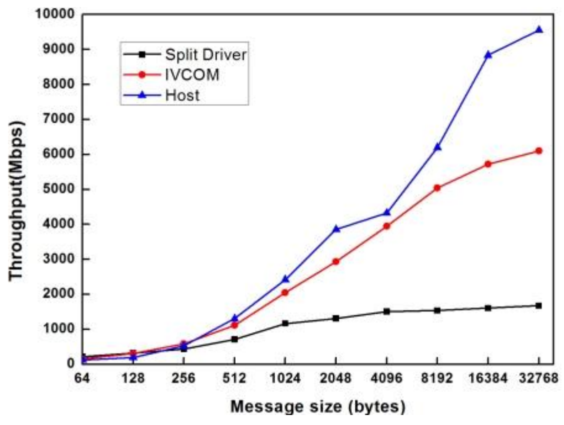


Fig. 7. Impact of message size on throughput

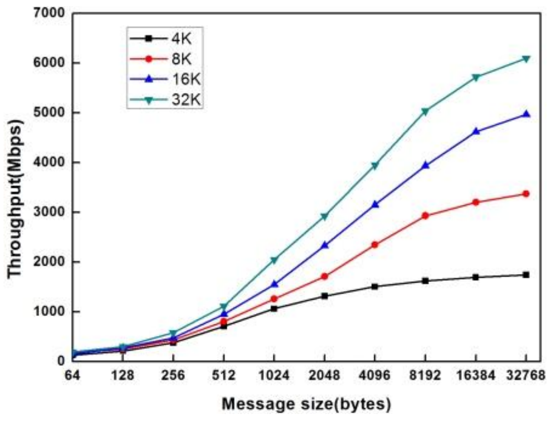


Fig. 8. Impact of circular queue size on throughput

set the circular queue size at 32K bytes, therefore, the throughput of IVCOM increases linearly and reaches the top when the message size is 32k bytes. As there is no shared memory space limitation in the Host scene, the throughput increases linearly along with the increase of message size.

We set the circular queue at size 4K, 8K, 16K and 32K bytes, and then we test IVCOM with different message size. The results are shown in Figure 8. When the circular queue is 4K bytes, the throughput increases slowly and almost reaches the peak at size 4K bytes. When the circular queue is 8K bytes, the throughput increases faster than 4K, and reaches its peak at size 8K bytes. Similarly 16K increases faster than 8K, and touches the top at size 16K bytes. With 32K bytes, we get the similar results. From this experiment we know that increasing the circular queue size has a positive impact on the achievable throughput. In other experiments, we set the circular queue size at 32K bytes.

We also measure the latency of IVCOM and split driver by using netperf UDP_RR test and netperf TCP_RR test as illustrated in Figure 9. When the message size is smaller than 4K bytes, the latency of split driver keeps steady. Once the message size is larger than 4K bytes, the latency increases rapidly. That is because split driver uses one page (4K bytes) to exchange data each time. If the message size is larger than one page size, then it needs to two pages or more to send data which leads to the rapid latency increase. Comparing to split driver, IVCOM has a much smaller latency. The latency keeps steady when the message size is smaller than the circular size. When the message size is too large, the latency of IVCOM also increases a litter which is endurable comparing to split driver.

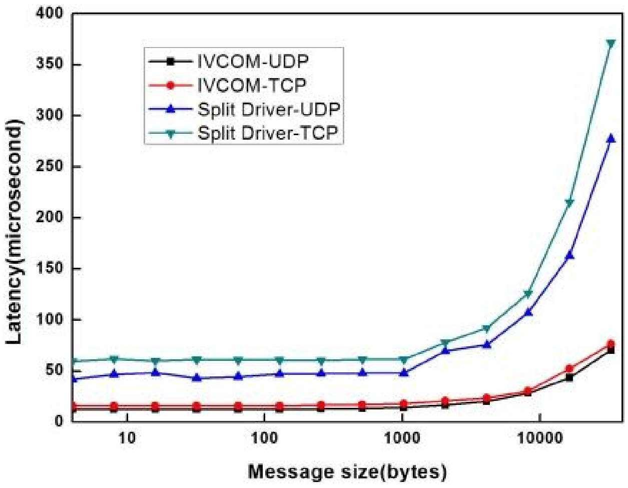


Fig. 9. Impact of message size on latency

4.2 The Impact of VM Amount and Virtual CPU Amount Per VM on Performance

We continue to measure the throughput with the increase of VM amount. In these four tests, each VM is assigned one virtual CPU. The test is based on netperf's UDP_ STREAM test and the result is illustrated in Figure 10. We use the Host test result which is of two processes in the same domain as a standard for comparison. From the figure, we can see that as the VM amount is increased, the throughput is decreased both in split driver and IVCOM and the throughput of Host is kept the same. It can be concluded from the experiment that the throughput of IVCOM is remain unchanged when the VM amount is less than or equal to four, and that is decreased in a linear manner by the amount of VM when the VM amount is more than four.

Finally, our test is focus on the influence about virtual CPU amount per VM. In our test, the machine on which domain0 is running has four processor cores, and at first, we simply estimate that if the virtual CPU amount of a domain U is more than the physical CPU amount, which is four in our experiment, the performance will decrease. The result can be seen from Figure 11. As the amount of virtual CPU in each

VM increases but doesn't reach the total amount of physical CPU, which is four CPU in our machine, the throughput of IVCOM increases greatly. And when the amount of virtual CPU is equal to or more than the physical CPU total amount, the throughput of IVCOM almost remains the same. So increasing the throughput of IVCOM by increasing the virtual CPU amount is effective only if that virtual CPU amount is less than physical CPU amount and our prediction at the beginning is not so correct. During the entire test, the throughput in split driver and Host (same as the prior experiment) keep steady regardless the amount of virtual CPU and its relationship with the amount of physical CPU.

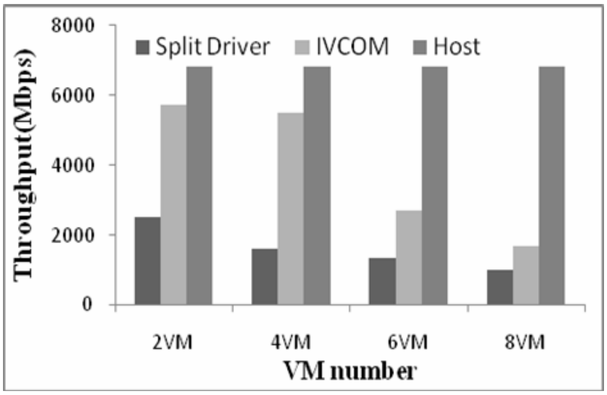


Fig. 10. Impact of VM number on throughput

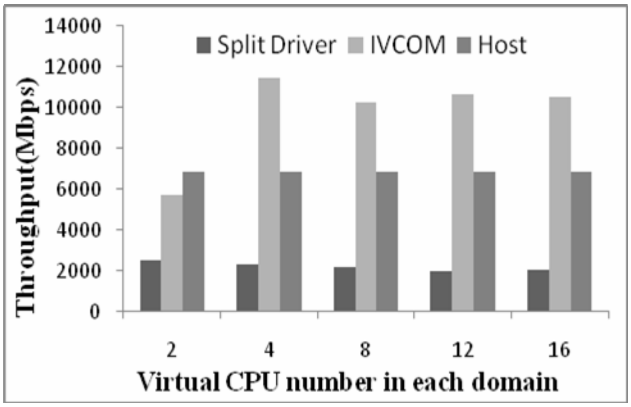


Fig. 11. Impact of virtual CPU number in each domain on throughput

5 Experiments in Full-Virtualization

In this section, we show the performance evaluation of IVCOM in full-virtualization environment. We configure the guest VMs on the test machine with 512MB memory. We compare the following two scenes:

- HVM_IVCOM: HVM to domain 0 network communication through IVCOM
- HVM_Tuning: HVM to domain 0 network communication through mode tuning

To carry out the experiments, we write a benchmark sending a 500M bytes data from HVM to domain 0. We use Xentrace to calculate the tuning times in two scenes. In our study, we focused on the total tuning times reduced by using IVCOM instead of traditional communication ways in HVM.

5.1 Experiments and Analysis

First of all, we use the traditional way to send 500M bytes data from HVM to domain 0. In each time, we adjust the sending data size to test the impact of size on VMentry/exit times.

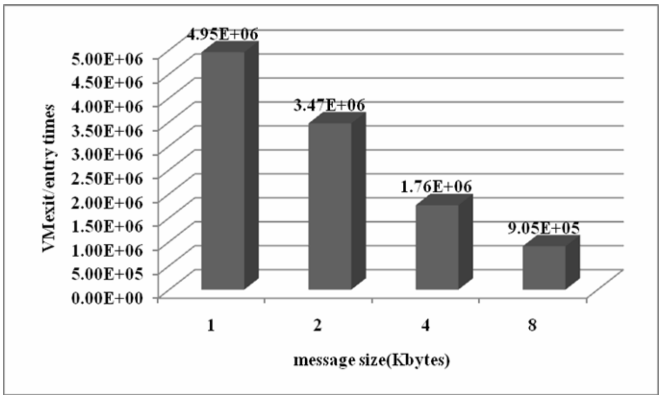


Fig. 12. Tuning times with HVM_Tuning

From Figure 12 we know that the VMentry/exit times increase along with the decrease of the message size. In another word, we can say VMentry/exit times increase along with the increase of message package number. This is because the domain needs to switch between HVM and domain 0 whenever there is a message package need to send. Therefore, it is better for HVM to send data in a big size of message to reduce the total tuning times. But, in most situations in HVM, the message size is small. For example, message size in socket communication is about 1400 bytes. IVCOM can solve this problem because it can use shared memory to exchange data between HVM and domain 0. There is almost no message size limitation only if there is enough space.

We repeat the experiment with IVCOM and the results are shown in Figure 13. As we have talked above, the message size is about 1400 bytes in traditional ways, and the corresponding number of tuning times is about 4.95E+6. We take this number as a reference for the results in Figure 13. When the shared memory is set to be one page, (as we can make the message size as large as the whole shared memory), the number of tuning times is 1.69E+6, which is only about one third of the number of tuning times in traditional ways. When the shared memory is set to be two pages, the number of tuning times is 8.85E+6, which is only about one fifth of the mode tuning times'

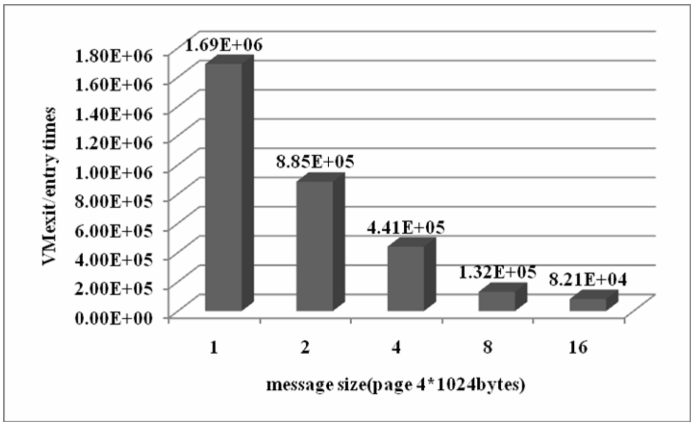


Fig. 13. Tuning times with HVM_IVCOM

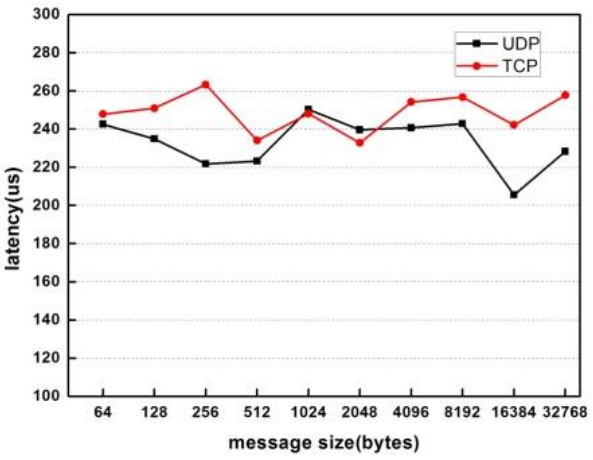


Fig. 14. Latency in traditional mode tuning

number. As the shared memory becomes larger, the number of mode tuning times decreases rapidly. This is mainly because it can send more data during each mode tuning. By this way, it will occur less mode tuning by using IVCOM instead of traditional ways when there is the same size data need to be exchanged between HVM and domain 0. Therefore, IVCOM can improve the performance by reducing the number of mode tuning times and improve the effective operations.

Through the experiments, we also find that the latency of IVCOM is bigger than the latency of traditional mode tuning. And the size of shared memory in IVCOM also affects the latency. The results are shown in Figure 14 and Figure 15.

From Figure 14 we know that the latency of traditional mode tuning is between 200 and 300 us while the latency of IVCOM is between 1300 and 1700 us when we set the shared memory 16 memory pages (each page is 4096 bytes). To have a further understand the effect of shared memory size on the latency, we repeat the experiment and adjust the shared memory size; the results are shown in Figure 16.

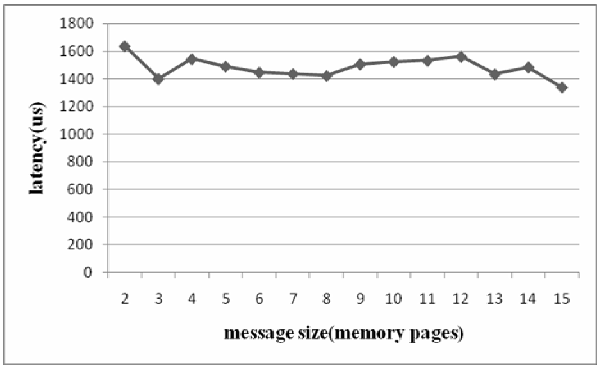


Fig. 15. Latency in IVCOM

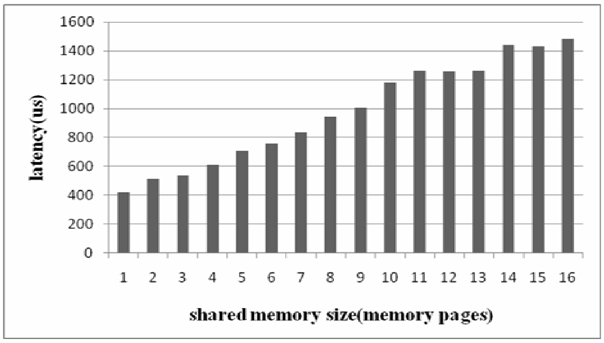


Fig. 16. Latency in IVCOM with different shared memory size

When the shared memory size is small, the latency of IVCOM is small. As when the shared memory size is one page (4096 bytes), the latency is only about 400 us. The latency is still endurable for most application. As the shared memory size increases, the latency also increases. The latency can touch 1500us when the shared memory size is 16 memory pages which is insufferable for many applications especially those latency sensitive applications.

Therefore, we have a further study on this problem and propose an N rank M/M/1 queuing module to reduce the communication latency for latency sensitive applications. The main idea of this module is to set up several IVCOM channels which have different memory size. For example, there are N IVCOM channels; we take each one as an M/M/1 queuing module. And the N channels make up of the N rank M/M/1 queuing module. When an application arrives, it calculates the average latency of each sub-module and chooses one channel whose latency can match the need of the application. By this way, it can solve the high latency problem in some extent. This is not a perfect solution, we still need further study to work out a better solution to solve the high latency problem.

6 Related Work

There have been some researches to improve the inter-VM communication performance in virtualization environment. For example, XWay [15], XenSockets [16] and IVC [17] have developed tools that are more efficient than traditional communication path that needs to via domain 0. XWay provides transparent inter-VM communication for TCP oriented applications by intercepting TCP socket calls beneath the socket layer. It requires extensive modifications to the implementation of network protocol stack in the core OS since Linux does not seem to provide a transparent netfilter-type hooks to intercept messages above TCP layer. XenSockets is a one-way communication pipe between two VMs which is based on shared memory. It defines a new kind of socket, with associated connection establishment and read-write system calls that provide interface to the underlying inter-VM shared memory communication mechanism. In order to use these calls, user applications and libraries need to be modified. XenSockets is suitable for applications that are high throughput distributed stream systems, in which latency requirement are relaxed, and that can perform batching at the receiver side. IVC is a user level communication library intended for message passing HPC applications. It can provide shared memory communication across VMs that reside within the same physical machine. It also provides a socket-style user-API using which an IVC aware application or library can be written. IVC is beneficial for HPC applications that can be modified to explicitly use the IVC API. In other application areas, XenFS [18] improves file system performance through inter-VM cache sharing. HyperSpector [19] permits secure intrusion detection through inter-VM communication. Prose [20] utilizes shared buffers for low-latency IPC in a hybrid microkernel-VM environment. Proper [21] introduces techniques that allow multiple PlanetLab services to cooperate with each other.

7 Conclusion

With the resuscitation of virtualization technologies and the development of multi-core technologies, using virtualization to enforce isolation and security among multiple cooperating components of complex distributed applications draws more and more attention. This makes it imminently for the virtualization technologies to enable high performance communication among them. In this paper, we presented the design and the implementation of a high performance inter-VM communication mechanism called IVCOM. It achieves high performance communication by bypassing the standard protocol stacks and establishing direct communication channel between VMs to exchange data. Evaluation using a number of benchmarks demonstrates a significant increase in communication throughput and reduction in inter-VM round trip latency. For future work, we are presently investigating if IVCOM can have a better performance when the message size is small. We also want to get a further study about the impact of multi-core and virtual multi-core on the performance of IVCOM which may help us enhance its performance on multi-core. In full-virtualization, although IVCOM can reduce the number of mode tuning times greatly, it also can import high latency. How to solve this problem is also one of our future works.

References

1. Figueiredo, R., et al.: Resource Virtualization Renaissance. IEEE Computer 38(5), 28–31 (2005)
2. VMware, `http://www.vmware.com/`
3. Sugerman, J., Venkitachalam, G., Lim, B.: Virtualizing I/O devices on VMware Workstation's Hosted Virtual Machine Monitor. In: USENIX Annual Technical Conference (2001)
4. Windows Virtual PC, `http://www.microsoft.com/windows/virtual-pc/default.aspx`
5. The User-mode Linux Kernel, `http://user-mode-linux.sourceforge.net`
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: 19th ACM Symposium on Operating Systems Principles (2003)
7. Whitaker, A., Shaw, M., Gribble, S.: Denali: Lightweight virtual machines for distributed and networked applications. In: The USENIX Annual Technical Conference, Monterey, CA (2002)
8. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: USENIX Annual Technical Conference, Boston, Massachusetts (2006)
9. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing performance overheads in the xen virtual machine environment. In: Virtual Execution Environments, VEE 2005 (2005)
10. Chisnall, D.: The definitive guide to the Xen Hypervisor. In: Chapter 7: Using Event Channels, November 2007. Prentice Hall, Englewood Cliffs (2007)
11. Huang, W., Koop, M.J., Panda, D.K.: Efficient One-Copy MPI Shared Memory Communication in Virtual Machines. In: IEEE CLUSTER 2008 (2008)
12. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, Chapter 9: Advanced Programmable Interrupt Controller (APIC), Order Number: 253668-029US (November 2008)
13. Netperf, `http://www.netperf.org/netperf/`
14. McVoy, L., Staelin, C.: lmbench: Portable tools for performance analysis. In: Proc. of USENIX Annual Technical Symposium (1996)
15. Kim, K., Kim, C., Jung, S.-I., Shin, H., Kim, J.-S.: Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen. In: Virtual Execution Environments, VEE 2008 (2008)
16. Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J.L.: Xensocket: A high-throughput interdomain transport for virtual machines. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 184–203. Springer, Heidelberg (2007)
17. Huang, W., Koop, M., Gao, Q., Panda, D.K.: Virtual machine aware communication libraries for high performance computing. In: SuperComputing, SC 2007, Reno, NV (November 2007)
18. XenFS, `http://wiki.xensource.com/xenwiki/XenFS`
19. Kourai, K., Chiba, S.: HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. In: Virtual Execution Environments, VEE 2005 (2005)
20. Hensbergen, E.V., Goss, K.: Prose i/o. In: First International Conference on Plan 9, Madrid, Spain (2006)
21. Muir, S., Peterson, L., Fiuczynski, M., Cappos, J., Hartman, J.: Proper: Privileged Operations in a Virtualised System Environment. In: USENIX Annual Technical Conference, Anaheim, California (2005)
22. Qumranet Inc. KVM: Kernel-based Virtual Machine, `http://kvm.sourceforge.net/`
23. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. In: Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, California, USA, October 21-25 (2006)

On the Effect of Using Third-Party Clouds for Maximizing Profit

Young Choon Lee[1], Chen Wang[2], Javid Taheri[1], Albert Y. Zomaya[1], and Bing Bing Zhou[1]

[1] Centre for Distributed and High Performance Computing,
School of Information Technologies,
The University of Sydney,
NSW 2006, Australia
{yclee,javidt,zomaya,bbz}@it.usyd.edu.au
[2] CSIRO ICT Center, PO Box 76,
Epping, NSW 1710, Australia
chen.wang@csiro.au

Abstract. Services in cloud computing systems are typically categorized into three types—software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). These services can be prepared in the form of virtual machine (VM) images; and they can be deployed and run dynamically as clients request. Since the cloud service provider has to deal with a diverse set of clients, including both regular and new/one-off clients, and their requests most likely differ from one another, the judicious scheduling of these requests plays a key role in the efficient use of resources for the provider to maximize its profit. In this paper, we address the problem of scheduling arbitrary service requests of those three different types—taking into account the maximization of profit—in cloud environments, and present the client satisfaction oriented scheduling (*CSoS*) algorithm. Our algorithm effectively exploits different characteristics of those three service types and the availability of third-party cloud service providers who have (or are capable of having) identical service offerings (using virtual machine images). Our main contribution is the incorporation of client satisfaction into our request scheduling; this incorporation enables to increase profit by avoiding the discontinuation of service requests from those unsatisfied clients due to the poor quality of service.

1 Introduction

Cloud computing has become a promising on-demand computing platform. A cloud is an aggregation of resources/services—possibly distributed and heterogeneous—provided and operated by an autonomous administrative body. Services offered by cloud providers can be classified into software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). Typical examples of these three types are service offerings of Salesforce.com [1], Microsft Azure [2], Google App Engine [3], Amazon EC2 and S3 [4, 5]. The primary driving force of this new computing paradigm is its cost effectiveness, i.e., the "pay-per-use" pricing model, which allows clients/users to flexibly rent a variety of computing resources and software packages as services. Typical examples of these service instances include processors, storage capacities, network bandwidths and application software development suites. Scalability, availability, fault-tolerance and flexibility are other key benefits [6, 7].

While IaaS requests are typically placed with fixed durations, requests of SaaS and PaaS are often made, and serviced until the provider is notified their discontinuation. This implies that, for a cloud service provider who has offerings of all three service types (IaaS, SaaS and PaaS), resource allocation for service requests in the latter two types may cause an inefficient use of resources leading to poor service to clients, since service termination times of the latter two types are hardly determined a prior.

In this study, we address the problem of scheduling arbitrary service requests of those three different types—taking into account the maximization of profit—in cloud environments. The cloud service provider has to deal with a diverse set of clients, including both regular and new/one-off clients, and their requests most likely differ from one another. A cloud in this study consists of homogeneous resources (e.g., an Amazon example; the homogeneity of a particular type of instances, such as small, large and extra large) and service requests arrive in a Poisson process. Based on these factors, dynamic scheduling/resource allocation is the only option.

We present the client satisfaction oriented scheduling (*CSoS*) heuristic as a novel profit-driven service-request scheduling algorithm. Our approach effectively exploits different characteristics of those three service types and the availability of third-party cloud service providers. Specifically, service requests, for which their deadlines cannot be met based on the current schedule of the provider, may be forwarded/outsourced to other (perhaps larger) cloud service providers (e.g., Amazon and Salesforce.com) transparently to clients. Here, our main objective is to maximize profit by accommodating as many service requests as possible maintaining a certain quality of service. This scenario might best suit to small and mid-size cloud providers. Our main contribution is the incorporation of client satisfaction into our request scheduling (to the best of our knowledge, the work in this study is the first attempt); this incorporation enables to increase profit by avoiding the discontinuation of service requests from those unsatisfied clients due to the poor quality of service.

The remainder of the paper is organized as follows. Section 2 describes related work. In Section 3, we define the profit model of a cloud service provider in detail. Our *CSoS* algorithm is presented in Section 4 followed by experimental results in Section 5. We summarize our work and draw a conclusion in Section 6.

2 Related Work

The cloud computing represents a trend of shifting data and computing away from personal computers and local servers to large scale remote data centres. The problem we deal with in this paper belongs to a common theme for efficiently managing data centres that host a variety of third-party services. To achieve efficiency, [4] profiles different types of services and uses the profiles to guide the placement of service components.

Market-based resource allocation methods introduce valuation functions to characterize the resource needs from different services [8, 10, 11, 12, 13, 14, 15, 16, 17, 18].

It is a promising approach as pricing can potentially reveal the needs of services accurately. Pricing models differentiate resource allocation mechanisms. *Muse* [8] assumes the service requests are small and the resource demand for service requests is stable. The valuation function of a resource is therefore derived from the throughput of request processing. The approaches in [10, 16] price a resource based on the proportion the resource is allocated to a service. Chun et al. [17] introduced time-varying resource valuation for jobs submitted to a cluster. The changing values of a resource to services requesting it were used for prioritizing and scheduling batch sequential and parallel jobs. The job with the highest value per time unit is put ahead of the queue to run. Irwin et al. [18] extended the time-varying resource valuation function to take account of penalty when the value was not realized. The optimization therefore also minimizes the loss due to penalty. The valuation function used in our model is time-varying, but the penalty is represented as the dissatisfaction of a service owner.

FirstPrice and *FirstProfit* are two common algorithms used in market-based resource allocation. *FirstPrice* orders the requests in the queue based on user-specified time-varying bids [17]. The request with the highest bid is allocated the resource first. *FirstProfit*, instead, maximizes the per-request profit for each request independently, i.e., the service request with the maximum profit obtainable is selected and scheduled onto a resource (resource/infrastructure instance) [15]. Our algorithm differs from *FirstPrice* and *FirstProfit* in the way we handle user dissatisfaction. Our algorithm minimizes the refusals of service requests as it incurs long term profit loss.

3 Profit Model of a Cloud Service Provider

A set of services $S = \{s_1, s_2, \ldots, s_n\}$ is hosted in a set of shared cloud resources (infrastructures) $C = \{c_1, c_2, \ldots, c_m\}$ (more often than not, $n > m$). A service s can spawn instances to run on any resource from a prepared VM image. We assume the following:

- The image of a service is available at any c_i, $0 < i \le m$, i.e., migrating a service from c_i to c_j does not incur any communication cost;
- The performance of a service is linear to the number of instances it runs;
- The requests of starting a new instance arrive as a Poisson process and the utilization of the instances of all services is similar (a service will not start a new instance if the utilization of its existing instances is low);
- The instance lifetime follows a certain distribution;
- Each service values its instances differently and the value of running an instance i is time-varying for a service as below:

$$v_{s,i}(st) = \begin{cases} V_s - \alpha_s(st - \underline{ST}_{s,i}), & \underline{ST}_{s,i} \le st < \overline{ST}_{s,i} \\ 0, & st < \underline{ST}_{s,i}, st \ge \overline{ST}_{s,i} \end{cases} \tag{1}$$

in which, st is the time that instance (s, i) is activated, $\underline{ST}_{s,i}$ and $\overline{ST}_{s,i}$ are the lower bound and the upper bound start times of instance (s, i), V_s is the maximum value obtainable serving instance (s, i), and α_s is the decay rate.

When $\overline{ST_{s,i}}$ passes and *(s, i)* has not been activated, the request from *s* is considered as being refused, which may increase the dissatisfaction of the service owner and cause him to reduce the average value of the requests sent to the infrastructure provider. We will evaluate this effect through simulation.

The profit of running instance (*s*, *i*) is proportional to equation (1) and can be represented as the following:

$$P(ST, ET) = \sum_{s=1}^{n} \sum_{i \in s} v_{s,i}(st_{s,i}) \cdot (et_{s,i} - st_{s,i}), \tag{2}$$

where $st_{s,i}, et_{s,i}$ are the actual start time and finish time of instance (*s*, *i*), respectively. The problem is to maximize the profit *P*(*ST*, *ET*) by producing a vector of $st_{s,i}$.

4 Client Satisfaction Oriented Scheduling for Maximizing Profit

In this section, we begin by discussing the rationale behind our client satisfaction oriented scheduling and detail the *CSoS* algorithm.

4.1 Rationale Behind *CSoS*

Clouds are primarily driven by economics—the pay-per-use pricing model similar to that for basic utilities, such as electricity, water and gas. While this pricing model is very appealing for both service providers and clients, fluctuation in service request volume and conflicting objectives between the two parties hinder its effective application. In other words, the service provider aims to accommodate/process as many requests as possible with its main objective maximizing profit; and this may conflict with client's performance requirements (e.g., response time).

Since resource availability in this study is limited to a certain degree, the service provider is likely to encounter situations in which it has to make decisions on the acceptance/refusal of service requests. This decision making is inevitable for the provider to ensure its service quality to remain at an acceptable level, i.e., profit gain is possible. In the meantime, service refusals may affect client satisfaction with the provider. Client satisfaction in our economic-driven cloud scenario can lead to decreases in service request volume; and this in turn results in profit loss or deficit (no profit at all).

We describe a typical case in our scenario as follows:

A cloud service provider offers a set of homogeneous resources for those three types of services. Although each resource may be a virtualized resource instance, the total number of resources in this study is assumed to be bounded. There are an arbitrary number of other cloud service providers for which identical or equivalent services are also offered. We assume their pricing model is comparable; that is, at least the provider pays no more than it receives from its clients.

Clients send service requests to the provider. The request rate of each client is modelled in a Poisson process with a specific mean inter-request time. With IaaS requests (processing times are specified/fixed) that a particular client sends, their processing times are modelled on the basis of the client's mean inter-request time to ensure the provider is not overwhelmed with requests. Note that the provider in our study is not very flexible with a number of available resources. Processing times of service requests follow exponential distribution. Each service request is accompanied with four parameters: (1) minimum processing time (*TMIN*), (2) allowable queuing delay (*d*), (3) maximum value (*V*), and (4) decay rate (α). Also note that SaaS and PaaS requests tend not to be bound with pre-determined values for these parameters; however, we relax this fact to a certain degree for the sake of problem complexity. That is, service requests of these two types have (rough) estimates for those parameters.

The provider allocates a resource to a given service request based on the profit obtainable. However, due primarily to the existence of services (of SaaS and PaaS) being processed, the decision is often difficult to make; and that request may be refused to be processed. This service refusal can cause profit decreases or loss/deficit both directly and indirectly. It is obvious that any profit obtainable from processing that service request is lost. By indirectly we mean that the client who requested that service may not return for some time or permanently, and thus, more and longer financial disadvantage incurs. This situation can be, however, alleviated to a certain degree by exploiting other service providers. Each of these providers in this study is characterized primarily by their service offerings and average queuing delay times. In order to avoid any loss, the original provider should make a decision on when that service is outsourced taking into account the average queuing delay of a given third-party provider and the allowable delay (specified by its client) associated with that service. When a service is further outsourced to one of these third-party providers, we assume no profit gain is possible for the original provider; however, this is still beneficial for the provider in terms of long-term profit gain.

4.2 CSoS

CSoS (Figure 1) proactively deals with fluctuation in service request volume adopting the use of third-party cloud service providers for service outsourcing. As mentioned earlier, the purpose of this outsourcing is not to maximize the profit from a particular service request, but to ensure client satisfaction to be maintained at an acceptable level. Specifically, the service is assigned onto an in-premises resource only if the assignment yields profit gain. Otherwise, the service is outsourced to a third-party cloud service provider with an assumption that no additional costs (over the value initially bound to that service request) incur.

A scheduling event in *CSoS* is initiated at the time a new service request arrives. If a resource is available for this request upon its arrival, the actual scheduling decision is made and finalized. However, if all resources are occupied with other services, *CSoS* makes an interim scheduling decision allocating the resource—among those processing SaaS and PaaS requests—for which the current service started the earliest hoping to get this current service completed before the new service's latest start time elapses. The latest start time of service s_i is defined as:

$$\overline{ST_{s,i}} = \underline{ST_{s,i}} + d_i \tag{3}$$

```
1. Let $s^*$ = a not-yet-started service for which $o_{s,i} \geq d_i$
2. if $s^* \neq \emptyset$ then
3.    Outsource $s^*$ to a third-party cloud service provider
4.    Go to Step 1
5. end if
6. Let $s^*$ = the service request with max unit value in $Q$
7. Let $c^*$ = the first resource available at the current time
8. if $c^* = \emptyset$ then
9.    Let $C'$ = a set of resources on which SaaS and PaaS requests are being processed
10.   Let $c^*$ = the resource in $C'$ for which the current service started the earliest
11. end if
12. Assign $s^*$ to $c^*$
13. Go to Step 1
```

Fig. 1. The *CSoS* algorithm

If none of resources is able to accommodate the newly arrived request, it waits until either a resource becomes available or the remaining time of its allowable delay reaches the mean queuing delay of third-party cloud service providers. The latter case triggers outsourcing. More formally, the outsourcing index of service s_i is defined as:

$$o_{s,i} = t - \underline{ST_{s,i}} \tag{4}$$

where t is the current time.

Note that there might be more than one service requests in the service queue Q; that is, more new requests may arrive while waiting for resource vacancies or outsourcing events. When there are more than one service requests waiting in the queue, *CSoS* selects the request with the maximum unit value (profit). More formally, the unit value of service s_i is defined as:

$$unit\_v_{s,i}(t) = \frac{v_{s,i}(t)}{(t - \underline{ST_{s,i}} + TMIN_i)} \tag{5}$$

5 Performance Evaluation

In this section, we first describe simulation settings and performance metrics, then present and discuss results obtained from our comparative evaluation study. Our performance evaluation study is conducted based on comparisons between two existing algorithms (*FCFS* and *FirstProfit*) and *CSoS*. *FCFS* is a simple, yet widely used/accepted scheme, which basically processes service requests in a first-come first-served fashion. *FirstProfit* prioritizes service requests by maximum value (maximum profit gain). It selects the service request for which the unit value (profit) is the largest.

5.1 Experimental Settings

The performance of *CSoS* was thoroughly evaluated based on a large set of simulations with a diverse range of settings. Parameters and their values in these settings were carefully chosen for realistic experiments. Simulations were facilitated using our discrete-event cloud simulator developed in C/C++.

The total number of experiments carried out is 126,000 (42,000 for each algorithm) using parameters as follows:

- 3 client satisfaction threshold values (90%, 95% and 99%)
- 5 discontinuation probabilities (10%, 20%, 30%, 40% and 50%)
- 4 different numbers of clients (100, 200, 400 and 800)
- 7 simulation durations (2,000, 4,000, 8,000, 12,000, 16,000, 20,000 and 30,000)

For each of these 420 base simulation settings, 100 variants were randomly generated primarily with different processing and arrival times. Note that client satisfaction threshold values and discontinuation probabilities are directly related. For example, in simulations with a client satisfaction threshold of 95 percent and a discontinuation probability of 20 percent, clients—whose service requests are refused or not processed for 5 or more percent of time—discontinue using the provider for their subsequent requests with a probability of 20 percent.

Now we describe and discuss the generation of parameters associated with service requests including maximum values, maximum acceptable mean response times, decay rates and arrival rates. The lower bound of maximum value of a service request s_i in our evaluation study is generated based on the following:

$$V_i^{lower} = TMIN_i u \tag{6}$$

where u is a unit charge set by the provider. u in our experiments was set to 1. The maximum value of an application and/or u in practice may be negotiated between the consumer and provider. While V_i^{lower} computed using Equation 6 is an accurate estimate, it should be more realistically modelled incorporating a certain degree of extra value; this is mainly because the absolute deadline of a service request ($TMIN_i$) cannot be practically met for various reasons including queuing delay. The extra value V_i^{extra} of s_i is defined as:

$$V_i^{extra} = \begin{cases} V_i^{lower} & , \quad d_i \le 0 \\ V_i^{lower} - V_i^{lower} \dfrac{d_i}{TMIN_i} & , \quad TMIN_t \ge d_i > 0 \\ 0 & , \quad d_i > TMIN_i \end{cases} \tag{7}$$

where d_i is an extra amount of delay time the client/service can afford. Now the upper bound of maximum value of s_i is defined as:

$$V_i^{upper} = V_i^{lower} + V_i^{extra} \tag{8}$$

The decay rate α_i of s_i is negatively related to allowable delay d_i and defined as:

$$\alpha_i = \frac{V_i^{upper}}{d_i} \tag{9}$$

Arrival times—for service requests from each client—are generated on a Poisson process with a mean value randomly generated from a uniform distribution.

5.2 Performance Metrics

Experimental results are analyzed using three different performance metrics, the average profit rate, the average response rate, and the average refusal / outsourcing rate. While profit is the primary performance index, response time and refusal / outsourcing rate are also important performance indicators. The profit rate of service s_i is defined as:

$$pr_i = \frac{P(ST, ET)}{V_i^{upper}} \tag{10}$$

The response rate of service s_i is defined as:

$$rr_i = \frac{t_i}{TMIN_i} \tag{11}$$

The response rate of service s_i indicates how much delay occurred based on the minimum processing time of s_i; hence, the smaller the better.

The refusal rate and outsourcing rate for a certain timeframe are defined as:

$$rfr = \frac{R}{N} \tag{12}$$

$$out = \frac{O}{N} \tag{13}$$

where R is the number of refused service requests, O is the number of outsourced service requests, and N is the total number of received service requests.

5.3 Results

Experimental results are summarized in Table 1 followed by results (Figure 2) based on each of those three performance metrics. Note that results shown in Figure 2c for *FCFS* and *FirstProfit* are based on their average refusal rates whereas those for *CSoS* are based on average outsourcing rate. *CSoS* in terms of both profit and response time demonstrated its superior performance compared with those two other algorithms. On average, *CSoS* outperformed *FCFS* and *FirstProfit* —in terms of particularly profit— by 48 percent and 26 percent, respectively. The major source of this performance gain is from the incorporation of client satisfaction with outsourcing when service

Table 1. Overall comparative results

algorithm / metric	*FCFS*	*FirstProfit*	*CSoS*
Average profit rate	52%	61%	77%
Average response rate	139%	177%	101%
Average refusal/outsourcing rate	23% / 0%	24% / 0%	0% / 29%

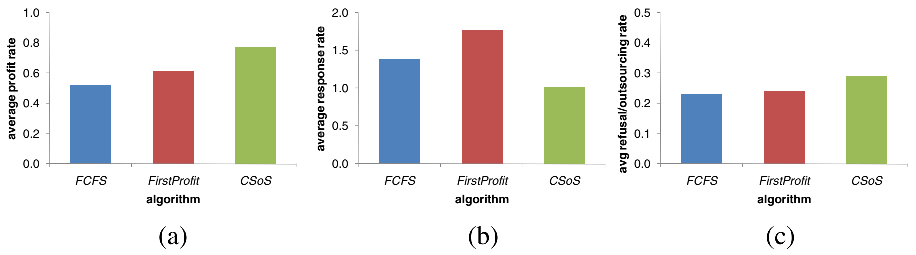


Fig. 2. Performance with respect to different metrics. (a) avg. profit rate. (b) avg. response rate. (c) avg. refusal/outsourcing rate.

requests overwhelm the provider. Although service refusals made by *FCFS* and *FirstProfit* are similar in the sense that no profit gain is possible from refused services, these refusals tend to yield a lower average incoming rate of service requests; this eventually contributes to decreases in profit.

6 Conclusion

In this paper, we have addressed the problem of scheduling service requests in cloud computing systems taking explicitly into account client satisfaction. Since the cloud service provider can hardly anticipate and/or model service request volume, there might be occasionally situations in which the provider is unable to process incoming service requests. We have investigated the effect of overloading with respect to profit and response time and devised a novel client satisfaction oriented scheduling heuristic adopting third-party service outsourcing. It has been identified that this outsourcing strategy is particularly effective in dynamic cloud environments. Our experimental results confidently confirmed this claim.

For our future work, we may consider a PaaS/IaaS request that requires multiple resources. A multi-player (multiple cloud service providers) game scenario will be investigated. Another issue that we need to look at is to identify the threshold point at which the cloud provider should purchase more servers to reduce costs associated with use of other clouds for outsourcing. For example, if costs using other clouds on average exceed costs to purchase and operate additional servers, the cloud provider should make a decision on how many additional servers he/she needs.

References

1. Salesforce.com, `http://www.salesforce.com`
2. Microsoft Azure Platform, `http://www.microsoft.com/windowsazure/`
3. Google App Engine, `http://code.google.com/appengine/`
4. Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2/`
5. Amazon Simple Storage Services, `http://aws.amazon.com/s3/`
6. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zahariam, M.: Above the clouds: A Berkeley view of Cloud computing, Technical report UCB/EECS-2009-28, Electrical Engineering and Computer Sciences, University of California at Berkeley, USA (2009)

7. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. ACM SIGCOMM Computer Comm. Review 39(1), 50–55 (2009)
8. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing Energy and Server Resources in Hosting Centers. In: Proc. 18th ACM symp. Operating systems principles (SOSP 2001), Banff, Canada, pp. 103–116 (2001)
9. Urgaonkar, B., Shenoy, P., Roscoe, T.: Resource Overbooking and application profiling in shared hosting platforms. In: Proc. 5th Symp. Operating Systems Design and Implementation (OSDI 2002), Boston, MA, USA, pp. 239–254 (2002)
10. Chun, B.N., Culler, D.E.: Market-based proportional resource sharing for clusters, Technical Report CSD-1092, University of California at Berkeley (2000)
11. Ferguson, D.F.: The application of microeconomics to the design of resource allocation and control algorithms, PhD thesis, Columbia University (1989)
12. Kurose, J.F., Simha, R.: A microeconomic approach to optimal resource allocation in distributed computer systems. IEEE Trans. on Computers 38(5), 705–717 (1989)
13. Lai, K., Rasmusson, L., Adar, E., Sorkin, S., Zhang, L., Huberman, B.A.: Tycoon: an Implementation of a Distributed, Market-based Resource Allocation System. Multiagent and Grid Systems 1(3), 169–182 (2005)
14. Yemini, Y.: Selfish optimization in computer networks. In: Proc. the 20th IEEE Conf. Decision and Control, pp. 281–285 (1981)
15. Popovici, F.I., Wilkes, J.: Profitable services in an uncertain world. In: Proc. the ACM/IEEE Int'l Conf. High Performance Networking and Computing, SC 2005 (2005)
16. Sherwani, J., Ali, N., Lotia, N., Hayat, Z., Buyya, R.: Libra: a computational economy-based job scheduling system for clusters. Softw. Pract. Exper. 34, 573–590 (2004)
17. Chun, B.N., Culler, D.E.: User-centric performance analysis of market-based cluster batch schedulers. In: Proc. IEEE/ACM Int'l Symp. Cluster Computing and the Grid, pp. 30–38 (2002)
18. Irwin, D.E., Grit, L.E., Chase, J.S.: Balancing risk and reward in a market-based task service. In: Proc. IEEE Symp. High Performance Distributed Computing, pp. 160–169 (2004)

A Tracing Approach to Process Migration for Virtual Machine Based on Multicore Platform

Liang Zhang[1], Yuebin Bai[1], and Xin Wei[2]

[1] School of Computer Science, Beihang University,
100191 Beijing, China
l_zhang@cse.buaa.edu.cn,
byb@buaa.edu.cn
[2] Beijing Institution of Information Control,
100037 Beijing, China
weixin@cse.buaa.edu.cn

Abstract. Recently, multicore processor and virtualization become popular in research and application. And an even newer tendency is to deploy virtualization on multicore processor platform. This means on a physical server, several isolated and high performance virtual environments are provided, and concurrent program has a chance to run in a multicore virtualized environment. But most virtual processor (VCPU) scheduler in VMM is not efficient in scheduling concurrent program with synchronization. And we have developed a VMM with a new VCPU scheduler to reduce the synchronization cost in some scenarios. As a component of this VMM, we need an approach to trace the processes migration in virtual machine and the mapping relationship between VCPUs and cores of physical processor to verify whether the new scheduler is effective and consistent with our initial idea. In this paper, we present such an approach and a demo Process Migration Tracing Engine for monitoring the migration of process on VCPU(s) and VCPU(s) on the cores of physical processor based on Linux 2.6 and Xen 3.2. We evaluate the impact of the engine on system performance and the results shows that this tracing approach and the tracing engine are effective and efficient.

1 Introduction

Nowadays, multicore processor is becoming more and more popular and mainstream both in home-used computers and enterprise servers. As the biggest two processor manufacturers in the world, Intel and AMD released their four-core processor lately respectively. Multicore processor creates a new type of computing environment which is more powerful, cheaper and more energy-saving than a common multiprocessor environment. More cores mean more computing resources, programs have a chance run in parallel concurrently and speed up comparing to former single-core environment. Also, virtualization is a broad term that refers to the abstraction of computer resources. Virtualization technique is a framework which creates multiple isolated and security functional partitions on a single physical machine in the manner of virtual machines monitor (VMM) or virtual environments (VEs). Virtualization technology offers a mixing platform shared by different system offering different functions such as database, web server, program debugging environment, running on a same hardware machine. Also, virtualization finds its way in system protection, fault tolerances, server integration and many other advanced enterprise level application. The popular commercial productions of virtual machine monitors include VMWare ESX Server [9] and so on, while there are also some open source VMMs such as Xen Hypervisor [5] and KVM [10].

Also, recently, more and more researches begin to cover the topic about the combining of the two, and this approach may be a good choice to exert their native power and compensate the shortcoming of each other. In this type of VMM, more computing resource is provided and every instance of virtual machine would also have a multicore environment, and in the term of virtual machine, it has more than one virtual CPU (VCPU). VMM schedules VCPUs and treats them much like thread or process. The VCPU schedule algorithm may be different in different VMMs, and in Xen hypervisor, the main schedule algorithm in current version 3.2 and above are Credit and SEDF (Simple Earlier Dead First) [6]. Credit tries to make every VCPU using CPU resource fairly while SEDF dynamically adjusts VCPU priority according to the workload. Both of them are proved effective and easy to implement in serial program environment, however, may bring a reduced performance in parallel or multithread applications.

So, at present, we have developed a new type of VMM with a new VCPU schedule algorithm, called co-scheduling, to make better utilization of the hardware resources on multicore system based on Xen hypervisor for concurrent program with synchronization in some special scenarios. As a component of this new VMM, we consequently need a new approach to trace the processes migration in virtual machine and the mapping relationship between VCPUs and cores of physical CPU from the hypervisor's point of view, in order to verify whether the new scheduler is effective and the result is consistent with our initial idea, and in this paper we introduce such a tracing approach in details and the implementation of a demo program, Process Migration Tracing Engine, to visually show the results.

The rest of this paper is organized as follow, Section 2, the method discussion in detail, Section 3, the experiment designed and implemented on Xen hypervisor 3.2.0 and Linux Kernel 2.6 to verify this tracing approach, and then we present the performance evaluation and analysis in Section 4. Section 5 introduces the related works about system status monitoring, and finally in Section 6, we conclude.

2 Approach to Trace Processes Migration for Virtual Machine

In this section, we explain our approach in detail. But at the very beginning, we must shortly introduce the relative architecture of Xen hypervisor and the role of our tracing approach in our new Xen-based experimental VMM in which the new schedule algorithm is applied.

2.1 Overview of the Tracing Approach

Xen hypervisor resides directly on the underlying hardware and performs all the privileged processor instructions [6]. Above the hypervisor, there are Xen guest domains.

Among them, Domain 0 (Dom0) is the most important domain providing device drivers and user interface and tools while other domains are *u*nprivileged, called Domain *U* (DomU). All the privileged requests from DomU(s) are transformed to and proceed by Dom0.

Related to our topic, Xen hypervisor emulates the virtual devices including virtual processors (VCPU) for domains [6]. And in the domains, they see VCPUs only but not underlying physical processors. Operating Systems (OSes) in domains schedule processes running on VCPUs, meanwhile Xen schedules VCPUs and maps them to the underlying physical processors. Our new under-developing VCPU schedule algorithm will be applied inside the layer of Xen hypervisor and responsible for VCPUs scheduling and mapping, then the approach presented in this paper is to monitor and display the schedule results and in Dom0, showing the processes migrations on VCPU(s) in DomU and the mapping relation between VCPU(s) and underlying physical processor cores in Xen hypervisor.

As a component of the VMM, the tracing approach will present both in Dom0 and DomU as a Process Migration Tracing Engine, interacting with Xen manage interfaces and tool and VCPU scheduler, the structural view of the tracing engine is displayed in the part surrounded by dash line in Fig. 1.

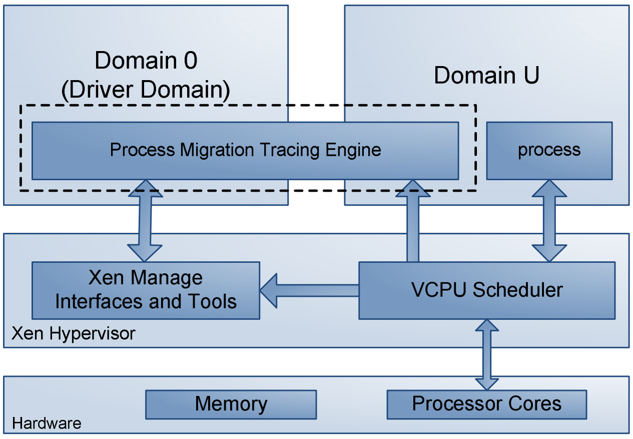


Fig. 1. Tracing Approach Engine View in Xen-based Environment

Generally speaking, the tracing approach can be broken down into three parts.

- From DomU's kernel, timely collect the ***runtime environment*** data then extract the information concerning processes and VCPU(s) schedule result;
- From DomU to Dom0, transport the information of processes and VCPU(s) schedule result;
- In Dom0, through interfaces and functions provided by Xen hypervisor, timely query the mapping relations between VCPU(s) in DomU and the core number of the underlying multicore processor.

And in the following subsections, we discuss the three parts in detail.

2.2 Collecting Information in DomU

In DomU, the OS schedules the VCPU(s) and assigns them to the processes to run on it/them. And in the OS, every process is an instance of a program which contains a set of instructions. In the way of construction, processes can be treated as an entry with some elements such as process id, process status, program counter, memory pointer and so on [7][8]. They, also can be called properties, tell a process from others during its life time and can be used for OS's control.

Minimally, a process must consist of the set of programs to execute and data associated with them. Thus, a process should have sufficient memory to hold the programs and data. Also, a stack is requested to trace the procedures calls and it is also a block of memory. Finally, as we mentioned above, there are a collection of properties for being controlled. In the way of implementation, these properties are represented by a data structure, called process control block (PCB) [7][8]. PCB is generated by OS when process is created. The information for identifying and scheduling a process is in the PCB partition and typically, they can be categorized into three groups, (1) process identification, they are static and seldom change during the process lifecycle, (2) processor state information, and (3) process control information. The information in group (2) and group (3) is changed frequently along with the processes schedule by the OS. And we can refer the whole of program, data, stack and PCB as ***process image*** [7]. In order to manage process images, OS organizes them in some manner, usually a queue-like structure. When a process is ready to run, its process image will be loaded into memory, and at other time, it, same with other process images, stays in secondly memory. Process image is very the ***runtime environment*** of a process and changes along with the process running and after being scheduled by the OS, some values, certainly including the processor state information and process control information, will be rewritten and new runtime environment information generated.

The OS manages where the processes list is located in memory, and most modern OS uses a table structure, often called control table, to maintain information including processes list. So, after getting the process table pointer in the control table entry, the OS then knows where the process queue is located.

Thus, the right way to finish our first step is to find the control table in the OS and get the entry of process table, then traverse the process table to extract the processor and process information from runtime environment of every process. In Linux, a process is presented with two data structures, one is `task_struct`, storing the static information about the process such as processor id, parent processor id, whereas the other is `thread_info`, containing the dynamic information about the process such as process status, process stack pointer address and the processor number assigned to the process after scheduling. The two data structure link to each other to be the role of PCB. Also, all the `task_struct` data structure of the processes alive is organized in a dual-cycled link list with the head list `&init_task` whose process id is `0`, which is first created when system boots.

2.3 Transporting Information from DomU to Dom0

In the point of Xen hypervisor, Dom0 and DomU is same in the concept of guest domain except that Dom0 has the ability to access the function provided by Xen hypervisor while DomU cannot. And no matter what information to transport, the information transportation process actually is a process of communication between two domains, or inter-domain communication.

For the demands of communications between domains, virtual network is a common approach applied by most VMMs. They are linked in a LAN and IP addressed are assigned to them, so that domains can exchange information as they are in a same physical network. A common socket is created, and then information is packed and delivered through the network protocol stack.

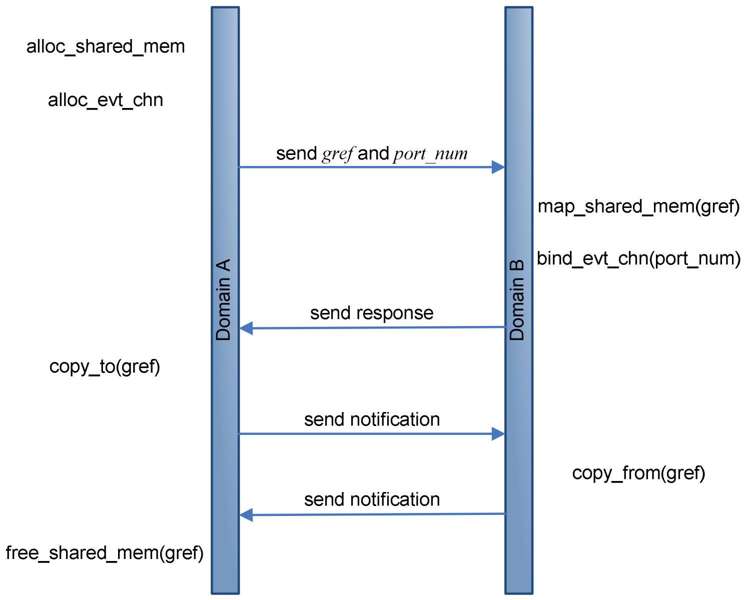


Fig. 2. Transporting Data through Shared Memory and Event Channel Mechanism

In Xen hypervisor, network devices are provided to DomUs in an abstract view and DomUs only know that they are using a network device but do worry about what specific type the device is. The physical network device is accessible only to Dom0. The communication between Dom0 and DomU is achieved by means of split-driver, DomU hosting the front-end of split-driver while Dom0 hosting the back-end of it [6].

Xen hypervisor provides two blocks of memory ring buffer for bulk data transport between Dom0 and DomU, one for sending and one for receiving. Shared memory and event channel mechanism are the fundamental of ring buffer. Memory is shared in granularity of page and is identified by an integer called *grant reference* (*gref*). Communication is bidirectional and can be initiated by any domain, DomU or Dom0. Assume that domain A is going to transport bulk data to domain B. Domain A firstly allocates a block of memory pages as shared memory. Domain A also allocates an unbound event channel and gets the *port number* (*port_num*). Then domain A passes the *gref* and *port_num* to domain B. After that, domain B connects to the unbound event channel by *port_num*, and notifies domain A that the event channel becomes bound. After that, domain A copies data to the memory pages pointed by the *gref*, and when finishing, through event channel, it notifies domain B to fetch data from the same memory region. Finally, domain B notifies domain A through event channel that copying is over, and then domain A removes *gref*. Fig. 2 describes the whole process.

Our information transport process is similar to the scenario description above, and here Dom0 is domain A and DomU is domain B in practice. Xen hypervisor, together with Linux kernel, provides some hypercalls and interfaces to manager shared memory, grant table accessing control and event channel operation [3].

2.4 Querying the Relationship between VCPU and Processor Core from Xen Hypervisor

As the most important part of common VMM, virtual processor scheduler is a critical factor to keep all guest OSes receiving the processor resource equally and running well [1]. And in VMM virtual processor(s) can be treated analogous to kernel thread(s) in a non-virtualized OS, meanwhile the program running on the virtual processor(s) in guest OS is much like the user-space program in an OS has no virtualization feature.

Processes in Xen domains are running on VCPU(s), it is the responsibility of Xen hypervisor to create, assign and schedule VCPU(s). In a multicore hardware environment, actually, scheduling is mapping the thread of VCPU(s) to underlying physical process cores.

Our new VCPU schedule algorithm is based on credit algorithm, so we must find some clues from how credit algorithm works so as to query the mapping relationship between VCPU(s) and processor core of multicore processor.

Every domain owns two properties, *weight* and *cap*, representing the share of overall physical processor time. Each physical processor (or each core in a multicore processor) manages a local queue containing the runnable VCPUs sorted by their priority. Every VCPU also has a *credit* value for determining its priority. Credit algorithm uses 30ms as a schedule cycle, choosing the VCPU, which owns a credit value more than 0, at the head of the queue to run. The scheduled VCPU then receives a time quantum of 30ms to run, and after that, it is inserted into the queue tail and waiting for another time to be scheduled [6].

It seems that no VCPU will migrates among physical processor cores because every VCPU is only dequeued and enqueued in the same run queue, but the fact is not always like this. When a physical processor core cannot find a VCPU ready to run on its local run queue, it will look for the queues on other processor cores for a ready VCPU and fetch the VCPU to run on itself. This mechanism guarantees that no processor core in a multicore processor idles when there is runnable work in the system. And also, due to this mechanism, VCPU migration will occur among processor cores within Xen hypervisor.

In the source code of Xen 3.2.0, there are some data structures used for credit algorithm for VCPU scheduling. The `struct` type variable `vcpu` is a fundamental one, within which are the relative members `vcpu_id` and `processor`, representing the id of the VCPU and the processor core number it is on during that runtime.

The migration of VCPU among physical processor cores happens within the hypervisor, and we obtain the VCPU information in Dom0. As an easy and abstract way, Xen hypervisor provides the function, together with other Xen manager functions, in programmable form interfaces in C and python language, to obtain the VCPU information in real-time. We can easy call these interfaces in program in Dom0 to talk to the hypervisor, getting what we want.

3 Software Design and Implementation

In this section, we present the details of software design and implementation of our tracing approach.

3.1 Function Module Level Design

The tracing approach is archived by the cooperation of Dom0 and DomU. As discussed in Section 2, in implementation, the idea of the tracing approach is presented in the Process Migration Tracing Engine resides both in Dom0 and DomU. Taking into account simplifying the implementation of the program and not modifying the code of Xen hypervisor and domain OSs, all the things are done with Linux kernel modules. And we define the basic function unit as a function module, in order to telling from the concept of module of Linux kernel. Fig. 3 shows the function module structure of Process Migration Tracing Engine in Dom0.

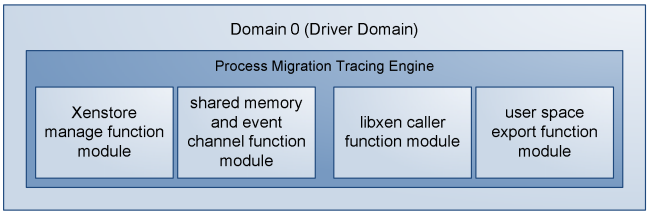

Domain 0 (Driver Domain)
Process Migration Tracing Engine
Xenstore manage function module
shared memory and event channel function module
libxen caller function module
user space export function module

Fig. 3. Function Modules of Process Migration Tracing Engine in Dom0

A kernel module resides on the Dom0's kernel, responsible to the task of allocating shared memory and event channel, passing the *gref* and *port_num* to DomU, passing and getting notifications with DomU, copying data from shared memory, and then removing the *gref* at the end. Fig. 4 shows the function module structure of Process Migration Tracing Engine in DomU.

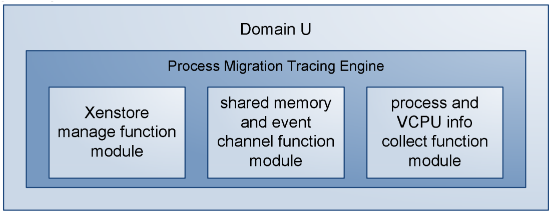

Domain U
Process Migration Tracing Engine
Xenstore manage function module
shared memory and event channel function module
process and VCPU info collect function module

Fig. 4. Function Modules of Process Migration Tracing Engine in DomU

A kernel module resides on the DomU's kernel, responsible to the task of timely querying the processes and VCPU(s) schedule results, binding event channel according to *port_num*, passing and getting notifications with Dom0 and then copying data to shared memory.

Some configuration information such as *gref* and *port_num* should be transport from Dom0 to DomU, we use Xenstore, which is a database of configuration information shared between domains, to finish that. Xenstore is designed only for small piece of data transportation rather than large bulk of data transportation, and for the transportation of processes and VCPU(s) schedule results, we use shared memory and event channel mechanism discussed in Section 2.3.

3.2 Dom0's Kernel Module Overview

In the Dom0 kernel module, we use interfaces

`__get_free_pages()`and `free_pages()`, `kmalloc()` and `kfree()`to allocate and free memory pages. We use hypercalls `gnttab_grant_foreign_access()`, `gnttab_end_foreign_access()`and `HYPERVISOR_grant_table_op()` to control the access of shared memory to get the value of *gref*. We use hypercall series `HYPERVISOR_event_channel_op()` to operate the event channel and get the *port_num* and exchange notification with DomU. We use `xenbus_directory()`, `xenbus_read()`and `xenbus_write()` to operate Xenstore database to transport *gref*, *port_num* and other configure data for negotiation with DomU.
Considering that the information transport from DomU to Dom0 is not so much, we only allocate one page, whose size is 4KB, used for shared memory.

3.3 DomU's Kernel Module Overview

In the DomU kernel module, we use interfaces

`alloc_vm_area()` and `free_vm_area()`, `kmalloc()` and `kfree()`, and hypercall `HYPERVISOR_grant_table_op()`to map shared memory region. We use hypercall series `HYPERVISOR_event_channel_op()` to bind event channel and exchange notification with Dom0.

Information must be collect timely at a frequency, so we create a kernel thread and make all the operation run on that kernel thread when the kernel module is inserted into the kernel of DomU, using the interface `kthread_run()`.

We use the same Xenstore interfaces as Dom0 to get *gref* of shared memory and *port_num* of unbound event channel for configuration negotiation with Dom0.

Because the information is collected timely, it should be considered that how to choose the interval. It cannot be too short, or a system performance reduce might be brought, meanwhile, it cannot be too long, either, or the information cannot retrieve the extract process schedule result. We choose this interval as long as the time quantum, 30ms, in credit algorithm for VCPU schedule. And in the experiments, we also change this interval and evaluate the impact on system performance.

4 Experiments and Performance Evaluation

4.1 Experiment Environment

All the experiments are performed on a Dell Optiplex 760MT Server with Intel® Core™ 2 Quad Q9400 2.66GHz processor and 2GB RAM. The processor has four cores and supports Intel® VT™ Technology. We install Novell openSUSE 11.0 on this server machine as Dom0, and the kernel source version is 2.6.25.5-1.1, then continue to set up Xen hypervisor 3.2.1_16881 and install openSUSE 11.0 as DomU without any unnecessary software installed and system services turned on. DomU has two VCPUs and 512MB RAM, and only text mode is available. In some test, we change the VCPU amount to one or the RAM amount to 256MB respectively.

4.2 Functional Verification

The basic test is to verify whether the function in the initial design can be achieved. We write a simple C program which only has a `while(true)` infinity loop in DomU and make it running in background.

After running all the kernel modules, we observe from Dom0, and we can get the list in the form of `[pid, vcpu_id, cpu_id]`, in which the three parameter stand for process id, VCPU id in DomU, and core number of multicore processor to which this VCPU is mapped at that time. After several observations, we can see that the C program randomly migrates between two VCPUs and the VCPUs are also randomly mapped among the four processor cores by Xen hypervisor.

And then we pin the C program to a VCPU and after several observations, we can only see that the VCPUs are also randomly mapped among the four processor cores whereas the C program remain running on the same VCPU.

4.3 Benchmark and Test Tools Summary

In the experiments, we mainly use these benchmarks.

Table 1. Benchmark Summary

Benchmark	Evaluation Point
lmbench	Transportation throughout from DomU to Dom0 Memory operation latency in DomU when collecting information
MBW	Memory bandwidth in DomU when collecting information

Lmbench [15] is a benchmark suite consisting of simple and portable benchmark for comparing and measuring different kinds of UNIX-liked system performance, and it is used in our experiment for evaluating transportation throughout from DomU to Dom0. MBW [16] is a small memory bandwidth benchmark through memory operations, and it is used in our experiment for evaluating the influence of information collecting on the memory operation in DomU.

And also, we use `xentop` command provided by Xen hypervisor to monitoring the VCPU utilization rate during information collection in DomU and information transportation from DomU to Dom0.

4.4 Results and Analysis

We run all the tests at least five times with the same configuration and report the average.

Firstly, we measure the VCPU cost with `xentop` command with the configuration that Dom0 has four VCPUs and DomU has two VCPUs, and then change the VCPU amount of DomU to one and perform the test again. All the tests are run without information data transportation from DomU to Dom0.

Fig. 5 shows the VCPU cost of Dom0 and DomU which has two VCPUs. With different information collecting interval, the Dom0's VCPU cost keep almost steady at a little more than 3.00%, whereas in DomU, the shorter the information collecting interval is, the more VCPU cost is, and it ranges from less than 12.00% to more than 4.00%.

Then we change the VCPU amount in DomU to one with other configuration unchanged and test the performance again. Same as the last test, Dom0's VCPU cost is holding at about 3.00% or a little more, and DomU's VCPU cost experiences an overall small drops, ranging from no more than 9.00% to less than 3.00%, and Fig. 6 shows the results.

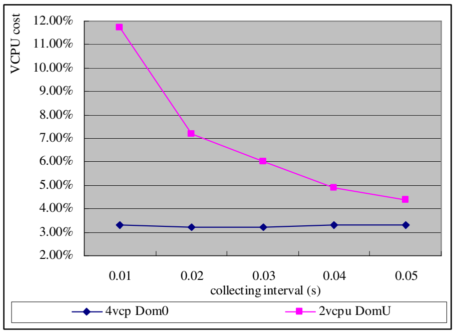


Fig. 5. VCPU cost (two VCPUs in DomU)

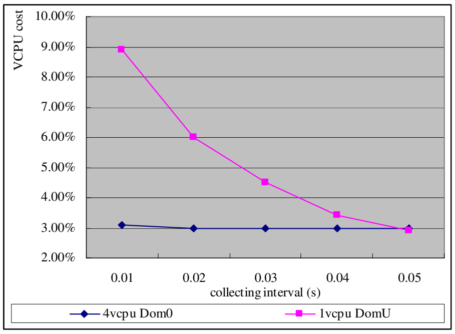


Fig. 6. VCPU cost (one VCPU in DomU)

And what is interesting is that more VCPUs in DomU lead a higher global VCPU cost. This experiment result helps us to determine how long the information collecting interval should be. It is obvious that the VCPU cost is quit high when the interval is 0.01s (10ms) and if the collecting work is running all the time along with DomU, it is a considerable system overhead which consumes ten percent of the global VCPU resource, although more accurate schedule information can be collected. And according to the principle of credit algorithm and our under-developing VCPU schedule algorithm, a VCPU is most likely to be scheduled at the end of a 30 ms-quantum, even if credit algorithm checks credit of VCPU every 10 ms. We also give up the longer interval such as 0.04s or 0.05s, as they may collect the processes and VCPU(s) schedule result with too much delay. So we choose 0.03s (30ms) as our default information collecting time interval based on the analysis above, and the VCPU cost whose value is around 5.00% is acceptable.

Table 2. Results of MBW benchmark

Data Size (MB)	Method	Collecting Only		Collecting and Transporting	
		Time (ms)	Date Rate (MB/s)	Time (ms)	Date Rate (MB/s)
4	memcpy	3.21	1245.83	3.25	1229.97
	dumb	3.08	1297.40	2.82	1417.54
	mcblock	0.51	7815.55	0.51	7817.08
8	memcpy	5.92	1351.79	5.94	1347.78
	dumb	6.04	1323.61	6.12	1306.87
	mcblock	1.00	7995.20	1.01	7936.51
10	memcpy	7.05	1418.18	7.25	1379.23
	dumb	7.39	1352.89	7.40	1351.90
	mcblock	1.25	8009.61	1.24	8055.42
20	memcpy	13.88	1440.95	13.76	1453.09
	dumb	13.96	1432.64	14.08	1420.07
	mcblock	2.48	8063.54	2.47	8107.67

Secondly, we measure memory performance with lmbench and MBW with only collecting information in DomU ongoing and then, measure it with both collecting information in DomU and transporting information from DomU to Dom0 ongoing.

From this table, we can see that the memory performance change little in the two conditions, with very tiny difference in operation time in microsecond level. This indicates that the shared memory mechanism is effective and the frequent memory reads and writes almost do not bring a memory performance decline. And the shared memory region is only in size of one page, its impact on the whole memory can be ignored.

Finally, we measure the throughput of information data transportation from DomU to Dom0 with lmbench to check the transportation efficiency of shared memory and event channel mechanism. The experiment is carried on with different data size every time DomU/Dom0 writes to/read from the 4KB shared memory block (We define this size as “package size” in the below analysis).

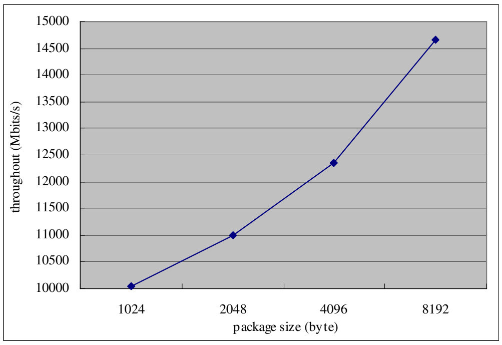


Fig. 7. Throughput of data transportation from DomU to Dom0

Fig. 7 shows the throughput of DomU-Dom0 data transportation, and with the package size increases, the throughout increases along with it. Based on this experiment result, we choose 8192Byte as our default package size in DomU's and Dom0's kernel modules.

5 Related Works

Some other works are done to monitor the system status, and in Linux system, /proc directory provides some useful entries to retrieve the runtime statistic information about OS. "Torsmo" developed by Hannu and Lauri [11] can show various information about the system and it's peripherals including number of processes running or sleeping, and its off-spring "conky" [12] achieves almost the same function. "sysstat" [13] obtains CPU statistics (global, per CPU and per Linux task / PID), including support for virtualization architectures. "htop" developed by Hisham Muhammad and his colleagues [14] is an interactive process viewer which is like the command top in Linux. However, these utilities do not focus on how to retrieve and trace the relationship between the processes and the corresponding mapping core of physical processor, and much less that the relationship crossing the VMM mapping from the virtual processor to physical processor. And our works not only pay close attention to trace that which process belongs to which processor core along with the system runs, but also expand this relationship from virtual processor to physical processor.

6 Conclusions

In order to verify the VCPU schedule algorithm in our under-developing VMM, in this paper, we present an approach for tracing processes migration on virtual processor(s) in guest OS and the mapping relation between virtual processors and cores of physical processor. And we also implement a demo program as a Process Migration Tracing Engine in Linux 2.6 and Xen hypervisor 3.2. The tracing engine shows that our approach can fill the gap between the guest OS and the underlying hardware by crossing through the intermediate lay of virtual machine monitor. The tracing engine shows it is an effective to trace process migration in virtual machine and the dynamic mapping relationship between virtual processor and cores of processor. And it is also effective to verify and display the VCPU schedule results of Xen and our newly developing Xen-based VMM.

We perform evaluations to measure the impact of the trace engine on system performance. And the experiment results show that the Process Migration Tracing Engine brings a very slim impact on the whole system performance and it is proved that the Process Migration Tracing Engine applying the process migration tracing approach is efficient.

References

1. Nesbit, K.J., Smith, J.E., Moreto, M., Cazorla, F.J., Ramirez, A., Valero, M.: Multicore Resource Management. In: IEEE Micro, pp. 6–16 (May-June 2008)
2. Payne, B.D., de A. Carbone, M.D.P., Lee, W.: Secure and Flexible Monitoring of Virtual Machines. In: Annual Computer Security Applications Conference (2007)

3. Shin, H.-S., Kim, K.-H., Kim, C.-Y., Jung, S.-I.: The new approach for inter-communication between guest domains on Virtual Machine Monitor. Computer and Information Sciences (2007)
4. Sweeney, P.F., Hauswirth, M., Diwan, A.: Understanding Performance of MultiCore Systems using Tracebased Visualization. In: STMCS 2006 (2006)
5. Xen Hypervisor Source Version 3.2.1 (2009), `http://www.xen.org`
6. Chisnall, D.: The Definitive Guide to the Xen Hypervisor. Prentice-Hall, Englewood Cliffs (2007)
7. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 8th edn. John Wiley & Sons, Chichester (2008)
8. Stallings, W.: Operating Systems: Internals and Design Principles, 5th edn. Prentice Hall, Englewood Cliffs (2005)
9. VMWare ESX Server, `http://www.vmware.com/products/vi/esx`
10. Linux KVM, `http://www.linux-kvm.org`
11. Torsmo, `http://torsmo.sourceforge.net/`
12. Conky, `http://conky.sourceforge.net/`
13. sysstat, `http://pagesperso-orange.fr/sebastien.godard/`
14. htop, `http://htop.sourceforge.net/`
15. Lmbench, `http://www.bitmover.com/lmbench/`
16. MBW, `http://ahorvath.home.cern.ch/ahorvath/mbw/`

Accelerating Dock6's Amber Scoring with Graphic Processing Unit

Hailong Yang, Bo Li, Yongjian Wang, Zhongzhi Luan, Depei Qian, and Tianshu Chu

Department of Computer Science and Engineering,
Sino-German Joint Software Institute, Beihang University,
100191 Beijing, China
{hailong.yang,yongjian.wang,tianshu.chu}@jsi.buaa.edu.cn,
{libo,zhongzhil,depeiq}@buaa.edu.cn

Abstract. In the drug discovery field, solving the problem of virtual screening is a long term-goal. The scoring functionality which evaluates the fitness of the docking result is one of the major challenges in virtual screening. In general, scoring functionality in docking requires large amount of floating-point calculations and usually takes several weeks or even months to be finished. This time-consuming disadvantage is unacceptable especially when highly fatal and infectious virus arises such as SARS and H1N1. This paper presents how to leverage the computational power of GPU to accelerate Dock6 [1]'s Amber [2] scoring with NVIDIA CUDA [3] platform. We also discuss many factors that will greatly influence the performance after porting the Amber scoring to GPU, including thread management, data transfer and divergence hidden. Our GPU implementation shows a 6.5x speedup with respect to the original version running on AMD dual-core CPU for the same problem size.

1 Introduction

Identifying the interactions between molecules is critical both to understanding the structure of the proteins and to discovering new drugs. Small molecules or segments of proteins whose structures are already known and stored in database are called ligands. While macromolecules or proteins associated with a disease are called receptors [4]. The final goal is to find out whether the given ligand and receptor can form a favorable complex and how appropriate the complex is, which may inhibit a disease's function and thus act as a drug.

Scoring is the step after docking which is involved evaluating the fitness for docked molecules and ranking them. A set of sphere-atom pairs will be on behalf of an orientation in receptor and evaluated with a scoring function on three dimensional grids. At each grid point, interaction values are to be summed to form a final score. These processes need to be repeated for all possible translations and rotations. Tremendous computational power is required, as scoring for each orientation needs large amount of CPU cycles, especially dealing with floating-point. The advantage of amber scoring is that both ligands and active sites of the receptor can be flexible during the evaluation. While the disadvantage is also obvious, it brings tremendous intensive floating-point computations. When performing amber scoring, it calculates the internal energy of the ligand, receptor and the complex, which can be broken down into three steps:

- Minimization
- molecule dynamics (MD) simulation
- more minimization using solvents

The computational complexity of amber scoring is very huge, especially in the MD simulation stage. Three grids which individually have three dimension coordinates are used to represent the molecule during the orientation such as geometry, gradient and velocity. In each grid, at least 128 elements are required to sustain the accuracy of the final score. During the simulation, scores are calculated in three nested loops, each of which walks through one of the three grids.

While many virtual screening tools such as GasDock [5], FTDock [6] and Dock6 can utilize multi-CPUs to parallel the computations, the incapacity of CPU in processing floating-point computations still remains untouched. Compared with CPU, GPU has the advantages of computational power and memory bandwidth. For example, a GeForce 9800 GT can reach 345 GFLOPS at peak rate and has an 86.4 GB/sec memory bandwidth, whereas an Intel Core 2 Extreme quad core processor at 2.66 GHz has a theoretical 21.3 peak GFLOPS and 10.7 GB/sec memory bandwidth. Another important factor why GPU is becoming widely used is that it is more cost-effective than CPU.

Our contributions in this paper include porting the original Dock6 amber scoring to GPU using CUDA, which can archive a 6.5x speedup. We analyze the different memory access patterns in GPU which can lead to a significant divergence in performance. Discussions on how to hide the computation divergence on GPU are made. We also conduct experiments to see the performance improvement.

The rest of the paper is organized as follows. In section II, an overview of Dock6's amber scoring and analysis of the bottleneck is given. In section III, we present the main idea and implementation of the amber scoring on GPU with CUDA, and details of considerations about performance are made. Then we give the results, including performance comparisons among various GPU versions. Finally, we conclude with discussion and future work.

2 Analysis of the Amber Scoring in Dock6

2.1 Overview

A primary design philosophy of amber scoring is allowing both the atoms in the ligand and the spheres in the receptor to be flexible during the virtual screening process, generating small structural rearrangements, which is much like the actual situation and gives more accuracy. As a result, a large number of docked orientations need to be analyzed and ranked in order to determine the best fit set of the matched atom-sphere pairs.

In the subsection, we will describe the amber scoring program flow and profile the performance bottleneck of the original amber scoring, which can be perfectly accelerated on GPU.

2.2 Program Flow and Performance Analysis

Figure 1 shows the steps to score the fitness for possible ligand-receptor pairs in amber. The program firstly performs conjugate gradient minimization, MD simulation, and more minimization with solvation on the individual ligand, the individual receptor, and the ligand-receptor complex, then calculates the score as follows:

$$\text{Ebinding} = \text{Ecomplex} - (\text{Ereceptor} - \text{Eligand}) . \tag{1}$$

The docked molecules are described using three dimension intensive grids containing the geometry, gradient or velocity coordinates information. The order of magnitude of these grids is usually very large. Data in these grids is represented using floating-point, which has little or no interactions during the computation. In order to archive higher accuracy, the scoring operation will be performed repeatedly, perhaps hundreds or thousands times.

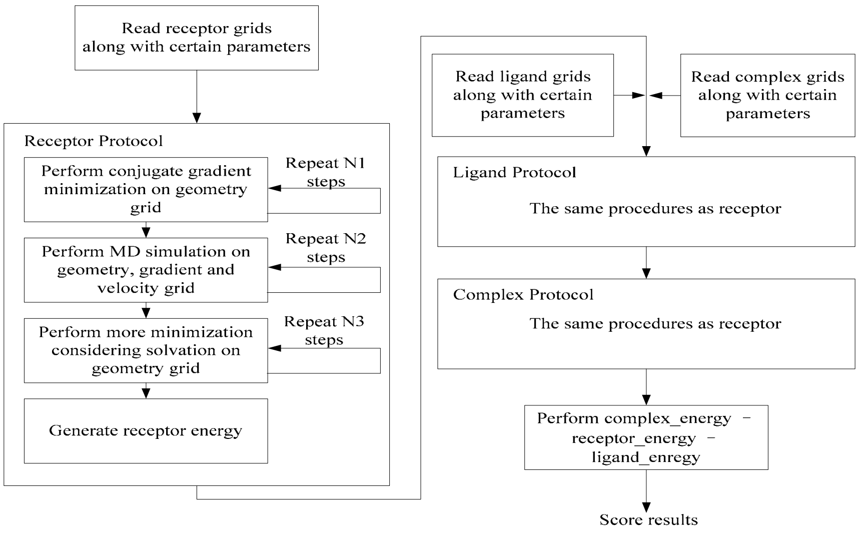

Read receptor grids along with certain parameters
Receptor Protocol
Perform conjugate gradient minimization on geometry grid
Repeat N1 steps
Perform MD simulation on geometry, gradient and velocity grid
Repeat N2 steps
Perform more minimization considering solvation on geometry grid
Repeat N3 steps
Generate receptor energy
Read ligand grids along with certain parameters
Read complex grids along with certain parameters
Ligand Protocol
The same procedures as receptor
Complex Protocol
The same procedures as receptor
Perform complex_energy − receptor_energy − ligand_enregy
Score results

Fig. 1. Program flow of amber scoring

Due to the characteristics of the amber scoring such as data independency and high arithmetic intensity, which are exactly the sweet spots of computing on GPU, it can be perfectly paralleled to leverage the computing power of GPU and gain preferable speedup.

3 Porting Amber Scoring to GPU

3.1 Overview

To determine the critical path of amber scoring, we conduct an experiment to make statistics about the cost of each step as Table 1 shows. We see that time spent on processing ligand is negligible, because ligand in docking always refer to small molecules or segments of protein whose information grids are small and can be calculated quickly. We also observe, however, that MD simulation on receptor and complex are the most time-consuming parts, which take up to 96.25 percentage of the total time. Therefore, in our GPU accelerated version, we focus on how to port the MD simulation to GPU, which could accelerate the bulk of the work.

Table 1. Run time statistics for each step of Amber scoring. 100 cycles are performed for minimization steps and 3,000 cycles for MD simulation step.

Stage		Run time (seconds)	Ratio of total (%)
Receptor protocol	gradient minimization	1.62	0.33
	MD simulation	226.41	45.49
	minimization solvation	0.83	0.17
	energy calculation	2.22	0.45
Ligand protocol	gradient minimization	≈0	0
	MD simulation	0.31	0.06
	minimization solvation	≈0	0
	energy calculation	≈0	0
Complex protocol	gradient minimization	8.69	1.75
	MD simulation	252.65	50.76
	minimization solvation	2.69	0.54
	energy calculation	2.22	0.45
Total		497.64	100

For the simplicity and efficiency, we take advantage of the Compute Unified Device Architecture (CUDA). We find the key issue to fully utilize GPU is high ratio of arithmetic operations to memory operations, which can be achieved through refined utilization of memory model, data transfer pattern, parallel thread management and branch hidden.

3.2 CUDA Programming Model Highlights

At the core of CUDA programming model are three key abstractions – a hierarchy of thread groups, shared memories and barrier synchronization, which provide finegained data parallelism, thread parallelism and task parallelism. CUDA defines GPU as coprocessors to CPU that can run a large number of light-weight threads concurrently. Threads are manipulated by kernels representing independent tasks mapped over each sub-problem. The kernel is invoked from the host side, most cases the CPU, as an asynchronized thread. The parallel threads collaborate through shared memory and synchronize using barriers.

In order to process on the GPU, data should be prepared by copying it to the graphic board memory first. Data transfer can be performed using deeply pipelined streams that overlap the kernel processing.The problem domain is defined in the form of a 2D grid of 3D thread blocks. The significance of thread block primitive is that it is the smallest granularity of work unit to be assigned to a single streaming multiprocessor(SM). Each SM is composed of 8 scalar processors (SP) that indeed run threads on the hardware in a time-slice manner. Every 32 threads within a thread block are grouped into warps. Within a warp the executions are in order, while beyond the warp the executions are out of order. However, if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously.

In addition to global memory, each thread block has its own private shared memory that is only accessible to threads within that thread block. Threads within a thread block could cooperate by sharing data among shared memory with low latency, while threads that belong to different thread blocks could only share data through global memory, which is slower by three orders of magnitude than shared memory. Synchronization within a thread block is implemented in hardware. Among thread blocks, synchronization can be achieved by finishing a kernel and starting a new one.

3.3 Parallel Thread Management

To carry out the MD simulation on GPU, a kernel needs to be written which is launched from the host (CPU) and executed on the device (GPU). A kernel is the same instruction set that will be performed by multiple threads simultaneously. By default, all the threads are distributed onto the same SM, which can't fully explore the computational power of the GPU. In order to utilize the SMs more efficiently, thread management must be taken into account.

We divide the threads into multiple blocks and each block can hold the same number of threads. In the geometry, gradient and velocity grids, 3D coordinates of atoms are stored sequentially and the size of the grid usually reach as large as 7,000. Calculation works are assigned to blocks on different SMs; each thread within the blocks computes the energy of one atom respectively and is independent of the rest (see Figure 2). We compose N threads into a block, which calculate N independent atoms in the grids. Assuming the grid size is M and M is divisible by N, there will happen to be M/N blocks.

While in most cases the grid size M is not divisible by N, we designed two schemes dealing with this situation. In the first scheme, there will be M/N blocks. Since there is M%N atoms left without threads to calculate, we will rearrange the atoms evenly to the threads in the last block. One more atom will be added to the threads in the last block until no atoms left, which is ordered by thread ID ascending. The second scheme is to construct M/N + 1 blocks. Each thread in the blocks still calculates one atom however the last block may contain threads with nothing to do. Control logic should be added to the kernel to judge whether the thread has some calculations or not through comparing the value of block ID * N + thread ID and M.

Our experiment proves the former scheme obtained better performance. This is caused by underutilized SM resources and branch cost in the second scheme. When there is a branch divergence, all the threads must wait until they reach the same instructions again. Synchronization instructions are generated by the CUDA complier automatically, which is time consuming.

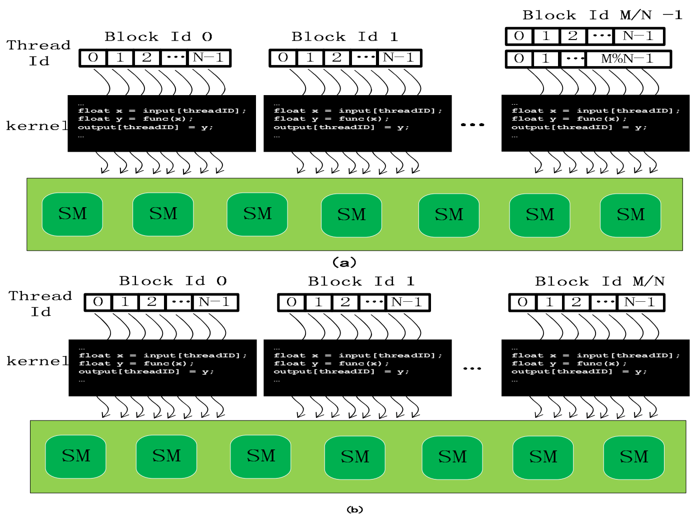


Fig. 2. Threads and blocks management about processing molecule grids on GPU: (a) blocks whose threads in the last block may calculate two atoms each (b) blocks whose threads in the last block may have nothing to do.

3.4 Memory Model and Data Transfer Pattern

The first step to perform GPU computations is to transfer data from host (CPU) memory to device (GPU) memory since the receptor, ligand and complex grids need to be accessible by all SMs during the calculations. There are two kinds of memory can be used to hold these grids. One is the constant memory, which can be read and written by the host but can only be read by the device. The other is the global memory, which can be read and written by both the host and device. One important distinction between the two memories is the access latency. SMs can get access to the constant memory magnitude order faster than the global memory. While the disadvantage of the constant memory is also obvious, it is much smaller, which is usually 64 KB compared to 512 MB global memory. Thus, a trade-off has to be made on how to store these grids.

During each MD simulation cycle, the gradient and velocity grids are read and updated. Therefore, they should be stored in global memory. While once entered the MD simulation process, the geometry grids are never changed by the kernel. Hence, they can be stored in constant memory (see Figure 3). Considering the out-bound danger which dues to the limited capacity of the constant memory, we observed the size of the each geometry grid. The receptor and complex geometry grids usually contain no more than 2,000 atoms each while the ligand geometry grid contains 700 atoms, which totally requires 2,000 * 3 * 4 * 2 + 700 * 3 * 4 bytes (56.4 KB) memory to store them. Since it is smaller than 64 KB, the geometry grids shall never go out-bound of the constant memory.

The time to transfer molecule grids from host to their corresponding GPU memory is likewise critical issue, which may degrade the benefit achieved from the parallel execution if without careful consideration [7]. For each MD simulation cycle, we

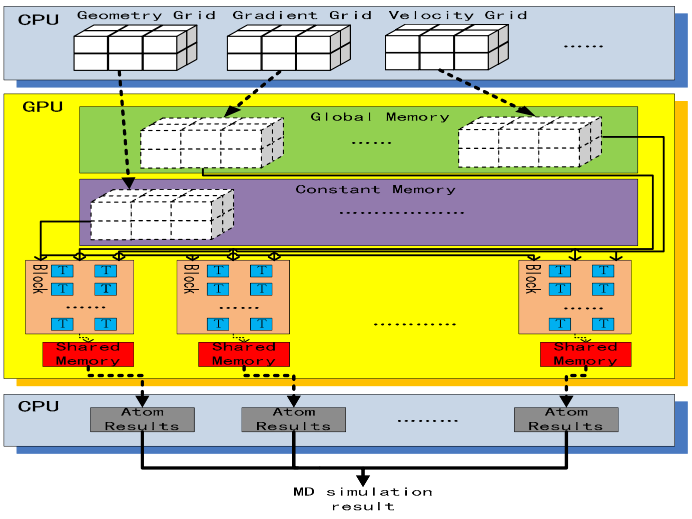


Fig. 3. Memory model and data transfer pattern during the MD simulation cycles. Grids are transferred only once before the simulation, which are stored in global memory and constant memory correspondingly. Atom results are first accumulated in the shared memory within the block. Then the accumulations per block are transferred into the host memory and summed up to the final MD simulation result for the molecule.

could transfer one single atom 3D coordinates in the geometry, gradient and velocity grids to device memory when they are required by the SMs. The other solution is to transfer the entire grids into the GPU memory before the MD simulation stage. When the simulation starts, these grids are already stored in device memory which can be accessed by simulation cycles performed on SMs.

Based on our experiment, significant performance improvements are obtained from the second scheme since the molecule grids are transferred only once for all before the MD simulation. When the calculations on the SMs are carried out, the coordinates of the atoms in the grids are already stored in the device memory. Therefore, the SMs don't have to halt and wait for the grids to be prepared, which obviously speeds up the parallel execution of the MD simulations by fully utilizing the SMs.

The MD simulations are executed parallel on different SMs, and threads within the different blocks are responsible for the calculations of their assigned atoms of the grids. The final simulation result is formed by accumulating all results. Our solution is to synchronize threads within the blocks, which generates atom results separately. Then a transformation is performed to store the atom results from shared memory to host memory in a result array, whose index is identical to the block ID. The molecule result shall be achieved by adding up all the elements in the array without synchronization since the results are transferred only when the calculations on device are accomplished.

3.5 Divergence Hidden

Another important factor that dramatically impact the benefits achieved by performing MD simulation on GPU is the branches. Original MD simulation procedure has involved a bunch of nested control logics such as bound of Van der Waals force and constrains of molecule energy. When the parallel threads computing on different atoms in the grids come to a divergence, a barrier will be generated that all the threads will wait until they reach the same instruction set again. The above situation can be time-consuming and outweigh the benefits of parallel execution.

We extract the calculations out of the control logic. Each branch result of the atom calculation is stored in a register variable. Inside of the nested control logic, only value assignments are performed, which means the divergence among all the threads will be much smaller, thus the same instruction sets can be reached with no extra calculation latency. Although this scheme will waste some computational power of the SMs since only a few branch results are useful in the end, it brings tremendous improvements in performance. These improvements can be attributed to that, in most cases, the computational power we required during the MD simulation is much less than the maximum capacity of the SMs. Hence, the extra calculations only consume vacant resources, which in turn speed up the executions. The feasibility and efficiency of our scheme have been demonstrated in our experiment.

4 Results

The performance of our acceleration result is evaluated for two configurations:

- Two cores of a dual core CPU
- GPU accelerated

The base system is a 2.7 GHz dual core AMD Athlon processor. GPU results were generated using an NVIDIA GeForce 9800 GT GPU card.

Table 2. CPU times, GPU times and speedups with respect to 3,000 MD simulation cycles per molecule protocol. The CPU version was performed using dual core, while GPU version with all superior scheme.

Stage		CPU	GPU	Speedup
Receptor protocol	gradient minimization	1.62	0.89	1.82
	MD simulation	226.41	31.32	7.23
	minimization solvation	0.83	0.15	5.53
	energy calculation	2.22	1.21	1.83
Ligand protocol	gradient minimization	≈0	0.02	——
	MD simulation	0.31	0.60	——
	minimization solvation	≈0	0.03	——
	energy calculation	≈0	≈0	0
Complex protocol	gradient minimization	8.69	2.88	3.02
	MD simulation	252.65	34.79	7.26
	minimization solvation	2.69	2.05	1.31
	energy calculation	2.22	1.47	1.51
Total		497.64	75.41	6.5

We referred to the Dockv6.2 as the original code, which was somewhat optimized in amber scoring. We also used the CUDAv2.1, whose specifications support 512 threads per block, 64KB constant memory, 16KB shared memory and 512MB global memory. Since double precision floating point was not supported in our GPU card, transformation to single precision floating point was performed before the kernel launched. With small precision losses, the amber scoring results were slightly different between CPU version and GPU version, which can be acceptable.

Table 2 compares the original CPU version with the GPU accelerated version in runtime for various stages. The MD simulation performed are 3,000 cycles each molecular stage. The overall speedup achieved for the entire amber scoring is over 6.5x.

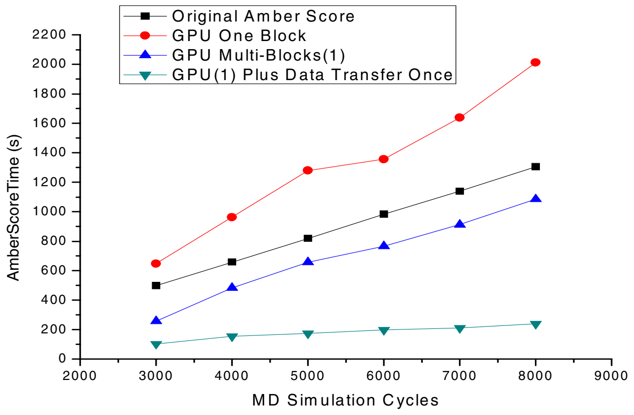


Fig. 4. Shown is a comparison of amber scoring time between original amber and different GPU versions whose speedup varies significantly as the MD simulation cycles increasing from 3,000 to 8,000

Figure 4 depicts the total speedups of different GPU schemes with respect to the range of increasing MD simulation cycles. As mentioned in section 3.3, the GPU version with only one block did not speedup, which was attributed to the poor management of threads since each block had a boundary of maximum active threads. In our experiment, GeForce 9800 GT specification limits each block can only hold 512 threads maximally. This limitation will force large amount of threads waiting on the only one block until other threads are served and release the SM resource. Since the MD simulation requires more than one block threads to calculate the atom results, the latency becomes more obviously as the MD simulation cycles scale. Fortunately, with multiple blocks, this kind of thread starvation latency can be greatly eased. Threads within multiple blocks can be scheduled onto different SMs so that calculations of independent atoms are executed parallel. The most significant performance improvements are achieved from transferring the molecule grids only once during the MD simulation in addition to the usage of multi-blocks.

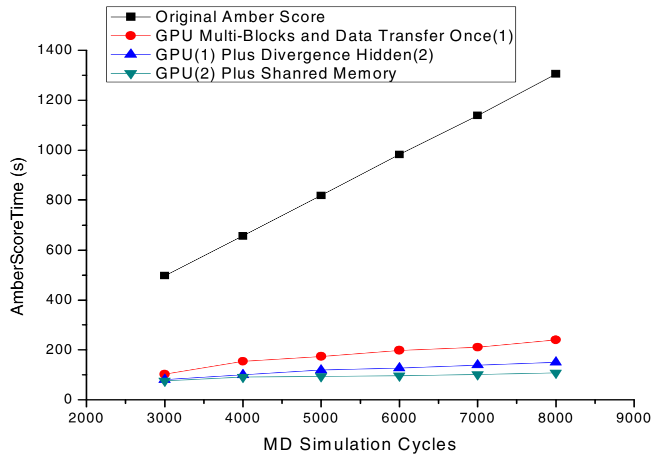


Fig. 5. Comparison of speedups among different GPU versions based on Figure 4 in addition to divergence hidden and shared memory

Figure 5 depicts the second speedup in performance comes from the utilizations of divergence hidden and synchronization on shared memory. Since the branch calculations are extracted out of the control logic and stored in temporary variables, only one single instruction will be performed which assigns corresponding values into the final result when divergences occur. This scheme greatly shortens the time consumed for all the threads to return to the same instruction sets. While threads within a block will accumulate atom simulation values into a partial result of molecule on shared memory, the result array transferred back to the host is very small. Performance improvements are obtained when summing up the elements in the array to form the molecule simulation result. We also notice that as the MD simulation cycles scales, the speedup becomes more considerable in our best GPU version.

5 Related Work

Exploiting GPUs for general purpose computing has recently gained popularity particularly as a mean to increase the performance of scientific applications. However most of the accelerations of science-oriented applications on GPU are in the fields of graphic processing and arithmetic algorithms. Kruger et al. [8] implemented linear algebra operators for GPUs and demonstrated the feasibility of offloading a number of matrix and vector operations to GPUs. Nathan Bell [9] demonstrated several efficient implementations of sparse matrix-vector multiplication (SpMV) in CUDA by tailoring the data access patterns of the kernels.

Studies on utilizing GPU to accelerate molecule docking and scoring problems are rare, the only work we find more related to our concern is in the paper of Bharat Sukhwani [10]. The author described a GPU-accelerated production docking code, PIPER [11], which achieves an end-to-end speedup of at least 17.7x with respect to a single core. Our contribution is different from the former study in two aspects. First, we focus our energy on flexible docking such as amber scoring while the previous study mainly work on rigid docking using FFT. Thus our work is more complex and competitive in the real world. Second, we noticed the logic branches in the parallel threads on GPU degraded the entire performance sharply. We also described the divergence hidden scheme and represented the comparison on speedup with and without our scheme.

Another attractive work needs to be mentioned is that Michael Showerman and Jeremy Enos[12] developed and deployed a heterogeneous multi-accelerator cluster at NCSA. They also migrated some existing legacy codes to this system and measured the performance speedup, such as the famous molecular dynamics code called NAMD[13, 14]. However, the overall speedup they achieved was limited to 5.5x since they could not utilize the computation power of GPU and FPGA simultaneously.

6 Conclusion and Future Works

In this paper we present a GPU accelerated amber score in Dock6.2, which achieves an end-to-end speedup of at least 6.5x with respect to 3,000 cycles during MD simulation compared to a dual core CPU. We find that thread management utilizing multiple blocks and single transferring of the molecule grids dominate the performance improvements on GPU. Furthermore, dealing with the latency attributed to thread synchronization, divergence hidden and shared memory can be elegant solutions which will additionally double the speedup of the MD simulation. Unfortunately the speedup of Amber scoring can't go much higher due to Amdahl's law. The limits are in multiple ways:

- With the kernel running faster because of GPU accelerating, the rest of the Amber scoring takes a higher percentage of the run-time
- Partitioning the work among SMs will eventually decrease the individual job size to a point where the overhead of initializing an SP dominates the application execution time

The work we presented in this paper only shows a kick-off stage of our exploration in GPGPU computation. We will proceed to use CUDA accelerating various applications with different data structures and memory access patterns and hope to be able to work out general strategies about how to use the manycore feature of GPU more efficiently.

Acknowledgment

Many thanks to Ting Chen for thoughtful discussions and comments about our implementation and paper work. This work was supported by the National Natural Science Foundation of China under the grant No. 90812001, Chinese National Programs for High Technology Research and Development under the grant No. 2006AA01A 124 and 2006AA01A118.

References

1. Dock6, `http://dock.compbio.ucsf.edu/DOCK_6/`
2. Wang, J., Wolf, R.M., Caldwell, J.W., Kollman, P.A., Case, D.A.: Development and testing of a general Amber force field. Journal of Computational Chemistry, 1157–1174 (2004)
3. NVIDIA Corporation Technical Staff.: Compute Unified Device Architecture - Programming Guide, NVIDIA Corporation (2008)
4. Kuntz, I., Blaney, J., Oatley, S., Langridge, R., Ferrin, T.: A geometric approach to macromolecule-ligand interactions. Journal of Molecular Biology 161, 269–288 (1982)
5. Lia, H., Lia, C., Guib, C., Luob, X., Jiangb, H.: GAsDock: a new approach for rapid flexible docking based on an improved multi-population genetic algorithm. Bioorganic & Medicinal Chemistry Letters 14(18), 4671–4676 (2004)
6. Servat, H., Gonzalez, C., Aguilar, X., Cabrera, D., Jimenez, D.: Drug Design on the Cell BroadBand Engine. In: Parallel Architecture and Compilation Techniques, September 2007, p. 425 (2007)
7. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: High-performance graphics coprocessor sorting for large database management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (2006)
8. Kruger, J., Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In: ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques (2003)
9. Nathan, B., Michael, G.: Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004 (Dec. 2008)
10. Bharat, S., Martin, C.H.: GPU acceleration of a production molecular docking code. In: Proceedings of 2nd Workshop on General Purpose Processing on GPUs, pp. 19–27 (2009)
11. PIPER, `http://structure.bu.edu/index.html`
12. Michael, S., Hwu, W.-M., Jeremy, E., Avneesh, P., Volodymyr, K., Craig, S., Robert, P.: QP: A Heterogeneous Multi-Accelerator Cluster. In: 10th LCI International Conference on High-Performance Clustered Computing (March 2009)
13. NAMD, `http://www.ks.uiuc.edu/Research/namd/`
14. Phillips, J.C., Zheng, G., Sameer, K., Kalé, L.V.: NAMD: Biomolecular Simulation on Thousands of Processors. In: Conference on High Performance Networking and Computing, pp. 1–18 (2002)

Optimizing Sweep3D for Graphic Processor Unit

Chunye Gong, Jie Liu, Zhenghu Gong, Jin Qin, and Jing Xie

Department of Computer Sciences, National University of Defense Technology, 410073 Changsha, China
leaf.gong@gmail.com, liujie@nudt.edu.cn

Abstract. As a powerful and flexible processor, the Graphic Processing Unit (GPU) can offer great faculty in solving many high-performance computing applications. Sweep3D, which simulates a single group time-independent discrete ordinates (S_n) neutron transport deterministically on 3D Cartesian geometry space, represents the key part of a real ASCI application. The wavefront process for parallel computation in Sweep3D limits the concurrent threads on the GPU. In this paper, we present multi-dimensional optimization methods for Sweep3D, which can be efficiently implemented on the fine grained parallel architecture of the GPU. Our results show that the performance of overall Sweep3D on CPU-GPU hybrid platform can be improved up to 2.25 times as compared to the CPU-based implementation.

Keywords: Sweep3D, neutron transport, graphics processor unit (GPU), Compute Unified Device Architecture (CUDA).

1 Introduction

When the first GPU was introduced in 1999, the GPU mainly has been used to transform, light and rasterize triangles in 3D graphics applications [1]. The performance of GPU doubles about every six to nine months, which outperforms CPU a lot [2]. The modern GPUs are throughput-oriented parallel processors that can offer peak performance up to 2.72 Tflops single-precision floating-point and 544 Gflops double-precision floating-point [3]. At the same time, the GPU programming models, such as NVIDIA's Compute Unified Device Architecture (CUDA) [4], AMD/ATI's Streaming Computing [5] and OpenCL [6], become more mature than before and simplify the processing of developing non-graphics applications. The enhancement of computing performance, development of programming model and software make GPU more and more suitable for general-purpose computing. At present, GPU has successfully applied to medical imaging, universe exploration, physics simulation, linear system solution and other computation intensive domains [7].

There is a growing need to accurately simulate physical systems whose evolutions depend on the transport of subatomic particles coupled with other complex physics [8]. In many simulations, particle transport calculations consume the majority of the computational resources. For example, the time devoted to particle transport problem in multi-physics simulations costs 50-80% of total execution time of many realistic simulations on DOE systems [9, 10]. So parallelizing deterministic particle transport calculations is recognized as an important problem in many applications targeted by the DOE's Accelerated Strategic Computing Initiative (ASCI). The benchmark code Sweep3D [11] represents the heart of a real ASCI application that run on the most powerful supercomputers such as Blue Gene [12] and Roadrunner [13]. So it is worthwhile to accelerate Sweep3D on high performance, low power consuming GPU.

In this paper we describe our experiences of developing Sweep3D implementation on the CUDA platform, analyze the bottleneck of our GPU execution. Our GPU version is based on the SIMD and uses the massive thread level parallelism of GPU. Efficiently using registers and thread level parallelism can improve performance. We use the repeated computing and shared memory to schedule 64 times more threads, which improves performance with $64n-cubed$ problem size. Our GPU version gets 2.25 times speedup as compared to the original single CPU core version.

2 An Overview of Sweep3D

Sweep3D [11] solves a three-dimensional neutron transport problem from a scattering source. The basis of neutron transport simulation is the time-independent, multigroup, inhomogeneous Boltzmann transport equation. The numerical solution to the transport equation involves the discrete ordinates (S_n) method and the procedure of source iteration. In the S_n method, where N represents the number of angular ordinates used, the angular-direction is discretized into a set of quadrature points. In the Cartesian geometries (XYZ), each octant of angles has a different sweep direction through the mesh, and all angles in a given octant sweep the same way. The sweep of S_n method generically is named wavefront [14]. The solution involves two steps: the streaming operator is solved by sweeps and the scattering operator is solved iteratively.

A S_n sweep for a given angle proceeds as follows. Every grid cell has 4 equations with 7 unknowns (6 faces plus 1 central) and boundary conditions complete the system of equations. The solution is by a direct ordered solve known as a sweep. Three known inflows allow the cell center and 3 outflows to be solved. Each cell's solution then provides inflows to 3 adjoining cells (I, J, and K directions). This represents a wavefront evaluation with recursion dependence in all 3 grid directions. Sweep3D exploits parallelism via a wavefront process. First, a 2D spatial domain decomposition onto a 2D array of processors in the I- and J-directions is used. Second, the sweeps of the next octant pair start before the previous wavefront is completed; the octant order required for reflective boundary conditions limits this overlap to two octant pairs at a time. The overall combination is sufficient to give good theoretical parallel utilization. The resulting diagonal wavefront is depicted in Fig. 1 just as the wavefront gets started in the 4th octant and heads toward the 6th octant [14, 15].

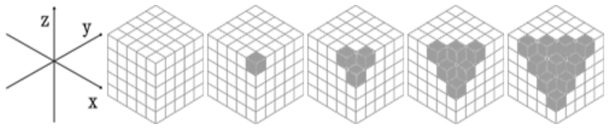


Fig. 1. The wavefront parallelism in Sweep3D

On single core, there is no communication in Sweep3D. The Sweep() subroutine, computational core of Sweep3D, takes about 97.70% of the whole runtime. The structure of the Sweep() subroutine is listed in Fig. 2. The jkm loop (line 7 - line 22) in the Sweep() subroutine takes 99.05 % of the subroutine runtime.

```
1  DO iq=1,8           ! octants
2   DO mo=1,mmo        ! angle pipelining loop
3    DO kk=1,kb        ! k-plane pipelining loop
4    RECV E/W          ! recv block I-inflows
5    RECV N/S          ! recv block J-inflows
6     DO idiag=1,jt+nk-1+mmi-1   !JK-diagonals
7      DO jkm=1,ndiag ! I-lines on this diagonal
8      DO i=1,it       ! source (from Pn moments)
9      ENDDO
10     IF .NOT.do_fixups
11      DO i=i0,i1,i2 ! Sn eqn
12      ENDDO
13     ELSE
14      DO i=i0,i1,i2 ! Sn eqn fixups
15      ENDDO
16     ENDIF
17     DO i=1,it       ! flux (Pn moments)
18     ENDDO
19     DO i=1,it       ! DSA face currents
20     ENDDO
21    ENDDO
22   SEND E/W          ! send block I-outflows
23   SEND N/S          ! send block J-outflows
24   ENDDO
25  ENDDO
26 ENDDO
```

Fig. 2. The structure of Sweep() subroutine

A complete wavefront from a corner to its opposite corner is an iteration. The I-lines that is the jkm loop in the iteration can be solved in parallel on each diagonal. As showed in Fig. 1, the number of concurrent threads is 1, 3, 6 and 10. The relationship between maximum concurrent threads ($MCT(n)$) and problem size $n - cubed$ ($n \in N$) shows in (1):

$$\lim_{n\to\infty} MCT(n) = 6n \tag{1}$$

Similarly, the relationship between average concurrent threads ($ACT(n)$) and problem size $n - cubed$ shows in (2):

$$\lim_{n\to\infty} ACT(n) = 3n \tag{2}$$

Many research works on particle transportation is focus on performance model, scalability and running on large scale parallel architecture [10, 16, 17]. The most similar and impressive work is that Petrini etc. implemented Sweep3D on CBE (Cell Broadband Engine) [18]. They exploited 5 dimensions of parallelism to get good performance. All the technologies make full use of CBE's processing element, data traveling and the hierarchical memory and get 4.5-20 times speedup compared with different kind of processors.

3 Architecture of Nvidia GT200 and CUDA

3.1 Architecture of Nvidia GT200

The architecture of GPU is optimized for rendering real-time graphics, a computation and memory access intensive problem domain with enormous inherent parallelism. Not like CPU, a much larger portion of a GPU's resources is devoted to data processing than to caching or control flow.

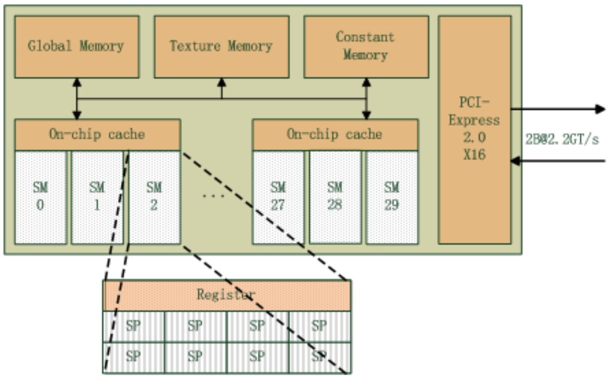


Fig. 3. GT200 structure

NVIDIA GT200 chip (Fig. 3) contains 240 Streaming-Processor (SP) cores running at 1.44 GHz. Physically, eight SPs form one Streaming Multiprocessors (SMs or Multiprocessors) and each SP run in Single Instruction Multiple Data (SIMD) manner. There are ten independent processing units called Thread Processing Clusters (TPC) and each TPC contains a geometry controller, a SM controller, three SMs and a texture unit. The Multiprocessors creates, manages, and executes concurrent threads in hardware with near zero scheduling overhead and can implement barrier synchronization. The Single Instruction Multiple Thread (SIMT) unit, which is akin to SIMD vector organizations, creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps [4, 19]. The particular and useful specifications for the GT200 of Tesla S1070 are listed in Table 1. The ideal peak performance depends on how many

Table 1. Technical specifications of a Tesla S1070 GPU

CUDA compute capability	1.3
Total amount of global memory	4G
Number of multiprocessors(SM)	30
Number of cores(SP)	240
Total amount of constant memory	64 KB
Total amount of shared memory per block	16 KB
Total number of registers available per block	16 KB
Warp size	32

operations can be performed per cycle. One stream processor technically support one MAD (multiply-add) and one MUL (multiply) per cycle, which would correspond to 1.04 Tflops per GPU. There is only one double-precision unit in a SM and the peak double-precision floating-point performance is about 78 Gflops per GPU. There are 4 GPUs in Tesla S1070. So the peak single- and double-precision floating-point performances are 3.73 to 4.14 Tflops and 311 to 345 Gflops [19].

The application performance on GPU is directly associated with Maximum Thread Blocks (MTB) per Multiprocessor. Compute capability 1.3 most run eight thread blocks (MTB_{warp}) because of the restriction of warp. As listed in Table 1, the shared memory and registers also limits maximum thread blocks. For example, the maximum thread blocks limited by registers (MTB_{reg}) show in (3):

$$MTB_{reg} = \frac{Totalnumberofregistersavailableperblock}{(Threadperblock) * (Registersperthread)} \tag{3}$$

3.2 Programming Model and Software

The programming model is a bridge between hardware and application. As a scalable parallel programming model, CUDA [4] does not abstract the whole computing system in an ideal level.The hybrid system is separated into host and device. CUDA uses kernel function, which is a SPMD (Single Program Multiple Data) computation with a potentially large number of parallel threads, to run efficiently on hardware. The concept of thread block in thread hierarchy makes CUDA programming model is independent of the number of a GPU's SMs.

CUDA exposes the Multiprocessors on the GPU for general purpose computation through a minimal set of extensions to C programming language. Compute intensive components of a program can run as kernel function. Kernel functions are executed many times with different input data. The software managed on-chip cache or shared memory in each SIMD core and barrier synchronization mechanism let local data sharing and synchronization in a thread block become a reality.

4 Multi-dimensional Optimization Methods

Although there is no data send and receive on single GPU through MPI, we still keep the process level parallelism as Fig. 2 shows. This will guarantee portability of existing software. As mentioned in Section 2, the jkm loop takes the most runtime. So the strategy of implementation on this hybrid system is GPU-centric. That is to say, we have the compute intensive part run on GPU and CPU do little computation. The optimizational architecture of hybrid computing of this application shows in Fig. 4.

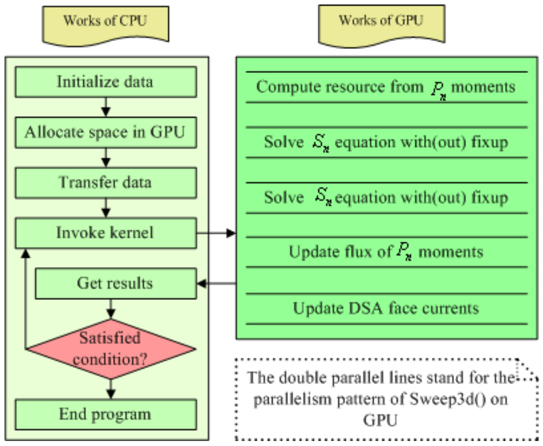

Works of CPU
Works of GPU
Initialize data
Allocate space in GPU
Transfer data
Invoke kernel
Get results
Satisfied condition?
End program
Compute resource from P_n moments
Solve S_n equation with(out) fixup
Solve S_n equation with(out) fixup
Update flux of P_n moments
Update DSA face currents
The double parallel lines stand for the parallelism pattern of Sweep3d() on GPU

Fig. 4. Architecture of hybrid CPU-GPU computing

4.1 Stage 1: Thread Level Parallelization

Fig. 2 indicates that there are two levels of parallelism of the subroutine Sweep(). One is the I-lines on JK-diagonals with MMI pipelining (the jkm loop, line 7 - line 20) that can be processed in parallel, without any data dependency. The other one is the inner loop in the jkm loop, including reading source from P_n moments (line 8, 9), updating flux of P_n moments (line 17, 18) and DSA face currents (line 19, 20). There are two reasons for why we don't exploite the parallelism of the inner loop. First, the inner loop is limited by the X dimension size of the space geometry (the value of 'it'). Another shortcoming of parallelizing the inner loop is that CPU calls kernel functions much more times and keeps too many temp local data. So exploiting the parallelism of the jkm loop becomes the suitable choice.

In the jkm loop, there are two main branches: not do fixups and do fixups (line 10) and the variable of data-dependent loops increases or decreases (line 11, 14). The branches don't affect the performance of kernel's execution, but make the whole kernel too big to debug. So we divide the jkm loop into four different kernels (jkmKernelNotFixUp, jkmKernelNotFixDown, jkmKernelFixUp,

Table 2. Runtime on CPU and GPU (seconds)

Size (-cubed)	64	128	192	256
CPU	2.43	28.72	107.85	254.33
Stage1	4.44	41.51	79.79	426.20

jkmKernelFixDown) and let CPU deal with branches. Each time CPU invokes "ndiag" threads which are divided into 1 to $\lceil \frac{ndiag*1.0}{threadBlockSize} \rceil$ thread blocks.

The runtime of $64n - cubed$ problem size on both CPU and GPU are listed in Table 2. When the problem size is $128 - cubed$, the runtime of the Sweep3D on GPU is 41.51 seconds which is slower than that of CPU. The following parts in Section 4 illustrate how to accelerate Sweep3D on Tesla S1070.

4.2 Stage 2: More Threads and Repeated Computing

Although we get 1.35 times speedup on problem size $192 - cubed$, the performance of $128 - cubed$ and $256 - cubed$ is relatively poor. Taking $192 - cubed$ as an example, the average number of concurrent threads is about 576. The jkmKernelNotFixUp and jkmKernelNotFixDown use 41 registers per thread while jkmKernelFixUp and jkmKernelFixDown use 52 registers per thread. According to (3), we have $\lfloor \frac{16KB}{64*41} \rfloor = 6$ and $\lfloor \frac{16KB}{64*52} \rfloor = 4$. Six or four blocks can concurrently run on every multiprocessor. But there are only 576 threads and $\lceil \frac{576}{64} \rceil = 9$ total blocks which can not make full use of 30 multiprocessors with $256 - cubed$ problem size, let alone smaller problem size.

There is no data-dependence in the loops of reading source from P_n moments, updating flux and DSA face currents. We use 64 times more threads to do the

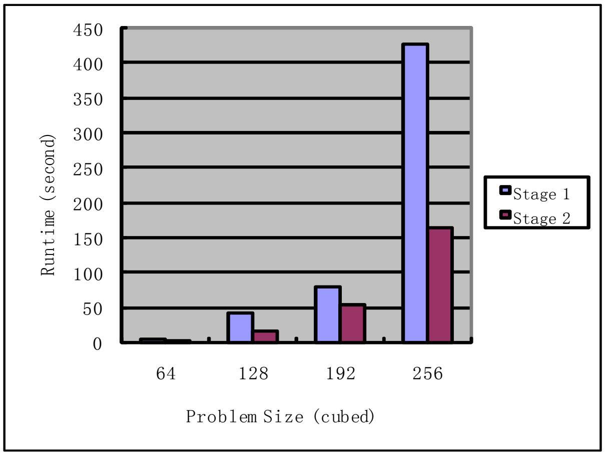


Fig. 5. Architecture of hybrid CPU-GPU computing

same work that one thread do in stage 1. As mentioned in Section 2, the average number of concurrent threads in $n-cubed$ is about $3n$ in stage 1. Here, invocating a kernel has 3n thread blocks with 64 threads in each block.

It's efficient to use the global memory bandwidth when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of 128 bytes. But the data-dependent S_n loop must be executed by one thread, so there are two barrier synchronizations before and after the S_n loop. The performance improvement is depicted in Fig. 5.

4.3 Stage 3: Using Shared Memory

The shared memory is on-chip and software managed cache that can reduce the memory access time of the reusing data. However, most computations in the program lack data reuse. It is important to exploit data reuse. We found that the results of computing source from P_n moments are reused in the S_n loop and medial results in S_n loop are reused in updating flux of P_n moments and DSA face currents. We utilize the shared memory to store the corresponding reusable vectors instead of accessing the global memory. A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing local or global memory, there are 400 to 600 clock cycles of memory latency in addition [4].

A multiprocessor takes 6 cycles per warp instead of the usual 4 to access the shared memory [20]. There are three reasons why using the shared memory can not run up 100 times faster than using global memory. First, because of the double data type, there exists two ways of bank conflicts in shared memory [4]. Second, the global memory access is coalesced and thread blocks scheduling can hide latency. Third, the real memory access operations are more complicated than theory analysis. The performance improvement is depicted in Fig. 6.

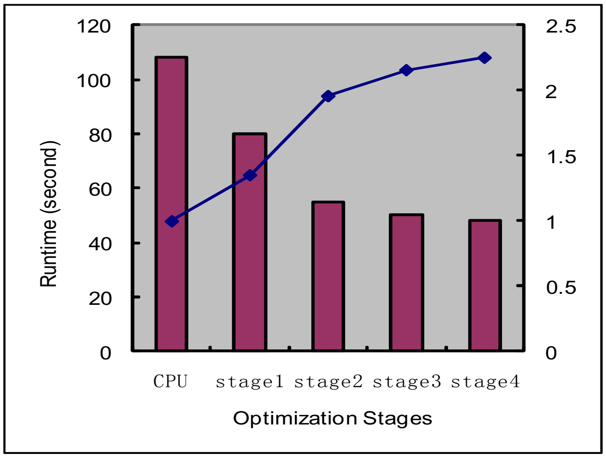


Fig. 6. Performance improvement using Shared memory with $192-cubed$ problem size

4.4 Stage 4: Other Methods

Other Methods include more work on GPU, communication overlapping computation, autotunner in the running time and using of texture memory. More work on GPU can avoid data movement between CPU and GPU. Communication overlapping computation can hide some time spending on communication. Autotunner is valuable when ndiag is very small. The texture memory has a high-speed cache that can reduce the memory access time of the neighboring data. All the methods above make the runtime of $256 - cubed$ reduce by 5.69 seconds.

5 Performance Results and Analysis

In this section, we compare the runtime of our GPU and CPU implementations from a wide range of problem sizes and present speedup and precision error.

The platform consists of Intel(R) Core(TM)2 Quad CPU Q6600 2.40 GHz processors, 5 GB of main memory, Red Hat Enterprise Linux Server release 5.2 operating system. The Tesla S1070 high performance computing platform consists of 4 GT200 GPU, clock rate 1.44 GHz, with 4 GB frame buffer memory each, making a total GPU memory of 16GB.

For the purpose of comparison, we measure the speedup provided by one GT200 GPU comparing to a serial code running on the CPU (consequently only one core). Both codes run on double-precision floating-point arithmetic, and are compiled using the GNU gfortran compiler version 4.1.2, and the nvcc compiler

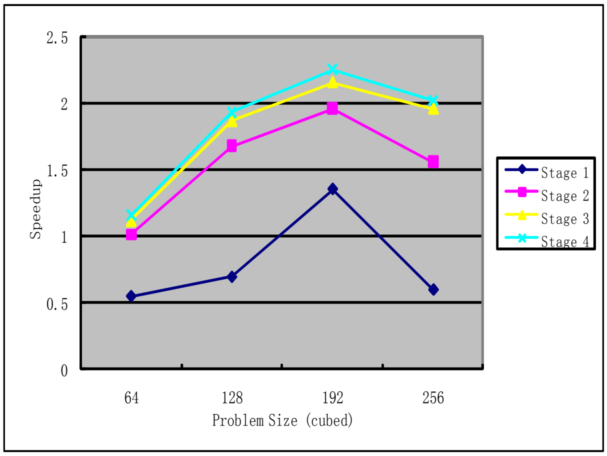


Fig. 7. Speedup of whole Sweep3D application on GPU. Stage 1 ports kernel part of Sweep3D to GPU and uses thread level parallelism. Stage 2 does some repeated computing and uses 64 times more threads to access global memory. Stage 3 uses shared memory to store local array. Stage 4 mainly puts additional work on GPU.

provided by CUDA version 2.1. In both cases, level two optimizations have been performed. Sweep3D runs 4 iterations, half with flux fixups and half without.

The performance improvement of each optimization stage is illustrated in Fig. 7.

Some double precision mathematical operations on GT200 are IEEE-754 round-to-nearest- even, or even worse. This kind of system error can not be avoided. The maximum error of 4 iterations at problem size $128 - cubed$ is smaller than 10-13, which is acceptable.

Petrini got that the CELL BE is approximately 4.5 and 5.5 times faster than the 1.9 GHz Power5 and 2.6 GHz AMD Opteron on $50 - cubed$ problem size. The optimizing runtime is 1.33 seconds on CELL BE without knowing how many iterations. Consequently the runtime of Power5 and Opteron are 5.985 seconds and 7.315 seconds. The runtime of Intel Q6600 2.40 GHz is 3.10 seconds with 12 iterations which is the default on $50 - cubed$ input size. So the CELL BE is about 2.33 times faster than Intel Q6600 processors. If the proportion can scale to $192 - cubed$ input size then the Sweep3D on GPU is almost as fast as on CELL BE. Of course the performance is only one advantage of GPU, the programming productivity and wildly available hardware are other two reasons for adoring GPU than CELL.

6 Conclusion and Future Work

We have presented an optimized GPU based implementation of Sweep3D and up to 2.25 times speedup is achieved compared to CPU implementation. Our implementation efficiently uses the features of hybrid system and explores its multi dimension optimizations. To the best of our knowledge, our work has revealed a new implementation of neutron transport simulation problem and wavefront process of parallelism on GPU. Other similar and complex wavefront algorithms or applications are also likely to benefit from our experience on CPU-GPU hybrid system.

As a part of the future work, first, Other advanced optimizations will be tried on Sweep3D(). Second, More experiments will be performed on other cooperating processing units such as SSE, FPGA and Clearspeed. Third, the scalability issues on heterogeneous GPU clusters will be carefully studied.

Acknowledgments. This research work is supported by the National Natural Science Foundation of China under grant No.60673150 and No.60970033, also by the National High Technology Research and Development Program of China under grant No.2008AA01Z137. We would like to thank the anonymous reviewers for their helpful comments.

References

1. Nguyen, H.: GPU Gems 3. Addison Wesley, Reading (2007)
2. Kirk, D.: Innovation in graphics technology. In: Talk in Canadian Undergraduate Technology Conference (2004)

3. AMD Corporation: ATI Radeon HD 5870 Feature Summary, http://www.amd.com/
4. NVIDIA Corporation: CUDA Programming Guide Version 2.1 (2008)
5. AMD Corporation: ATI Stream Computing User Guide Version 1.4.0a (2009)
6. Munshi, A.: The OpenCL Specification Version: 1.0. Khronos OpenCL Working Group (2009)
7. NVIDIA Corporation: Vertical solutions on CUDA, http://www.nvidia.com/object/vertical_solutions.html
8. Mathis, M.M., Amato, N., Adams, M., Zhao, W.: A General Performance Model for Parallel Sweeps on Orthogonal Grids for Particle Transport Calculations. In: Proc. ACM Int. Conf. Supercomputing, pp. 255–263. ACM, New York (2000)
9. Hoisie, A., Lubeck, O., Wasserman, H.: Scalability analysis of multidimensional wavefront algorithms on large-scale SMP clusters. In: The 7th Symposium on the Frontiers of Massively Parallel Computation, pp. 4–15. IEEE Computer Society, Los Alamitos (1999)
10. Hoisie, A., Lubeck, O., Wasserman, H.: Performance and scalability analysis of teraflop- scale parallel architectures using multidimensional wavefront applications. International Journal of High Performance Computing Applications 14(4), 330–346 (2000)
11. Los Alamos National Laboratory: Sweep3D, http://wwwc3.lanl.gov/pal/software/sweep3d/
12. Davis, K., Hoisie, A., Johnson, G., Kerbyson, D.J., Lang, M., Pakin, M., Petrini, F.: A Performance and Scalability Analysis of the BlueGene/L Architecture. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, pp. 41–50 (2004)
13. Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C.: Entering the petaflop era: the architecture and performance of Roadrunner. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (2008)
14. Lewis, E.E., Miller, W.F.: Computational Methods of Neutron Transport. American Nuclear Society, LaGrange Park (1993)
15. Koch, K., Baker, R., Alcouffe, R.: Solution of the First-Order Form of Three-Dimensional Discrete Ordinates Equations on a Massively Parallel Machine. Transactions of American Nuclear Society 65, 198–199 (1992)
16. Mathis, M.M., Kerbyson, D.J.: A General Performance Model of structured and Unstructured Mesh Particle Transport Computations. Journal of Supercomputing 34, 181–199 (2005)
17. Kerbyson, D.J., Hoisie, A.: Analysis of Wavefront Algorithms on Large-scale Two-level Heterogeneous Processing Systems. In: Workshop on Unique Chips and Systems, pp. 259–279 (2006)
18. Petrini, F., Fossum, G., Fernandez, J., Varbanescu, A.L., Kistler, N., Perrone, M.: Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In: The 21th International Parallel and Distributed Processing Symposium (2007)
19. NVIDIA Corporation: NVIDIA Tesla S1070 1U Computing System, http://www.nvidia.com/object/product_tesla_s1070_us.html
20. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (2008)

Modular Resultant Algorithm for Graphics Processors

Pavel Emeliyanenko

Max-Planck-Institut für Informatik, Saarbrücken, Germany
asm@mpi-inf.mpg.de

Abstract. In this paper we report on the recent progress in computing bivariate polynomial resultants on Graphics Processing Units (GPU). Given two polynomials in $\mathbb{Z}[x, y]$, our algorithm first maps the polynomials to a prime field. Then, each modular image is processed individually. The GPU evaluates the polynomials at a number of points and computes univariate modular resultants in parallel. The remaining "combine" stage of the algorithm is executed sequentially on the host machine. Porting this stage to the graphics hardware is an object of ongoing research. Our algorithm is based on an efficient modular arithmetic from [1]. With the theory of displacement structure we have been able to parallelize the resultant algorithm up to a very fine scale suitable for realization on the GPU. Our benchmarks show a substantial speed-up over a host-based resultant algorithm [2] from CGAL (www.cgal.org).

Keywords: polynomial resultants, modular algorithm, parallel computations, graphics hardware, GPU, CUDA.

1 Overview

Polynomial resultants play an important role in the quantifier elimination theory. They have a comprehend applied foreground including but not limited to topological study of algebraic curves, curve implitization, geometric modelling, etc. The original modular resultant algorithm was introduced by Collins [3]. It exploits the "divide-conquer-combine" strategy: two polynomials are reduced modulo sufficiently many primes and mapped to homeomorphic images be evaluating them at certain points. Then, a set of univariate resultants is computed independently for each prime, and the result is reconstructed by means of polynomial interpolation and the Chinese Remainder Algorithm (CRA). A number of parallel algorithms have been developed following this idea: those specialized for workstation networks [4] and shared memory machines [5, 6]. In the essence, they differ in how the "combine" stage of the algorithm (polynomial interpolation) is realized. Unfortunately, these algorithms employ polynomial remainder sequences [7] (PRS) to compute univariate resultants. The PRS algorithm, though asymptotically quite fast, is sequential in nature. As a result, the Collins' algorithm in its original form admits only a *coarse-grained* parallelization which is suitable for traditional parallel platforms but not for systems with the massively-threaded architecture like GPUs (Graphics Processing Units).

That is why, we have decided to use an alternative approach based on the theory of *displacement structure* [8] to compute univariate resultants. This method reduces the problem to matrix computations which commonly map very well to the GPU's threading model. The displacement structure approach is traditionally applied in a floating-point arithmetic, however, using square-root and division-free modifications [9], we have been able to adapt it to work in a prime field. As of now, the research is carried out to port the remaining algorithm stages (polynomial interpolation and the CRA) to the GPU. Modular computations still constitute a big challenge on the GPU, see [10,11]. Our algorithm uses the fast modular arithmetic developed in [1] which is based on mixing floating-point and integer computations, and is supported by the modified CUDA [12] compiler[1]. This allowed us to benefit from the multiply-add capabilities of the graphics hardware and minimize the number of instructions per modular operation, see Section 4.3.

The rest of the paper is structured as follows. In Section 2 we state the problem in mathematically rigorous way and give an overview of the displacement structure which constitutes the theoretical background for our algorithm. Section 3 surveys the GPU architecture and CUDA programming model. In Section 4 we present the overall algorithm and discuss how it maps to the graphics hardware. Finally, Section 5 provides an experimental comparison of our approach with a host-based algorithm and discusses feature research directions.

2 Problem Statement and Mathematical Background

In this section we define the resultant of two polynomials and give an introduction to the theory of displacement structure which we use to compute univariate resultants.

2.1 Bivariate Polynomial Resultants

Let f and g be two polynomials in $\mathbb{Z}[x, y]$ of y-degrees p and q respectively: $f(x, y) = \sum_{i=0}^{p} f_i(x)y^i$ and $g(x, y) = \sum_{i=0}^{q} g_i(x)y^i$. Let $r = res_y(f, g)$ denote the resultant of f and g with respect to y. The resultant r is defined as the determinant of $(p+q) \times (p+q)$ Sylvester matrix S:

$$r = det(S) = det \begin{bmatrix} f_p & f_{p-1} & \dots & f_0 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & & \ddots & & \vdots \\ 0 & \dots & 0 & f_p & f_{p-1} & \dots & f_0 \\ g_q & g_{q-1} & \dots & g_0 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & & \ddots & & \vdots \\ 0 & \dots & 0 & g_q & g_{q-1} & \dots & g_0 \end{bmatrix}.$$

[1] http://www.mpi-inf.mpg.de/~emeliyan/cuda-compiler

Accordingly, the resultant of two monic polynomials f/f_p and g/g_q relates to $res_y(f,g)$ as follows:

$$res_y(f,g) = f_p^q g_q^p \cdot res_y(f/f_p, g/g_q).$$

Note that, the resultant is a polynomial in $\mathbb{Z}[x]$. Using modular and evaluation homomorphisms one can effectively avoid the arithmetic in polynomial domain as discussed in Section 4.

2.2 Displacement Structure and the Generalized Schur Algorithm in Application to Polynomial Resultants

We consider a strongly regular matrix $M \in \mathbb{Z}^{n\times n}$[2]. The matrix M is said to have a *displacement structure* if it satisfies the displacement equation:

$$\Omega M \Delta^T - FMA^T = GJB^T,$$

where $\Omega, \Delta, F, A \in \mathbb{Z}^{n\times n}$ are lower-triangular matrices, $J \in \mathbb{Z}^{r\times r}$ is a signature matrix, G and $B \in \mathbb{Z}^{n\times r}$ are *generator matrices*, such that GJB^T has a *constant* rank $r < n$. Then, r is called a *displacement rank* of M. We refer to [8,13] on the algorithms for general displacement structure and focus our attention on resultants.

Let $f, g \in \mathbb{Z}[x]$ be two polynomials of degrees p and q respectively, and $S \in \mathbb{Z}^{n\times n}$ be the associated Sylvester matrix ($n = p + q$). The matrix S is structured and has a displacement rank 2. It satisfies the displacement equation: $S - ZSZ^T = GJB^T$, where Z is a down-shift matrix zeroed everywhere except for 1's on its subdiagonal, $J = I \oplus -I \in \mathbb{Z}^{2\times 2}$. Accordingly, $G, B \in \mathbb{Z}^{n\times 2}$ are generators defined as follows:

$$B^T = \begin{bmatrix} f_p \ldots f_{p-q+1} & f_{p-q} & f_{p-q-1} \ldots f_0 & 0 \ldots 0 \\ g_q \ldots \quad g_1 & g_0 - f_p & -f_{p-1} \quad \ldots & -f_1 \end{bmatrix} \qquad \begin{array}{c} G \equiv 0 \text{ except for} \\ G_{0,0} = 1,\ G_{q,1} = -1 \end{array}$$

Our goal is to obtain an LDU^T-factorization of the matrix S, where the matrices L and U are lower triangular with unit diagonals, and D is a diagonal matrix. Having this factorization, the resultant is: $det(S) = det(D) = \prod_i^n d_{ii}$ (the product of diagonal entries of D).

The *generalized Schur algorithm* computes the matrix factorization by iteratively computing the *Schur complements* of leading submatrices. The Schur complement R of a submatrix A in M arises in the course of a block *Gaussian elimination* performed on the rows of matrix M, and is defined as:

$$R = M - CA^{-1}B, \quad \text{where} \quad M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}.$$

The main idea of the algorithm is to operate on low-rank matrix generators instead of the matrix itself giving an asymptotically fast solution. After n iterations the algorithm returns the Schur complement of an $n \times n$ leading submatrix

[2] In other words, a matrix whose leading principal minors are non-singular.

expressed in terms of matrix generators. In each step it brings the generators to a *proper* form. Let (G_i, B_i) denote the generators in step i. A proper form generator $\overline{G}_i$ has only *one* non-zero entry in its first row. The transformation is done by applying non-Hermitian rotation matrices Θ_i and Γ_i[3] to G_i and B_i respectively:

$$(G_i\Theta_i)^T = \overline{G}_i^T = \begin{bmatrix} \delta^i & a_1^i & a_2^i & \dots \\ 0 & b_1^i & b_2^i & \dots \end{bmatrix} \text{ and } (B_i\Gamma_i)^T = \overline{B}_i^T = \begin{bmatrix} \zeta^i & c_1^i & c_2^i & \dots \\ 0 & d_1^i & d_2^i & \dots \end{bmatrix}.$$

Once the generators are in proper form, it follows from the displacement equation that: $d_{ii} = \delta^i\zeta^i$. The next generator G_{i+1} is obtained from $\overline{G}_i$ by shifting down the column with first non-zero entry while keeping the other column intact (for explanations please refer to [8]):

$$\begin{bmatrix} 0 \\ G_{i+1} \end{bmatrix} = Z\overline{G}_i \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \overline{G}_i \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \text{ where } Z \text{ is a down-shift matrix.}$$

The generator B is processed by analogy. Remark that, the size of generators is *decreased* by one in each step of the algorithm.

2.3 Non-Hermitian Division-Free Rotations

Here the term non-Hermitian means that we apply rotation to a non-symmetric generator pair (G, B). Our task is to find matrices Θ and Γ satisfying: $\begin{bmatrix} a & b \end{bmatrix} \Theta = \begin{bmatrix} \alpha & 0 \end{bmatrix}$, $\begin{bmatrix} c & d \end{bmatrix} \Gamma = \begin{bmatrix} \beta & 0 \end{bmatrix}$ with $\Theta J \Gamma^T = J$. It is easy to check that these equations hold for the following matrices:

$$\Theta = \begin{bmatrix} c & -b/D \\ -d & a/D \end{bmatrix}, \quad \Gamma = \begin{bmatrix} a/D & -d \\ -b/D & c \end{bmatrix}, \quad \text{where} \quad D = ac - bd.$$

Note that, these formulae contain divisions which is undesirable as we are going to apply the algorithm in a finite field. Similar to Givens rotations [9], we use the idea to defer the division until the end of the algorithm by keeping a common denominator for each generator column. In other words, we express the generators in the following way:

$$G^T = \begin{bmatrix} 1/l_a & 0 \\ 0 & 1/l_b \end{bmatrix} \begin{bmatrix} a_0 & a_1 & \dots \\ b_0 & b_1 & \dots \end{bmatrix} \text{ and } B^T = \begin{bmatrix} 1/l_c & 0 \\ 0 & 1/l_d \end{bmatrix} \begin{bmatrix} c_0 & c_1 & \dots \\ d_0 & d_1 & \dots \end{bmatrix},$$

Then, the generator update $(\overline{G}, \overline{B}) = (G\Theta, B\Gamma)$ proceeds as follows:

$$\begin{aligned} \overline{a}_i &= l_a(a_ic_0 - b_id_0) \quad \overline{b}_i = l_b(b_ia_0 - a_ib_0)\text{ , where } G = (a_i, b_i), \\ \overline{c}_i &= l_c(c_ia_0 - d_ib_0) \quad \overline{d}_i = l_d(d_ic_0 - c_id_0)\text{ , where } B = (c_i, d_i). \end{aligned}$$

It can be shown that the denominators are *pairwise* equal, thus, we can keep only two of them. They are updated as follows: $\overline{l}_a = \overline{l}_d = \overline{a}_0$, $\overline{l}_c = \overline{l}_b = l_al_c^2$.

[3] Such matrices must satisfy: $\Theta J \Gamma^T = J$ to ensure that the displacement equation holds after transformation. In other words, we get: $G\Theta J(B\Gamma)^T = GJB^T$.

Apparently, the denominators must be non-zero to prevent the algorithm from the failure. This is guaranteed by the *strong-regularity* assumption introduced in the beginning. However, this is not always the case for Sylvester matrix. In Section 4.4 we discuss how to deal with this problem.

3 GPU Architecture and CUDA Framework

In this section we consider GPUs with NVIDIA Tesla architecture. The GPU comprises a set of Streaming Multiprocessor (SMs) which can execute vertex and fragment shaders as well as general purpose parallel programs. As an example, the GTX 280 contains 30 SMs. The GPU execution model is known as *single-instruction multiple-thread* or SIMT. It means that the SM applies an instruction to a group of 32 threads called *warps* which are always executed synchronously. If thread code paths within a warp diverge, the SM executes all taken paths serially. Different warps can execute disjoint paths without penalties.

On the top level, threads are grouped in a programmer defined *grid* of *thread blocks*. Such a model creates potentially unlimited parallel resources exploited dynamically by the target hardware. A thread block can contain up to 512 threads which can communicate using fast on-chip shared memory and synchronization barriers. The code running on the GPU is referred to as a *kernel* which is launched on a grid of thread blocks. Different blocks run completely independent from each other: data movement between thread blocks can be realized by splitting a program in two or more kernel launches[4].

CUDA memory model is built on five memory spaces. Each thread has a statically allocated fast local storage called *register file*. Registers is a scarce resource and should be used carefully to prevent spilling. The SM has a fixed amount of per-block on-chip *shared memory* (16 Kb). Shared memory is divided in 16 banks to facilitate concurrent access. The GPU has two cached memory spaces – read-only *constant* and *texture* memory – that are visible to all thread blocks and have a lifetime of an application. The remaining read-write *global memory* is also visible to the entire grid but is not cached on the device. It is crucial to stick to *memory coalescing* patterns in order to use the global memory bandwidth effectively.

4 Mapping Resultants Algorithm to Graphics Hardware

In this section we consider the algorithm step-by-step. We start with a high-level overview, then consider computation of univariate resultants and 24-bit modular arithmetic. Finally, we discuss the main implementation details and outline some ideas about the polynomial interpolation on the GPU.

4.1 Algorithm Overview

Our approach follows the "divide-conquer-combine" strategy of Collins' modular algorithm. At the beginning, the input polynomials are mapped to a prime field

[4] Block independence guarantees that a binary program will run unchanged on the hardware with any number of SMs.

for sufficiently many primes. The number of primes depends on the height of the resultant coefficients which is given by Hadamard's bound, see [14]. For each prime m_i we compute resultants at $x = \alpha_0, x = \alpha_1, \ldots \in \mathbb{Z}_{m_i}$. The degree bound (the number of evaluation points α_k) can be devised from the entries of Sylvester matrix [14]. The resultant $r \in \mathbb{Z}[x]$ is reconstructed from modular images using polynomial interpolation and the CRA. The first part of the algorithm is run on the GPU: we launch a kernel on a 2D grid $N \times S$,[5], where one thread block evaluates polynomials and computes one univariate resultant. The univariate resultant algorithm will be discussed in Section 4.2.

In order for the algorithm to work properly, we need to handle "bad" primes and evaluation points adequately. For two polynomials $f, g \in \mathbb{Z}[x, y]$ as defined in Section 2.1, *a prime m is said to be bad if $f_p \equiv 0 \bmod m$ or $g_q \equiv 0 \bmod m$.* Similarly, *an evaluation point $\alpha \in \mathbb{Z}_m$ is bad if $f_p(\alpha) \equiv 0 \bmod m$ or $g_q(\alpha) \equiv 0 \bmod m$.* "Bad" primes can be discarded quite easily: we do this during the initial modular reduction of polynomial coefficients prior to the grid launch. To deal with "bad" evaluation points we *enlarge* the grid by a small amount of excessive points (1–2%) such that, if for some points the algorithm fails, we still have enough information to reconstruct the result. The same technique is used to deal with *non-strongly regular* Sylvester matrices, see Section 2.3. In fact, non-strong regularity corresponds to the case where polynomial coefficients are related via some non-trivial equation which occurs rarely in practise and is confirmed by our tests (see Section 5). Indeed, if for some α_k Sylvester matrix is ill-conditioned, instead of using intricate methods, we simply ignore the result and take another evaluation point. In a very "unlucky" case when we cannot reconstruct the resultant due to the lack of points, we launch another grid to compute extra information.

It is worth mentioning, that the another interesting approach to compute polynomial resultants is given in [15]. It is based on modular arithmetic and linear recurring sequences. Although, this algorithm seemingly has a connection to the PRS, it is yet unclear whether it can be a good candidate for realization on the GPU.

4.2 Univariate Resultant Algorithm

The resultant $res(f, g) \bmod m_i$ at $x = \alpha_j$ is computed using the method explained in Section 2.2. In each iteration the algorithm multiplies the generators by rotation matrices collecting one factor of the resultant per iteration. Then, the generator columns are shifted down to obtain the generators for the next iteration. After $n = p + q$ iterations (notations are as in Section 2.2), the product of the factors yields the resultant.

The original algorithm can largely be improved. First, we write the generators as pairs of column vectors: $G = (a, b)$, $B = (c, d)$. Now, remark that, at the beginning $G \equiv 0$ except for two entries: $a_0 = 1$, $b_q = -1$. If we run the algorithm on monic polynomials[6], we can observe that the vectors a, b and c stay *constant*

[5] Here N denotes the number of moduli, and S – the number of evaluation points.

[6] In other words, on polynomials with unit leading coefficients.

during the first q iterations of the algorithm (except for a single entry a_q). Indeed, because polynomials are monic, c_0 and d_0 are initially ones, and so is the denominator of the rotation matrices (see Section 2.3): $D = a_0c_0 - b_0d_0 = c_0 \equiv 1$. Thus, we can get rid of the denominators completely which greatly simplifies the rotation formulae. Moreover, the first q factors of the resultant returned by the algorithm are *unit*, therefore we can skip them. However, we need to multiply the resultant by $f_p^q g_q^p$ as to compensate for running the algorithm on monic polynomials. The pseudocode is given below:

```
1: procedure RESULTANT_UNIVARIATE($f$ : Polynomial, $g$ : Polynomial)
2:     $p = \mathrm{degree}(f),\ q = \mathrm{degree}(g),\ n = p + q$
3:     $f \leftarrow f/f_p,\ g \leftarrow g/g_q$                          ▷ convert polynomials to monic form
4:     $G = (\mathbf{a}, \mathbf{b}),\ B = (\mathbf{c}, \mathbf{d})$          ▷ set up generators: see Section 2.2 for details
5:     for $j = 0$ to $q - 1$ do                                  ▷ first q iterations are simplified
6:         $\mathbf{d}_i \leftarrow \mathbf{d}_i - \mathbf{c}_i\mathbf{d}_j$ for $\forall i = j+1 \ldots n-1$      ▷ multiply by the rotation matrix
7:         $\mathbf{a}_q = \mathbf{d}_j$                                        ▷ update a single entry of a
8:         $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i, \mathbf{c}_{i+1} \leftarrow \mathbf{c}_i$ for $\forall i = j+1 \ldots n-2$      ▷ shift down the generators
9:     end for
10:    $l_a = 1,\ l_c = 1,\ res = 1,\ l_{res} = 1$              ▷ denominators and resultant are set to 1
11:    for $j = q$ to $n - 1$ do
12:        for $i = j$ to $n - 1$ do                  ▷ multiply the generators by rotation matrices
13:            $s = l_a(\mathbf{a}_i c_j - \mathbf{b}_i d_j),\ \mathbf{b}_i = l_c(\mathbf{b}_i a_j - \mathbf{a}_i b_j),\ \mathbf{a}_i = s$
14:            $t = l_c(\mathbf{c}_i a_j - \mathbf{d}_i b_j),\ \mathbf{d}_i = l_a(\mathbf{d}_i c_j - \mathbf{c}_i d_j),\ \mathbf{c}_i = t$
15:        end for
16:        $l_c = l_a l_c^2,\ l_a = \mathbf{a}_j, res = res \cdot \mathbf{c}_j,\ l_{res} = l_{res} \cdot l_c$        ▷ update the denominators
17:        $\mathbf{a}_{i+1} \leftarrow \mathbf{a}_i, \mathbf{c}_{i+1} \leftarrow \mathbf{c}_i$ for $\forall i = j \ldots n-2$          ▷ shift down the generators
18:    end for
19:    return $res \cdot f_p^q \cdot g_q^p / l_{res}$                              ▷ return the resultant
20: end procedure
```

We will refer to iterations $j = 0 \ldots q-1$ and $j = q \ldots n-1$ as type S and T iterations respectively. For division in lines 3 and 19 we use the modified Montgomery modular inverse [16] with improvements from [17]. The number of iterations of this algorithm is bounded by moduli bitlength (24 bits), see Appendix A.

4.3 24-Bit Modular Arithmetic on the GPU

Modular multiplication is a challenging problem due to the limited hardware support for integer arithmetic. The GPU natively supports only 24-bit integer multiplication realized by mul24.lo and mul24.hi instructions[7]. However, the latter instruction is *not* exposed by CUDA API. To overcome this limitation, the authors of [10] propose to use slow 32-bit multiplication, while the tests from [11] show that 12-bit arithmetic is faster because modular reduction can be done in floating-point without overflow concerns.

[7] They return 32 least and most significant bits of the product of 24-bit operands respectively.

We use the arithmetic based on mixing floating-point and integer computations [1] which is supported by the patched CUDA compiler[8]. In what follows, we will refer to umul24 and umul24hi as intrinsics for mul24.lo and mul24.hi respectively. The procedure MUL_MOD in Algorithm 1 computes $a \cdot b \bmod m$ for two 24-bit residues. The idea is to split the product as follows: $a \cdot b = 2^{16}hi + lo$ (32 and 16 bits), and then use a congruence ($0 \le \lambda < m$):

$$2^{16}hi + lo = (m \cdot l + \lambda) + lo \equiv_m \lambda + lo = 2^{16}hi + lo - m \cdot l = a \cdot b - l \cdot m = r$$

It can be checked that $r \in [-2m + \varepsilon; m + \varepsilon]$ for $0 \le \varepsilon < m$. Thus, r fits in a 32-bit word and it suffices to compute only 32 *least significant bits* of products ($a \cdot b$ and $m \cdot l$) as shown in line 5 of the algorithm. Finally, the reduction in lines 6–7 maps r to the valid range $[0; m-1]$.

The next procedure SUB_MUL_MOD is an extended version of MUL_MOD which is used to implement matrix rotations: it evaluates $(x_1y_1 - x_2y_2) \bmod m$ (see Section 2.3). The algorithm computes the products x_1y_1 and x_2y_2, and subtracts partially reduced residues. Adding $m \cdot 100$ in line 14 is necessary to keep the intermediate result positive since umul24 operation in line 16 cannot handle negative operands. In total, line 14 is compiled in 4 multiply-add (MAD) instructions[9]. The remaining part is an inlined REDUCE_MOD operation (see [1]) with a minor change. Namely, in line 15 we use the mantissa trick [18] to multiply by $1/m$ and round the result down using a single MAD instruction.

4.4 Putting It All Together

Having all the ingredients at hand, we can now discuss how the algorithm maps to the GPU. The GPU part of the algorithm is realized by two kernels, see Figure 1 (a). The first kernel calculates modular resultants, while the second one eliminates zero denominators from the input sequence and multiplies resultants by respective modular inverses $\mathsf{l}_{\mathsf{res}}^{-1}$. Grid configuration for each kernel launch is shown to the left.

The number of threads per block for the **first kernel** depends on the maximal y-degree of polynomials being processed. We will use the notation: $p = deg_y(f)$, $q = deg_y(g)$, where $f, g \in \mathbb{Z}[x, y]$ and $p \ge q$. We provide three kernel instantiations for different polynomial degrees: kernel **A** with 64 threads per block for $p \in [32, 63]$; **B** – 96 threads for $p \in [64, 95]$; and **C** – 128 threads for $p \in [96, 127]$. One reason behind this configuration is that we use $p + 1$ threads to evaluate coefficients of f at $x = \alpha_i$ in parallel using Horner form (the same for g). The resultant algorithm consists of one outer loop split up in iterations of type S and T, see Section 4.1. The inner loop of the algorithm is completely vectorized: this is another reason why we need the number of threads to match the polynomial degree. Remark that, in each iteration the size of the generators decreases by

[8] http://www.mpi-inf.mpg.de/~emeliyan/cuda-compiler

[9] The graphics hardware supports 24-bit integer as well as floating-point MADs. The compiler aggressively optimizes subsequent multiply and adds to use MAD instructions.

Algorithm 1. 24-bit modular arithmetic on the GPU

```
1: procedure MUL_MOD(a, b, m, invm)                        ▷ $\mathsf{invm} = 2^{16}/\mathsf{m}$ (in floating-point)
2:     $\mathsf{hi} = \mathsf{umul24hi(a, b)}$                                  ▷ high 32 bits of the product
3:     $\mathsf{prodf} = \mathsf{fmul\_rn(hi, invm)}$                          ▷ multiply in floating-point
4:     $\mathsf{l} = \mathsf{float2uint\_rz(prodf)}$               ▷ integer truncation: $\mathsf{l} = \lfloor \mathsf{hi} \cdot 2^{16}/\mathsf{m} \rfloor$
5:     $\mathsf{r} = \mathsf{umul24(a, b)} - \mathsf{umul24(l, m)}$     ▷ now $\mathsf{r} \in [-2\mathsf{m} + \varepsilon; \mathsf{m} + \varepsilon]$ with $0 \le \varepsilon < \mathsf{m}$
6:     if $\mathsf{r} < 0$ then $\mathsf{r} = \mathsf{r} + \mathsf{umul24(m, 0x1000002)}$ fi    ▷ multiply-add: $\mathsf{r} = \mathsf{r} + \mathsf{m} \cdot 2$
7:     return $\mathsf{umin(r, r - m)}$                          ▷ return $\mathsf{r} = \mathsf{a} \cdot \mathsf{b} \bmod \mathsf{m}$
8: end procedure
9: procedure SUB_MUL_MOD(x1, y1, x2, y2, m, invm1, invm2)
10:    $\mathsf{h1} = \mathsf{umul24hi(x1, y1)},\ \mathsf{h2} = \mathsf{umul24hi(x2, y2)}$          ▷ two inlined MUL_MOD's
11:    $\mathsf{pf1} = \mathsf{fmul\_rn(h1, invm1)},\ \mathsf{pf2} = \mathsf{fmul\_rn(h2, invm1)}$              ▷ $\mathsf{invm1} = 2^{16}/\mathsf{m}$
12:    $\mathsf{l1} = \mathsf{float2uint\_rz(pf1)},\ \mathsf{l2} = \mathsf{float2uint\_rz(pf2)}$
13:            ▷ compute an intermediate product r, $\mathsf{mc} = \mathsf{m} \cdot 100$:
14:    $\mathsf{r} = \mathsf{mc} + \mathsf{umul24(x1, y1)} - \mathsf{umul24(l1, m)} - \mathsf{umul24(x2, y2)} + \mathsf{umul24(l2, m)}$
15:    $\mathsf{rf} = \mathsf{uint2float\_rn(r)} * \mathsf{invm2} + \mathsf{e23}$        ▷ $\mathsf{invm2} = 1/\mathsf{m},\ \mathsf{e23} = 2^{23},\ \mathsf{rf} = \lfloor \mathsf{r}/\mathsf{m} \rfloor$
16:    $\mathsf{r} = \mathsf{r} - \mathsf{umul24(float\_as\_int(rf), m)}$                     ▷ $\mathsf{r} = \mathsf{r} - \lfloor \mathsf{r}/\mathsf{m} \rfloor \cdot \mathsf{m}$
17:    return $(\mathsf{r} < 0\ ?\ \mathsf{r} + \mathsf{m} : \mathsf{r})$
18: end procedure
```

1, and so is the number of working threads, see Figure 1 (b). To achieve higher thread occupancy, we unroll the type S iterations by the factor of 2. In this way, we double the maximal degree of polynomials that can be handled, and ensure that all threads are occupied. Moreover, at the beginning of type T iterations we can guarantee that not less than half of threads are in use in the corner case. We keep the column vectors a and c of the generators $G = (a, b)$ and $B = (c, d)$ in shared memory because they need to be shifted down in each iteration. The vectors b and d are located in register space. Accordingly, each iteration (of type S or T) consists of fetching current first rows of G and B (these are shared by all threads), transforming the generators using SUB_MUL_MOD operation, saving computed factors in shared memory (only for type T iterations), and shifting down the columns a and c, see Figure 1 (b). Also, during type T iterations we keep track of the size of generators and switch to iterations *without* sync on crossing the *warp* boundary[10]. The final result is given by the product $f_p^q g_q^p \cdot \prod_i d_{ii}$ (see Section 2.2). We compute this product efficiently using "warp-sized" reductions [19]. The idea is to run prefix sums for different warps separately omitting synchronization barriers, and then combine the results in a final reduction step, see Figure 1 (c). The **second kernel**, launched with 128 threads, runs stream compaction for each modulus in parallel, and then computes modular inverses for the remaining entries. Stream compaction algorithm is also based on "warp-sized" reductions[11].

[10] Warp, as a minimal scheduling entity, is always executed synchronously, hence, shared memory access not need to be synchronized.

[11] Stream compaction can be regarded to as an exclusive prefix sum of 0's and 1's where 0's correspond to elements being eliminated.

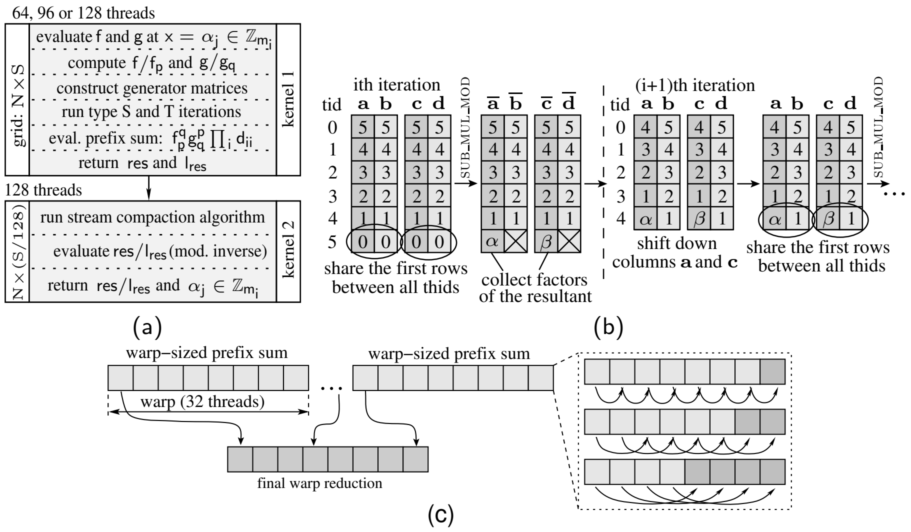

64, 96 or 128 threads
evaluate f and g at $x = \alpha_j \in \mathbb{Z}_{m_i}$
compute f/f_p and g/g_q
construct generator matrices
run type S and T iterations
eval. prefix sum: $f_p^q g_q^p \prod_i d_{ii}$
return res and l_{res}
grid: N×S
kernel 1
128 threads
run stream compaction algorithm
evaluate res/l_{res} (mod. inverse)
return res/l_{res} and $\alpha_j \in \mathbb{Z}_{m_i}$
N×(S/128)
kernel 2
(a)
ith iteration
tid
SUB_MUL_MOD
share the first rows between all thids
collect factors of the resultant
(i+1)th iteration
shift down columns a and c
share the first rows between all thids
(b)
warp–sized prefix sum
warp (32 threads)
final warp reduction
(c)

Fig. 1. (a) GPU part of the algorithm consisting of two kernel launches (N – number of moduli, S – number of eval. points); (b) vector updates during the type T iterations (**tid** denotes the thread ID); (c) warp-sized reduction (prefix sum)

4.5 Polynomial Interpolation

Now we sketch some ideas on how to realize polynomial interpolation efficiently on the GPU. The task of interpolation is to find a polynomial $f(x)$, $deg(f) \leq n$, satisfying the following set of equations: $f(\alpha_i) = y_i$, for $0 \leq i \leq n$. The polynomial coefficients a_i are given by the solution of an $(n+1) \times (n+1)$ *Vandermonde* system:

$$V\mathbf{a} = \mathbf{y}, \text{ where } V \text{ is a Vandermonde matrix: } V_{ij} = \alpha_i^j \ (i, j = 0, \ldots, n).$$

Vandermonde matrix is structured and has a displacement rank 1. Thus, we can adapt the generalized Schur algorithm to solve the linear system in a small parallel time. Namely, if we apply the algorithm to the following matrix embedding:

$$M = \begin{bmatrix} V & -\mathbf{y} \\ I & \mathbf{0} \end{bmatrix},$$

then after $n+1$ steps we obtain the Schur complement R of a submatrix V which is equal to: $R = 0 - IV^{-1}(-\mathbf{y}) = V^{-1}\mathbf{y}$, i.e., the solution of a Vandermonde system.

5 Experiments and Conclusion

We have tested our algorithm on the *GeForce GTX 280* graphics processor. As a reference implementation we have use the resultant algorithm [2] from CGAL[12]

[12] www.cgal.org

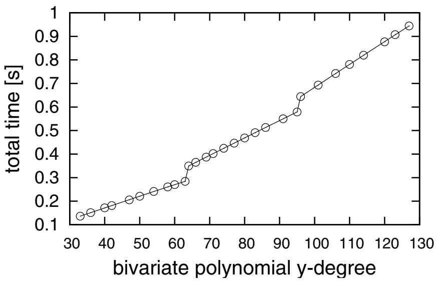
Table 1. Timings. *First column.* p and q: polynomial y-degrees; **points:** # of evaluation points; **moduli:** # of 24-bit moduli; *Second column.* **eval:** polynomial evaluation; **res:** univariate resultants; **interp:** polynomial interpolation; **CRA:** Chinese remaindering; **ERR:** # of wrong entries computed

parameters setup	CPU timing breakdown	CPU eval + resultant	GPU eval + resultant	ratio	ERR
$p: 32,\ q: 32$ points: 970 moduli: 96	eval: 29.5 s, res: 25.4 s interp: 27.6 s, CRA: 0.63 s total: 83.8 s	54.9 s	131.6 ms	**417x**	1
$p: 50,\ q: 43$ points: 940 moduli: 101	eval: 26.1 s, res: 47.0 s interp: 26.9 s, CRA: 0.69 s total: 101.0 s	73.1 s	175.7 ms	**415x**	0
$p: 63,\ q: 63$ points: 1273 moduli: 136	eval: 55.1 s, res: 148.4 s interp: 65.7 s, CRA: 1.32 s total: 271.1 s	203.5 s	393.5 ms	**517x**	3
$p: 70,\ q: 64$ points: 1354 moduli: 102	eval: 48.7 s, res: 129.2 s interp: 57.2 s, CRA: 1.0 s total: 236.1 s	177.9 s	492.6 ms	**361x**	4
$p: 95,\ q: 95$ points: 1152 moduli: 145	eval: 45.0 s, res: 292.5 s interp: 57.8 s, CRA: 1.3 s total: 397.1 s	337.5 s	752.5 ms	**448x**	2
$p: 120,\ q: 99$ points: 1549 moduli: 130	eval: 71.3 s, res: 461.2 s interp: 93.3 s, CRA: 1.53 s total: 627.9 s	532.5 s	1363.6 ms	**390x**	3

(Computational Geometry Algorithms Library) run on the 2.5Ghz *Quad-Core Intel Xeon E5420* with 12MB L2 cache and 8Gb RAM under 32-bit Linux platform. The code has been compiled with '*–DNDEBUG –O3 –march=core2*' options. We have benchmarked the first two stages of the host-based algorithm (evaluate + resultant) and compared them with our realization. Table 1 summarizes the running time for different configurations. The GPU timing includes the time for GPU–host data transfer for objective comparison. The number of evaluation points has been increased by 2% to accommodate "unlucky" primes. The total number of evaluation points for which the algorithm fails is given by the column **ERR** in the table. The tests confirm that, indeed, this occurs rarely on the average. Observe that, the maximal speed-up is attained for $p = \{63, 95\}$ (see Table 1): this is no surprise as these parameters correspond to the full thread occupancy. In total, one can see that our algorithm outperforms the CPU implementation by a large factor. Moreover, due to the vast amount of blocks executed[13], the algorithm achieves a full utilization of the GTX280 graphics card and we expect the performance to scale well on forthcoming GPUs.

The left graph in Figure 2 examines the performance depending on the polynomials y-degree (which are chosen to be equal) with the number of moduli and evaluation points fixed to 128 and 1000 respectively. One can see that the

[13] Recall that, the grid size equals to 'number of moduli' $\times$ 'number of points'.



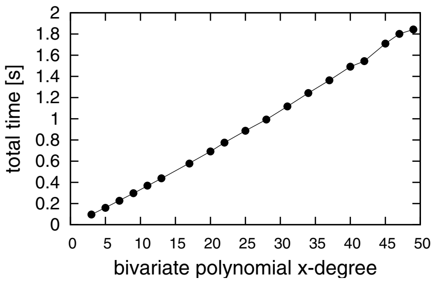


Fig. 2. The running time as a function of the polynomials' y-degree (left) and x-degree (right)

performance scales linearly with the degree. This is an expected behavior because the algorithm consists of one outer loop (while the inner loop is vectorized). Performance degradation at the warp boundary (64 and 96) is due to switching to a larger kernel once all thread resources are exhausted. It might be possible to smooth "stairs" by dynamically balancing the thread workload. The second graph in Figure 2 evaluates how the running time grows with the x-degree (and y-degree fixed). Linear dependency is because of the fact that the x-degree only causes the number of evaluation points (one grid dimension) to increase while the number of moduli remains the same.

To conclude, we have identified that with the approach of displacement structure we can harness the power of GPUs to compute polynomial resultants. Our algorithm has achieved a considerable speed-up which was previously beyond the reach of traditional serial algorithms. Certainly, this is only the first step in realization of a complete and robust resultant algorithm on graphics hardware. From Table 1 one can see that polynomial interpolation could be quite expensive. Nevertheless, our benchmarks clearly show that graphics processors have a great performance potential in such a not yet well-explored application domain. Moreover, with the ideas from Section 4.4, we are currently underway to realize polynomial interpolation on the GPU.

References

1. Emeliyanenko, P.: Efficient multiplication of polynomials on graphics hardware. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 134–149. Springer, Heidelberg (2009)
2. Hemmer, M.: Polynomials, CGAL - Computational Geometry Algorithms Library, release 3.4. CGAL, Campus E1 4, 66123 Saarbrücken, Germany (January 2009)
3. Collins, G.E.: The calculation of multivariate polynomial resultants. In: SYMSAC 1971, pp. 212–222. ACM, New York (1971)
4. Bubeck, T., Hiller, M., Küchlin, W., Rosenstiel, W.: Distributed Symbolic Computation with DTS. In: IRREGULAR 1995, pp. 231–248. Springer, London (1995)
5. Schreiner, W.: Developing a distributed system for algebraic geometry. In: EURO-CM-PAR 1999, pp. 137–146. Civil-Comp Press (1999)

6. Hong, H., Loidl, H.W.: Parallel computation of modular multivariate polynomial resultants on a shared memory machine. In: Buchberger, B., Volkert, J. (eds.) CONPAR 1994 and VAPP 1994. LNCS, vol. 854, pp. 325–336. Springer, Heidelberg (1994)
7. Geddes, K., Czapor, S., Labahn, G.: Algorithms for computer algebra. Kluwer Academic Publishers, Dordrecht (1992)
8. Kailath, T., Ali, S.: Displacement structure: theory and applications. SIAM Review 37, 297–386 (1995)
9. Frantzeskakis, E., Liu, K.: A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing. IEEE Transactions on Signal Processing 42, 2455–2469 (1994)
10. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
11. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 350–367. Springer, Heidelberg (2009)
12. NVIDIA: CUDA Compute Unified Device Architecture. NVIDIA Corp. (2007)
13. Chandrasekaran, S., Sayed, A.H.: A Fast Stable Solver for Nonsymmetric Toeplitz and Quasi-Toeplitz Systems of Linear Equations. SIAM J. Matrix Anal. Appl. 19, 107–139 (1998)
14. Monagan, M.: Probabilistic algorithms for computing resultants. In: ISSAC 2005, pp. 245–252. ACM, New York (2005)
15. Llovet, J., Martínez, R., Jaén, J.A.: Linear recurring sequences for computing the resultant of multivariate polynomials. J. Comput. Appl. Math. 49(1-3), 145–152 (1993)
16. de Dormale, G., Bulens, P., Quisquater, J.J.: An improved Montgomery modular inversion targeted for efficient implementation on FPGA. In: IEEE International Conference on FPT 2004, pp. 441–444 (2004)
17. Savas, E., Koc, C.: The montgomery modular inverse-revisited. IEEE Transactions on Computers 49(7), 763–766 (2000)
18. Hecker, C.: Let's get to the (floating) point. Game Developer Magazine, 19–24 (1996)
19. Hillis, W.D., Steele Jr., G.L.: Data parallel algorithms. Commun. ACM 29, 1170–1183 (1986)

A Montgomery Modular Inverse Algorithm

To realize efficient Montgomery modular inverse on the GPU, we have combined the algorithm from [16] with ideas from [17], and took advantage of the fast 24-bit integer multiplication supported by the GPU. The algorithm comprises two stages. In the first stage we iteratively compute $x^{-1}2^k \bmod m$[14], where $s \leq k \leq 2s$ and $s = \lceil \log_2 m \rceil$. Then, we run two Montgomery multiplications by the powers of two to get rid of 2^k factor. The pseudocode is given below:

```
1: procedure MONTGOMERY_INVERSE(x, m, mu)          ▷ computes $x^{-1} \bmod m$
2:     $v = x,\ u = m,\ s = 1,\ r = 0,\ k = 0$          ▷ x is a 24-bit residue modulo m
3:     repeat                    ▷ first stage: compute $r = x^{-1}2^k \bmod m$ iteratively
4:         $tmprs = r$
5:         if $v \bmod 2 = 1$ then
6:             $safeuv = v$
7:             if $(v$ xor $u) < 0$ then $v = v + u$ else $v = v - u$ fi
8:             if $(v$ xor $safeuv) < 0$ then $u = safeuv,\ tmprs = s$ fi
9:             $s = s + r$
10:        fi
11:        $v = v/2,\ r = tmprs \cdot 2,\ k = k + 1$
12:    until $v \neq 0$
13:    $r = m - r$                              ▷ second stage: get rid of $2^k$ factor
14:    if $r < 0$ then $r = r + m$ fi                ▷ $r = x^{-1}2^k \bmod m,\ 24 \leq k \leq 48$
15:    if $k > 24$ then     ▷ first multiply: $r = (x^{-1}2^k)(2^{-m}) = x^{-1}2^{k-m} \pmod m$
16:        $c = umul24(r, mu)$                               ▷ $mu = -m^{-1} \bmod 2^{24}$
17:        $lo = umul24(c, m),\ hi = \_umul24hi(c, m)$          ▷ $(hi, lo) = c \cdot m$ (48 bits)
18:        $lo = umul24(lo, 0x1000001) + r$                  ▷ $lo = (lo \bmod 2^{24}) + r$
19:        $r = hi/2^8 + lo/2^{24},\ k = k - 24$                      ▷ $r = (lo, hi)/2^{24}$
20:    fi
21:          ▷ second Montgomery multiply: $r = (x^{-1}2^k)(2^{m-k})(2^{-m}) = x^{-1} \pmod m$
22:    $c = r \cdot 2^{24-k},\ d = umul24(c, mu)$                    ▷ $mu = -m^{-1} \bmod 2^{24}$
23:    $lo = umul24(d, m),\ hi = umul24hi(d, m)$               ▷ $(hi, lo) = d \cdot m$ (48 bits)
24:    $d = r/2^k, lo = lo \bmod 2^{24}$
25:    $lo = umul24(c, 0x1000001) + lo$                     ▷ $lo = (c \bmod 2^{24}) + r$
26:    $r = hi/2^8 + d + lo/2^{24}$
27:    return r
28: end procedure
```

[14] The number of iterations is bounded by moduli bit-length (24 bits).

A Novel Scheme for High Performance Finite-Difference Time-Domain (FDTD) Computations Based on GPU

Tianshu Chu[1], Jian Dai[2], Depei Qian[1], Weiwei Fang[3], and Yi Liu[1]

[1] School of Computer Science and Engineering, Beihang University, 100191, Beijing, P.R. China
{tianshu.chu,depei.qian,yi.liu}@jsi.buaa.edu.cn
[2] School of Electronics and Information Engineering, Beihang University, 100191, Beijing, P.R. China
daijian2003cs@163.com
[3] School of Computer and Information Technology, Beijing Jiaotong University, 100044, Beijing, P.R. China
fangvv@gmail.com

Abstract. Finite-Difference Time-Domain (FDTD) has been proved to be a very useful computational electromagnetic algorithm. However, the scheme based on traditional general purpose processors can be computationally prohibitive and require thousands of CPU hours, which hinders the large-scale application of FDTD. With rapid progress on GPU hardware capability and its programmability, we propose in this paper a novel scheme in which GPU is applied to accelerate three-dimensional FDTD with UPML absorbing boundary conditions. This GPU-based scheme can reduce the computation time significantly, while obtaining high accuracy as compared with the CPU-based scheme. With only one AMD ATI HD4850 GPU, when computational domain is up to (180×180×180), our implementation of the GPU-based FDTD performs approximately 93 times faster than the one running with Intel E2180 dual cores CPU.

Keywords: computational electromagnetic, FDTD, GPGPU, parallel computing, ATI Stream.

1 Introduction

First introduced by Yee in 1966 [1], the algorithm of Finite-Difference Time-Domain (FDTD) has received significant research attention in the electromagnetic community. It has been proved to be a powerful method for solving time domain electromagnetic problems.

In recent years, the algorithm has been further developed by many researchers. Some sub-technologies for FDTD have been put forward to improve the algorithm greatly, such as conformal FDTD, local sub-cell FDTD, ADI-FDTD, parallel FDTD, PSTD and hybrid FDTD-FE technology. With the help of these sub-technologies, FDTD has been applied to more domains, including electromagnetic scattering, antennas design, etc [2].

Unfortunately, for large-scale application of FDTD, there are still some restrictions. Among them, requirement to large system memory and long computing time is especially prohibitive. To overcome those drawbacks, researchers have proposed some parallel FDTD algorithms. Those algorithms make use of system memory and computing resource of the cluster to meet the need of large memory and to reduce the simulation time [3]. However, as the number of compute nodes in the cluster increases, the efficiency of communication among nodes will go down. Thus, reducing computing time for FDTD becomes a tough issue for engineering application.

The high computational power of graphics processor unit (GPU) has been noticed by many researches in recent years. The GPU-based FDTD algorithm was first proposed by Sean Krakiwsky in 2004 [4], and then improved by some other researchers [5-10]. In comparison with CPU, GPU is of the capability of parallel computing with multi-pipelines and greater data throughput [11]. It can achieve very significant performance improvement over CPU-based FDTD computing [12].

Nevertheless, GPU computing has not been very popular among researchers, due to the demand of profound knowledge about GPU hardware design and graphics API (i.e. OpenGL). Introduction of CUDA (Compute Unified Device Architecture) and Stream, by the leading GPU manufacturer Nvidia and ATI, brought GPU computing into a new era. They provide researchers a more accessible way to bring the power of GPU into full play. To the best of our knowledge, very few GPU-based FDTD schemes are implemented on ATI architecture. In this paper, we propose a novel scheme for high performance FDTD computing based on ATI Stream.

Although great achievement on FDTD algorithm for GPU application has been made, there are still rooms for performance improvement and problems needed to be resolved. This paper presents a step forward and proposes a new array of optimization techniques and algorithms to further improve the performance of GPU-based FDTD. In addition, the UPML [2] absorbing boundary condition applied in the scheme makes it possible to solve practical problem with high performance.

The rest of this paper is organized as follows. Section 2 provides an overview of both the basic theory of FDTD algorithm and the GPU computing. A novel practical tuning technique for three-dimensional FDTD with UPML is presented in Section 3. Section 4 evaluates our optimized FDTD algorithm based on GPU against the CPU-based scheme in both accuracy and speed-up ratio. In addition, a case study is presented to prove the validity of our scheme. Finally Section 5 concludes this paper and looks into other issues for further improvement of the performance of FDTD.

2 Background

2.1 The System Model

In order to elucidate resources available to a programmer, we shortly describe the ATI HD4850 GPU, on which our schemes were implemented. HD4850 is a product of AMD Corp. which is a single-GPU card published in 2008. The central core of HD4850 is RV770, which has ten SIMD engines, each with 16 thread processors, and each thread processor contains five stream cores. It means that RV770 has 800 stream cores which are used for parallel computation [13]. The detailed parameters are shown in table 1.

Table 1. The RV770 parameters of HD4850 video card

Stream Processor – RV770			
Core freq	625MHz	Thread processors	16
Core crafts	55nm	Memory freq	1986MHz
Transistor amount	0.956bil	Memory size	512M
Memory type	GDDR3	Memory bit-width	256 bit
Texture unit	40	Stream Core	800

The HD4850 GPU is supported by ATI Stream programming platform. The implementation in this paper is developed with a high-level language, ATI Brook+ [13].

For performance comparison between CPU and GPU, a serial code is implemented and executed on Intel Dual E2180 processor.

2.2 The Basic Scheme of FDTD Algorithm

Maxwell's equations are the basic and significant equations defining macroscopically electromagnetic phenomenon. However, it would be difficult or impossible to solve Maxwell's equations for arbitrary model spaces. FDTD is a direct time domain solution to Maxwell's curl equations, which are given as follows:

$$\nabla\times\vec{H}(\vec{r},t)=\varepsilon\frac{\partial\vec{E}(\vec{r},t)}{\partial t}+\sigma\vec{E}(\vec{r},t) \tag{1}$$

$$\nabla\times\vec{E}(\vec{r},t)=-\mu\frac{\partial\vec{H}(\vec{r},t)}{\partial t}-\sigma_m\vec{H}(\vec{r},t) \tag{2}$$

Formula (1) and (2) can be respectively cast into three scalar partial differential equations in the Cartesian coordinates as follows [2].

$$\frac{\partial H_x}{\partial t}=\frac{1}{\mu}\left(\frac{\partial E_y}{\partial z}-\frac{\partial E_z}{\partial y}-\sigma_m H_x\right) \tag{3}$$

$$\frac{\partial H_y}{\partial t}=\frac{1}{\mu}\left(\frac{\partial E_z}{\partial x}-\frac{\partial E_x}{\partial z}-\sigma_m H_y\right) \tag{4}$$

$$\frac{\partial H_z}{\partial t}=\frac{1}{\mu}\left(\frac{\partial E_x}{\partial y}-\frac{\partial E_y}{\partial x}-\sigma_m H_z\right) \tag{5}$$

$$\frac{\partial E_x}{\partial t}=\frac{1}{\varepsilon}\left(\frac{\partial H_z}{\partial y}-\frac{\partial H_y}{\partial z}-\sigma E_x\right) \tag{6}$$

$$\frac{\partial E_y}{\partial t}=\frac{1}{\varepsilon}\left(\frac{\partial H_x}{\partial z}-\frac{\partial H_z}{\partial x}-\sigma E_y\right) \tag{7}$$

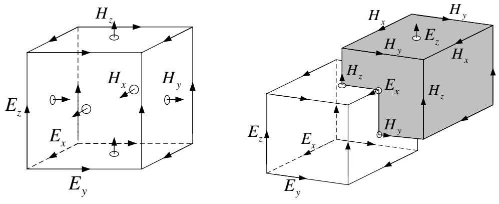
$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z\right) \tag{8}$$

Formulas (3)-(8) can be modified according to discrete time domain and spatial domain. Below is the resulting *x*-directed *E* field equation; the other 5 field components are similar [2].

$$Ex^{n+1}_{(i+1/2,j,k)} = \frac{\varepsilon_x - 0.5\Delta t\sigma_x}{\varepsilon_x + 0.5\Delta t\sigma_x} Ex^{n}_{(i+1/2,j,k)} +$$

$$\frac{\Delta t}{\varepsilon_x + 0.5\Delta t\sigma_x}\left(\frac{Hz^{n+1/2}_{(i+1/2,j+1/2,k)} - Hz^{n+1/2}_{(i+1/2,j-1/2,k)}}{\Delta y} - \frac{Hy^{n+1/2}_{(i+1/2,j,k+1/2)} - Hy^{n+1/2}_{(i+1/2,j,k-1/2)}}{\Delta z}\right) \tag{9}$$

Another important FDTD concept is known as the Yee Space Grid, shown in Figure 1.



(a) Electric grid cell (b) Electric grid and magnetic grid

Fig. 1. Relationship between electric and magnetic field for the format of Yee

The characteristic of the Yee Space Grid is the interleaved placement of the *E* and *H* fields. The *E* fields are interleaved with *H* fields by half of the time step and so are the *H* fields. *E* and *H* are centered in the surface of each other. Thus, each *E* field component is surrounded by four *H* field components and vice versa.

Many schemes which use GPU to accelerate FDTD computation do not take PML region. It means that they cannot be used to solve the problems of electromagnetic wave propagation in free space. Therefore, in this paper, we adopt UPML to improve those schemes. With UPML, the formula of E_x can be given as follows [2]:

$$Ex^{n+1}_{(i+1/2,j,k)} = C1(m)Ex^{n}_{(i+1/2,j,k)} + C2(m)Dx^{n+1}_{(i+1/2,j,k)} - C3(m)Dx^{n}_{(i+1/2,j,k)} \tag{10}$$

$$Dx^{n+1}_{(i+1/2,j,k)} = CA(m)Dx^{n}_{(i+1/2,j,k)} +$$

$$CB(m)\left(\frac{Hz^{n+1/2}_{(i+1/2,j+1/2,k)} - Hz^{n+1/2}_{(i+1/2,j-1/2,k)}}{\Delta y} - \frac{Hy^{n+1/2}_{(i+1/2,j,k+1/2)} - Hy^{n+1/2}_{(i+1/2,j,k-1/2)}}{\Delta z}\right) \tag{11}$$

where the variables *CA(m)*, *CB(m)*, *C1(m)*, *C2(m)*, and *C3(m)* are set as:

$$CA(m) = \frac{\kappa_y(m)/\Delta t - \sigma_y(m)/2\varepsilon_0}{\kappa_y(m)/\Delta t + \sigma_y(m)/2\varepsilon_0} \tag{12}$$

$$CB(m)=\frac{1}{\kappa_y(m)/\Delta t+\sigma_y(m)/2\varepsilon_0} \tag{13}$$

$$C1(m)=\frac{\kappa_z(m)/\Delta t-\sigma_z(m)/2\varepsilon_0}{\kappa_z(m)/\Delta t+\sigma_z(m)/2\varepsilon_0} \tag{14}$$

$$C2(m)=\frac{\kappa_x(m)/\Delta t+\sigma_x(m)/2\varepsilon_0}{\varepsilon_1\kappa_z(m)/\Delta t+\varepsilon_1\sigma_z(m)/2\varepsilon_0} \tag{15}$$

$$C3(m)=\frac{\kappa_x(m)/\Delta t-\sigma_x(m)/2\varepsilon_0}{\varepsilon_1\kappa_z(m)/\Delta t+\varepsilon_1\sigma_z(m)/2\varepsilon_0} \tag{16}$$

In those equations, $m = (i + 1/2, j, k)$, $\varepsilon, \sigma, \kappa$ are the parameters of UPML. Other fields can be obtained in the similar way [2].

The FDTD algorithm is a process of alternate *E* field updating and *H* field updating to solve the Maxwell's equations. At time step *n*, it performs *E* field update equations for each cell. Then it performs *E* field update equations for each cell at time step *n* + 1/2.

The computational domain is a three-dimensional grid. Each cell in the grid has its own material type, which determines the dielectric properties. After initialization, the basic serial FDTD algorithm used is shown in Figure 2.

```
for(n = 0; n < time_steps; n++){
 for(k = 0; k < z_dim; k++)
     for(j = 0; j < y_dim; j++)
    for(i = 0; i < x_dim; i++){
       /* E field updates */
       update $e_x$[i][j][k];
       update $e_y$[i][j][k];
       update $e_z$[i][j][k];
       /* H field updates */
       update $h_x$[i][j][k] ;
       update $h_y$[i][j][k];
       update $h_z$[i][j][k];
    }
}
```

Fig. 2. High-level overview of basic serial FDTD algorithm

From the discussion above, we can learn that the memory requirement is determined by the size of the problem space, and the computing time is determined by the size of computational domain and total time steps.

3. FDTD Computation Based on GPU

In this section, we will discuss the details of implementing the FDTD algorithm on GPU.

3.1 Parallel Updating by Means of Thread

The bottleneck of FDTD is that the update operations are performed in tight loops. Since the value of E_z (or H_z) of a cell in FDTD is only related to the magnetic field (or electric field) of the closest neighboring cells as well as its previous value, we need only the values of H_x and H_y (E_x and E_y) surrounding the cell and the value of E_z (H_z) at the previous time step in calculation. Thus, if we divide the task into E update and H update, each of them can be executed in parallel on GPUs.

Take update of E for example. In our scheme, we define a three-dimensional computing domain: $n_x \times n_x \times n_z$. Let variables n_x, n_y, n_z denote the number of Yee cells in direction x, y, z, respectively. The data storage is continuous along z direction. Unlike the case in CPU execution where $E_{0,0,0}$ is calculated before $E_{0,0,1}$, the execution of GPU kernel [14] creates independent threads that compute and write at every location in E. The hardware takes the place of the nested for-loop.

3.2 Vectorization

One feature of the ATI stream processors is that each thread processor is capable of performing parallel operations. In addition, the Brook+ can support vector data types of up to four elements, such as float4, to match the hardware architecture [14]. Using vector data types is of two advantages. First, because every memory fetch instruction takes at least one cycle, the vector fetches can make more efficient use of the fetch resource. Second, as the inputs and outputs are vector data types, the domain size of the execution could decrease and fewer threads are executed by the stream processor. Thus, to maximize performance of our scheme, we can vectorize fetches and threads, making efficient use of those architecture features.

As in our scheme, the inputs and outputs are set as float4 instead of float. Therefore, the GPU kernel can issue a fetch for float4 type in one cycle versus four separate float fetches. In addition, the kernel combines four threads into a single thread and writes out four results with the vector data type float4. And then the domain size of the execution decreases to $n_x \times n_y \times (n_z/4)$.

Furthermore, since the maximum output of the threads can be a vector of eight float4, we can put more work into a thread and combine the output of individual computation by vectorization [14]. In our scheme, the E field variables, such as E_x, E_y and E_z are all computed in the same kernel (and therefore thread) simultaneously, which can reduce the number of memory fetches and stream core operations.

Figure 3 shows how update of E is assigned to GPU threads. The update of H is assigned in the same way.

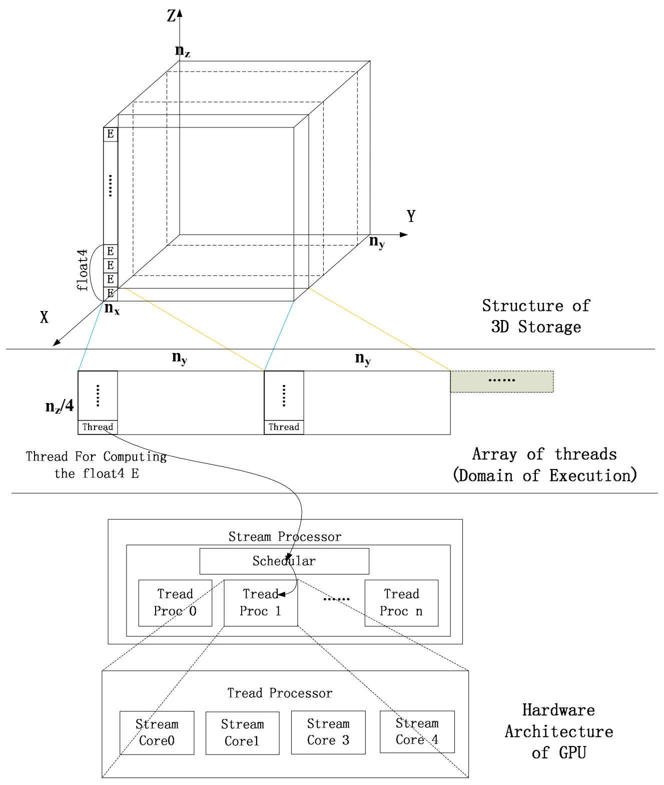


Fig. 3. GPU threads performing parallel FDTD

In our scheme of FDTD computation, along the direction *z*, to calculate the value of $E_{i,j,k}$, we need the difference between two adjacent elements, $H_{i,j,k}$ and $H_{i,j,k-1}$. However, using float4 type, if H_1 minus H_2, it means that the elements in the H_1 subtract the corresponding elements in H_2. To address the issue, we establish another float4 data structure to store float4 H_{temp} shown in figure 4. Each time when it is needed to compute the value of *E*, we use float4 *H* minus float4 H_{temp} to accomplish adjacent element subtraction.

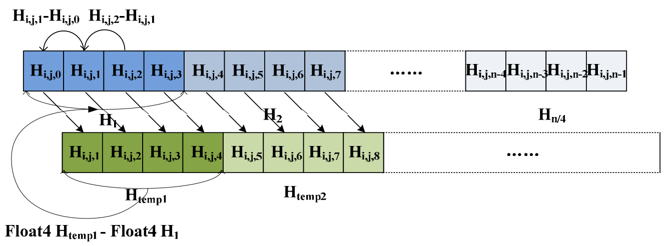

$H_{i,j,1}-H_{i,j,0}$ $H_{i,j,2}-H_{i,j,1}$
$H_{i,j,0}$ $H_{i,j,1}$ $H_{i,j,2}$ $H_{i,j,3}$ $H_{i,j,4}$ $H_{i,j,5}$ $H_{i,j,6}$ $H_{i,j,7}$ …… $H_{i,j,n-4}$ $H_{i,j,n-3}$ $H_{i,j,n-2}$ $H_{i,j,n-1}$
H_1
H_2
$H_{n/4}$
$H_{i,j,1}$ $H_{i,j,2}$ $H_{i,j,3}$ $H_{i,j,4}$ $H_{i,j,5}$ $H_{i,j,6}$ $H_{i,j,7}$ $H_{i,j,8}$ ……
H_{temp1}
H_{temp2}
Float4 H_{temp1} - Float4 H_1

Fig. 4. Adjacent element subtraction along direction *z*

Through the transmission function, the problem space can be read into GPU and then iterative computation of FDTD algorithm with UPML will be performed in time step. The high-level overview of the GPU code is shown in Figure 5. *B* represents the magnetic flux, and *F* and *G* are the parameters of UPML.

```
I. Main Processing
1. Use StreamRead() to transfer data from CPU to GPU;
2. for(timestep=0; timestep<nstep; timestep++){
3.   ecomp_gpu_kernel();
4.   hcomp_gpu_kernel();
5. }
6. Use StreamWrite() to transfer data from GPU to CPU;

II ecomp_gpu_kernel()
1. float2 index = indexof();//Get (x, y) in array of
   threads
2. if(need the difference of E along with z
   direction){
3.   Implement $E_{temp}$ scheme;
4.   Compute float4 F[index], float4 G[index], float4
   H[index];
5. }
6. else{
7.   Compute float4 F[index], float4 G[index], float4
   H[index];
8. }
9. Add excitation;

III.hcomp_gpu_kernel()
1. float2 index = indexof();//Get (x, y) in array of
   threads
2. if(need the difference of H along with z
   direction){
3.   Implement $H_{temp}$ scheme;
4.   Compute float4 B[index], float4 H[index];
5. }
6. else{
7.   Compute float4 B[index], float4 H[index];
8. }
```

Fig. 5. High-level overview of GPU codes

4 Performances Evaluating

To prove that our scheme has practical value with UPML, and can achieve better performance and perfect accuracy compared with CPU based scheme, we present some numerical experiments below.

4.1 Analysis of Performance on GPU-Based FDTD

Unlike the CPU computation, the GPU-based FDTD scheme must spend time for data-transfer through the bus. It will influence the total execution time. Reference 15 shows that steady state performance is achieved when the initialization cost is amortized over many iterations (>1,000) and approaches zero per iteration. In our paper, this assertion proves to be valid for the ATI HD4850 GPU. Thus, we fixed the time iteration steps as 2000.

We use the term Mcell/s 16 to measure the computing performance of GPU. It describes simulation speed as follow.

$$\text{Speed}\left[\text{Mcells/s}\right] = \frac{I \times J \times K \times n_{it}}{10^6 \times T_s} \tag{17}$$

In this equation, $I \times J \times K$ denotes the problem size, n_{it} describes the iteration steps which has been set to 2000, and T_s represents the simulation time.

In order to analyze the new scheme, we choose a simple and typical example of FDTD: electromagnetic wave radiation of line excitation source. The computing domain is (100×100×100). The spatial step is 0.05m. The time step is determined by the following formula.

$$\Delta t = 0.9 \times \frac{1}{c\sqrt{\left(1/\Delta x\right)^2 + \left(1/\Delta y\right)^2 + \left(1/\Delta z\right)^2}} \tag{18}$$

The waveform of excitation is set as below:

$$e(t) = A_0 \exp\left[-\left(\frac{t-3T}{T}\right)^2\right]\sin\left[\omega_0 t\right] \tag{19}$$

where $A_0 = 2000$, $T = 2e^{-9}s$, $\omega_0 = 2e^9$.

As shown in Figure 6, the GPU-based FDTD has a good UPLM, which absorbs electromagnetic wave well as that no electromagnetic wave reflects backwards.

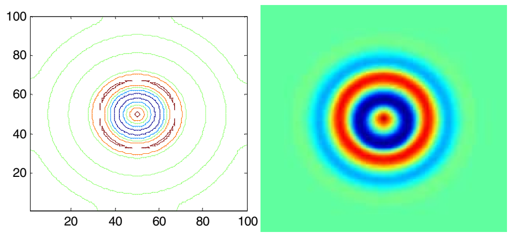


Fig. 6. Simulation results of GPU-based FDTD (n_z=50, *x-y* plane)

To analyze the precision of the new scheme on GPU, we assume that the result of CPU is accurate, and calculate the absolute error and relative error on the surface in Figure 6. The absolute error is around 10^{-4} and the relative error is lower than $4‰$. The difference is caused by the compiler and internal function of the GPU architecture.

Thus, for FDTD computing, the result of GPU is acceptable. However, if the time step and problem size grow up, the error will accumulate. Therefore, when applying the scheme in a practical project, researchers need to consider the error range for the validity of the result.

Figure 7 shows the performance comparison between GPU-based and CPU-based FDTD. We can see that, when the problem size is small, the GPU-based scheme does not show the benefit of parallel computing and the performance is even worse than that of the CPU-based one because the models are not big enough to saturate the GPUs fragment processors [15]. However, when the problem size increases and all the data fit into the GPU memory, the performance of GPU-based scheme raises sharply while the performance of the CPU-based scheme remains the same. Summary of achieved speed-up compared to CPU-based approach is presented in table 2. The highest speed-up reaches to 93.

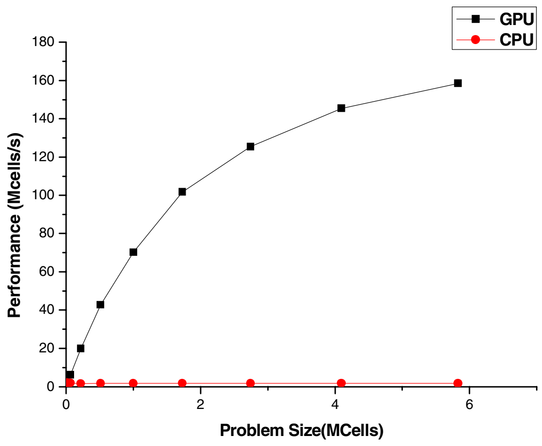


Fig. 7. Performance comparison between massively parallel GPU FDTD and serial CPU FDTD for different problem size

Table 2. Computing speed-up comparing to the CPU

Problem Size	Speed-Up
20×20×20	0.4
40×40×40	3
60×60×60	12
80×80×80	25
100×100×100	41
120×120×120	56
140×140×140	76
160×160×160	82
180×180×180	93

In addition, as the performance curve for our GPU-based scheme seems to follow a certain law, we try to utilize the curve-fitting technique to obtain the formula to describe the relationship between scheme performance y and problem size x. The initial fitting function is set as:

$$y = a\log_{10}(x-b)+c \tag{20}$$

The Levenberg-Marguardt algorithm [17] is then applied to obtain the value of a, b and c, which are listed as follows:

$$a = 144,\ b = -4.39e5,\ c = -815 \tag{21}$$

Figure 8 shows the final fitted curve as well as data samples from experiments with different problem size. Thus, we can easily estimate the performance value y by any given input problem size x.

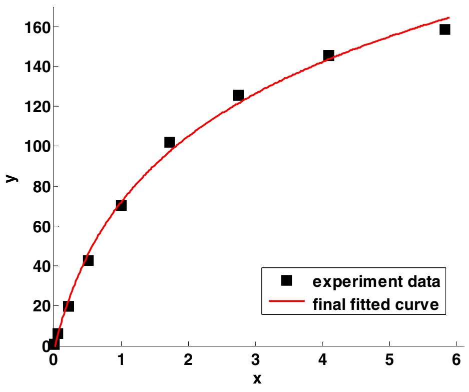


Fig. 8. Final fitted curve of scheme performance

4.2 Numerical Example Computing with GPU

In this section, we will present a case study of electromagnetic radiation of a computer box, which applies GPU to accelerate FDTD computing, to prove the validity of our scheme. This sample shows the electromagnetic radiation of a personal computer and the distribution of electromagnetic field in the computer.

The size of computer is shown in Figure 9. The spatial step is 0.004m. The waveform of excitation is set as:

$$e(t) = A_0 \exp\left[-\left(\frac{t-3T}{T}\right)^2\right]\sin[\omega_0 t] \tag{22}$$

where $A_0 = 3773.6$, $T = 2e^{-10}s$, $\omega_0 = 3e^8$.

In this case study, the computing domain is 100×100×100. The running speed of GPU-based scheme is about 45 times faster than that of the CPU-based. Figure 10 shows the computing results. The results show the distribution of electromagnetic field at the 600th time step. From these figures, we can see the distribution of electromagnetic

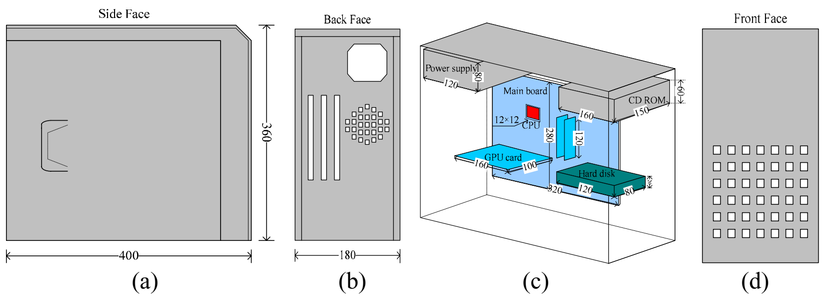


Fig. 9. The three-dimensional plot of computer box and its size. (a) the side face of box. (b) the back face of box. (c) the inner structure of computer. (d) the front face of box covering with plastic.

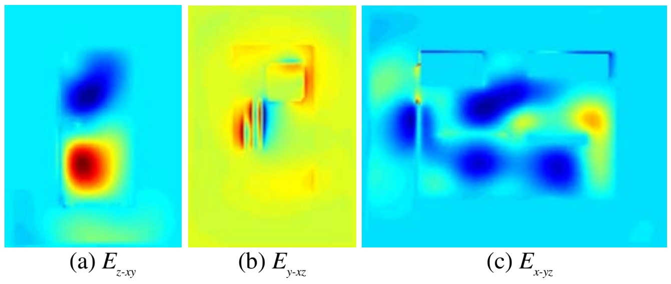


Fig. 10. The distribution of electromagnetic field at 600th time step with GPU computing. (a) the surface of n_z=80. (b) the surface of n_y=30. (c) the surface of n_x=50.

field and the interaction with computer structure clearly. Then we can analyze the electromagnetic compatibility of a computer box and develop electromagnetic protection schemes for the computer box.

5 Conclusions

This paper presents a practical implementation of FDTD with UPML using GPUs. We present a highly optimized scheme for GPU that achieves approximately 93 times speed-up in comparison with the Intel E2180 dual cores CPU. In addition, the scheme proves to be valid and works perfectly in a real project of simulating electromagnetic radiation of the computer box.

Starting from the scheme that we have described above, there are obviously a number of possible directions for future research. One is to utilize CPU and the system memory to assist GPU computing in order to obtain a higher performance. Another approach is to use extra GPUs to set up a GPU-cluster, which may have a much better price/performance ratio.

Acknowledgement

This work was supported by the competition committee of AMD China University Accelerated Computing Application Contest, the National Hi-tech R&D Program of

China (863 program) under grant No.2009AA01Z107, the National Science Foundation of China (NSFC) under grant No.60873053, and the International S&T Cooperation Program of China under grant No.010S2010GR0607.

References

1. Yee, K.S.: Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. IEEE Transaction on Antennas and Propagation AP-14(3), 302–307 (1966)
2. Taflove, A., Hagness, S.C.: Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd edn. Artech House, Norwood (2005)
3. Yu, W., Mittra, R., Su, T., Liu, Y., yang, X.: Parallel finite-difference time-domain method. Artech House, Norwood (2006)
4. Krakiwsky, S.E.: Acceleration of Finite-Difference Time-Domain Electromagnetic Simulations Using Graphics Processor Units, Ph.D. dissertation, Dept. Elect. Comp. Eng. University of Calgary, Alberta, CA (2004)
5. Inman, M.J., Elsherbeni, A.Z., Maloney, J.G., Baker, B.N.: GPU based FDTD solver with CPML boundaries. In: 2007 IEEE Antennas and Propagation Int. Symp., June 9-15, pp. 5255–5258 (2007)
6. Price, D.K., Humphrey, J.R., Kelmelis, E.J.: GPU-based accelerated 2D and 3D FDTD solvers. In: Osinski, M., Henneberger, F., Arakawa, Y. (eds.) Proceedings of the SPIE, Presented at Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, Physics and Simulation of Optoelectronic Devices XV, February 2007, vol. 6468, p. 646806 (2007)
7. Poman, S.: Time-Domain Computational Electromagnetics Algorithms for GPU Based Computers. In: EUROCON, Warsaw, Poland, September 2007, pp. 1–4 (2007)
8. Baron, G.S., Fiume, E., Sarris, C.D.: Graphics hardware accelerated multiresolution time-domain technique: development, evaluation and applications. IET Microwaves, Antennas & Propagation 2(3), 288–301 (2008)
9. Valcarce, A., De La Roche, G., Zhang, J.: A GPU approach to FDTD for Radio Coverage Prediction. In: IEEE 11th International Conference on Communication Systems, Guangzhou, China (November 2008)
10. Balevic, A., Rockstroh, L., Tausendfreund, A., Patzelt, S., Goch, G., Simon, S.: Acceleration Simulations of Light Scattering based on Finite-Difference Time-Domain Method with General Purposed GPUs. In: CSE 2008, 11th IEEE International Conference on Computational Science and Engineering, Sao Paulo, July 2008, pp. 327–334 (2008)
11. Pharr, M. (ed.): GPU Gems 2. Addison Wesley, Upper Saddle River (2005)
12. Sypek, P., Dziekonski, A., Mrozowski, M.: How to Render FDTD Computations More Effective Using a Graphics Accelerator. IEEE Transactions on Magnetics 45(3) (March 2009)
13. AMD Corp, `http://www.amd.com`
14. AMD Corp. ATI Stream Computing User Guide
15. Adams, S., Payne, J., Boppana, R.: Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors. In: HPCMP Users Group Conference 2007 (2007)
16. Acceleware, `http://www.acceleware.com`
17. Levenberg, K.: A method for the solution of certain non-linear problems in least-squares. Quart. Appl. Math. 2, 164–168 (1944)

A Proposed Asynchronous Object Load Balancing Method for Parallel 3D Image Reconstruction Applications

Jose Antonio Alvarez-Bermejo and Javier Roca-Piera[⋆]

Dept of Computer Architecture and Electronics
Universidad de Almería
Ctra. Sacramento S/N. 04120
Spain

Abstract. Scientific applications usually exhibit irregular patterns of execution and high resource usage. Parallel architectures are a feasible solution to face these drawbacks, but porting software to parallel platforms means the addition of an extra layer of complexity to scientific software. Abstractions such as Object Orientation and models like the concurrent object model may be of great help to develop scientific parallel applications. The shared nature of parallel architectures and the stochastic condition of parallel schedulers underline the adaptivity as a desired feature for parallel applications. Load Balancers are key for achieving adaptivity, and benefit from object oriented models in issues like load migration. In this paper we present our experiences when porting scientific software using the concurrent object abstraction and a method to asynchronously invoke load balancers.

1 Introduction

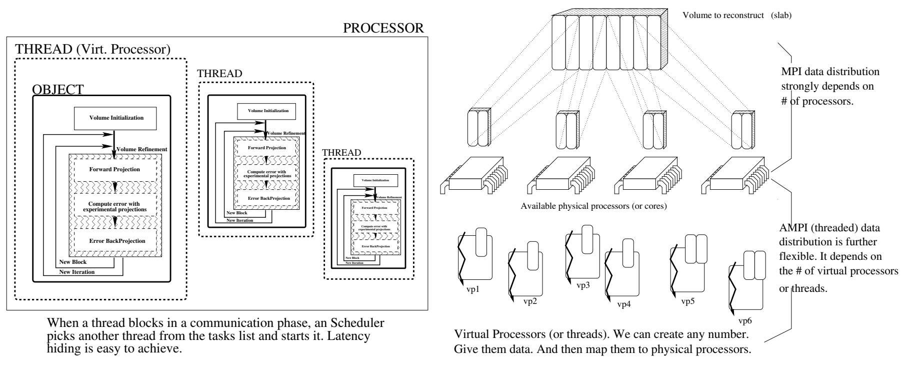
A high percentage of the scientific code is CPU-stressing, iterative and irregular and parallelism is almost the unique path for an acceptable solution. Much effort and programming abilities are needed to parallelize efficiently [1]. The application's irregular execution patterns make it difficult to assure optimal runs on parallel machines, therefore adaptivity is a desired issue [2]. It is a fact that prior to writing parallel code there is so much time spent in optimizing the algorithm and very little attention is paid to data structures. Our main aim was trying to program efficiently, shifting the programming paradigm and minimizing the gap from algorithm to implementation using more expressive paradigms [3]. We shifted to a particular object oriented paradigm [4] and used the charm framework [5]. This paper gathers the idea of exploiting concurrence and parallelism efficiently using the object orientation pattern. On a second stage, this paper shows load balancing techniques used to achieve adaptivity and finally presents a method to avoid interferences from the load balancer at fixed intervals, which

[⋆] This work has been funded by grant TIN 2008-01117 from the Spanish Ministry of Science and Innovation.

may diminish the performance. This paper is organized as follows, section 2 introduces the iterative reconstruction method. Section 3 shows improvements achieved when porting our code to the new paradigm. Section 4 exposes how we reached adaptivity and the technique to avoid fixed-interval interferences from the load balancer, and in Section 5 we summarize the conclusions.

2 The Iterative Reconstruction Problem

Series expansion reconstruction methods assume that a 3D object, or function f, can be approximated by a linear combination of a finite set of known and fixed basis functions, with density x_j. The aim is to estimate the unknowns, x_j. These methods are based on an image formation model where the measurements depend linearly on the object in such a way that $y_i = \sum_{j=1}^{J} l_{i,j} \cdot x_j$, where y_i denotes the i^{th} measurement of f and $l_{i,j}$ the value of the i^{th} projection of the j^{th} basis function. Under those assumptions, the image reconstruction problem can be modeled as the inverse problem of estimating the x_j's from the y_i's by solving the system of linear equations aforementioned. Assuming that the whole set of equations in the linear system may be subdivided into B blocks, a generalized version of component averaging methods, *BICAV* [1] can be described. The processing of all the equations in one of the blocks produces a new estimate, see Figure 1(a). All blocks are processed in one iteration of the algorithm. These techniques produce iterations which converge to a weighted least squares solution of the system. A volume can be considered made up of 2D slices. The use of the spherically symmetric volume elements (blobs) [1], makes slices interdependent because of blob's overlapping nature. The amount of communications is proportional to the number of blocks and iterations (as sketched in Figure 1(a)). Reconstruction yields better results as the number of blocks is increased. The main drawback of iterative methods is their high computational requirements. These demands can be faced by means of parallel computing and


PROCESSOR
THREAD (Virt. Processor)
OBJECT
Volume Initialization
Volume Refinement
Forward Projection
Compute error with experimental projections
Error BackProjection
New Block
New Iteration
THREAD
THREAD
When a thread blocks in a communication phase, an Scheduler picks another thread from the tasks list and starts it. Latency hiding is easy to achieve.
Volume to reconstruct (slab)
MPI data distribution strongly depends on # of processors.
Available physical processors (or cores)
AMPI (threaded) data distribution is further flexible. It depends on the # of virtual processors or threads.
vp1
vp2
vp3
vp4
vp5
vp6
Virtual Processors (or threads). We can create any number. Give them data. And then map them to physical processors.

(a) Virtual MPI processes (b) Volume decomposition

Fig. 1. Coarse grain approach

efficient reconstruction methods with fast convergence. The parallel iterative reconstruction method has been implemented following the Single Program Multiple Data (SPMD) approach.

3 Shifting the Programming Paradigm

Clusters are processing platforms where highly demanding resources problems like *BICAV* can be efficiently solved. But multicores are appearing on the scene, they may use the same memory address space for their cores, inviting to think in a threaded programming model but the problem of how to exploit data caches and how to adapt parallel HPC applications is still an issue. Doubtlessly new abstractions are needed in order to maintain performance gains in HPC. The Object Oriented programming paradigm offers a flexible method to describe parallel computations. Rewriting legacy scientific applications is not advisable, but a rapid way to exploit this kind of new parallelism is by virtualizing processes [5] (see Figure 1(a)).

The Coarse Grain Approach

BICAV was reimplemented [6] using AMPI [5], which is a *framework* that allows the virtualization of MPI processes. Concurrence is then a consequence of having more virtual processors than physical processors. In our application the data is distributed as depicted in Figure 1(b). Communications between neighbor nodes to compute the forward-projection or the error back-projection for a given slab and to keep redundant slices updated are mandatory. The communication rate increases with the number of blocks and iterations. There will be almost no penalty for those virtual processors containing non boundaries slices because the communication will be carried out within the node. Table 1 underlines the gains in the coarse−grain implementation versus MPI. The test reconstructed two volumes, a 256x256x256 volume and a 512x512x512 volume, the number of blocks (K) was set to the maximum (K=256 and K=512, respectively). The efficiency was defined, for these tests, in terms of the relative idle time computed per processor. Table 1 presents the relative difference (columns *Idle%*) among

Table 1. % Relative differences between CPU and WALL times

	K 256 (volume 256)		K 512 (volume 512)	
	MPI	AMPI	MPI	AMPI
Procs	Idle%	Idle%	Idle%	Idle%
2	2.9	0.1	2.8	0.0
4	3.5	0	3.2	0.0
8	5.6	0	4.7	0.3
16	17.1	0.8	9.7	0.7
32	62.4	1.5	32.3	0.2

cputime and walltime for both problem sizes. For the new version, the computed walltime and cputime are almost the same, so cpu was not idling. MPI version behaved worst as the number of processors grew up. Our version (used 128 virtual processors) seized concurrence at maximum. Experiences were performed on a cluster with (*32* computing nodes with two Pentium IV xeon 3.06 Ghz with 512KB L2 Cache and a 2GB sdram).

A Finer Grain Approach

Objects are good for new architectures: they protect their internal data and protect their internal state and do not share it. Objects properties define a scenario based in local and separated environments so objects can be executed in parallel. Objects are structural units and concurrent units. *Parallel Objects* (PO)[7] is an object model where parallelism as well as non determinism can be expressed easily. With this finer-grain implementation, intra-object and inter-object parallelism were exploited. The former with concurrent methods provided by the language, the latter was achieved implementing a High Level Parallel Composition (where internal schedulers are provided together with concurrence control mechanisms) following the Farm Pattern [8], where a group of concurrent objects work in parallel (see Figure 2) under the guidance of a master object (see figure 2). To control the concurrence we used MAXPAR [4]. The implementation used a concurrent object framework [5], built atop a runtime that *abstracts the beneath architecture.* This runtime offers scheduling capabilities avoiding local non−determinism as the scheduler is part of the runtime itself.

The Figure 2 shows a *main object* that controls how the program advances. The node objects are placed one per processor, they serve as middleman between slaves and master. Slaves simply do their task concurrently, independently from

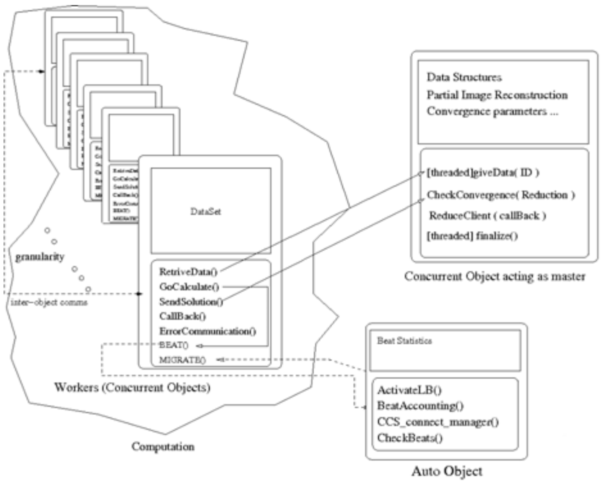


Fig. 2. Object oriented version : *auto* object and visualization

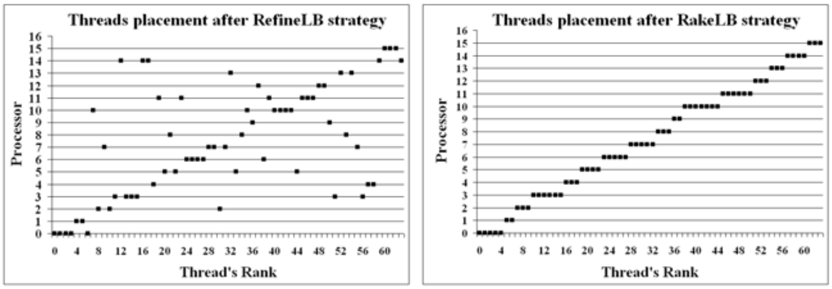
Table 2. Walltime computed in a cluster and in a multicore processor

	Cluster		Multicore	
Obj−Prc	MPI WallTime	Objects WallTime	MPI WallTime	Objects WallTime
2	366	42	111,449	4,1
4	86	26	75,651	2,891
8	46	16	45,496	2,697
16	26	14	34,136	2,4
32	n/p	n/p	33,25	2,121

the processor they are located in. A mechanism for providing *automatic adaptivity* was designed (see Section 4.1) using a special automated object (see *auto* object in Figure 2) that drive the load balancing (see Section 4). The table 2 shows the behaviour of the reconstruction in MPI and in Charm++ on a cluster and multicore system. In multicore, the column titled *Obj−Prc* refers to the number of MPI processes (MPI version) and objects created in the Object Oriented version. When running on the cluster, this column means physical processors, in this platform four worker objects per processor were always used. The reconstruction in the O.O. version is message-driven, where a message activates an object's service. As Table 2 shows, the object-oriented version behaves better than its MPI counterpart. Using a cluster, the main harm that this scenario suffers from is the network latencies. Communicating two MPI processes is costly. Our new version has better granularity and concurrence, this helps for hiding latencies. Nevertheless when running the application in the multicore (Intel Core 2 Quad Q6600), the MPI version reaches a point (8 processes) where the cache contention affects dramatically the performance. Also one may note that MPI is based on passing messages and although the network remains untouched, when sharing memory this message passing is translated into a copy from private to shared memory where conflicts also exist.

4 Load Balancing a Mean for Adaptive Applications

Developing a parallelizable application means adding an extra layer of complexity to the software development process. This complexity refers not only to determine when a certain operation will be processed but where. Making an application adaptive is a two steps procedure, the first one is the heuristic, the second step has to do with how to migrate the computation. Load migration is a scheme reached by consensus for facing imbalance. Conventional strategies indicate the amount of computation units to be moved but say nothing about which of them should be moved to preserve locality and performance. An strategy to preserve locality (RakeLB [9]) was implemented as a centralized load balancer [6], its behaviour was compared with standard centralized strategies like Refine and Greedy [10] where Greedy strategy does not consider any previous



(a) Disorder after RefineLB (b) Locality preservation after RakeLB

Fig. 3. Background load and resulting data redistribution after load balancing

thread-processor assignment, simply builds two queues, processors and threads, and reassigns load to reach average. Refine, in contrast, only migrates objects from overloaded processors until the processor load is pushed under average. Preserving data locality and minimizing latencies (see Section 3) are two issues exploited by RakeLB. We tested $BICAV$ to evaluate the dynamic load balancing algorithms. After migration, RakeLB and RefineLB reacted alike, with a fine load distribution as shown by the σ value. The initial imbalance value (workload placed statically in each node) reflected by σ (standard deviation of the whole system load, normalized to the average load) was over 0.5. After applying the load balancer a similar σ value was achieved (0.051 for GreedyLB and 0.045 for both RakeLB and RefineLB). GreedyLB reached a good load balancing but the distribution of the objects was remarkably messy (see Figure 3) and negative for performance. Figure 3, shows an ordered distribution for RakeLB, aspect that turns out to be an issue for $BICAV$'s performance, as can be seen in Table 3 first row.

4.1 Asynchronous Load Balancer Calls

Objects give the chance of using load balancing policies to better adapt the application to the current computing scenario. Iterative applications must statically specify points in its timeline where the balancer must interfere, otherwise there is no way to collect performance data, evaluate it and propose migrations. So what about creating an object that acts like a coach? An object that is able to detect performance decrements and then invoke the load balancer? This method works as a *swimming team*, the proposed object controls the expected time lapse, as a coach does. If a *slow swimmer enters your lane* you will experience worst time lapses. In our case the *solution* is already implemented in the load balancing strategy, so the coach just need to invoke it. Probably the slow *swimmer* will not be removed away because it does not belong to our team (computing set) but my objects can be moved to more appropriated lanes for fast objects. In Figure 4 several cases are shown. Each lane can be considered as a node (cluster)

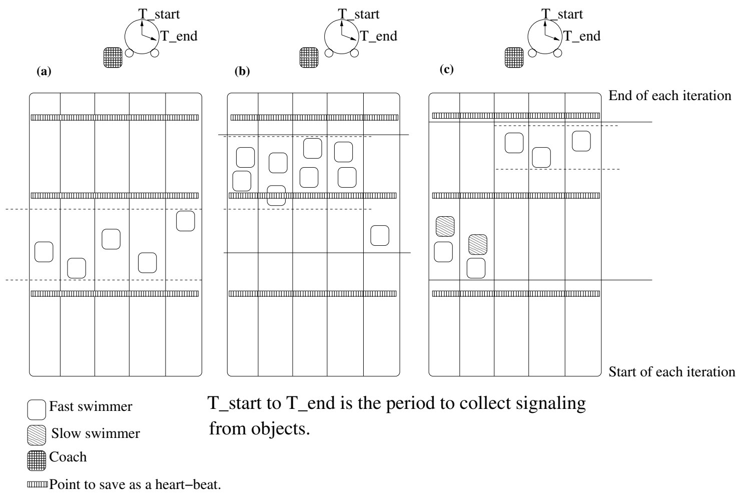

T_start
T_end
(a)
(b)
(c)
End of each iteration
Start of each iteration
Fast swimmer
Slow swimmer
Coach
Point to save as a heart–beat.
T_start to T_end is the period to collect signaling from objects.

Fig. 4. Swimmers based model

or a core (multicore), and even the three *pools* may be the case of a cluster of multicore nodes. In Figure 4, (a) represents a computation performing *normally*, no need for load balancing. Each time that an object passes through the stripped line, a *beat* is sent to the coach. (b) shows a case where an object is running on a slower node or core, or may be processing a dense data-set so it progresses slower, the coach should invoke the balancer. (c) identifies a case where our the computation is harmed by a third party, it might be useful to invoke the LB. If we were not using this method, in (a) the load balancer would have been invoked once per a number of iterations, unnecessarily. In (b) and (c) the situation might appear at some point in time, so there is no need to implement *periodic* checks. This method is more flexible than stopping at load balancing barriers. Results in section 4 were obtained with just one load balancer call, we will compare against that. Table 3 shows the application *walltimes* as percentages taking as a reference the walltime of the application using the Greedy strategy when the load balancer is invoked just once in the traditional way (Greedy 1 LB call). In a Load Balancer call, the whole computation is stopped into a barrier, the control is passed to the runtime that loads the heuristic, retrieves the object's traces and decide if the mapping should stay as it is or it should migrate. It is a good idea

Table 3. Walltime ratio for *BICAV*, pool method

	Greedy	Refine	Rake
Master's Walltime 1 LB call	100	66.59	50.29
Master's Walltime 1 LB call per 50 it.	101.5	67.4	50.36
Master's Walltime 1 LB call per 20 it.	101.62	67.59	50.41
Master's Walltime Asynchronous LB call	100.8	66.63	50.3

```
Data: $trigger \longleftarrow NumberOfBeatsToCollect$
Result: Invoke load balancer if the beat slows down
if $(--trigger==0)$ then
    lapse ← gettime();
    if $(CollectFirstTime)$ then
        auto.getFasterTime(lapse);
    if $lapse-idealLapse > \epsilon$ then
        objects[].signalObjectsLBCallRecommended();
    $trigger \leftarrow NumberOfBeatsToCollect$ ;
```

Algorithm 1. Algorithm for the method beat(), *auto* object

not to abuse from LB calls, our method places an object (*auto object*) per core / node. Computing objects send a simple lowpriority message each time that the object passes through the check points in the code. The *auto object* collects a predefined number of beats, computes the time to collect them and it the time intervals are worst each time, then the LB is invoked. Algorithm 1 describes the basic workings of the *auto-object*.

5 Conclusions

Object oriented abstractions can efficiently exploit parallelism. As a consequence latency hiding and adaptivity issues are easier to achieve. The adaptivity is usually affected by the application's irregular and iterative nature as well as by the heterogeneity of the parallel platform. Load balancing strategies must be carefully devised to be locality aware, we have shown that this benefits the walltime of the application. Load balancers must at some point in time, interfere the application to collect data and re-balance the problem, this means loosing flexibility by having to specify statically the points in time when the load balancer should enter in action. A flexible method to invoke load balancers was presented and shown to be effective.

References

1. Fernandez, J.J., Lawrence, A.F., Roca, J., Garcia, I., Ellisman, M.H., Carazo, J.M.: High-performance electron tomography of complex biological specimens. Journal of Structural Biology 138(1-2), 6–20 (2002)
2. Qin, X., Jiang, H., Manzanares, A., Ruan, X., Yin, S.: Communication-aware load balancing for parallel applications on clusters. IEEE Transactions on Computers 59(1), 42–52 (2010)
3. Álvarez, J.A., Roca-Piera, J., Fernández, J.J.: From structured to object oriented programming in parallel algorithms for 3d image reconstruction. In: POOSC 2009: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing, pp. 1–8. ACM, New York (2009)
4. Corradi, A., Leonardi, L.: Concurrency within objects: layered approach. Inf. Softw. Technol. 33(6), 403–412 (1991)

5. Kalé, L.V., Krishnan, S.: Charm++: A portable concurrent object oriented system based on c++. In: OOPSLA, pp. 91–108 (1993)
6. Alvarez, J.A., Roca, J., Fernández, J.J.: A load balancing framework in multithreaded tomographic reconstruction. In: Proceedings of the International Conference ParCo 2007, Aachen-Julich (September 2007) (in press)
7. Corradi, A., Leonardi, L.: Po: an object model to express parallelism. SIGPLAN Notices 24(4), 152–155 (1989)
8. López, M.R., Tunón, M.I.C.: An approach to structured parallel programming based on a composition. In: CONIELECOMP, vol. 42. IEEE Computer Society, Los Alamitos (2006)
9. Fonlupt, C., Marquet, P., Dekeyser, J.l.: Data-parallel load balancing strategies. Parallel Computing 24, 1665–1684 (1996)
10. Aggarwal, G., Motwani, R., Zhu, A.: The load rebalancing problem. J. Algorithms 60(1), 42–59 (2006)

A Step-by-Step Extending Parallelism Approach for Enumeration of Combinatorial Objects

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. We present a general step-by-step extending approach to parallel execution of enumeration of combinatorial objects (ECO). The methodology extends a famous enumeration algorithm, *OrderlyGeneration*, which allows concurrently generating all objects of size $n + 1$ from all objects of size n. To the best of our knowledge, this is the first time there is an attempt to plug parallel computing into *OrderlyGeneration* algorithm for ECO problem. The potential impact of this general approach could be applied for many different servants of ECO problem on scientific computing areas in the future. Our work has applied this strategy to enumerate Orthogonal Array (OA) of strength t, a typical kind of combinatorial objects by using a implementation with MPI paradigm. Several initial results in relation to speedup time of the implementation have been analyzed and given significant efficiency of the proposed approach.

1 Introduction

Enumeration of combinatorial objects (ECO) remains an important role in combinatorial algorithms. Many scientific applications have been using results from typical servants of enumeration of combinatorial objects, such as maximal clique enumeration (MCE), hexagonal system enumeration and enumeration of orthogonal arrays. For example, the solutions of MCE related problems are used to align 3-dimensional protein structures [CC05] and to find clusters of orthologous genes [PSK+07]. Hexagonal systems play an important topic in computational and theoretical chemistry [BCH03] whilst orthogonal arrays could be applied in Design of Experiment [LYPP03] and software testing [LM08].

Usually, we are interested in enumerating or producing precisely one representative from each isomorphism class. In many cases, the only available methods for enumeration base on the exhaustive generation for counting the objects. Several general serial algorithms have been proposed for enumeration of combinatorial objects ([McK98], [Far78], [Rea79], [AF93]) and we call them *isomorph-free exhaustive generation* algorithms.

One of the major characteristics of ECO is the huge effort needed to complete the computation concerned. Therefore, a parallelism method for solving ECO could allow us to reduce the execution time significantly and generate new results using the power of high performance computing system.

Strengthened by the above objective, the general parallelism approach presented in this paper efficiently decomposes the computation-intensive nature of ECO problem. All objects of a new size will be concurrently enumerated from all generated objects of the previous size, not from scratch. This saves a lot of cost for regenerating old objects of previous size, especially when the target size S is huge. Moreover, the trivial data parallelism strategy applied for each extending step could gives an efficient speedup and the scalability.

The rest of this paper is organized as follows: Section 2 presents an overview about general algorithms for isomorph-free exhaustive generation. In Section 3, a step-by-step extending parallelism approach for ECO will be proposed. Since there are many different servants of ECO problem, we choose a specific servant of ECO and apply our general proposed approach to do some experiments. This case study is discussed in Section 4 in which we applied the proposed general approach for enumeration of orthogonal array of strength t, a specific kind of combinatorial objects. Detail of this implementation and some initial results will be given on Section 5. And finally, some conclusion will be discussed in Section 6.

2 Overview and Related Work

2.1 Serial Algorithms for Isomorph-Free Exhaustive Generation

The objective of isomorph-free exhaustive generation of combinatorial objects is to generate a representative for each of the isomorphism classes of those objects. For construction of objects, the most natural and widely used method is backtracking [Wal60]. On the other hand, most methods proposed for isomorphism rejection could be classified in two types. These are *OrderlyGeneration* method which has been proposed by Read [Rea79] and Faradev [Far78] and the canonical augmentation method which has been proposed by McKay [McK98]. The thorough discussion of these methods could be seen in [KO06] and [MS08]. Note that there are the so called "method of homomorphisms" (Laue & others [GLMB96]) but it uses a more algebraic approach, not the search-tree model.

The most common method is *OrderlyGeneration* which was independently introduced at the same time in 1978 by Read [Rea79] and Faradev [Far78]. Basically, it uses the idea that there is a canonical representative of every isomorphism class that is the object that needs to be generated. Usually, the canonical object is the isomorphic object that is extremal in its isomorphism class (largest lexicographically or smallest lexicographically). The algorithm will backtrack if a subobject is not canonical. The canonical labeling and the extensions of an object must be defined in order to ensure that each canonically labeled object is the extension of exactly one canonical object.

The second method is the canonical augmentation which has been proposed by McKay [McK98], where generation is done via a canonical construction path, instead of a canonical representation. In this method, objects of size k are generated from objects of size $k-1$, where only canonical augmentations are accepted.

Hence the canonicity testing is substituted by testing the augmentation from the smaller object is a canonical one. McKay's method is related to the *reverse search* method of Avis and Fukuda [AF93]. Both are based on the idea of defining a tree structure on a set of objects with a function for deciding parenthood for objects. However, they differ in that Avis and Fukuda's method is not concerned with eliminating isomorphs, but simply repeated objects.

It is noteworthy to note that those algorithms are isomorph exhaustive generation and based on a famous association rule mining algorithm, a priori [AS94], which generate all objects of size k from all objects of size $k-1$.

2.2 Two Issues of Concern

Before we propose our small-step parallelism approach, we will discuss further about the properties of the *OrderlyGeneration* algorithms.

Note that the *OrderlyGeneration* algorithm allows for generation from scratch when called with the root parameters [] and $n = 0$ and it will finish when reaching the target size S. Such characteristic results in two issues that need to be taken into account. First, 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. Second, in practice, sometimes when size S is huge, the computation cost at such one extending step (generating objects of size k from objects of size $k-1$ with $k \leq S$) is also very large (see Section 5). So in this case it could be worth applying parallel computing for solving the high calculation cost issue.

3 A Proposed Approach

The above two issues motivate us to find out a parallel method for decomposing the generation process into small separated computation steps and try to reuse old calculated results before generating for the new size. Fortunately, this can be done by using a special characteristic of *OrderlyGeneration* algorithm.

An important characteristic of the *OrderlyGeneration* algorithm is that a canonical object size k is guaranteed to be an extension of exactly one previously canonical object of size $k-1$. The outcome of this characteristic is all canonical objects of size k can be generated by extending all canonical objects of size $k-1$. In the point of data parallelism view, this is a very important characteristic that could be used to exploit the parallel computing on the generation. In particular, the *OrderlyGeneration* algorithm will generate canonical objects in an increasing way, in which it begins with object of size 0, generates step by step all canonical objects of size k from all canonical objects of size $k-1$. The generation is repeated until all target canonical objects of a target size, S, are reached. The search tree space of the *OrderlyGeneration* algorithm is given in Fig. 1

We propose a novel small-step parallelism approach for ECO based on the above important characteristic of *OrderlyGeneration* algorithm. The main idea

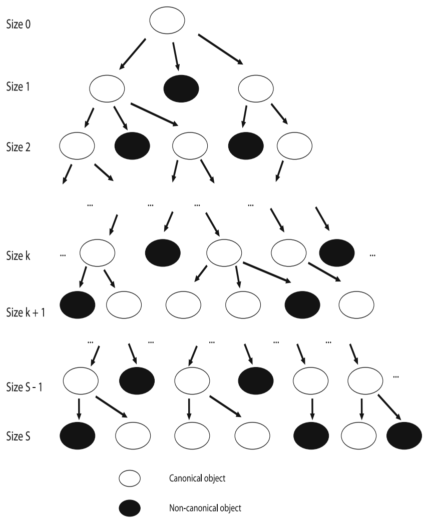


Fig. 1. Search tree space of the *OrderlyGeneration* algorithm

is that we divide the generation in separated small steps. At each step, we just extend concurrently all canonical objects of size k from all canonical objects of size $k - 1$. All canonical objects of size k will be stored and reused for the next step. In particular, we propose basically a general step-by-step parallelism approach for enumeration of combinatorial objects as follows:

1. At the initial stage, using the original serial algorithm *OrderlyGeneration* to generate all canonical objects of an initial size k_0, which will be chosen depending on the specific kind of combinatorial object.
2. The data parallelism strategy is applied to generate all canonical objects of size k from all canonical objects of size $k - 1$. All results will be stored and reused in the next step.
3. The step-by-step extending phases continue until the all the canonical objects of the target size S are reached.

The most important phases is the generation of objects of size k from all object of size $k - 1$, in which canonical objects of size k will be generated and stored concurrently. On section 5, we will discuss about some methods could be used for domain decomposition on this phase.

With the proposed approach, we have some advantages. Most importantly, with the reusing of canonical objects, all objects of each level will be generated precisely one time. All objects of a new size will be enumerated from all generated objects of the previous size, not from scratch. This save a lot of cost for regenerating old objects of previous size, especially when size S is huge. Moreover, the data parallelism strategy gives a efficient speedup in each extending step (see Section 5).

4 A Case Study: Enumeration of Orthogonal Array of Strength *t*

Since there are many different kinds of ECO problems, we choose a specific servant of ECO and apply our general proposed approach to do some experiments. On this paper, our work has applied the proposed approach for enumeration of Orthogonal Array (OA) of strength t, a special kind of combinatorial objects. It is noteworthy that OA has many applications in Design of Experiment [LYPP03] and software testing [LM08]. On this section we first present the notation of OA of strength t and then, the *MSC* algorithm, an OrdelyGeneration algorithm for enumerating orthogonal array, which is proposed by Eric, Pieter and Man [SEN10] will be presented. Finally, we will discuss about how our approach is applied for enumeration of orthogonal array of strength t.

4.1 Notation of Orthogonal Array

We present a clear definition of OA of strength t which could be found in [Ngu08].

We denote d finite sets $Q_1, Q_2, ..., Q_d$ as $factors$ where d is a finite number. The elements of a factor are called *levels*. The $(full)$ $factorial$ $design$ with respect to these factors is the Cartesian product $D = Q_1 \times \cdots \times Q_d$. A fractional design or $fraction$ F of D is a subset consisting of elements of D (possibly with multiplicities). We denote $r_i := |Q_i|$ as the number of levels of the i*th* factor. Let $s_1 > s_2 > \cdots > s_m$ be the distinct factor sizes of F, and suppose that F has exactly a_i factors with s_i levels. We call the partition

$r_1 \cdot r_2 \cdots r_d = s_1^{a_1} \cdot s_2^{a_2} \cdot \cdots \cdot_m^{a_m}$

the design type of F.

A subfraction of F is obtained by choosing a subset of the factors (columns), and removing the other factors. If a fraction is a multiple of a full design, i.e. it contains every possible row with the same multiplicity, we call it *trivial.* With a natural number t, a fraction F is called *t-balanced* if, for each choice of t factors, the corresponding subfraction is trivial. In other words, every possible combination of coordinate values from a set of t factors occurs equally often. A t-balanced fraction F is also called *an OA of strength t*. If F has N rows, we write $F = OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}; t)$.

For instance, the following array is an OA of strength 3 but not strength 4, $OA(16; 3 \cdot 2^3; 3)$:

$$F = \begin{bmatrix} 0\,0\,0\,0\,1\,1\,1\,1\,2\,2\,2\,2\,3\,3\,3\,3 \\ 0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\,0\,1\,1 \\ 0\,0\,1\,1\,0\,0\,1\,1\,1\,1\,0\,0\,0\,0\,1\,1 \\ 0\,1\,1\,0\,0\,0\,1\,1\,1\,0\,0\,1\,0\,1\,0\,1 \end{bmatrix}^T$$

We say that a triple of column vectors $X,\ Y,\ Z$ are *orthogonal* if each possible tuple (x, y, z) in $[X|Y|Z]$ appears with the same frequency. So an array has strength 3 if, and only if, every triple of columns in the array is orthogonal.

4.2 An OrdelyGeneration Algorithm for Enumerating Orthogonal Array

Recently, Eric, Pieter and Man ([SEN10]) have proposed an algorithm named *Minimum Complete Set (MSC)* for finding lexicographically-least orthogonal arrays. With that algorithm, several orthogonal arrays with distinct specific types have been generated and enumerated.

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 corespondent to a specific design type will be called *lexicographically minimum in column (LMC)matrix* and it is the only representative of this isomorphic class. Certainly, the concept of *LMC* matrix is equivalent with the concept of canonical object in the general *OrderlyGeneraion* algorithm. In other words, *LMC* matrix is a specific canonical object in the context of orthogonal array.

Finding lexicographically-least orthogonal array algorithm. The *MCS* backtracking algorithm has been used 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 reach the target column size S. The detail of this algorithm is so complicated on [SEN10]. Hence, we summarize the outline of the *MCS* algorithm as below:

```
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

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 reach the target column size S.

4.3 Applying Our Proposed Approach for Enumeration of Orthogonal Arrays

Since enumeration of orthogonal array is a typical kind of the general ECO and *MCS* algorithm is an specific *OrderlyGeneration* algorithm, our proposed parallel computing approach will be applied for solving this issue and we use *MCS* algorithm as an original sequence algorithm for our approach. In particular, our proposed method was applied specifically for enumeration of orthogonal array of strength t 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 size S_0 ($S_0 < S$).

5 Algorithm Design Details

The most important phase on our proposed approach is the extending phase, in which all *LMC* matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m+1}; t)$ will be generated concurrently from all *LMC* matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}; t)$. The initial implementation of this phase will be presented on this section.

5.1 Domain Decomposition

At the beginning, all *LMC* matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}; t)$ are stored in an input file. We need useful methods to deliver input matrices to all processes.

Naive method. The basic method for domain decomposition is dividing equally all input for matrices. In particular, a single process, such as the process with rank 0, will read all input matrices from the input file and and deliver evenly *LMC* matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m}; t)$ to all other processes. For each input, each process utilizes *MCS* algorithm to generate *LMC* matrices of $OA(N; s_1^{a_1} \cdot s_2^{a_2} \cdots s_m^{a_m+1}; t)$. Note that there are no any dynamic load balancing scheme deployed on this method. The inputs are just distributed equally for all processes at the start time.

Master-slave method. There is another method using a single process for dynamic load balancing for all other processes. In particular, the process rank 0, called *master*, after reading all input matrices from the input file will distribute one input matrix for each other process, called *worker*, at a time to generate new results. After finish utilizing the single input, each worker will request a new input from master. Master accepts the request and sends another new input. This work continues until there are no more input at master.

In fact, the load balancing is better when we use the *Master-slave* method, this could be seen in Figure 2 when we do the experiment:

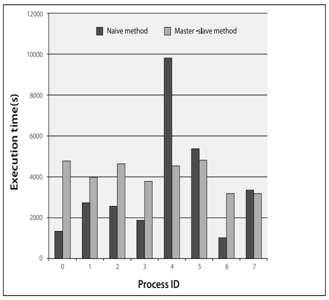


Fig. 2. Execution time of each process

Random pooling method. Besides using one process as a master for dynamic load balancing, some other methods also could be used. One of them is *random pooling* method, which is referred as one of the most efficient methods of requesting work when the underlying architecture of the computing system is now known [KGR94]. The idea of *random pooling* method is after finish exploring all of input matrices which had been assigned, the idle process will request for more inputs to explore from another randomly chosen process. It seems to us that *random pooling* could be a useful method for dynamic load balancing.

Work-stealing method. Besides *master-slave* and *random pooling* method, there is a popular method for dynamic load balancing named *Work-stealing* algorithm. The nature of work-stealing method is so simple. At each time step, each empty process will send a request to one other processor, which is chosen usually at random. Each non-empty processor having received at least one such request will select one of the requests. Now each empty process with its request being accepted will "steals" tasks (matrices) from the other process. Since work-stealing method is a stable [BF01] and scalable [DLS+09] algorithm for domain decomposition, it could be also a good choice for us for dynamic load balancing our general proposed approach.

At this time, the decision for choosing the proper domain decomposition and dynamic load balancing will depend on the specific ECO problem. On this initial experiment, we choose *Master-slave* as a method for dynamic load balancing.

5.2 Processing Outputs

Remind that our approach will generate all canonical objects of size k from all canonical objects of size $k-1$ and all results will be stored and reused in the next step. Hence, the storage of all generated outputs is an important factor that need to be concerned. The best choice is that all outputs which are generated by distinct processes will be stored concurrently on a single file. On this experiment, we choose a basic method that each time a new *LMC* matrix is generated, the process will write the result to the shared log file of the cluster system by basically using the *print* function of *C* language. However, with the support of parallel file system, *MPI-IO* [Mes97] could be a helpful tool for us to improve further the performance of concurrent I/O tasks.

5.3 Some Initial Experiment Results

Our work has been executed on the *Hercules* cluster of La Trobe university, Australia. A small experiment has been executed to simplify the evaluation. At the initial stage, the *MCS* algorithm is used to generate all 89 *LMC*-matrices of $OA(72; 3\cdot 2^4; 3)$. After that, those 89 *LMC* matrices of $OA(72; 3\cdot 2^4; 3)$ have been chosen as inputs for enumerating all non-isomorphism class of $OA(72; 3\cdot 2^5; 3)$. Note that the result is we have 27730 *LMC* matrices of $OA(72; 3\cdot 2^5; 3)$. The number of processes have been doubled up to 16. On each experiment, we collect the maximal execution time, total execution time and the results are recorded in Table 1:

Table 1. Execution time

Number of processes	Maximal execution time (minutes)
2 (1 + 1)	465.9’
4 (1 + 3)	156.23’
8 (1 + 7)	79.6.
16 (1 + 15)	57.45’

Note that, with the experiment using two processes, actually there is only one worker do the generation whilst the master just send every input for the worker at a time. With the experiment using four processes, there are three worker processes and one master and so on.

Since we use the master-slave method for our experiments, we always need at least two processes (one master and at least one worker) for every parallelism experiment. That’s why we are more concerned with the relative speedup of the algorithm with the initial 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 (ideal is 2)
4 (1 + 3)	2.98
8 (1 + 7)	1.97
16 (1 + 15)	1.38

With using 89 matrices inputs of $OA(72; 3 \cdot 2^4; 3)$ for all experiments, the results show that the speedup is really scalable with 8 processes. However, the speedup is reduced significantly when we doubled the process up to 16. This is because there is a really big difference of the exploring times for inputs. For example, the exploring times when we execute the *MCS* function on each input are recorded in the Figure 3.

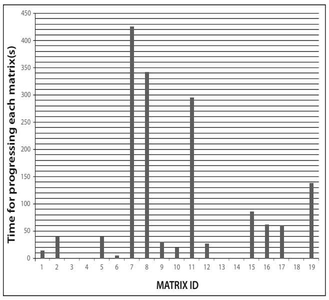


Fig. 3. Time for processing each input

As you can see in Figure 3, with a simple input, the exploring time could be about 1 second, however, with a more complex input matrix, the exploring time could be about 430 seconds. Note that the execution time of a process includes the summary of all the exploring times for all input matrices which had been assigned. When we just have 89 input matrices delivering for 16 processes, it is possible to have a situation in which the total exploring time for all inputs of a process is less than the exploring time just for one input of another process. This affects a lot to the load balancing of the algorithm. In fact, the execution time when we use 16 processes are significantly distinct as the results recorded in Figure 4.

Finally, it is noteworthy to analyze more about the cost for each extending phase. Using 89 input matrices of $OA(72; 3 \cdot 2^4; 3)$ to generate 27730 LMC matrices of $OA(72; 3 \cdot 2^5; 3)$ took us 80 minutes using 8 processes(see Table 1). Quantitatively, using 27730 LMC matrices of $OA(72; 3 \cdot 2^5; 3)$ to generate all LMC matrices of $OA(72; 3 \cdot 2^6; 3)$ with using 8 processes could take us about $(27730/89)* \ 80 =$ 25.000 minutes. This number shows that it is worth for applying the parallel computing for reducing the time taken for each extending phase.

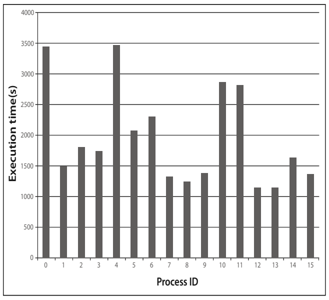


Fig. 4. Execution Time of 16 processes

6 Conclusion and Future Work

The parallelism approach presented in this paper is able to handle the computing-intensive of ECO problem. The approach utilizes the *OrderlyGeneration* original method of enumerating of combinatorial objects as a foundation to step-by-step extending generation.

With the proposed approach, we have some advantages. Most importantly, with the reusing of canonical objects, all objects of each level will be generated precisely one time. All objects of a new size will be enumerated from all generated objects of the previous size, not from scratch. This saves a lot of cost for regenerating old objects of previous size, especially when size S is huge. Moreover, on the point of data parallelism view, the reusing of previous objects for generating all objects of the next size strategy certainly gives us a great chance to apply useful data parallelism techniques in each extending step.

The experiments done on this current work are just on the initial stage, in which we use the Master-slave method for domain decomposition. It is because, on this paper we just aim to show the potential usefulness of our proposed method on applying parallel computing for enumeration of combinatorial objects. Certainly, we could explore advanced domain decomposition techniques such as random pooling or work-stealing method to improve the load balancing of the experiments. Besides, with applying work-stealing algorithm or a random pooling algorithm for domain decomposition, the speedup analysis of the initial experiment would provide a much better understanding of the performance gains (and the communication overhead) if done in contrast with the sequential algorithm, i.e. absolute speedup instead of relative speedup as we have done on this initial experiment.

Moreover, with the results gained on the initial work, we believe that I/O time spent writing and reading results, plus the communication load on the master node might be a dominant factor on the speed of the algorithm, especially when the search space is wide. Hence, some I/O optimization issues could be applied on the future, such as using *MPI-IO* to improve the performance of I/O tasks.

Finally, the potential usefulness of our approach could be applied for many special kinds of ECO. For example, besides the *MCS* algorithm for enumeration of orthogonal array of strength t, we have some specific *OrderlyGeneration* algorithms for special ECO problems, such as an *OrderlyGeneration* algorithm for *classifying triple systems* [KTR09]. Hence, we could apply our parallelism method for such *classifying triple systems* problem.

Acknowledgment

The research was carried out whilst the first author was supported by a La Trobe University Tuition Fee Remission, Postgraduate Research Scholarship and eResearch grant by La Trobe University. We also would like to thank anonymous reviewers have given us useful comments to improve further this final paper.

References

[AF93] Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Applied Mathematics 65, 21–46 (1993)

[AS94] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: APriori, pp. 487–499 (1994)

[BCH03] Brinkmann, G., Caporossi, G., Hansen, P.: A survey and new results on computer enumeration of polyhex and fusene hydrocarbons. Journal of Chemical Information and Computer Sciences 43(3), 842–851 (2003)

[BF01] Berenbrink, P., Friedetzky, T.: The natural work-stealing algorithm is stable. In: FOCS 2001: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science, Washington, DC, USA, p. 178. IEEE Computer Society, Los Alamitos (2001)

[CC05] Chen, Y., Crippen, G.M.: A novel approach to structural alignment using realistic structural and environmental information. Protein Science 14(12), 2935–2946 (2005)

[DLS$^+$09] Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: SC 2009: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11. ACM, New York (2009)

[Far78] Faradzev, I.A.: Constructive enumeration of combinatorial objects. problemes combinatoires et theorie des graphes collogue interat. CNRS 260, 131–135 (1978)

[GLMB96] Grner, T., Laue, R., Meringer, M., Bayreuth, U.: Algorithms for group actions: Homomorphism principle and orderly generation applied to graphs. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 113–122. American Mathematical Society, Providence (1996)

[KGR94] Kumar, V., Grama, A.Y., Rao, V.N.: Scalable load balancing techniques for parallel computers. Journal of Parallel and Distributed computing, 60–79 (1994)

[KO06] Kaski, P., Ostergard, P.R.J.: Classification algorithms for codes and designs. Algorithms and Computation in Mathematics 15 (2006)

[KTR09] Khosrovshahi, G.B., Tayfeh-Rezaie, B.: Classification of simple 2-(11,3,3) designs. Discrete Mathematics 309(3), 515–520 (2009); International Workshop on Design Theory, Graph Theory, and Computational Methods - IPM Combinatorics II, International Workshop on Design Theory, Graph Theory, and Computational Methods

[LM08] Lazic, L., Mastorakis, N.: Orthogonal array application for optimal combination of software defect detection techniques choices. W. Trans. on Comp. 7(8), 1319–1336 (2008)

[LYPP03] Lee, K.-H., Yi, J.-W., Park, J.-S., Park, G.-J.: An optimization algorithm using orthogonal arrays in discrete design space for structures. Finite Elements in Analysis and Design 40(1), 121–135 (2003)

[McK98] McKay, B.D.: Isomorph-free exhaustive generation. J. Algorithms 26(2), 306–324 (1998)

[Mes97] Message-Passing Interface Forum. MPI-2.0: Extensions to the Message-Passing Interface, ch. 9. MPI Forum (June 1997)

[MS08] Moura, L., Stojmenovic, I.: Backtracking and isomorph-free generation of polyhexes. In: Nayak, A., Stojmenovic, I. (eds.) Handbook of Applied Algorithms: Solving Scientic, Engineering, and Practical Problems, pp. 39–83. John Wiley & Sons, Chichester (2008)

[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)

[PSK$^+$07] Park, B.-H., Samatova, N.F., Karpinets, T., Jallouk, A., Molony, S., Horton, S., Arcangeli, S.: Data-driven, data-intensive computing for modelling and analysis of biological networks: application to bioethanol production. Journal of Physics: Conference Series 78, 012061 (6p.) (2007)

[Rea79] Read, R.C.: Every one a winner. Ann Discrete Math., 107–120 (1979)

[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)

[Wal60] Walker, R.J.: An enumerative technique for a class of combinatorial problems. In: Proc. Sympos. Appl. Math., vol. 10. American Mathematical Society, Providence (1960)

A Study of Performance Scalability by Parallelizing Loop Iterations on Multi-core SMPs

Prakash Raghavendra, Akshay Kumar Behki, K. Hariprasad, Madhav Mohan, Praveen Jain, Srivatsa S. Bhat, V.M. Thejus, Vishnumurthy Prabhu

Department of Information Technology, National Institute of Technology Karnataka, Surathkal
srp@nitk.ac.in, aksbeks@gmail.com, harinitk2007@gmail.com, madhavmon@gmail.com, prgu_jain@yahoo.com, bhat.srivatsa@gmail.com, thejus.vm@gmail.com, prabhuvishnumurthy@gmail.com

Abstract. Today, the challenge is to exploit the parallelism available in the way of multi-core architectures by the software. This could be done by re-writing the application, by exploiting the hardware capabilities or expect the compiler/software runtime tools to do the job for us. With the advent of multi-core architectures ([1] [2]), this problem is becoming more and more relevant. Even today, there are not many run-time tools to analyze the behavioral pattern of such performance critical applications, and to re-compile them. So, techniques like OpenMP for shared memory programs are still useful in exploiting parallelism in the machine. This work tries to study if the loop parallelization (both with and without applying transformations) can be a good case for running scientific programs efficiently on such multi-core architectures. We have found the results to be encouraging and we strongly feel that this could lead to some good results if implemented fully in a production compiler for multi-core architectures.

1 Introduction

Parallel processing requires program logic to have zero dependency between the successive iterations of a loop. To run a program in parallel we can divide the task between multiple threads or processes executing in parallel. We can also go up to the extent of running these parallel pieces of code simultaneously on different nodes in a high speed network. However the amount of parallelization possible depends on the program structure as well as the hardware configuration.

OpenMP [3] is an Application Program Interface (API) specification for C/C++ or FORTRAN that may be used to explicitly direct multi-threaded, shared memory parallelism. It is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. It has been implemented on most of the popular compilers like GNU (gcc), Intel, IBM, HP, Sun Microsystems compilers.

Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or FORTRAN source code. The use of the pre-processor directive (starting with #) along with the OpenMP directive instructs the compiler during pre-processing to implement parallel execution of the code following the OpenMP directive.

There are various directives available, one of them being '#pragma omp parallel for' whose implementation was our prime interest in the project. The '#pragma' directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. The '#pragma omp parallel for' directive instructs the compiler that all the iterations of the 'for' loop following the directive can be executed in parallel. In that case, OpenMP compiler will generate code to spawn optimized number of threads based on the number of cores available. Consider the following example of a typical C/C++ program using OpenMP pragmas:

```
#include <omp.h>
main ()  {
 int var1, var2, var3;
 /*** Serial code ***/
       .
       .
 /*** Beginning of parallel section. Fork a team of threads ***/
 /*** Specify variable scoping ***/
     #pragma omp parallel private(var1, var2) shared(var3)
        {
        /*** Parallel section executed by all threads ***/
               .
               .
        /*** All threads join master thread and disband ***/
        }

 /*** Resume serial code ***/
     .
     }
```

The variables var1 and var2 are private to each of the threads spawned and the variable var3 is shared among all the threads. The intent in this work is to study some OpenMP programs and see if these scale well on multi-core architectures. Further, we would also like to parallelize non-parallel loops by applying transformations (using known techniques like unimodular and GCD transformations [5] [6] [7]) and see if they too scale well on such architectures. We used some known OpenMP pragmas as case studies and implemented them in our compiler to study the performance. In Section 2, we describe the way we implemented these OpenMP pragmas. In Section 3, we discuss unimodular transformations which we used to parallelize non-parallel loops. In Section 4, we tabulate and explain all the results. Section 5 concludes the paper and suggests some directions for future work.

2 Implementation of OpenMP Pragmas in gcc

The parallel portions of the program can be run on different threads. For this we have to implement pthread library function calls in C. Our two approaches were: by using a runtime library, and second, by using a 'wrapper' which calls gcc. In the first approach, we planned to have our own library functions which would have to be called in the same way as OpenMP. We felt that this might not have much impact on performance and therefore, we resolved this to do at source level. We would call gcc on the resultant program for further code generation and optimization for the target platform.

Hence we opted for the second approach in which we intended to develop a wrapper on gcc which would take the file having OpenMP directives as the input file and automatically generate the code which implemented multithreading. The wrapper on gcc should be able to search for the OpenMP directives in the input file (program) and replace them with appropriate thread calls. This could be easily achieved using a shell script, since we were working in a Linux environment. The script searches for '#pragma' omp and removes it from the code, and the loop which follows this directive will be implemented in a function to be called by the thread. The loop which needs to be parallelized is divided into several segments each of which is executed by a thread. The kernel implicitly assigns individual threads to available cores. Threads can be implemented in two ways: Static and Dynamic assignment of threads. In static assignment, the total number of iterations of the loop is divided equally among the different threads. However, in dynamic assignment, we dynamically assign loop iterations to different threads as and when the threads complete their previous tasks. The static implementation of threads was observed to be faster than its dynamic counterpart since there is no explicit necessity to handle the tasks given to each of the threads. But in this case the number of iterations of the loop must be divisible by the number of threads we create, at compile time itself. We used the following standard for POSIX threads.

```
pthread_create(&thread_ID,NULL,function,&value);
```

This function creates a thread with the thread ID as 'thread_ID', with default characteristics (specified by NULL as second parameter), executes the function 'function ()' in the thread and passes the value 'value' (usually a structure) to the function.

```
pthread_join(thread_ID, &exit_status);
```

This function waits for the thread with thread ID 'thread_ID' to complete its execution and collects its exit status (return value if any) in 'exit_status'. This also helps in synchronizing more than one thread.

When the OpenMP pragmas are non-nested, then the case is very simple and all we have to do is to put the code following the directives in separate functions and call the functions using threads. Whereas when there are nested

OpenMP directives, then we will have to create threads again inside the outer thread functions, resulting in nested thread functions. As an added feature to the script, the number of threads to be implemented can be specified as a command line argument, default being two. We run the script as follows:

```
Syntax: ./script.sh <input_file> [no_of_threads]
```

3 Unimodular Transformations

Unimodular transformation is a loop transformation defined by a unimodular matrix. To test the validity of a general unimodular transformation, we need to know the distance vectors of the loop. After a valid transformation, if a loop in the transformed program carries no dependency, then it can execute in parallel.

Many researchers have applied unimodular transformations for parallelization of loops [11] [12]. Such transformations, though have limitations on applicability due to the model on which they work, can be applied very elegantly onto loops, which give various forms of loops executable in parallel. These types can be generalized by a set of non-parallel loops sandwiched between set of outer parallel loops and set of inner parallel loops. The beauty of this technique is that we can control on how many of such inner and outer loops we want depending upon the number of schedulable resources that we have. If the input loop does not belong to the model or if the number of dependencies are more than what is allowed by the model, then we will not be able to get completely parallel loops. In that case, we have to make these run with explicit communication, as explained in some works like [12]. In such methods, we allow the loops to have dependencies, however we still run these in parallel and making explicit synchronization wherever necessary so that the loops run correctly honoring all data dependencies. The extension of this technique would be one in [13] where they optimize the layout of arrays for such loops.

Program Model. Our model program [6] is the loop nest L= (L1, L2,...Lm):

```
L1:  do I1 = p1, q1
L2:  do I2 = p2, q2
   . .
   . . .
Lm:  do Im = pm, qm
     H (I1, I2,..., Im)
     Enddo
     .
     Enddo
     Enddo
```

An iteration H(i) is executed before another iteration H(j) in L iff i < j. Take any m*m unimodular matrix U. Let LU denote the program consisting of the iterations of L, such that an iteration H(i) is executed before another iteration

H(j) in LU iff iU < jU. LU can be written as a nest of m loops with an index vector K=(K1,K2,...Km) defined by K=IU. The body of the new program is H(KU-1) which will be written as HU (K). The program LU is the transformed program defined by U, and the transformation L -> LU is the unimodular transformation of L defined by U. The idea here is to transform the non-parallel loop into another base so that the same loop would become parallel (and can be executed in parallel on target machine).

4 Results

We studied the results as two problems, first to study the OpenMP ready programs with explicit parallel loops, and second, loops which are not parallel as given, but may need some transformations (like unimodular transformations) to make them run in parallel. We will show the results for the first case in Section 4.1 and show the results of next case in Section 4.2.

4.1 Study of OpenMP Parallel Programs

We took OpenMP programs like matrix multiplication with array sizes of 1800 * 1800 and LU factorization (array sizes of 1800*1800). We tested these programs with different number of threads (2, 4, 6, 8, 16, etc) on IBM Power 5 server and the performances were noted.

The IBM Power 5 server we used (IBM 9131 52A) has 4 physical processing units (8 cores). The maximum available physical processing units were 3.8 since the rest was used for VIOS (Virtual I/O Server). Each processing unit has 2 cores. The virtual machine on which the programs were tested was configured for two settings:
Profile 1: Virtual Processing Units: 3 (6 cores)
Physical Processing Units: 3 (6 cores)
Profile 2: Virtual Processing Units: 8 (16 cores)
Physical Processing Units: 3.8 (approx. 8 cores)

Matrix multiplication. The program snippet which performs matrix multiplication implemented with threads is as follows :

```
#pragma omp parallel for private(i,j,k)
  for(i = 0; i < 1800; i++){
    for(j = 0; j < 1800;j++){
      c[i][j] = 0;
        for(k = 0; k < 1800; k++){
          c[i][j] += a[i][k] * b[k][j];
 }}}
```

Table-1 shows the statistics for the above program using Profile-1 and Profile-2. Graph-1 shows the graph corresponding to Profile-1. Graph-2 shows the graph corresponding to Profile-2.

Table 1. Scale-ups for the above program with Profile-1 and Profile-2

Profile-1			Profile-2		
Threads	Time in s	Scale-up	Threads	Time in s	Scale-up
1	239.21	1	1	236.01	1
2	119.67	2	2	118.19	2
3	80	2.99	3	79.12	2.98
4	77.8	3.07	4	59.71	3.95
6	60.91	3.93	6	61.23	3.85
8	64.94	3.68	8	60.8	3.88
10	63.13	3.79	10	58.58	4.03
12	60.72	3.94	12	56.4	4.18
18	60.38	3.96	18	51.41	4.59
100	61.33	3.9	100	51.39	4.59
300	66.74	3.58	300	51.85	4.55

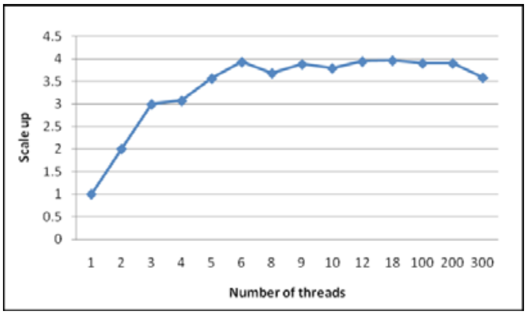


Fig. 1. Graph-1

From the observations, it can be concluded that the scale up increases with the number of threads as long as the number of threads is less than or equal to the number of cores available. We see that we can never approach the ideal speed up of N where N is the number of cores, since there is always some synchronization to be done, which would slow down the program execution.

Comparing the maximum scale up produced in the two cases (profile-1: 3.96, profile-2: 4.59), though the number of cores in the second case is twice that of the first one, considerable amount of scale up is not obtained. This is because in the second case 16 is the number of virtual cores and not the physical cores (which is actually 8). And hence one core is taking care of two threads, which leads to overhead due to switching of threads.

LU Factorization. The code snippet which performs LU factorization, implemented with threads is as follows :

```
#pragma omp parallel for private(i,j,k)
   for(k = 0; k < 1800; k++){
      for(i=k+1 = 0; i < 1800;i++){
            a[i][k] = a[i][k] / a[k][k];
   }
   for(i=k+1 = 0; i < 1800;i++){
   for(j=k+1 = 0; j < 1800;j++){
            a[i][j] = a[i][j] - a[i][k] * a[k][j];
   }}}
```

Table-2 shows the statistics for the above program using Profile-1 and Profile-2. Graph-3 shows the graph corresponding to Profile-1. Graph-4 shows the graph corresponding to Profile-2.

It is clearly observed in all of the graphs that when the number of threads is increased beyond a certain limit (number of cores available), the scale up remains almost a constant. But if it is further increased to a much higher value,

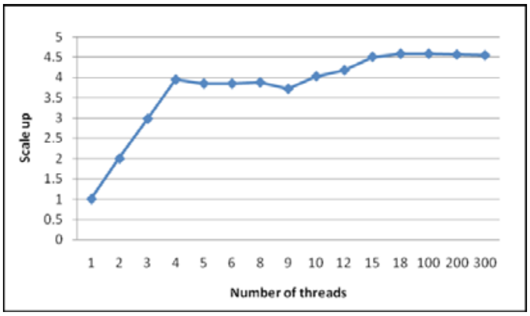


Fig. 2. Graph-2

Table 2. Scale-ups for the above program with Profile-1 and Profile-2

Profile-1			Profile-2		
Threads	Time in s	Scale-up	Threads	Time in s	Scale-up
1	57.16	1	1	57.13	1
2	50	1.14	2	49.95	1.14
3	40.2	1.42	3	40.17	1.42
4	33.03	1.73	4	33	1.73
6	24.25	2.36	6	24.65	2.32
8	21.57	2.65	8	20.74	2.75
10	19.79	2.89	10	18.76	3.05
12	18.8	3.04	12	17.12	3.34
18	16.28	3.51	18	15.37	3.72
100	15.69	3.64	100	11.97	4.77
300	16.03	3.57	300	12.14	4.71

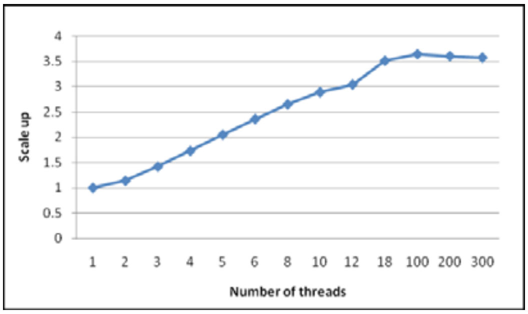


Fig. 3. Graph-3

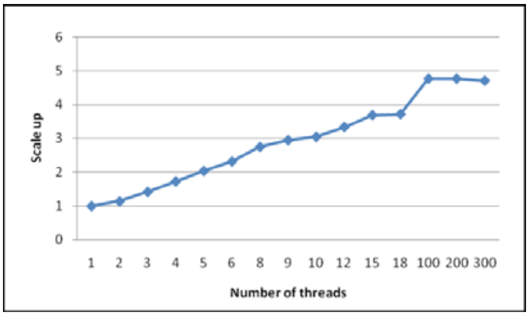


Fig. 4. Graph-4

the scale up would decrease. This is because of the delay (overhead) produced due to switching of the large number of threads.

As we know the loop following the OpenMP directives will be executed in parallel. We studied the effects of parallelizing different sets of loops which are nested. Consider the following code snippet :

```
for(i = 0; i < 1800; i++){
  for(j = 0; j < 1800;j++){
     c[i][j] = 0;
     for(k = 0; k < 1800; k++){
       c[i][j] += a[i][k] * b[k][j];
 }}}
```

The time taken for a single thread to complete the above loops was 239.21 seconds. When the outer most 'for' loop (loop counter 'i') was parallelized with 4 threads, the time taken was found to be 79.72 seconds. When the second 'for' loop (loop counter 'j') was parallelized with 4 threads, the time taken was found to be 101.56 seconds. When the inner most 'for' loop (loop counter 'k') was parallelized with 4 threads, the time taken was found to be 526.6 seconds.

The parallelization achieved by multithreading the outer ‘for’ loop is called ‘Coarse Granular Parallelization’. It takes the least time because each thread executes large chunk of calculations and hence overhead due to switching of threads is minimized since each thread once created handles significant part of the program. The parallelization achieved by multithreading the inner ‘for’ loop is called ‘Fine Granular Parallelization’. It takes more time because each thread executes smaller chunk of calculations and hence overhead is more due to repeated thread creation and destruction for very small tasks.

When the outer most and the second ‘for’ loop were parallelized with two threads each, the time taken was found to be 63.47 seconds, which is slightly better than that obtained by parallelizing the outer most loop alone with four threads (79.72 s). When the outer most and the inner most ‘for’ loop were parallelized with two threads each, the time taken was found to be 197.5 seconds. When the second and the inner most ‘for’ loop were parallelized with two threads each, the time taken was found to be 201.64 seconds.

The above data further proves higher performance gain of coarse granular parallelization over fine granular parallelization. There are a number of limitations in case of multithreading. There is an upper limit on the number of parallel threads we can create, since all these threads belong to a single process and each process has an upper limit on the memory it can use, depending on the hardware architecture. Each thread has its own program stack. So having a large number of threads in a single process may lead to memory insufficiency due to the large number of stacks required. Moreover, the best performance is obtained when the number of threads equals the number of actual cores available on the system.

4.2 Study of Loop Transformations on Multi-core

Loops with independent iterations can be easily parallelized unlike ones with dependencies. But some loops with dependencies can still be made independent to some extent by applying mathematical transformations, which then can be parallelized ([7]) . The transformations are usually specific to the function the body of the loop performs. Some specific examples are discussed below.

Inner loop parallelization. [6] Consider the following double loop which has dependencies :

```
for(i = 1; i < 1000; i++){
  for(j = 1; j < 1000;j++){
    for(l = 0; l < 10000; k++)// this loop is just to increase
     the calculations
      a[i][j] = a[i-1][j] + a[i][j-1];
 }}
```

This code takes 87.51 seconds to complete. This can be transformed into an independent loop given by:

```
for(k = 2; k < 1999; k++){
  k_1=1>(k-999)?1:(k-999);
  k_2=999>(k-1)?(k-1):999;
  for(k_1 = k_1; k_1 <= k_2;k_1++){
    for(l = 0; l < 10000; k++)// this loop is just to increase
     the calculations
      a[k-k_1][k_1] = a[k-k_1-1][k_1] + a[k-k_1][k_1-1];
 }}
```

In the above code, only the inner most loop is independent and hence can be run in parallel. When executed with two threads, the time taken is 52.17 seconds, which is faster compared to the time taken by dependent loops though the parallelization is fine granular. In this case, the number of iterations executed by the second 'for' loop is least for the first and the last iteration of the outer most 'for' loop and highest in between and the gradient is linear. Hence in order to obtain higher efficiency, the first and last few iterations (depending upon the amount of work being done) of the outer most 'for' loop can be executed sequentially and the rest in parallel or we can make use of dynamic threads. Dynamic threads results are in table-3.

Table 3. Scale-ups for Dynamic threads program

Threads	Time in s
2	95.33
4	41.53
8	39.44

Outer loop parallelization. [6] Consider the following code which has dependencies :

```
for(i = 6; i <= 500; i++){
  for(j = i; j <= (2*i)+4; j++){
    for(l = 0; l < 10000; k++)// this loop is just to increase
    the calculations
      a[i][j] = a[i-2][j-3] + a[i][j-6];
 }}
```

This code takes 32.96 seconds to complete. This can be transformed into an independent code given by :

```
for(y_1 = 0; y_1 <= 1; y_1++){// parallelizable loop
for(y_2 = 0; y_2 <= 1; y_2++){ // parallelizable loop
for(k_1 = ceil((6-y_1)/2.0); k_1 < floor((10-y_1)/2.0); k_1++){
for(k_2 = ceil(y_1+2*k_1-y_2);
k_2 < floor((2*y_1+4*k_1-4-y_2)/3.0); k_2++){
a[y_1+2*k_1][y_2+3*k_2] = a[y_1+2*k_1-2][y_2+3*k_2-3] +
a[y_1+2*k_1][y_2+3*k_2-6];
 }}}}
```

When executed with two threads, the time taken is 20.28 seconds, which is faster compared to the time taken by dependent loops. Since it is a case of coarse granular parallelization, increasing the number of threads gives better performance. The above program with maximum 6 threads gives a scale up of 3 (13.35 s) which is not up to the expectations since the number of inner loop iterations is greater than that of the original program.

5 Conclusion

In this study, we did two things. First, study performance of a few OpenMP ready parallel programs on multi-core machines (up to 8 physical cores). For this, we implemented the OpenMP compiler over the gcc compiler. Second, we extended our work to also include non-parallel loops and used some transformations to make these loops run in parallel. We were delighted to see that both these cases proved that running these (with or without transformations) can give us significant performance benefits on multi-core architectures. The trend today is to have more and more cores and in that context, this work would be quite relevant. In future, we would like to extend our work to more non-parallel loops, which may run in parallel with some explicit synchronization on multi-core.

References

1. AMD Multi-core Products (2006), http://multicore.amd.com/en/products/
2. Multi-core from Intel Products and Platforms (2006), http://www.intel.com/products/processor/
3. OpenMP, http://www.openmp.org
4. Wolfe, M.J.: Techniques for improving the inherent parallelism in programs. Technical Report 78-929, Department of Computer Science, University of Illinois at Urbana-Champaign (July 1990)
5. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley, Reading
6. Banerjee, U.K.: Loop Transformations for Restructuring Compilers: The Foundations. Kluwer Academic Publishers, Norwell (1993)
7. Banerjee, U.K.: Loop Parallelization. Kluwer Academic Publishers, Norwell (1994)
8. Pthreads reference, https://computing.llnl.gov/tutorials/pthreads/
9. DHollander, E.H.: Partitioning and Labelling of loops by Unimodular Transformation. IEEE Transactions on Parallel and Distributed Systems 3(4) (1992)
10. Saas, R., Mutka, M.: Enabling unimodular transformation. In: Supercomputing 1994, November 1994, pp. 753–762 (1994)
11. Banerjee, U.: Unimodular Transformations of Double Loop. In: Advances in Languages and Compilers for Parallel Processing, pp. 192–219 (1991)
12. Prakash, S.R., Srikant, Y.N.: An Approach to Global Data Partitioning for Distributed Memory Machines. In: IPPS/SPDP (1999)
13. Prakash, S.R., Srikant, Y.N.: Communication Cost Estimation and Global Data Partitioning for Distributed Memory Machines. In: Fourth International Conference on High Performance Computing, Bangalore (1997)

Impact of Multimedia Extensions for Different Processing Element Granularities on an Embedded Imaging System

Jong-Myon Kim

School of Computer Engineering and Information Technology,
University of Ulsan,
San 29, Mugeo-2 Dong, Nam-Gu, Ulsan, South Korea, 680-749
Tel.: +82-52-259-2217; Fax: +82-52-259-1687
jongmyon.kim@gmail.com

Abstract. Multimedia applications are among the most dominant computing workloads driving innovations in high performance and cost effective systems. In this regard, modern general-purpose microprocessors have included multimedia extensions (e.g., MMX, SSE, VIS, MAX, ALTIVEC) to their instruction set architectures to improve the performance of multimedia with little added cost to microprocessors. Whereas prior studies of multimedia extensions have primarily focused on a single processor, this paper quantitatively evaluates the impact of multimedia extensions on system performance and efficiency for different number of processing elements (PEs) within an integrated multiprocessor array. This paper also identifies the optimal PE granularity for the array system and implementation technology in terms of throughput, area efficiency, and energy efficiency using architectural and workload simulation. Experimental results with cycle accurate simulation and technology modeling show that MMX-type instructions (a representative Intel's multimedia extensions) achieve an average speedup ranging from 1.24× (at a 65,536 PE system) to 5.65× (at a 4 PE system) over the baseline performance. In addition, the MMX-enhanced processor array increases both area and energy efficiency over the baseline for all the configurations and programs. Moreover, the highest area and energy efficiency are achieved at the number of PEs between 256 and 1,024. These evaluation techniques composed of performance simulation and technology modeling can provide solutions to the design challenges in a new class of multiprocessor array systems for multimedia.

Index terms: image and video processing, multimedia extensions, multiprocessor arrays, grain size determination, technology modeling.

1 Introduction

The growing popularity of multimedia has created new demand for portable electronic products [1]. These applications, however, demand tremendous computational and I/O throughput. Moreover, increasing user demand for multimedia-over-wireless capabilities on embedded systems places additional constraints on power, size, and weight.

Application-specific integrated circuits (ASICs) can meet the needed performance and cost goals for such embedded imaging systems. However, they provide limited, if any, programmability or flexibility needed for varied application requirements.

General-purpose microprocessors (GPPs) offers the necessary flexibility and inexpensive processing elements, and multimedia extensions to GPPs have improved the performance for multimedia applications with litter added cost to the processors. Examples include Intel MMX [2], SSE, and SSE-2 [3], Hewlett Packard MAX-2 for the PA-RISC architecture [4], Sun VIS for SPARC [5], Alpha MVI [6], and Motorola ALTIVEC for PowerPC architecture [7]. These extensions exploit subword parallelism by packing several small data elements (e.g., 8-bit pixels) into a single wide register (e.g., 32-, 64-, and 128-bit) while processing these separate elements in parallel. The designers of digital signal processors (DSPs), such as the Texas Instruments TMS320C64x families [8] and the Analog Devices TigerSharc processor [9], have followed the trend.

However, despite some performance improvements through multimedia extensions, neither GPPs nor DSPs will be able to meet the much higher levels of performance required by emerging multimedia applications on higher resolution images. This is because they lack the ability to exploit the full data parallelism available in these applications.

Among many computational models available for imaging applications, single instructions multiple data (SIMD) processor architectures [15] are promising candidates for embedded multimedia systems since they replicate the datapath, data memory, and I/O to provide high processing performance with low node cost. Whereas instruction-level or thread-level processors use silicon area for large multiported register files, large caches, and deeply pipelined functional units, SIMD processor arrays contain many more simple processing elements (PEs) for the same silicon area. As a result, SIMD processor arrays often employ thousands of PEs with the data I/O to minimize storage and data communication requirements. While it is evident that the overall performance improvement is achieved with increasing number of PEs (or parallelism), no general consensus has been reached that what granularity of processors and memories on an array system is best for multimedia capabilities and what kind of impact does multimedia extensions have on performance and both area and energy efficiency for different PE granularities (or the number of PEs).

This paper presents the impact of multimedia extensions for different number of PEs in an integrated processor array and determines the most efficient PE granularity which delivers the required processing performance with the lowest cost and longest battery life for multimedia applications. Prior studies have primarily focused on a single processor whereas this study quantitatively evaluates the impact of the multimedia extensions on performance and efficiency metrics for different PE granularities. Unlike many architectural parameters, analysis of multimedia extensions for different PEs is difficult since each PE configuration has a different amount of local memory and it significantly affects both hardware and software design. This paper effectively evaluates the effects of multimedia extensions for different PE granularities using application retargeting simulator combined with both area and energy technology modeling. Experiment results indicate that MMX-type instructions (a representative Intel's multimedia extensions) achieve an average speedup ranging from 1.24× (at a 65,536 PE system) to 5.65× (at a 4 PE system) over the baseline due to more subword parallelism. In addition, the MMX-enhanced processor array increases both area and energy efficiency over the baseline for all the configurations and programs because it achieves higher sustained throughput with a small increase in the system area and power. Moreover, the most efficient operation is achieved at the number of PEs between 256 and 1,024. Using these evaluation techniques, a new class of multiprocessor array systems for imaging applications can be designed.

The rest of this paper is organized as follows. Section II presents the design methodology and exploration strategy. Section III analyzes execution performance and efficiency for each case, and Section IV concludes this paper.

2 Methodology

2.1 Simulation Infrastructure

Figure 1 shows an overview of our simulation infrastructure which is divided into three levels: application, architecture, and technology. At the application level, an instruction-level SIMD processor array simulator has been used to profile execution statistics, such as cycle counts, dynamic instruction frequencies, and PE utilization, for the two different versions of the programs: (1) baseline ISA without subword parallelism (base-SIMD) and (2) baseline plus MMX-type ISA (MMX- SIMD). At the architecture level, the heterogeneous architectural modeling (HAM) of functional units for SIMD arrays [10] has been used to calculate the design parameters of the modeled architectures. For the design parameters of the MMX functional unit (FU), Verilog models for the baseline and MMX FUs were implemented and synthesized

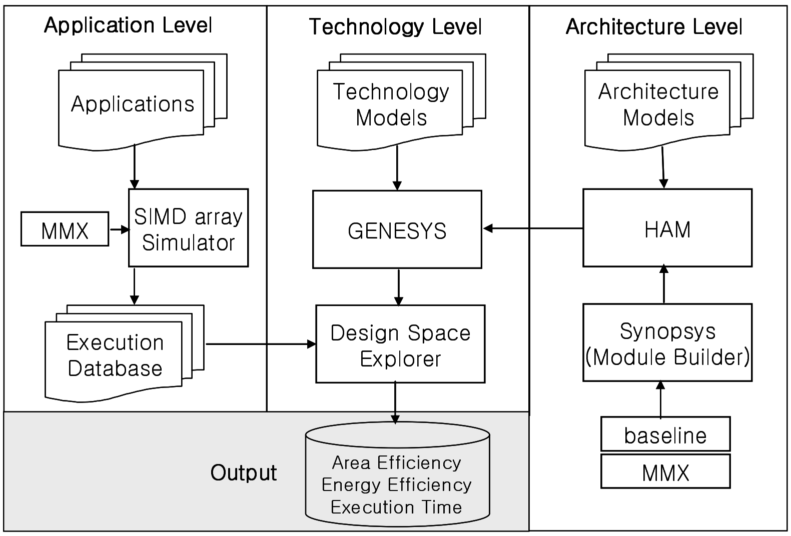


Fig. 1. A simulation infrastructure for the design space exploration of modeled architectures

with the Synopsys design compiler (DC) using a 0.18-micron standard cell library. The reported area specification of the MMX FU was then normalized to the baseline FU, and the normalized number was applied to the HAM tool to determine the design parameter of the MMX-SIMD array. The design parameters are then passed to the technology level. At the technology level, the Generic System Simulator (GENESYS) [11] has been used to calculate technology parameters (e.g., latency, area, power, and clock frequency) for each configuration. Finally, the databases (e.g., cycle times, instruction latencies, instruction counts, area, and power of the functional units), obtained from the application, architecture, and technology levels, were combined to determine execution times, area efficiency, and energy efficiency for each case. The next section evaluates the system area and power of the modeled architectures.

2.2 Modeled Processor Array Architectures

A SIMD processor array architecture shown in Figure 2 is used as our baseline architecture for this study. This SIMD processor architecture is symmetric, having an array control unit (ACU) and an array consisting of from a few ten to a few thousand PEs. Depending on the number of PEs in a system, each PE in different configurations supports a different size of pixels and a different local memory to store input images and temporary data produced during processing. This study assumes that each PE is associated with a specific portion of an image frame with sensor subarrays, allowing streaming pixel data to be retrieved and processed locally. Each PE has a reduced instruction set computer (RISC) datapath with the following minimum characteristics:

- ALU – computes basic arithmetic and logic operations,
- MACC – multiplies 32-bit values and accumulates into a 64-bit accumulator,
- Sleep – activates or deactivates a PE based on local information,
- Pixel unit – samples pixel data from the local image sensor array,
- ADC unit – converts light intensities into digital values,
- Three-ported general-purpose registers (16 32-bit words),
- Small amount of local storage (64 32-bit words),
- Nearest neighbor communications through a NEWS (north-east-west-south) network and serial I/O unit.

In this study, eight different PE granularities (in the number of PEs) are used for the performance and efficiency evaluation of MMX-type instructions in which each PE in different PE configurations supports a different number of pixels. For a total system calculations, an image size of 256 × 256 is used. The resulting number of PEs for a fixed 256 × 256 pixel system is defined as $N_{PE} = 2^{2(8-i)}$, $i = 0,\ldots,7$ in which each PE supports 2^{2i}, $i = 0,\ldots,7$, respectively. To support the different PE granularities within the array, we have modified hardware and software design of the architecture. In addition, we have added a MMX-type functional unit to the system to evaluate

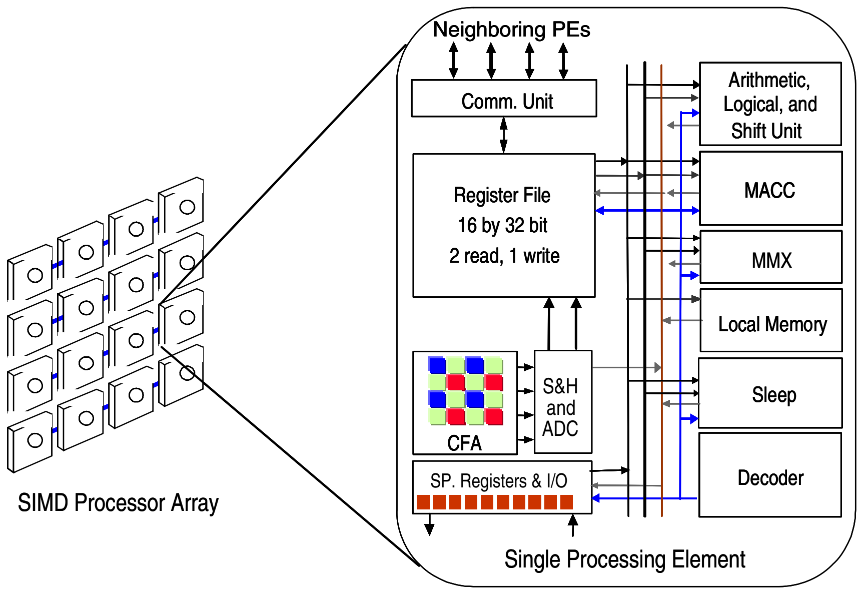


Fig. 2. A block diagram of a SIMD processor array and a processing element used in this study

Table 1. Modeled system parameters

Parameter	Value							
Number of PEs	65,536	16,384	4,096	1,024	256	64	16	4
Pixels/PE	1	4	16	64	256	1,024	4,096	16,384
Memory/PE [word]	8	16	64	256	1,024	4,096	16,384	65,536
VLSI Technology	100 nm							
Clock Frequency	500 MHz							
Interconnection Network	Mesh							
intALU/intMUL/Barrel Shifter/intMACC/Comm	1 / 1 / 1 / 1 / 1							

the impact of the MMX-type instructions for each system configuration. Each configuration has a different amount of local memory. Table 1 describes all the system configurations and their local memory sizes.

2.3 Benchmark Applications

To evaluate the modeled architectures, an application suite was selected, retargeted, and simulated using the instruction-level simulator. The suite includes two imaging applications: vector quantization (VQ) and motion estimation (ME). These applications form significant components of many current real-world workloads such as image and video compression. For a fair performance comparison, the applications were retargeted and optimized for each PE configuration individually.

2.3.1 Vector Quantization

Full search vector quantization (FSVQ), a promising candidate for low rate and low power image and video compression, was selected for a case study. It has a computationally inexpensive decoding process and low hardware requirement for decompression, while still achieving an acceptable picture quality at high compression ratios [12][13]. However, the encoding process is computationally very intensive. Computational cost can be reduced by using suboptimal approaches such as tree-searched vector quantization (TSVQ) [12]. This study prefers to overcome the computational burden by using a parallel implementation of FSVQ on a SIMD processor array system. FSVQ is defined as the mapping of k-dimensional vectors in the vector space $\mathbf{R}^k$ into a finite set of vectors $\mathbf{V} = \{\mathbf{c}_i, i=1,\ldots,N\}$, where N is size of the codebook. Each vector $\mathbf{c}_i=(\mathbf{c}_1,\ldots,\mathbf{c}_N)$ is called as a code vector or codeword. Only index i of the resulting code vector is sent to the decoder. At the decoder, an identical copy of the codebook is looked up by a simple table-lookup operation.

In this implementation, we use a codebook of 256 4×4 code vectors designed off-line through a standard Linda-Buzo-Gray (LBG) training process to achieve a 0.5 bit per pixel encoding for an image in 24-bit data, using 4×4 (k = 16). In the 2-D case, non-overlapping vectors are extracted from the input image by grouping a number of contiguous pixels to retain available spatial correlation of data. The input blocks are then compared with the codebook in a parallel systolic fashion, with a large number of them compared at any given time in parallel. A key enabling role is played by the toroidal structure of the interconnection network, which enables communication among the nodes on opposite edges of the mesh. The most time-critical operation for this implementation is the distortion calculation between a 4×4 input block and a local codeword. The distortion can be efficiently calculated with the SAD (sum of absolute differences) instruction by comparing pairs of sub-elements in the two source registers while accumulating each result in the packed accumulator. Table 2 shows a comparison of instruction counts using the baseline and MMX-type ISAs for a full search VQ operation of 4×4 pixels. The instruction count decreases 81% with MMX-type instructions over the baseline performance.

Table 2. A comparison of instruction counts using the baseline and MMX-type ISAs for a VQ operation of 4×4 pixels

	Baseline	MMX-type
ALU	483	37
MEM	80	34
MASK	34	-
MMX	-	17
Scalar Instructions	34	34
Total	631	122

2.3.2 Motion Estimation

Motion estimation (ME) is a core building block in several video compression standards (e.g., H.26x and MPEG). Compression is achieved through a block-matching algorithm (BMA) that subdivides the current frame into small reference blocks and then finds the best match for each block among the available blocks in the previous frame. In this implementation, the macroblock size of 16 × 16 pixels and the search range of ±8 were used. Since the objective of this study is to achieve accurate motion estimates, both luminance and chrominance components in the program were used rather than only the luminance component in the standard BMA. In particular, the proposed approach, called *full search vector BMA* (FSVBMA), uses both luminance and chrominance components while arriving at one motion vector for all components, improving the accuracy of the process and the overall video quality [14]. The matching criterion of the FSVBMA for motion estimation is defined as

$$\begin{aligned} MAD(m,n) = &\sum_{i=0}^{M-1}\sum_{j=0}^{N-1} | y(i+m, j+n) - x(i,j) |_Y \\ &+ \sum_{i=0}^{M-1}\sum_{j=0}^{N-1} | y(i+m, j+n) - x(i,j) |_{Cb} \\ &+ \sum_{i=0}^{M-1}\sum_{j=0}^{N-1} | y(i+m, j+n) - x(i,j) |_{Cr} \,, \end{aligned} \tag{1}$$

$$-p \leq m, n \leq p \,, \tag{2}$$

$$\boldsymbol{v} = \arg\min_{-p\leq m,n\leq p} MAD(m,n), \tag{3}$$

where *x(i,j)* is the reference block of size *M* × *N* pixels at coordinates *(i,j)*, *y(i+m,j+n)* is the candidate block within a search area in the previous frame, *p* is the search range, *(m,n)* represents the candidate displacement vector, and ***v*** is the motion vector.

Clearly, the most time-critical operation is the mean absolute differences (MAD) computation that involves a reference block of pixels and all the candidate blocks of pixels in the search area. Similar to the VQ implementation, the MAD block is efficiently processed with the SAD instruction by comparing pairs of the sub-elements in the two source registers (one containing pixels within the candidate block; the other containing pixels within the reference block) while accumulating each result in the packed accumulator. This process is iterated until all the candidate blocks are compared by the reference block. Table 3 shows a comparison of instruction counts using

Table 3. A comparison of the number of instructions using the baseline and MMX-type ISAs for a MAD operation of 16×16 pixels

	Baseline	MMX-type
ALU	392	42
MEM	33	33
MASK	48	-
COMM	6	6
MMX	-	16
Scalar Instructions	33	33
Total	512	130

the baseline and MMX-type ISAs for a MAD computation of 16×16 pixels. The instruction count decreases 75% with MMX-type instructions over the baseline version.

2.4 System Area and Power Evaluation Using Technology Modeling

Figures 3 and 4 present the distribution of each functional unit's area and power, respectively, for the SIMD processor array with and without MMX. Each bar divides the system area and power into register file, arithmetic and logical unit (ALU), multiply-accumulate unit (MACC), shifter, memory, communication, decoder, PE activity control unit (sleep), and serial unit. For the number of PEs at or below 256, both baseline and MMX array's area and power are dominated by local memory. Above this point, however, MMX requires higher system area and power than the baseline since the area overhead of the MMX execution unit is significant. These system area and power results are combined with application simulations to determine both area and energy efficiency for each case, which is presented next.

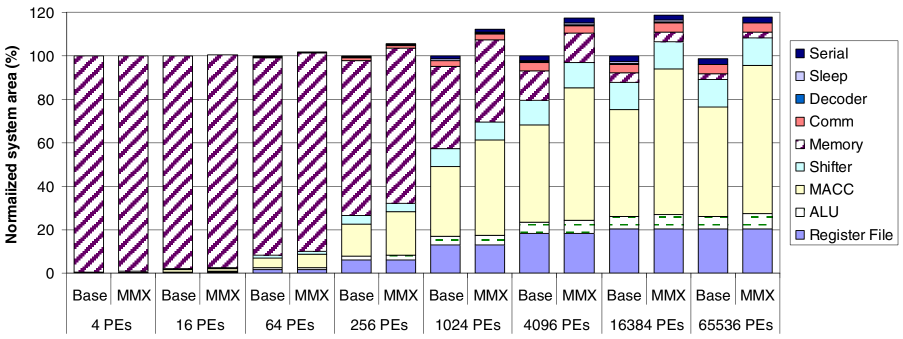


Fig. 3. Impact of MMX on system area for different number of PEs

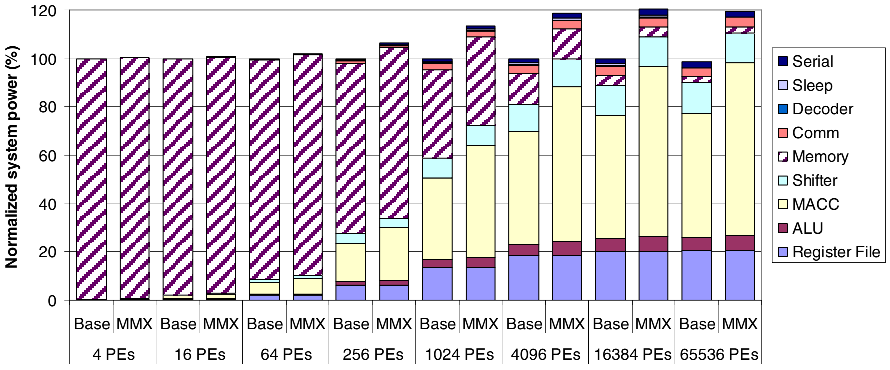


Fig. 4. Impact of MMX on system power for different number of PEs

3 Experimental Results

Application simulation and technology modeling are used to determine performance and efficiency for each PE configuration for the chosen workloads. The execution time, sustained throughput, area efficiency, and energy efficiency of each case form the basis of the study comparison, defined in Table 4.

Table 4. Summary of evaluation metrics

execution time	sustained throughput	area efficiency	energy consumption
$t_{exec} = \frac{C}{f_{ck}}$	$\eta_E = \frac{O_{exec} \cdot U \cdot N_{PE}}{Energy} [\frac{\text{Gops}}{\text{Joule}}]$	$\eta_A = \frac{Th_{sust}}{Area} [\frac{\text{Gops}}{\text{s} \cdot \text{mm}^2}]$	$Energy = Power \cdot t_{exec} [J]$

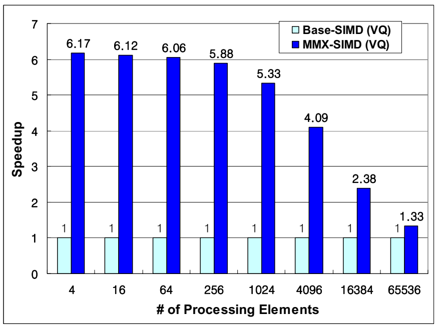
C is the cycle count, f_{ck} is the clock frequency, O_{exec} is the number of executed operations, and N_{PE} is the number of processing elements. Note that since each MMX instruction executes more operations (typically four times) than a baseline instruction, we assume that each MMX and baseline instruction executes four and one operation, respectively, for the sustained throughput calculation.

3.1 Execution Performance

This section evaluates the impact of MMX-type instructions on processing performance for each case.

3.1.1 Impact of MMX on Processing Performance

Figure 5 shows execution performance (speedup in executed cycles) for different number of PEs with and without MMX. As expected, MMX outperforms the baseline for all the PE configurations because of more subword parallelism. However, the slope is not equal. For the number of PEs above 4,096, less speedup efficiency is achieved. This is because high inter-PE communication operations are involved in the task, which are not affected by MMX.



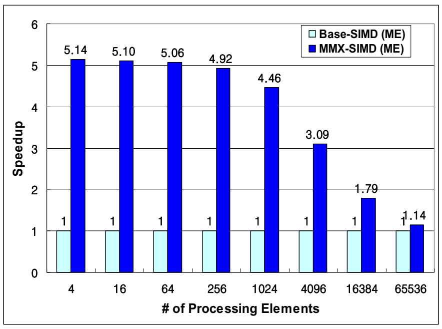


Fig. 5. Speedups for different PE configurations with and without MMX

3.1.2 Impact of MMX on System Utilization

System utilization is calculated as the average number of active processing elements MMX increases system utilization over the baseline, shown in Figure 6. This is because MMX compare instructions allow multiple conditional (MASK) instructions with one equivalent MMX instruction, reducing PE idle cycles based on the local information. As with the vector instruction count, MMX is less effective at reducing PE idle cycles for the number of PEs above 1,024 because of high inter-PE communication operations that are not affected by MMX. The next sections discuss energy and area efficiency results.

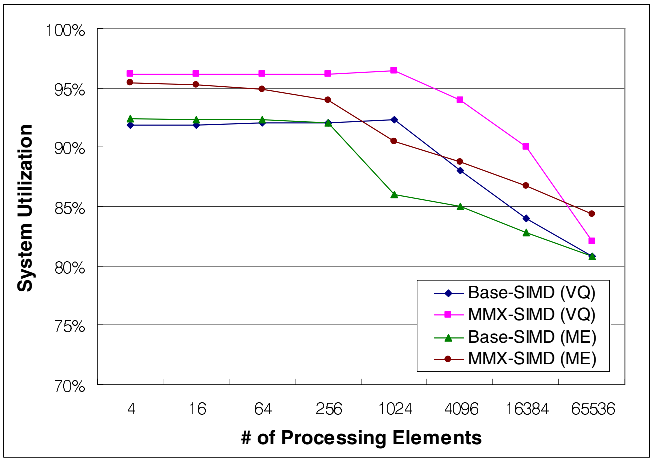


Fig. 6. System utilization for different PE configurations with and without MMX

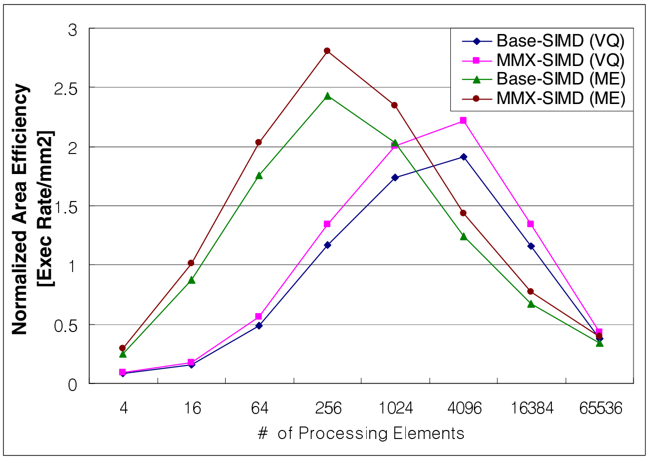


Fig. 7. Normalized area efficiency of MMX versus the number of PEs

3.2 Area Efficiency

Area efficiency is the task throughput achieved per unit of area. Figure 7 presents area efficiency for each case. Maximum area efficiency of both MMX and baseline for VQ and ME is achieved at PEs = 4,096 and 256 due to the inherent definition of FSVQ and ME encoding, respectively. MMX increases area efficiency over the baseline for all the PE configurations and programs. This is because MMX achieves higher sustained throughput with smaller area overhead. Increasing area efficiency improves component utilization for given system capabilities.

3.3 Energy Efficiency

Energy efficiency is the task throughput achieved per unit of Joule. Figure 8 presents energy efficiency for each case. As with area efficiency, maximum energy efficiency of both MMX and baseline for VQ and ME is achieved at PEs = 4,096 and 256 due to the inherent definition of FSVQ and ME encoding, respectively. MMX increases energy efficiency over the baseline for all the PE configurations and programs. This is because MMX achieves higher sustained throughput with a small increase in the system power. Increasing energy efficiency improves sustainable battery life for given system capabilities.

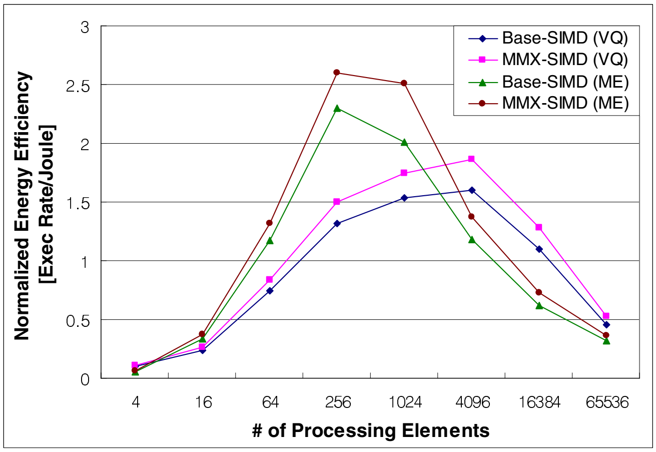


Fig. 8. Normalized energy efficiency of MMX versus the number of PEs

4 Conclusions

Continued advances in multimedia computing will rely on reconfigurable silicon area usage within an integrated pixel processing array. This paper explores the impact of MMX on performance and both area and energy efficiency for different PE granularities and identifies the most efficient PE configuration for a specified SIMD processor array and implementation technology. Experimental results using architectural and workload simulation indicate that the MMX-enhanced processor array decreases execution time for all the configurations and programs over the baseline performance due to more subword parallelism. In addition, the MMX-enhanced processor array increases both area and energy efficiency over the baseline for all the configurations and programs because it achieves higher sustained throughput with a small increase in the system area and power. Moreover, the most efficient operation for VQ and ME encoding is achieved at the number of PEs between 256 and 4,096, respectively. These evaluation techniques can provide solutions to the design challenges in a new class of multiprocessor array systems for multimedia.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. R01-2008-000-20493-0).

References

1. Agerwala, T., Chatterjee, S.: Computer architecture: challenges and opportunities for the next decade. IEEE Micro, 58–69 (May-June 2005)
2. Peleg, A., Weiser, U.: MMX technology extension to the Intel architecture. IEEE Micro 16(4), 42–50 (1996)
3. Raman, S.K., Pentkovski, V., Keshava, J.: Implementing streaming SIMD extensions on the Pentium III processor. IEEE Micro 20(4), 28–39 (2000)
4. Lee, R.B.: Subword parallelism with MAX-2. IEEE Micro 16(4), 51–59 (1996)
5. Tremblay, M., O'Connor, J.M., Narayanan, V., He, L.: VIS speeds new media processing. IEEE Micro 16(4), 10–20 (1996)
6. Sites, R. (ed.): Alpha Reference Manual. Digital, Burlington (1992)
7. Nguyen, H., John, L.: Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In: Proc. Intl. Supercomputer Conference, June 1999, pp. 11–20 (1999)
8. TMS320C64x DSP Technical Brief, `http://www.ti.com/sc/docs/products/dsp/c6000/c64xmptb.pdf`
9. Fridman, J., Greenfield, Z.: The TigerSHARC DSP architecture. In: Proc. IEEE/ACM Intl. Sym. on Computer Architecture, May 1999, pp. 124–135 (1999)
10. Chai, S.M., Taha, T.M., Wills, D.S., Meindl, J.D.: Heterogeneous architecture models for interconnect-motivated system design. IEEE Trans. VLSI Systems, special issue on system level interconnect prediction 8(6), 660–670 (2000)
11. Nugent, S., Wills, D.S., Meindl, J.D.: A hierarchical block-based modeling methodology for SoC in GENESYS. In: Proc. of the 15th Ann. IEEE Intl. ASIC/SOC Conf., September 2002, pp. 239–243 (2002)
12. Nozawa, T., et al.: A parallel vector-quantization processor eliminating redundant calculations for real-time motion picture compression. IEEE J. Solid-State Circuits 35(11), 1744–1751 (2000)
13. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 3rd edn. Prentice Hall, Englewood Cliffs (2008)
14. Tremeau, A., Plataniotis, K., Tominaga, S.: Color in Image and Video Processing. Special issue of EURASIP Journal on Image and Video Processing (2008)
15. Chiu, J.-C., Chou, Y.-L., Tzeng, H.-Y.: A Multi-streaming SIMD Architecture for Multimedia Applications. In: Proceedings of the 6th ACM conference on Computing frontiers, pp. 51–60 (2009)

Reducing False Aborts in STM Systems

Daniel Nicácio and Guido Araújo

Institute of Computing
UNICAMP
{dnicacio,guido}@ic.unicamp.br

Abstract. Transactional memory (TM) continues to be the most promising approach replacing locks in concurrent programming, but TM systems based on software (STM) still lack the desired performance when compared to fine-grained lock implementations. It is known that the critical operation in TM systems is to ensure the atomicity and isolation of concurrently executing threads. This task is known as the read/write-set validation. In attempt to make this process as fast as possible, STM systems usually use ownership tables to perform conflict detection, but this approach generates false positive occurrences, which result in false aborts. This paper shows the real impact of false aborts and how its relevance increases along with the number of concurrent threads, showing it is an essential factor for TM systems. We propose two different techniques to avoid false aborts, showing its benefits and limitations. The first is a collision list attached to the existing hash table. The second is a full associative memory mapping between the addresses and its version information. We achieved significant performance improvements in some STAMP benchmark programs, resulting in speedups up to 1.5x. We also show that speedups become higher when the number of parallel threads increases.

1 Introduction

Multi-core chips are a solid reality now, being present in desktops, servers and even embedded systems. Although multi-cores chips have largely been adopted, parallel programming has not. The use of locks has been the most widely used tool for controlling access to shared data, but while coarse-grained locks are easy to manage, it limits the parallelism and has a poor performance. On the other hand, fine-grained locks perform quite well, but designing such a system is already known as a very complex task better left to experts only. Computer scientists are looking for a programming tool which combines the low complexity of coarse grained locks and the minimal contention of fine grained locks. The transactional memory programming paradigm proposed by Herlihy and Moss [1] seems to be the most promising accepted answer.

Transactional Memory (TM) allows the programmer to mark code segments as transactions without worrying about interactions between concurrent operations. Transactions are automatically executed as an atomic operation and in parallel with other transactions as long they do not conflict. A conflict occurs if two or more ongoing transactions try to access the same data and at least one of them modifies it. There are many design proposals for TM systems: (a) based on hardware transactional memory (HTM) [1,2]; (b) pure software transactional memory (STM) [3,4]; and (c) hybrid techniques (HyTM) that combine hardware and software [5,6]. TM achieves its goal by lessening the burden of parallel programming, but its performance is not yet adequate.

It is known that the read/write set validation task is critical in STM systems. It consists of (1) keeping track of every read and write operation for each ongoing transaction, usually done by storing it in read/write sets; (2) verifying each element in a write-set if it is also present in the read-set or write-set of another transaction; if this happens, a conflict is detected and one of the transactions involved must be aborted. In an attempt to make the verification process as fast as possible, STM systems typically uses techniques susceptible to false positive answers (e.g. memory mapping using a hash table), thus generating unnecessary false aborts.

This paper shows that false aborts are a real threat for STM systems, and as the number of concurrent threads increases, this problem becomes even more critical. Empirical tests show that the probability of a conflict to occur is at least proportional to the number of parallel threads executing. In order to evaluate this, we analyzed the effects of false aborts, measuring the false abort ratio and the time spent on false aborted transactions. Then, we propose two solutions to address this problem and show their benefits and limitations: (1) a collision list for the commonly used hash table and (2) a full associative mapping between the addresses and its version information.

In Section 2 we list related works to false aborts, false conflicts, and conflict detection techniques. In Section 3 we explain the conflict detection process and provide quantitative data to support the need of a conflict detection technique free of false positives. Section 4 shows in detail the two techniques mentioned earlier, explaining how they work and why they tend to become even more powerful as the number of cores in a single chip increases. In Section 5 we present the experimental results of the proposed techniques. And in Section 7 we conclude this work.

2 Related Work

Zilles and Rajwar [7] have a similar work regarding false conflict rate. They demonstrated through an analytical model validated by statistical data that a tagless ownership table results in an alias-induced conflict rate. This rate increases approximately as the square of both concurrency and size of the transaction data footprint. This demonstration was based on a simulated environment. They suggest a tagged ownership organization for alias elimination. Our work differs in many aspects: (1) while they used analytical and statistical models, we made practical experiments on commonly used TM benchmarks using a well known STM system; (2) they aimed at hybrid TM system, and we used a STM model; (3) although they suggested the use of a tagged table, they did not show the results (time benefits and overhead) of such technique; and finally, (4) we also implemented a full associative memory mapping technique to avoid the aliasing of a tagless table.

Xiaoqiang, Lin, and Lunguo [8] focus on high transaction abort ratio, which degrades the overall performance of high contention workloads. Their experimental results show that a STM system implementation does not scale well. The number of transactions aborted every second increases for each CPU added. According to the authors, the problem is that concurrent threads with high contention hurt performance by increasing the number of conflicts, which in turn wastes resources through aborted transactions. To overcome this issue, they tried to reduce the number of conflicts between transactions, using a causal consistency memory model with weaker semantics. We agree with them on how the number of conflicts increases as the number of cores grows. In this paper we approach the problem from a different angle, reducing the number of conflicts generated by false positives.

There are some related works to conflict detection techniques. Agrawal, Fineman and Sukha [9] proposed a conflict detection technique called XConflict for a TM system designed for nested parallel transactions. Their algorithm dynamically divides the computation into traces, where each trace consists of memory operations, then XConflict manages conflicts only between traces. Unfortunately, the performance analysis of their model does not include checking for conflicts with multiple readers and possible aborts, what could significantly increase the runtime.

Shriraman and Dwarkadas [10] analyzed the interplay between conflict resolution time and contention management policy in the context of hardware-supported TM systems. Although they did not change the way conflict detection is performed, they claimed that Lazy policies provide higher performance than Eager polices. They also evaluated a mixed conflict detection mode that detects write-write conflicts eagerly, while detecting read-write conflicts lazily. Atoofian, Baniasadi and Coady [11] also discussed the efficiency of Lazy and Eager polices, and proposed a new read validation policy that adapts to the workload.

3 The False Abort Issue

This section shows how false aborts increases as the number of cores raises. Furthermore, we measure the time wasted in false aborted transactions, highlighting the importance of effective conflict detection techniques.

The work presented in this paper was implemented in TL2 [12] STM system, considered a state-of-the-art STM implementation. TL2 stores a read/write set for each ongoing transaction. These sets store every address read and/or written by the respective transaction. In order to always keep the memory consistent, avoiding *zombie* transactions, the system checks if every load and write operation is valid; this checking is called conflict detection. Conflict detection is also done immediately before the transaction commits (to assure nothing became inconsistent during the transaction execution). This task is performed as follows: (1)

for every read operation, the system verifies the version-clock associated to the address being read and compare it to the transaction version-clock; (2) before the transaction commits, the version-clock of every address in the read set is rechecked. If any of them has changed to an invalid state, a write operation was executed in the meantime, changing the address version-clock and producing a conflict. In such case, the transaction must be aborted.

Trying to make this process as fast as possible, TL2 uses a hash function to map memory addresses to its respective version-clock. As a hash function is used, alias problem can occur, as two different addresses might be mapped to the same entry. Suppose the hash function maps addresses X and Y to the same hash table entry. As a consequence, if transaction T1 reads address X, then transaction T2 writes in Y; when T1 tries to commit, it will detect a conflict in X and will abort. But the conflict is a false conflict, since X and Y were two completely different memory positions. When a transaction is aborted due to a false conflict, we have a *false abort*. Most of the STM systems [13,14,15] use such techniques susceptible to false aborts.

We have made modifications to TL2 so as to measure the time spent in each transaction. Next, we classified transactions in three groups: committed, which successfully committed; true aborted, which aborted due to a true conflict; and false aborted, which aborted due to a false conflict. The number of occurrences and the time spent in each type of transaction were measured. The environment used to run these tests is described in Section 5.

The graphic of Figure 1 shows, for seven STAMP [16] benchmarks, the percentage of started transactions that ended up in a false abort and the percentage of time spent in transactions that ended up in a false abort. As shown, when running up to eight parallel threads, around one quarter of aborted transactions on Kmeans High (26%) and Kmeans Low (18%) are false aborted transactions. This number is still high for Labyrinth (8%) and Vacation High (6%). Other programs have 0.6% false aborted transactions on average. All programs have presented an increasing pattern of false aborts as the number of threads increase.

The time shown in this graphic corresponds exclusively to the time spent in transactional code (including the STM management overhead during its execution). Running with eight parallel threads, Labyrinth spends 22% of its time on false aborted transactions, Kmeans High 11%, Genome, Bayes and Vacation High between 5% and 10%, and the rest of the programs 1.7%. Again, the figure clearly shows an increase in the number of false aborts as the number of threads increase. The benchmark SSCA2 did not presented any aborts in our tests; hence it is not displayed in the graphic.

The graphic reveals that false aborts exist in a substantial number for some programs. As mentioned, the time spent in false aborted transactions for program Labyrinth was high; avoiding false aborts would probably provide a speedup around 22% in this program. Also, when a false abort is eliminated, it brings a number of positive consequences: (1) the time spent on the transaction is not wasted, eliminating the need of redoing computation, (2) the roll-back process

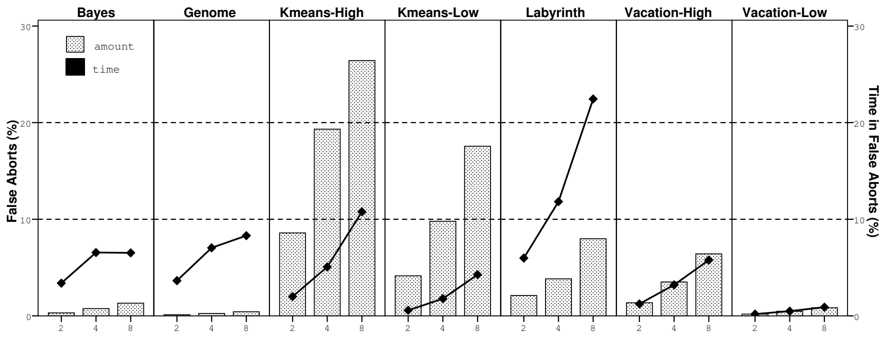


Fig. 1. The percentage of false aborted transactions and the time percentage spent in false aborted transactions of seven STAMP programs

to restore the previous processor's state is eliminated, and (3) as we decrease the time on transactions, we also diminish the chances of real conflicts to occur.

It is worth noting that the number of false aborted transactions has no direct relation to the time spent on false aborted transactions. The size of a transaction has direct influence on the time needed by the roll-back process, which produces high cost aborts. For example, Labyrinth has long transactions, while Kmeans High has very short transactions. This explains why Kmeans High, despite its high number of false aborted transaction, does not spend much time of its execution on aborts.

4 Conflict Detection Free of False Aborts

As described above, the performance of STM may be improved by avoiding false aborts. To achieve that, false aborts must be avoided by using a low overhead technique to be of any worth. In the next two sub-sections we present two different ways to do it. The first one is based on hash table conflict disambiguation, and the second uses a full associative memory mapping.

4.1 A Hash Table Collision List

Zilles and Rajwar [7] proposed a chaining hash table to eliminate false conflicts, but they did not implement it to see its benefits, and they focused on hybrid systems. Our implementation follows the same idea they proposed, but some modifications were necessary due to some technical restrictions and the fact that we tried to change the TL2 algorithm as minimally as possible.

TL2 uses a 4MB hash table capable of storing 1000 entries. Each entry is a 32 bit word. Depending on the value of the least significant bit, the 30 most significant bits contain a version-clock or a pointer to a read/write set element.

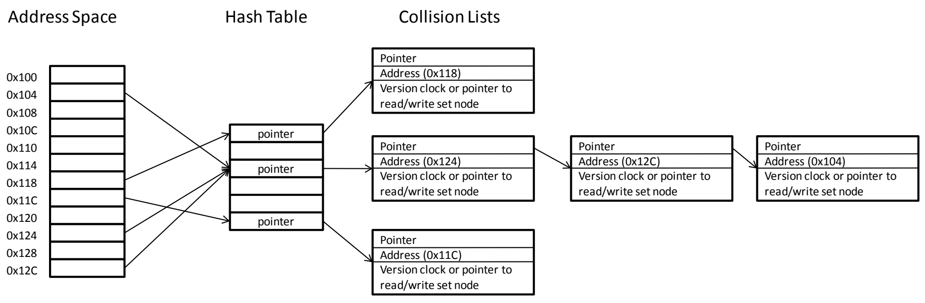

Address Space
Hash Table
Collision Lists
0x100
0x104
0x108
0x10C
0x110
0x114
0x118
0x11C
0x120
0x124
0x128
0x12C
pointer
pointer
pointer
Pointer
Address (0x118)
Version clock or pointer to read/write set node
Pointer
Address (0x124)
Version clock or pointer to read/write set node
Pointer
Address (0x12C)
Version clock or pointer to read/write set node
Pointer
Address (0x104)
Version clock or pointer to read/write set node
Pointer
Address (0x11C)
Version clock or pointer to read/write set node

Fig. 2. The hash table collision list for STM systems

First, we thought about using the second least significant bit to hold information in case the entry had a collision or not (a collision bit). This way, a level of indirection would be needed only when a collision was present. Unfortunately, in the TL2 context this approach seems to be impossible. Due to the irreversible feature of hash functions, if there is a version clock in the entry, there is no way to know the address associated to that clock; therefore, there is no available information to set the collision bit properly.

The implementation used in this paper stores a pointer to a collision list at each hash table entry. A collision list node is a structure containing: (1) a pointer for a possible next node, (2) the memory address associated to the node, and (3) the 32 bit word previously present in the hash table. The implementation overhead come as the additional level of indirection added to the scheme and the memory needed to build the collision list node (each node requires 12 bytes). Figure 2 depicts on collision list approach.

4.2 A Full Associative Memory Mapping

Our second approach to avoid false aborts was to hold a unique information for each address writen and/or read by a transaction. In order to do this, a full associative memory mapping is used.

First, the address is divided in two parts: the *highAddr* (8 most significant bits) and the *lowAddr* (bit 2 to bit 24). An array of 256 positions (*highAddrList*) is used to represent the *highAddr*. Each position points to its associated *lowAddrList*, each *lowAddrList* has 2^{22} entries and each entry holds the version clock or the pointer to a read/write set node used in the TL2. The *lowAddrList* has only 2^{22} elements, and not 2^{24}, because TL2's API defines the minimum data granularity as a word.

Figure 3 shows an example where address 0x030011F0 is mapped. In our example, the *highAddr* has value 3, it indicates that the fourth entry of the *highAddrList* holds the pointer to the array that represents the *lowAddr*. The *lowAddr* has value 1148, so the 1149^{th} entry of *lowAddrList* holds the information used by TL2.

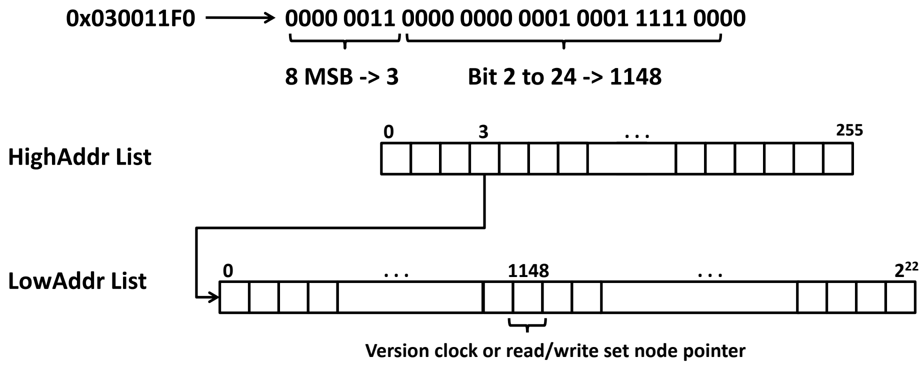


Fig. 3. The full associative memory mapping scheme

Notice that we divided the address in 8/24 bits instead of half. This decision was based on the locality principle, which states [17] that memory references tend to access smaller regions of the total memory. It means that, if we divide it in 8/24, only a few *lowAddrLists* will be needed to map all referenced address. If we divide the address in half, a lot of smaller *lowAddrLists* would be needed, and they would be spread in the memory space. Again, based on the locality principle, to access big contigous regions of memory is more efficient than access a lot of small spread regions of memory.

The *lowAddrLists* are allocated on demand, making the total memory needed by this technique proportional to the application data footprint. Its performance is also linked to the amount of memory allocated, since the allocation process is expensive. Furthermore, all allocated *lowAddrLists* cannot be deallocated until the end of the application. It might be prohibitive for some applications, but, if memory is available, this technique proved to be very efficient.

The *lowAddrLists* could be deallocated if and only if we guarantee that all of its addresses are not in use by any transaction at a given moment. Some techniques of garbage collection were tested, but none was viable to use in the memory transaction context.

5 Experimental Results

In this section we preset the speedup obtained by both techniques: hash table collision list and full associative memory mapping. The results use the eager conflict detection policy (the results using the lazy conflict detection policy were very similar, and due to space limitation, are not shown here). Both, TL2 system and STAMP, were compiled with ICC (Intel C Compiler) using -O3 flag. Our experiments and results were measured on a Intel dual-Xeon (eight 2.0 GHz cores) machine, 1MB L2 cache available for each core. Every result presented here corresponds to the average of 30 runs, and had its standard deviation measured and analyzed.

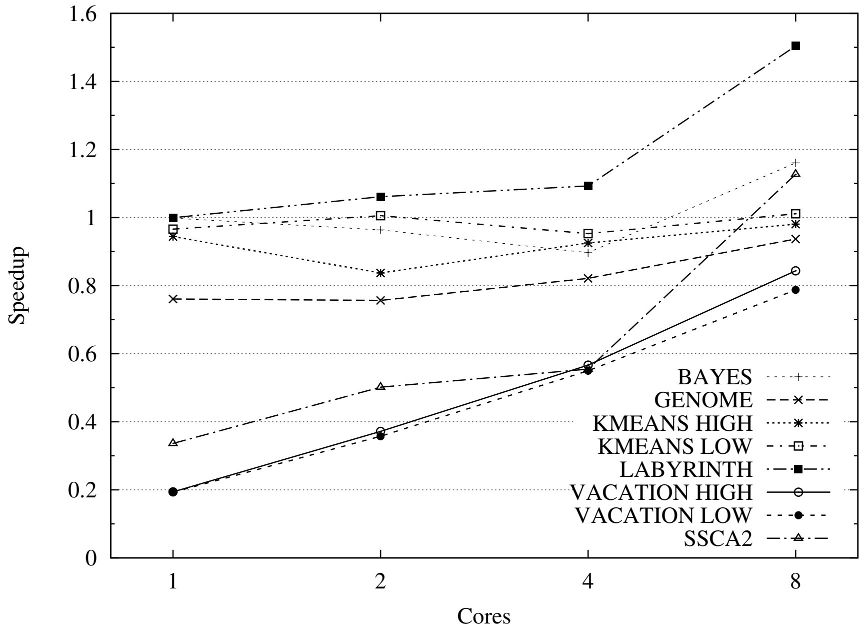


Fig. 4. The hash table collision list speedup on the STAMP benchmark

This evaluation was done using programs from the STAMP benchmark. The eight STAMP programs include a gene sequencing program (Genome), a bayesian learning network (Bayes), a network intrusion detection algorithm (Intruder), a k-means clustering algorithm (KMeans), a maze routing algorithm (Labyrinth), a set of graph kernels (SSCA2), a client-server reservation system simulating SpecJBB (Vacation), and a Delaunay mesh refinement algorithm (Yada). Of those, Yada and Intruder failed to run on our hardware, using the out of the box TL2 version available. Hence, we do not report results for these two benchmarks. For the remaining benchmarks we report the achieved speedups.

The graphic on Figure 4 shows the hash table collision list speedup on the STAMP benchmark. Most programs start with a slowdown when running only one or two threads, but when the number of threads goes to 4 and 8, the slowdown diminishes and in some cases (Bayes, KmeansLow, Labyrinth and SSCA2) the speedup comes to evidence. The maximum speedup achieved on this set was 1.5x on Labyrinth. VacationHigh and VacationLow start with a major slowdown when running the collision list technique. this happens due to the huge data footprint of those programs, but even they tend to be benefited by this technique as more cores are added. The relevant information extracted from this result is the clear speedup tendency in almost all programs, making the future 16 core tests very promising.

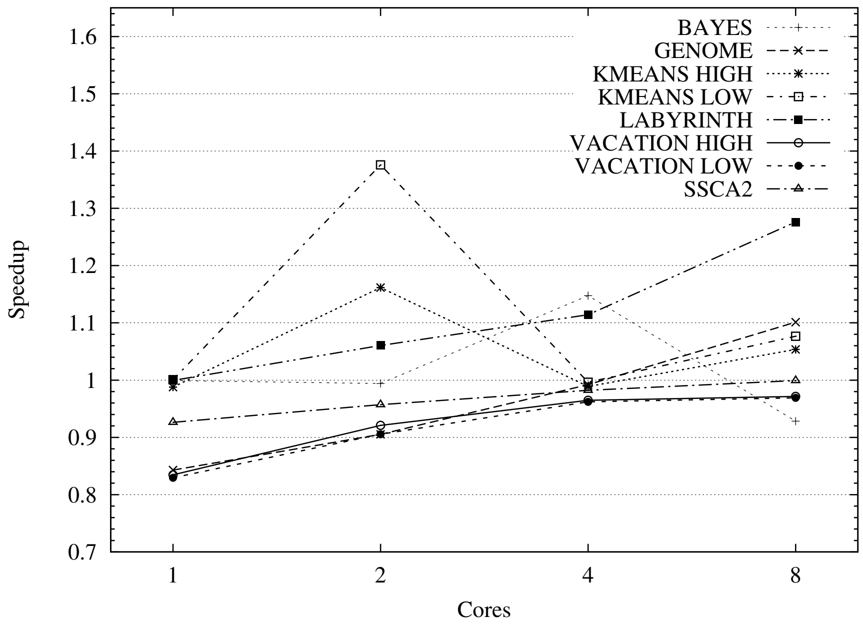


Fig. 5. The full associative memory mapping speedup on the STAMP benchmark

The graphic on Figure 5 shows speedup when using the full associative memory mapping technique on the STAMP benchmark. Again, with only one and two cores, some programs present a slowdown; as in the collision list technique, it comes from the overhead introduced by both techniques. The speedup tendency also appears here, as the number of working cores increases, the number of false conflicts also increases. When running eight parallel threads, only VacationHigh, VacationLow and Bayes showed slowdowns, and the maximum slowdown was only 0.92x on Bayes. All the remaining programs had speedups, the maximum speedup obtained was 1.28x on Labyrinth, followed by 1.10x on Genome and 1.08x on KmenasLow.

The overall result shows that the higher the system contention, the more it benefits from a conflict detection technique free of false positive occurrences. If the speedup tendency keeps its pace, which will likely happen, most programs will present a major speedup when executing 16 threads.

6 Further Discussion

In Section 3 we show that programs waste execution time in false aborts. Results obtained with the two techniques went accordingly to what we would expect, programs with higher wasted times showed the best performance improvement (i.e. Labyrinth and Genome). However, programs Kmeans-High and Vacation-High also had a significant amount of time wasted in false aborted transactions, but have not benefited as expected. In this section we analyze what these programs have in common to present this behavior.

Minh et al. [16] provided a detailed characterization of applications included in STAMP benchmark. Among other features, they provide the number of instructions per transaction (mean). Labyrinth and Genome have large transactions, more than 1700 instructions per transaction in Genome and 680000 in Labyrinth. With such large transactions, a transaction abort cost is high due to the need roll back, making it a good target for a conflict detection technique free of false conflicts. On the other hand, Kmeans-High has only 150 instructions per transactions, resulting in low cost aborts.

Also according to Minh et al. [16], Kmeans-High and Kmeans-Low spend just a small amount of their total execution time in transactional code. This is the main reason why our techniques have not improved nor impacted their performance.

Program Vacation-High had a major slowdown when executing with the hash table collision list technique. Its performance was hurt due to its huge data footprint. With a lot of different addresses to be mapped to the hash table, the number of chained entries in the hash table raises. When the conflict detection is finished, the whole chain of entries must be verified, considerably increasing the technique overhead.

7 Conclusion

This paper put to evidence, through experimental analysis, the false abort issue, highlighting its real threat for high contention systems. We measured the number and the time spent in transactions that ended up in a false abort. Up to 26% of the started transactions ended in a false abort. The time percentage wasted on such transactions reached up to 22%.

To overcome this obstacle, we proposed two solutions: a collision detection list for the commonly used hash table, and a full associative memory mapping scheme. Both solutions avoid false conflicts and were implemented in the TL2 STM system and evaluated through the STAMP benchmark.

Results showed the utility of a collision detection list and the full associative mapping scheme and how they become important as the number of parallel threads increases. The maximum speedup achieved was 1.5x when running eight parallel threads. Most programs had significant performance improvements. Programs which were not improved tend to expose speedup on 16+ cores processors.

Acknowledgements

The authors would like to thank CNPq (process 142869/2009-0) for funding this project.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
2. 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 consistency. In: ISCA 2004: Proceedings of the 31st annual international symposium on Computer architecture, Washington, DC, USA, p. 102. IEEE Computer Society, Los Alamitos (2004)
3. Shavit, N., Touitou, D.: Software transactional memory. In: PODC 1995: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pp. 204–213. ACM, New York (1995)
4. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003: Proceedings of the twenty-second annual symposium on Principles of distributed computing, pp. 92–101. ACM, New York (2003)
5. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 336–346. ACM, New York (2006)
6. Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: ISCA 2007: Proceedings of the 34th annual international symposium on Computer architecture, pp. 69–80. ACM, New York (2007)
7. Zilles, C., Rajwar, R.: Implications of false conflict rate trends for robust software transactional memory. In: IISWC 2007: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization, Washington, DC, USA, pp. 15–24. IEEE Computer Society, Los Alamitos (2007)
8. Xiaoqiang, Z., Lin, P., Lunguo, X.: Lowering conflicts of high contention software transactional memory. In: CSSE 2008: Proceedings of the 2008 International Conference on Computer Science and Software Engineering, Washington, DC, USA, pp. 307–310. IEEE Computer Society, Los Alamitos (2008)
9. Agrawal, K., Fineman, J.T., Sukha, J.: Nested parallelism in transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 163–174. ACM, New York (2008)
10. Shriraman, A., Dwarkadas, S.: Refereeing conflicts in hardware transactional memory. In: ICS 2009: Proceedings of the 23rd international conference on Supercomputing, pp. 136–146. ACM, New York (2009)
11. Atoofian, E., Baniasadi, A., Coady, Y.: Adaptive read validation in time-based software transactional memory, pp. 152–162 (2009)
12. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Proc. of the 20th Intl. Symp. on Distributed Computing (2006)
13. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 26–37. ACM, New York (2006)

14. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA 2003: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pp. 388–402. ACM, New York (2003)
15. 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: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 187–197. ACM, New York (2006)
16. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proceedings of The IEEE International Symposium on Workload Characterization (September 2008)
17. Denning, P.J.: The locality principle. Commun. ACM 48(7), 19–24 (2005)

Fault-Tolerant Node-to-Set Disjoint-Path Routing in Hypercubes

Antoine Bossard, Keiichi Kaneko, and Shietung Peng

Graduate School of Engineering
Tokyo University of Agriculture and Technology
Tokyo 184-8588, Japan
{50008834706@st,k1kaneko@cc}.tuat.ac.jp
Department of Computer Science
Hosei University
Tokyo 184-8584, Japan
speng@k.hosei.ac.jp

Abstract. Hypercube-based interconnection networks are one of the most popular network topologies when dealing with parallel systems. In a hypercube of dimension n, node addresses are made of n bits, and two nodes are adjacent if and only if their Hamming distance is equal to 1. In this paper we introduce a fault-tolerant hypercube routing algorithm which constructs n minus the number of faulty nodes f disjoint paths connecting one source node to $k = n - f$ destination nodes in $O(n^2)$ time complexity.

Keywords: Interconnection networks, hypercube, routing algorithm, disjoint paths.

1 Introduction

Parallel processing has become an increasingly important part of computer science nowadays. The majority of computers, even personal computers, contains now several CPU cores. If desktop computers are bundled with a few (two, four), today large scale computers (supercomputers) are built with hundreds of thousands of processors.

For example, the IBM Roadrunner which as of November 2008 ranked no.1 in the TOP500 list [1] embodies 130,464 cores. Also, early 2009, a new supercomputer "Sequoia" has been announced by IBM, with a computing power equivalent to that of 2 million laptops, where the current Roadrunner has a computing power similar to that of about 100,000 laptops [2]. In this era of massively parallel processing systems, it is easy to understand that networking a such huge amount of CPUs is a major issue to retain high performance. Beyond choosing an efficient interconnection network, network routing is also a key problem to be able to use as efficiently as possible such large amount of processors.

Hypercube topology is one of the most used interconnection networks when dealing with massively parallel architectures, for example the Intel iPSC [3] or the SGI Origin 2000 [4]. Such large scale parallel systems are error prone, that's why algorithms designed for these systems should be tolerant against faults [8], [9]. Hence we propose in this paper a fault-tolerant routing algorithm in hypercube which will disjointly route from one common source node to at most n destinations nodes, where n is the dimension of the hypercube.

The node-to-set disjoint-path routing problem is to find node-disjoint paths between one common source node and several distinct destination nodes in polynomial time [12], [13], [14].

This algorithm adopts a much simpler approach than the one given by Peng *et al.* in [10] or Rabin in [7], and besides, while retaining the optimal $O(n^2)$ time complexity, our algorithm allows an additional parameter specifying which neighbour nodes of the source to use, it is the restriction set X.

The rest of the paper is structured as follow. Section 2 recalls some important definitions and properties of hypercubes. Also, some general definitions and lemmas needed later in this paper will be introduced. In Section 3, the node-to-set routing algorithm is described, with an additional variation. The correctness and complexities of the algorithm are studied in Section 4. An example is given in Section 5 and Section 6 finally concludes this paper.

2 Preliminaries

A hypercube of dimension n, also called an n-cube, consists of nodes whose addresses can be represented by n bits. Such a hypercube has thus a total of 2^n nodes. Also, two nodes in a hypercube are adjacent if and only if their addresses have a Hamming distance of 1. Hence, we can deduce the following important properties of an n-cube H_n: each node of H_n has a degree n, H_n has a diameter equal to n, and finally it is n-connected. Another fundamental property of hypercubes is that they are symmetric graphs [5].

Now a property of hypercubes extensively used in this paper: hypercubes have a recursive structure: considering a certain dimension $0 \leq i \leq n-1$, a hypercube H_n of dimension n (n-cube) consists of two subcubes H^0_{n-1} and H^1_{n-1} (also more generally denoted H^0 and H^1), both of dimension $n-1$, respectively containing the nodes of H_n whose i-th bit is set to 0 or 1.

Let us recall some general definitions. A path P is defined as an alternate sequence of nodes and edges as follow: $P := \boldsymbol{a}_1, (\boldsymbol{a}_1, \boldsymbol{a}_2), \boldsymbol{a}_2, \ldots, \boldsymbol{a}_{k-1}, (\boldsymbol{a}_{k-1}, \boldsymbol{a}_k), \boldsymbol{a}_k$. In this paper, a path will be often shortened to $P := \boldsymbol{a}_1 \rightarrow \boldsymbol{a}_2 \rightarrow \ldots \rightarrow \boldsymbol{a}_k$, or even to its simplest expression $P := \boldsymbol{a}_1 \rightsquigarrow \boldsymbol{a}_k$. The length of a path is defined as its number of edges, denoted $L(P)$. A path is said fault-free if it contains no faulty node. Two paths are disjoint if they have no node in common (excepted the source node $\boldsymbol{s}$ in our problem).

Gu and Peng described in [6] a fault-tolerant node-to-node routing algorithm in hypercube H_n, returning a path of length at most $n+2$ with a time complexity of $O(n)$. Gu and Peng described in [10] a fault-tolerant node-to-set routing algorithm in hypercube H_n, returning paths of length at most $n+2$ with a time complexity of $O(|F|n)$. We put these two results respectively in Lemma 1 and Lemma 2.

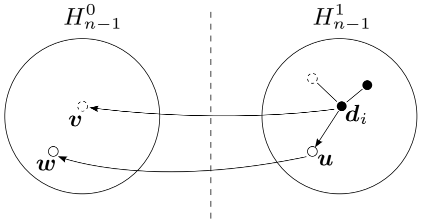


Fig. 1. Fault-free path of length at most 2 linking $\boldsymbol{d}_i \in H^1_{n-1}$ to H^0_{n-1}. (faulty nodes are dashed, nodes of D are black).

Lemma 1. [6] In an n-dimensional hypercube H_n, given a source node $\boldsymbol{s}$, a destination node $\boldsymbol{d}$ and a set of faulty nodes F, $|F| \leq n-1$, we can find a fault-free path connecting $\boldsymbol{s}$ and $\boldsymbol{d}$ of length at most $n+2$ in $O(n)$ time complexity.

Lemma 2. [10] In an n-dimensional hypercube H_n, given a source node $\boldsymbol{s}$, a set of destination nodes D and a set of k faulty nodes F, $|D| + |F| \leq n$, we can find $|D|$ fault-free paths connecting $\boldsymbol{s}$ to each node of D of length at most $n+2$ in $O(kn)$ time complexity.

Lemma 3. In an n-dimensional hypercube H_n, we can find $k \leq n - |F|$ fault-free disjoint paths of length at most 2 connecting k nodes of H^1_{n-1} to H^0_{n-1} (and vice-versa).

(Proof). We assume that H_n is divided into two subcubes H^0_{n-1} and H^1_{n-1} with respect to the dimension h. Let us call $D = \{\boldsymbol{d}_1, \ldots, \boldsymbol{d}_k\} \subset H^1_{n-1}$ the nodes we want to map into H^0_{n-1}. Each $\boldsymbol{d}_i$ has a degree n and we can construct n paths Q_j $(1 \leq j \leq n)$ which are disjoint except for $\boldsymbol{d}_i$ from $\boldsymbol{d}_i$ to n distinct nodes in H^1_{n-1}:

$$Q_j : \begin{cases} \boldsymbol{d}_i \to \boldsymbol{d}_i \oplus 2^j \to \boldsymbol{d}_i \oplus 2^j \oplus 2^h & (1 \leq j \neq h \leq n) \\ \boldsymbol{d}_i \to \boldsymbol{d}_i \oplus 2^h & (j = h) \end{cases}$$

Since the sum of the number of the nodes in D except for $\boldsymbol{d}_i$ and the number of the faulty nodes is at most $n-1$, that is, $|D \setminus \{\boldsymbol{d}_i\}| + |F| \leq (k-1)+(n-k) = n-1$, there is at least one path in Q_j's that does not include any node in $D \setminus \{\boldsymbol{d}_i\}$ nor F. See Figure 1. Practically, we use the path $\boldsymbol{d}_i \to \boldsymbol{d}_i \oplus 2^h$ of length 1 if it is available. □

3 Fault-Tolerant Node-to-Set Disjoint-Path Hypercube Routing

In this section, we propose an algorithm FTN2S which finds inside an n-cube H_n, disjoint paths connecting one common source node $\boldsymbol{s}$ and k destination nodes $D = \{\boldsymbol{d}_1, \ldots, \boldsymbol{d}_k\}, k \leq n - |F|$, where F is the set of faulty nodes, in $O(kn)$ time complexity. Because of the symmetric structure of hypercubes, we can assume without loss of generality that the source node $\boldsymbol{s}$ is always $00\ldots0$.

The first subsection introduces an extended version of the regular fault-tolerant node-to-node routing in hypercube of Lemma 1. The second subsection focuses on the fault-tolerant node-to-set routing itself, describing the basic algorithm introduced in this paper. A third subsection presents the capability of this algorithm to handle a given optional argument X a set of fault-free neighbors of $\boldsymbol{s}$ the source node, so that each of them is used by one of the paths constructed. Pseudocode of this algorithm is given in Algorithm 1.

3.1 Extended Fault-Tolerant Node-to-Node Routing in Hypercubes

We introduce below an extended version of the regular fault-tolerant node-to-node routing algorithm in hypercube.

Lemma 4. Given a source node $\boldsymbol{s}$, a set of destination nodes D and a set of faulty nodes F, such that $|F| \leq n-1$ and $|D| \leq 2^n - |F|$ (ie. no restriction on the size of D), we can construct one fault-free path between $\boldsymbol{s}$ and one undecided node of D, of length at most $n + \frac{|F|}{2} + 2$ in $O(n)$ time complexity.
(Proof) Let us describe such an algorithm. The main idea is a classical divide-and-conquer approach, reducing the original hypercube H_n into two subcubes of smaller dimension $n-1$, until either one the subcubes is fault-free or only one target remains. During this process, the source node may need to be mapped to the opposite subcube if the algorithm is to be recursively applied to the subcube not containing $\boldsymbol{s}$.

Case 0 - $|F| = 0$

Apply a shortest-path routing algorithm to connect $\boldsymbol{s}$ to the closest destination node.

Case 1 - $|D| = 1$

Apply the regular fault-tolerant node-to-node routing algorithm to connect $\boldsymbol{s}$ to the unique destination node.

Case 2 - Otherwise

Pick one of the n dimensions such that after reducing H_n along this bit into two $(n-1)$-dimensional subcubes H^0_{n-1} and H^1_{n-1}, the following condition hold:

$$F \cap H^1_{n-1} \neq \emptyset$$

Case 2-1 - $H^1_{n-1} \supset D$

Case 2-1-1 - $H^0_{n-1} \cap F = \emptyset$

Map one $\boldsymbol{d} \in H^1_{n-1}$ onto a $\boldsymbol{d}' \in H^0_{n-1}$ using one edge, and if $\boldsymbol{d}' \neq \boldsymbol{s}$, link them by applying a shortest-path routing algorithm inside H^0_{n-1}.

Case 2-1-2 - Otherwise

Map $\boldsymbol{s} \in H^0_{n-1}$ onto a node $\boldsymbol{s}' \in H^1_{n-1}$ with a fault-free path of length at most 2. If $\boldsymbol{s}' \in D$ we are done, otherwise apply this algorithm recursively on H^1_{n-1}.

Case 2-2 - Otherwise (ie. $H^0_{n-1} \cap D \neq \emptyset$)
Apply this algorithm recursively on H^0_{n-1}.

We can always map $\boldsymbol{s} \in H^0_{n-1}$ onto a node $\boldsymbol{s}' \in H^1_{n-1}$ with a fault-free path of length at most 2 since we have initially $|F| < n$, and each reduction puts at least one faulty node into H^1, so decreasing the hypercube dimension by one during a reduction is always done in parallel with moving one faulty node into H^1. Hence there always remain at least 1 fault-free path of maximal length 2 to map $\boldsymbol{s}$ onto $\boldsymbol{s}'$ in H^1.

Finding a dimension which satisfies the two above conditions can be done in $O(n)$ time during a preprocessing task to identify suitable dimensions. Hence the time complexity of this extended fault-tolerant node-to-node algorithm stays $O(n)$, that is the complexity given by Lemma 1.

Regarding path length, we note that there will be at most $\lfloor \frac{|F|}{2} \rfloor$ $(= \rho)$ hypercube reductions performed, hence $\boldsymbol{s}$ will be mapped to $\boldsymbol{s}'$ at most ρ times. Since the regular fault-tolerant node-to-node routing in H_n outputs a path of maximal length $n + 2$, we have for this extended version a length of at most $2\rho + (n - \rho) + 2 \leq n + \frac{|F|}{2} + 2$.

3.2 Fault-Tolerant Node-to-Set Routing in Hypercubes with Restriction

This algorithm follows a recursive approach to perform a fault-tolerant disjoint-path routing inside a hypercube H_n. As explained in Section 2, hypercubes have a recursive structure and we use this property to apply a divide-and-conquer strategy to solve our routing problem. After performing such a reduction along a dimension $0 \leq i \leq n-1$, we obtain two subcubes of lower dimension, we call H^0_{n-1} (abbreviated H^0) the subcube containing $\boldsymbol{s}$ the source node and H^1_{n-1} (abbreviated H^1) the other subcube. The algorithm introduced in this section allows us to specify a set $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{|D|}\}$ of fault-free neighbors of $\boldsymbol{s}$, which must be used by the constructed paths such that every path, excepted one, always starts with a subsequence $\boldsymbol{s} \rightarrow \boldsymbol{x}_i$.

Case 0 - If the hypercube is fault-free, we apply the fault-tolerant node-to-set routing algorithm in hypercube of Lemma 2 to connect $\boldsymbol{s}$ and all the destination nodes plus, as fake destination nodes, the neighbours of $\boldsymbol{s}$ not in X. Finally we discard the paths not starting with a sequence $\boldsymbol{s} \rightarrow \boldsymbol{x}_i$.

Case 1 - Otherwise, we choose a dimension such that after reducing H_n along this bit, we have at least one faulty node into H^1_{n-1}. Then we distinguish two cases:

Case 1.1 - If the neighbour of $\boldsymbol{s}$ which is in H^1 is not an element of X, then we only perform a back-mapping operation to route all the destination nodes of H^1 back into H^0, with fault-free paths of length at most 2.

Case 1.2 - Otherwise, that is the neighbour of $\boldsymbol{s}$ in H^1 is an element of X, we connect it to one destination node of H^1 using the extended fault-tolerant node-to-node routing algorithm of Lemma 4, specifying as destinations all the destination nodes of H^1 plus their fault-free neighbours, and finally we map back into H^0 all the destinations nodes of H^1 remaining (ie. not reached by the extended node-to-node algorithm called previously). If there is no destination node in H^1 we route one of H^0 into H^1 in at most 2 edges.

Finally we apply recursively this routine to the subcube H^0_{n-1}, with the set of the destination nodes now containing destination nodes present in H^0 plus the images of the destination nodes mapped back from H^1_{n-1}.

4 Correctness and Complexities

First, let us recall that a hypercube reduction uses exactly one bit of the n bits initially available in node addresses (assume we are working in H_n).

For each reduction performed, the algorithm previously described in 3.2 always put at least one faulty node inside Ht before the recursive call, which as a result decreases the number of faulty nodes for the next reductions by at least one. That's why the algorithm will perform at most F hypercube reductions. So for one reduction, $|F|$ decreases by at least one ($|D|$ may also decrease). Now considering the initial condition $|D|+|F| \leq n$, and a fortiori $|F| \leq n$, we see that the n bits of node addresses, or in other words n reductions, suffice to construct the $|D|$ disjoint paths.

Let us now analyze the maximal path length returned and the time complexity of the algorithm described in 3.2.

First let us recall that the extended fault-tolerant node-to-node routing algorithm of Lemma 4 returns a path of length at most $n + \frac{|F|}{2} + 2$ in $O(n)$ time complexity. We also recall that we perform at most $k = |F|$ reductions, that is a destination node can be mapped back at most k times. And finally, in the worst case, we will call the extended fault-tolerant node-to-node routing algorithm if we cannot reach a fault-free hypercube.

Hence we can express the maximal path length with the following recursive expression. $L(n, k)$ represents the maximal path length when the algorithm is applied onto a n-cube with k faulty nodes.

$$\begin{aligned} L(n,0) &= n + 2 \\ L(n,k) &= n + \frac{k}{2} + 2 \\ L(n,k) &= 2 + L(n-1, k-1) \end{aligned}$$

From this discussion, we understand that the worst case occurs when a destination node is mapped back during all the $k-1$ hypercube reductions and is finally

Algorithm 1. FTN2S(H_n, $\boldsymbol{s}$, $D = \{\boldsymbol{d}_1, \ldots, \boldsymbol{d}_k\}$, F [, $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$])

```
/* $X$ is an optional array of $k$ fault-free neighbours of $\boldsymbol{s}$. */

if $X$ undefined then /* Initialization of $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$ needed */
   $i := 0$
   while $|X| < k$ do
      $\boldsymbol{x}_{cdt} := \boldsymbol{s} \oplus 2^i$
      if $\boldsymbol{x}_{cdt} \notin F$ then
         $X := X \cup \{\boldsymbol{x}_{cdt}\}$
      end if
      $i := i + 1$
   end while
end if

if $F = \emptyset$ then /* Case 0 */
   Apply the fault-tolerant hypercube node-to-set routing algorithm of Lemma 2 to
   connect $\boldsymbol{s}$ to $D$, marking the neighbours of $\boldsymbol{s}$ not in $X$ as faulty.

else /* Case 1 */
   Reduce $H_n$ using the dimension $i$ such that $H^1_{n-1} \cap F \neq \emptyset$;

   if $\boldsymbol{s}' \notin X$ then
      for all $\boldsymbol{d}_i \in D \cap H^1_{n-1}$ do /* back mapping */
         Route $\boldsymbol{d}_i \in H^1_{n-1}$ to $\boldsymbol{d}'_i \in H^0_{n-1}$ in at most 2 edges such that $\boldsymbol{d}'_i \notin F \cup D$;
         $D' := D' \cup \{\boldsymbol{d}'_i\}$
      end for;

   else /* $\boldsymbol{s}' \in X$ */
      if $D \cap H^1 = \emptyset$ then
         Route one $\boldsymbol{d} \in H^0_{n-1}$ onto a node $\boldsymbol{d}' \in H^1_{n-1}$ in at most 2 edges.
      end if
      Connect $\boldsymbol{s}'$ and $\boldsymbol{d}' \in H^1_{n-1}$ with extended fault-tolerant node-to-node routing
      inside $H^1_{n-1}$.
      for all $\boldsymbol{d}_i \in D \cap H^1_{n-1}, \boldsymbol{d}_i \neq \boldsymbol{d}'$ do /* back mapping */
         Route $\boldsymbol{d}_i \in H^1_{n-1}$ to $\boldsymbol{d}'_i \in H^0_{n-1}$ in at most 2 edges such that $\boldsymbol{d}'_i \notin F \cup D$;
         $D' := D' \cup \{\boldsymbol{d}'_i\}$
      end for;
   end if

   FTN2S($H^0_{n-1}$, $\boldsymbol{s}$, $(D \cap H^0_{n-1}) \cup D'$, $F \cap H^0_{n-1}$, $X$)
end if
```

reached by the extended node-to-node routing algorithm after being routed inside H^1 in at most 2 edges (assuming $\boldsymbol{s}' \in X$ after this reduction). Hence we obtain the following maximal path length.

$$2(k-1) + (n - (k-1) + \frac{1}{2} + 2) + 2$$
$$\leq n + k + 4$$

Regarding time complexity, we understand that building each path requires a linear time complexity $O(n)$. Therefore the algorithm described in 3.2 has a total time complexity of $O(n^2)$.

5 Example

Let $n = 8$, that is we perform routing inside H_8. Let the source node be $\boldsymbol{s} = 0000\ 0000$. Let the destination, faulty and restriction nodes be as follow.

$$D = \{\ \boldsymbol{d}_1 = 0101\ 1100,\ \boldsymbol{d}_2 = 1010\ 1010,\ \boldsymbol{d}_3 = 0101\ 0101,\ \boldsymbol{d}_4 = 0011\ 1100\ \}$$
$$F = \{\ \boldsymbol{f}_1 = 1011\ 1111,\ \boldsymbol{f}_2 = 0001\ 0100,\ \boldsymbol{f}_3 = 1000\ 0000,\ \boldsymbol{f}_4 = 1111\ 0000\ \}$$
$$X = \{\ \boldsymbol{x}_1 = 0100\ 0000,\ \boldsymbol{x}_2 = 0001\ 0000,\ \boldsymbol{x}_3 = 0000\ 0100,\ \boldsymbol{x}_4 = 0000\ 0001\ \}$$

The routing algorithm applied to this example is illustrated in Table 1.

Table 1. Routing example inside H_8

	Dimension	$\in H^0_{n-1}$	$\in H^1_{n-1}$	Path P_i constructed
1st red.	0100 0000 ($\boldsymbol{x}_1$)	$\boldsymbol{d}_2, \boldsymbol{d}_4, \boldsymbol{f}_1, \boldsymbol{f}_2, \boldsymbol{f}_3$	$\boldsymbol{d}_1, \boldsymbol{d}_3, \boldsymbol{f}_4$	$\boldsymbol{s} \rightarrow \boldsymbol{x}_1 \rightarrow \ldots \rightarrow \boldsymbol{d}_1$ **
map back $\boldsymbol{d}_3 \in H^1_{n-1}$ to $\boldsymbol{d}'_3 \in H^0_{n-1}$				
2nd red.	0001 0000 ($\boldsymbol{x}_2$)	$\boldsymbol{d}_2, \boldsymbol{f}_3$	$\boldsymbol{d}_4, \boldsymbol{d}'_3, \boldsymbol{f}_1, \boldsymbol{f}_2$	$\boldsymbol{s} \rightarrow \boldsymbol{x}_2 \rightarrow \ldots \rightarrow \boldsymbol{d}_4$ **
map back $\boldsymbol{d}'_3 \in H^1_{n-1}$ to $\boldsymbol{d}''_3 \in H^0_{n-1}$				
3rd red.	1000 0000 ($\boldsymbol{x}_3$)	$\boldsymbol{d}''_3$	$\boldsymbol{d}_2, \boldsymbol{f}_3$	$\boldsymbol{s} \rightarrow \boldsymbol{x}_3 \rightarrow \ldots \rightarrow \boldsymbol{d}_2$ *
H^0_{n-1} is fault-free				$\boldsymbol{s} \rightarrow \boldsymbol{x}_4 \rightarrow \ldots \rightarrow \boldsymbol{d}''_3 \rightarrow \boldsymbol{d}'_3 \rightarrow \boldsymbol{d}_3$

* Fault-tolerant node-to-node routing (Lemma 1).
** Extended fault-tolerant node-to-node routing (Lemma 4).

6 Conclusion

We first described in this paper an extended fault-tolerant node-to-node routing algorithm in hypercubes, connecting one source node to an undecided destination node member of a set of candidates destination nodes, creating a path of length at most $n + \frac{|F|}{2} + 2$ in $O(n)$ time complexity. We then introduced a fault-tolerant routing algorithm solving the node-to-set disjoint-path problem in hypercubes, with the ability to specify the neighbors of $\boldsymbol{s}$ to use. This algorithm finds paths of length at most $n + k + 4$ in $O(n^2)$ time complexity.

Solving routing problems with a similar restriction on neighbors of $\boldsymbol{s}$ inside other recursive interconnection networks should be considered for future work.

Acknowledgment

This study is partly supported by the Fund for Promoting Research on Symbiotic Information Technology of Ministry of Education, Culture, Sports, Science and Technology (MEXT) Japan. It is also partly supported by a Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science (JSPS) under Grant No. 19500022.

References

1. White, A., Grice, D. (IBM): Roadrunner: Science, Cell and a Petaflop/s. In: International Supercomputing Conference (2008)
2. EE Times: U.S. taps IBM for 20 petaflops computer (2009), http://eetimes.com/news/design/showArticle.jhtml?articleID=213000489
3. Vanvoorst, B., Seidel, S., Barscz, E.: Workload of an ipsc/860. In: Proc. Scalable High-Performance Computing Conf., pp. 221–228 (1994)
4. SGI: Origin2000 Rackmount Owner's Guide, 007-3456-003 (1997), http://techpubs.sgi.com/
5. Saad, Y., Schultz, M.H.: Topological Properties of Hypercubes. IEEE Transactions on Computers 37(7), 867–872 (1988)
6. Gu, Q., Peng, S.: An Efficient Algorithm for Node-to-node Routing in Hypercubes with Faulty Clusters. The Computer Journal 39(1), 14–19 (1996)
7. Rabin, M.A.: Efficient dispersal of information for security. Journal of ACM 36(2), 335–348 (1989)
8. Kaneko, K.: An algorithm for node-to-set disjoint paths problem in burnt pancake graphs. IEICE Trans. Inf. and Systems E86-D(12), 2588–2594 (2003)
9. Gargano, L., Vaccaro, U., Vozella, A.: Fault tolerant routing in the star and pancake interconnection networks. Inf. Processing Letters 45(6), 315–320 (1993)
10. Gu, Q., Peng, S.: Node-to-set and Set-to-set Cluster Fault Tolerant Routing in Hypercubes. Parallel Computing 24(8), 1245–1261 (1998)
11. Latifi, S., Ko, H., Srimani, P.: Node-to-set Vertex Disjoint Paths in Hypercube Networks. Technical Report CS-98-107
12. Bossard, A., Kaneko, K., Peng, S.: Node-to-set Disjoint-path Routing Metacube. In: Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (2009)
13. Bossard, A., Kaneko, K., Peng, S.: A Node-to-set Disjoint-path Routing Algorithm in Metacube. In: Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms and Networks (2009)
14. Bossard, A., Kaneko, K., Peng, S.: Node-to-set Disjoint Paths Problem in Perfect Hierarchical Hypercubes. In: Proceedings of the 9th Parallel and Distributed Computing and Networks (2010)

AirScope: A Micro-scale Urban Air Quality Management System

Jung-Hun Woo[1], HyungSeok Kim[2], Sang Boem Lim[1,*], Jae-Jin Kim[3], Jonghyun Lee[1], Rina Ryoo[1], and Hansoo Kim[1]

[1] Department of Advanced Technology Fusion at Konkuk University
Seoul, Korea
{jwoo,sblim,lejohy}@konkuk.ac.kr, joinrina814@naver.com, lunahife@gmail.com

[2] Department of Internet & Multimedia Engineering, Konkuk University
Seoul, Korea
hyuskim@konkuk.ac.kr

[3] Department of Environmental Atmospheric Sciences, Pukyong National University
Busan, Korea
jjkim@pknu.ac.kr

Abstract. Monitoring air quality for daily life gets more and more attention. Especially in modern urban environment with high skylines and unexpected pollution events, it is important to have micro-scale monitoring in addition to traditional monitoring methods. We propose a micro-scale modeling system as well as a micro-scale air quality monitoring system, which comprises as a micro-scale air quality management system, named AirScope. AirScope consists of CFD-based air quality modeling, USN-based sensor monitoring, and multi-modal interaction platform. In this paper, we present a brief overview of AirScope and several aspects of constructed initial indoor test environment with a few validity tests. The proposed system will be extended to an outdoor real-world testbed with most of modern urban elements.

1 Background

The urban environment is where an increasing share of the world's population resides, where most commercial energy is consumed, and where the impacts of pollution are felt the most. Rapid economic growth in urban Asia has attracted millions of rural residents to metropolitan environments [1]. Changing standards of living in the urban centers have fueled increasing energy demand often associated with unchecked emissions from automobiles, domestic heating, and small-scale industries. Asian urban centers, prone to air pollution, incur hundreds of millions of dollars in health and economic damages [2].

[*]Corresponding author.

Presently, the urban air pollution problems in Asia are continuing to increase and air pollutants originating from urban regions are recognized as increasing sources of regional- and global-scale pollution [3].

Seoul Metropolitan Area (SMA), which is located center of Korean Peninsula, has been suffering from such air pollution problems. Regional air pollution transport from China as well as local pollution sources, such as thermal power plants and mobile sources, has also become a major contributor to increasing human health effects in the urban environments of SMA. In addition to those problems, micro-scale air pollution, which lasts for minutes to hours and covers several kilometers, exist in SMA. The examples of micro-scale of air pollution are fugitive dust from construction fields or accidental release of hazardous pollutants. Unlike regional to local scale air pollution, which is traditionally well established research areas in SMA, micro-scale air pollution has not been researched very much. One of the major reasons is that monitoring and modeling methods developed for bigger scales are not appropriate for micro-scale phenomena. The needs for more detailed information for public services, however, have been increasing as the needs for real time emission estimate is increasing and the urban building geography is being more complex. In this reason, micro-scale air monitoring and modeling system need to be developed to improve our level of understanding about the micro-scale air pollution, hence provide better and detailed public information services.

2 Related Work

CitySense [4] is an open, urban-scale sensor network testbed, focused to establish in Cambridge, Massachusetts. It has been developed by researchers at Harvard University and BNN Technologies. Their goal is to develop and adjust innovative wireless monitoring system using powered Ethernet (or USB) with an XML data interface. There will be at least 100 nodes embedded Linux having wireless function and various sensor modules for temperature, traffic, parking conditions, etc. Sensors will be installed on light poles, private or public buildings to monitor required data. They intend to an open testbed, so that researchers from all over the world can access monitoring data and evaluate wireless networking. They also provide overall information from sensors through website; http://www.citysense.net.

Anthony Steed, et al. [5] studies about urban pollution, focusing carbon monoxide (CO) with multiple mobile sensors which are equipped with GPS receiver. Sensors are carried by pedestrians or mounted on vehicles to see the distributed CO concentration along and cross the streets. In this way, they try to create maps of pollution variation at each street, so that it makes it possible to understand urban air pollution, especially CO. They collect sensor data through network to desktop, but a wireless networking is not developed yet. So transferring data is possible, only when the device is synchronized with a desktop. They also have visualization plan and grid service. In the data-logging mode, it can provide synchronized database services and 2D, 3D grid services for visualization.

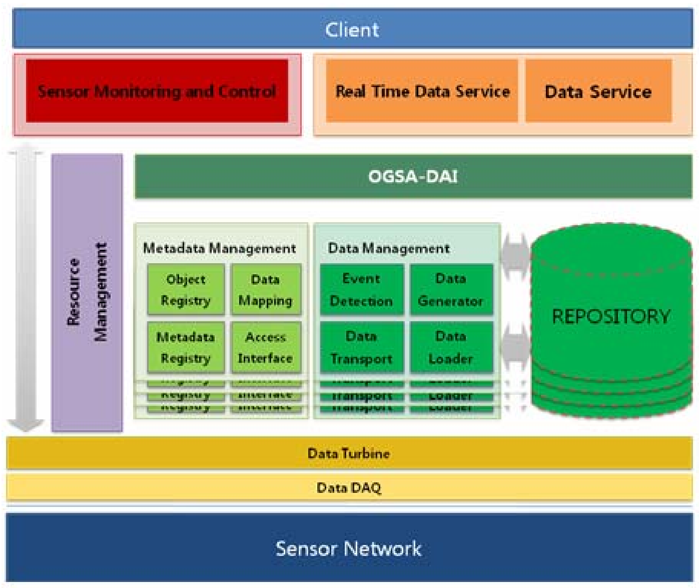


Fig. 1. AirScope Architecture

3 Micro-scale Air Quality Management System

Micro-scale air quality management system (named AirScope) is a need-based monitoring and modeling system to meet public needs for air quality information, and complement the limitations of existing monitoring and modeling systems. The proposed system is constructed upon a sensor network to monitor micro-scale air quality. In addition to sensor network, the proposed system consists of middleware, CFD-based air quality modeling, GIS-based 3D virtual environment model, and air quality data management system. The system also provides visualization and interaction platform based on virtual reality (VR) technology.

Architecture of AirScope system is shown in Figure 1. This system consists of three layers. First layer is sensor network layer. In this layer, data acquisition (DAQ) is collect data from sensors and load data to DataTurbine [6]. Second layer is middleware layer that provides functionalities like management of data and resources. Last layer is used to present data to the users in different ways. By using data stored in repository or using real time streaming data, we provide sensor monitoring and control service, real time data service, history data services, high-capacity data process and distributed query process. We also provide CFD-based Air Quality Modeling, 3D GIS, and VR services.

Each middleware components and services will be described in following sections.

3.1 3D GIS for Air Quality Monitoring and Modeling

Our system can be connected with GIS to visualize detailed data in a given domain. By establishing 3D building GIS, we can see the overall array of buildings in a micro-scale domain. So we can understand dispersion of air pollutants and complex wind path due to building blocking effects. To set up a sensor deployment plan, we generated grid map of 100m and 200m sizes on top of the 2D digital map in consideration of the Zigbee sensor's wireless network range. The resulting structure GIS with gird maps help us planning a sensors deployment, considering landscape and structures. And then, we establish 3D GIS map with height information based on 2D map to attach sensors in and near Konkuk University. Established 3D GIS map is shown in Figure 2.

3D GIS map can be used as an input data to Computational fluids dynamics (CFD) air quality modeling and visualization of CFD output data. In CFD modeling, results will be employed on 2D and 3D GIS maps with real dimensions of building structures. CFD modeling output, therefore, will be used to study the dispersion of pollutants from a more realistic urban area [1].

3.2 3D GIS for Air Quality Monitoring and Modeling

Air quality in micro-scale usually varies dynamically with short time and small space. Understanding air quality which depends solely on the monitoring is difficult and would not be reliable enough. The air quality modeling technique, therefore, should be very beneficial to incorporate. A Computational Fluids Dynamics (CFD) air quality model is our model of choice. Many air quality models are developed for the larger scales; hence they treat a complex building geography as simple parameters, such as roughness length scale. We're using a high-resolution CFD-based air quality modeling, which can represents a complex building geography, to understand and predict air quality in more realistic way, so that we can provide predicted air quality through website, mobile devices and so on. Also, modeling results can be compared with USN-based monitoring data. In this way, modeling devices will be improved and we can provide more accurate and detailed air quality information to the public.

The computational fluid dynamics (CFD) model used in this study considers a three-dimensional, non-hydrostatic, non-rotating, incompressible airflow system. The model includes the k-ε turbulence closure scheme based on the renormalization group (RNG) theory and employs wall functions at the solid surfaces.

The tentative CFD model domain ranges is show in Figure 2. The horizontal grid interval is 10 m in the x-direction and 10 m in the y-direction and the horizontal grid dimension is 200×200. In the vertical, a non-uniform grid system with 62 layers will be employed, in which the vertical grid interval is uniform with 5 m up to the 33rd layer, increases with an expansion ratio of 1.1 from the 34th to the 41st layer, and is then uniform with 10.72 m from the 42nd to the 62nd layer. The domain size is ~2000 m in the x-direction, ~2000 m in the y-direction, and ~440 m in the z-direction. The time step used is 1 second.

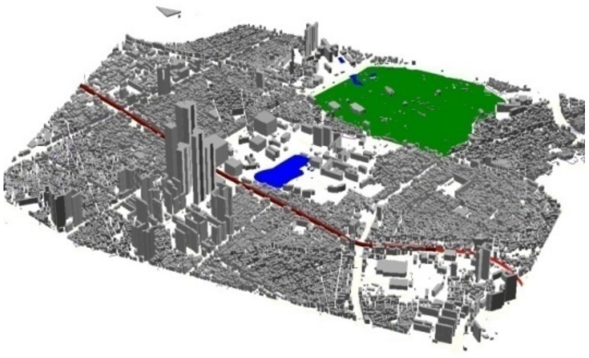


Fig. 2. 3-Dimensional Buildings GIS Map near Konkuk University

3.3 Air Quality Data Management

Environment monitoring system and analysis system are need to various sensors which will produce different types of data. This will cause the problem when we are trying to share data among different systems and scientists. To resolve this problem we need abstract to standard sensor model. Fortunately Open Geospatial Consortium (OGC) developed sensor modeling language called SensorML [7] and Observation and Measurement (O&M) [8] for observation and measurement data modeling. Our system uses sensorML and O&M for standard sensor modeling and observation and measurement data modeling.

Figure 5 describes USN sensor data model of AirScope System. This data model consists of SENSOR_DATA, SENSORML, O&M, SENSOR_STAUS and SENSORML_INDEX. SENSOR_DATA describes the data that produced by the sensors. This data is used directly at real time monitoring system and is encoded into O&M and SENSOR_STATUS which is used for Resource management. O&M contains real-data value such as TIME_STAMP, RESULT and reference to SENSORML. SENSORML includes meta-information of O&M. SENSORML and O&M are XML documents and managed Berkeley XML DB. SENSORML_INDEX has information about SENSORML_ID, O&M_ID and SENSOR_ID. This table provides relation of data.

3.4 Air Quality Visualization

The visualization system transforms the sensor data into intuitive visual form in multi-modal way. Users can get overview of multiple sensor values by navigating into the virtual world. The direct manipulation allows users to remotely control the sensor itself through VR interfaces.

The visualization system is composed of three modules devoted to network management, resource management, and visualization.

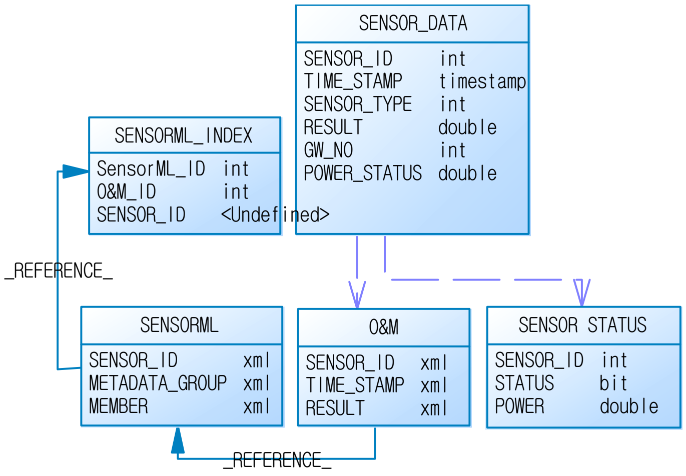


Fig. 3. USN sensor data conceptual design

Network management module treats all of the on-line transactions between visualization system and GSN. It encodes sensor manipulation data which will be sent to GSN or decodes retrieved data.

All of the sensor data that visualization system uses are managed by the resource management module. The module loads sensor data from GSN. It processes user actions along with sensor data updates to and from GSN. Data stored in resource management module are transformed in to visualization forms by various methods according to its usage.

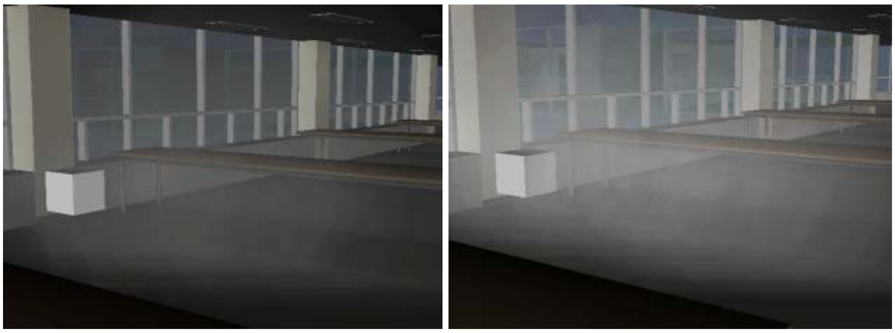


Fig. 4. Sensor data representation using particle system: White box is locator of the sensor. Density and transparency of particles will be changed related to data from the sensor.

Visualization module presents sensor data to users using VR-based multi-modal framework. Users not only can explore the 3D virtual world to get sensor information, but also can perform direct manipulation to the sensor itself. To present air quality data effectively for casual users, we selected a few different methods. First of all, a single element of sensed data is represented as a particle with density and transparency parameters, which are related to sensor data of air quality. By changing parameters of particles, we can represent different air conditions like infusibility, density, fluidity and so on. Also, we use graphs and tables are available for experts who want to get detailed data. Isosurface rendering will be also employed. The multi-modal framework supports visual, haptic and auditory rendering for those data.

User Interaction module is a part of the visualization module. It deals with inputs from users and returns result through visualization system. As a default device set, users can explore virtual space and can manipulate sensors with those devices. In specialized environment, like large-scale multi-display, we are developing devices with gesture-based interaction with haptic feedback or wand-type interfaces.

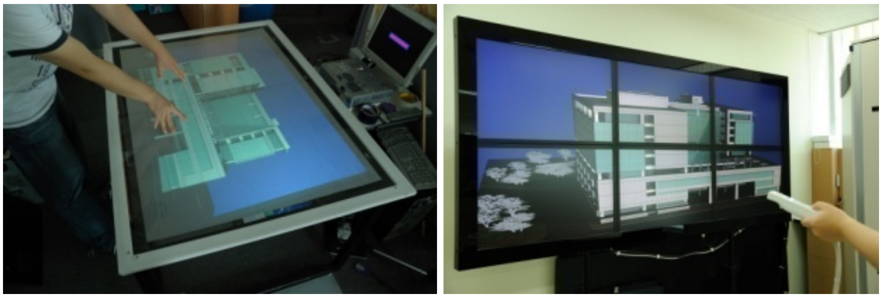


Fig. 5. Interaction example with table top and wand type interface

4 Conclusions and Future Work

In this paper, we present a framework for micro-scale air quality monitoring and sensor management system. The air quality is being simulated by CFD-based air quality model and will be validated with USN-based sensor data and GIS model of the environment.

The developed system is validated in a preliminary test environment. The test result shows that it is feasible to adopt sensor networks for micro-scale monitoring.

For extended real-world testbed in the future, 2km × 2km domain around Konkuk University campus Complex (KUC) is selected. KUC is composed of residential & commercial area, hospital, university campus, subway stations, and so on. KUC is a densely-packed area, which has many facilities that intensive monitoring is needed.

Initial sensor placement plan was made in consideration of building complexity and wireless communication requirements. Sensors will be installed major buildings inside of campus. The effect of water (lake), students' activities, and driver training institute can be monitored. Enhancements on the core modules will also be investigated such as enhanced interaction devices, mobile sensors, display devices, and etc. Finally, we will also build web-based information portal to be used by citizens whenever the information is needed.

Acknowledgement

This research was supported by the Seoul R&BD Program(GS070167).

References

1. Guttikunda, S.K., Carmichael, G.R., Calori, G., Eck, C., Woo, J.-H.: The contribution of megacities to regional sulfur pollution in Asia. In: Atmospheric Environment 37 (2003)
2. OECD, Ancillary benefits and costs of greenhouse gas mitigation. In: Proceedings of an IPCC Co-sponsored Workshop, Washington, DC, USA, March 27-29 (2000)
3. Streets, D.G., Carmichael, G.R., Amann, M., Arndt, R.L.: Energy consumption and acid deposition in Northeast Asia. Ambio 28, 135 (1999)
4. CitySense Homepage, `http://www.citysense.net`
5. Steed, A., Spinello, S., Croxford, B., Greenhalgh, C.: e-Science in the Streets: Urban Pollution Monitoring. In: Proceedings of the 2nd UK e-Science All Hands Meeting (2003)
6. DataTurbine homepage, `http://www.dataturbine.org`
7. Open GIS® Sensor Model Language (SensorML) Implementation Specification, OGC Geospatial Consortium INC. (2007)
8. Open GIS® Observation and Measurements (O&M) Implementation Specification, OGC Geospatial Consortium INC. (2007)

Design of a Slot Assignment Scheme for Link Error Distribution on Wireless Grid Networks*

Junghoon Lee[1], Seong Baeg Kim[2,**], and Mikyung Kang[3]

[1] Dept. of Computer Science and Statistics, [2] Dept. of Computer Education,
Jeju National University, 690-756, Jeju Do, Republic of Korea
[3] University of Southern California - Information Sciences Institute, VA22203, USA
jhlee@jejunu.ac.kr, sbkim@jejunu.ac.kr, mkkang@east.isi.edu

Abstract. This paper designs and measures the performance of an efficient routing scheme and corresponding slot assignment schedule capable of efficiently handling the difference in the error rate of each link on the wireless mesh network. Targeting at the grid topology common in most modern cities, a split-merge operation masks the channel error by making an intermediary node receive a message simultaneously in two directions and then integrating a secondary route, required to have the same length with the primary route, to the communication schedule. Adding virtual links from level 1 to level 3 to the 3×3 grid and estimating their error rates based on those of the surrounding links, the proposed scheme applies the well-known shortest path algorithm to decide the best route and also to finalize the complete a slot assignment for the WirelessHART protocol. The performance measurement result obtained by simulation using a discrete event scheduler demonstrates that the proposed scheme can find the promising path, improving the delivery ratio by up to 5.21 %.

1 Introduction

The vehicular network can have a great variety due to a bunch of available wireless communication technologies and widely open possibilities to organize the network. Some prospective communication technologies include IEEE 802.11, DSRC (Dedicated Short Range Communication), Zigbee, and WirelessHART [1,2]. Undoubtedly, a new method keeps appearing. There can be a lot of communication styles according to the system goal of the vehicular network and

* 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-2010-(C1090-1011-0009)).

Following are also results of a study on the "Human Resource Development Center for Economic Region Leading Industry" Project, supported by the Ministry of Education, Science & Technology(MEST) and the National Research Foundation of Korea(NRF).

** Corresponding author.

its application, ranging from the ad-hoc mode for instant accident information propagation to the infrastructure-based mode for ubiquitous real-time vehicle tracking. Here, vehicles do not solely constitute the vehicular network, which includes many static elements such as gateways, traffic lights, and wireless nodes installed in the fixed locations. In addition to responding to the request from the vehicles passing by, each static node needs to communicate with each other [3].

The static gateway is placed somewhere in the street to meet a given goal such as vehicle connectivity or driving safety. The traffic light can desirably accommodate a communication interface as it has sufficient power provision and is generally installed at a secure place. It is very rare to see a traffic light fail in our everyday life. With the stable connection on the traffic light network, it is also possible to implement a monitor-and-control operation when the vehicular network includes sensors and actuators. Practically, a lot of devices such as speed detectors, pollution meters, and a traffic signal controllers, can be systematically integrated into the vehicular network. Vehicles can also carry a sensor device and report the collected sensor data to a static node when connected via ad-hoc communication. The monitor-and-control application runs on top of a specific network topology using a deterministic and robust wireless communication protocol. Particularly, in a road network of urban areas, where each crossing has a traffic light or signal, the traffic light network has grid topology [4].

In the mean time, the WirelessHART standard provides a robust wireless protocol for various process control applications [5]. For the sake of overcoming the transient instability of wireless channels, WirelessHART puts a special emphasis on reliability by mesh networking, channel hopping, and time-synchronized messaging. This protocol has been implemented and is about to be released to the market soon [6]. Moreover, the WirelessHART protocol can be extended to enhance reliability for the grid network. The split-merge operation makes it possible for a sender node to send a message in two directions in a single time slot as well as a receiver node to receive from one of the possible intermediary nodes [7]. Its advantage can be much improved with an efficient slot assignment scheme considering the split-merge operation in grid networks. In this regard, this paper is to address how to define and add a virtual link to the traffic light network to apply the existing routing schemes.

This papers is organized as follows: After issuing the problem in Section 1, Section 2 introduces the background of this paper focusing on the WirelessHART protocol and reviews the split-merge operation. Section 3 proposes the virtual link management scheme for the grid style traffic light network. Section 4 shows the simulation result, and Section 5 summarizes and concludes this paper.

2 Related Work

Communication on the grid topology is extensively studied in the wireless mesh network area, and its main concern is efficient routing inside the grid to achieve various optimization goals, like minimizing end-to-end delays, ensuring fairness among nodes, and minimizing power consumption [8]. The channel assignment problem is NP-hard [9], so even for the case of no interference in the network, the optimal schedule cannot be computed in polynomial time. Many greedy algorithms are applied for the static channel assignment, like the well-known coloring algorithm [10]. That is, colors or time slots must be assigned such that no two adjacent nodes in the network have the same color. For slot-based access, most schemes focus on how to assign a color on the diverse topology and how to cope with the topology change. Some of them addresses how to locally decide the route when just partial topology information is available. Anyway, existing schemes have not dealt with the dynamic channel selection within a single slot and the subsidiary slot allocation.

N. Chang considered transmission scheduling based on optimal channel probing in a multichannel system, where each channel state is associated with a probability of transmission success and the sender just has partial information [11]. To compensate for the overhead and resource consumption of the channel probing procedure, this work proposed a strategy that determines which channel to probe based on a statistical channel model. Using this estimation, the sender or scheduler can pick the channel to use for transmission. Even though the concept of channel probing is very prospective, this scheme only focuses on the dynamic operation of communication procedure, so it is not suitable for process control. Moreover, the network access is not predictable, as it does not consider the route length but just the probabilistic estimation on successful transmissions in the selection of an available channel.

A slot management on WirelessHART was considered along with a mathematical framework in terms of modeling and analysis of multi-hop communication networks [12]. This model allows us to analyze the effect of scheduling, routing, control decision, and network topology. In this model, each node has at least two neighbor choices to route a packet for any destination nodes. Hence, the time slot schedule must explicitly have two paths for each source and destination pair. That is, regardless of whether the first path successfully transmits a message, the secondary route redundantly delivers the same message. This scheme can integrate any style of an alternative route such as a node-disjoint or link-disjoint path, but bandwidth waste is unavoidable and slot allocation can get too complex.

Lee et al. proposed a channel selection scheme based on the CCA result in the preliminary version of this paper [7]. The idea of CCA-based channel switch is sustained to this paper. However, this approach restrictively assumed that the slot error rate of each link is the same and the split-merge operation, which will be described later, is simply placed to the rectangle closer to the destination. As contrast, this paper generalizes to the case that each channel has its own slot error rate and addresses how to apply the channel switch operation in each control message path, along with extensive performance analysis results.

3 WirelessHART and Traffic Light Network

The WirelessHART standard is defined over the IEEE 802.15.4 GHz radioband physical link, allowing up to 16 frequency channels spaced by 5 MHz guard band [2]. The link layer provides a deterministic slot-based access mechanism on top of the time synchronization function carried out continuously during the whole network operation time [6]. According to the specification, the size of a single time slot is 10 ms, and to meet the robustness requirement of industrial applications, a central controller node coordinates routing and communication schedules. For more reliable communication, CCA (Clear Channel Assessment) before each transmission and channel blacklisting can also be used to avoid specific area of interference and also to minimize interference to others. Here, channel probing can be accomplished by CCA, RTS/CTS handshaking, and so on [13].

The traffic light network looks like a grid network in modern cities, as each traffic light node is placed at each crossing of the Manhattan-like road network, as shown in Figure 1(a) [4]. Each node can exchange messages directly with its vertical and horizontal neighbors. Two nodes in the diagonal of a rectangle do not have a direct connection, as there may be obstacles like a tall building that blocks the wireless transmission. In this network, the central controller is assumed to be located at the fringe of rectangular area, for this architecture makes the determination of the communication schedule simple and systematic. In Figure 1(a), $N00$ is the controller node. It can be generalized by the grid partition illustrated in Figure 1(b), where each of 4 quadrants can be mapped or transformed to the network shown in Figure 1(a) by making the controller be placed at the left top corner.

In WirelessHART, if the channel status is not good, the sender simply discards the transmission, wasting a slot time and possibly making the subsequent transmission schedule quite complex. In case the sender can attempt on alternative

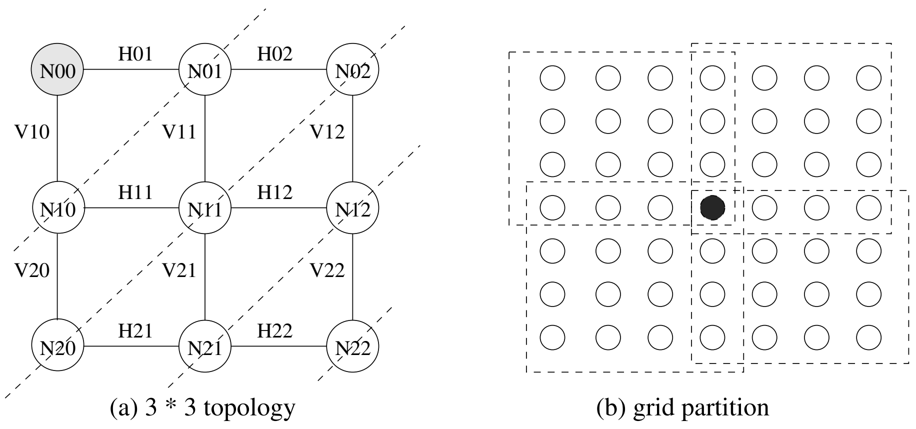


Fig. 1. Traffic light network

route in the same slot, the slot waste can be saved. To integrate an alternative path to the communication schedule, their lengths must be same. The main idea of the split-merge operation can be described by an example network of Figure 1(a). In this figure, consider the transmission from $N00$ to $N11$ which has two 2-hop paths, namely, $N00 \rightarrow N10 \rightarrow N11$, and $N00 \rightarrow N01 \rightarrow N11$. If the network manager selects the first as the primary route and allocates the slots, the schedule will include ($N00 \rightarrow N10$) and ($N10 \rightarrow N11$) at times slots, say i and $i+1$, respectively. For this communication to be successful, $V10$ at slot i and $H11$ at slot $i+1$ should be both clear. Otherwise, the transmission will fail.

Our previous paper has proposed the split-merge operation for better reliable communication [7]. The sender, $N00$, senses a channel status for $V10$ at slot i. If it is clear, it sends to $N10$ according to the original schedule. Otherwise, instead of discarding the transmission, it sends to $N01$ after switching channels. Here, $N01$ (the receiver on the alternative path) as well as $N10$ (the receiver on the primary path) must listen to the channel simultaneously at slot i, and this is the overhead cost for enhanced reliability. At slot $i+1$, either $N10$ or $N01$ sends the packet to $N11$. $N01$ on the secondary route must send after a small delay $TsRxOffset$, which is specified in the original standard and can include one channel switch time. $N11$ first tries to receive from $N10$ on the primary route. If the packet arrives, it receives as scheduled. Otherwise, namely, the node switches channel to $H01$ on the secondary route. After all, the path is split and merged over each rectangle. However, this scheme didn't consider where to put the split-merge operation along the path, as it assume that every link has the same link quality.

4 Routing Scheme

In the control loop scenario, traffic goes from and to $N00$. Namely, each node sends and receives a message to and from the controller node once in the period specified by the system requirement. Even if it is desirable to take the route which has the minimum number of hops to the destination, another detour can have advantage in terms of delivery ratio and transmission delay. If we consider the case of $N21$, it has many paths to $N00$. A path is to be selected by the scheduler mainly periodically according to the change in the error characteristics of each link. Each link has its own error characteristics due to different power level, obstacle distribution, and so on. The change of link error characteristics can be estimated in many ways [11], but we assume that the probing result is always correct, as the correctness of channel probing is not our concern. The routing scheme is designed for 3×3 grid, as WirelessHART restricts the maximum number of hops to 4 and there are so many virtual links for a larger grid.

To begin with, we denote the error rate of link L by $E(L)$. Intuitively, L may be the horizontal or vertical link. For example, $E(V21)$ denotes the error rate of the link from $N21$ to $N11$ and it has the same meaning with $E(N21 \rightarrow N11)$. However, L can be extended to represent a virtual link which consists of both normal and virtual links. That is, even though there is no direct connection between two nodes in diagonal lines, the two-hop link between them can be considered to be a single link due to the split-merge operation. The split-merge operation works either for the uplink from Nij to $Ni-1 \cdot j-1$ or for the downlink from Nij to $Ni+1 \cdot j+1$. The transmission fails only of two paths are simultaneously not good in the split-merge operation. As a result, the error rate of this virtual link can be estimated as follows:

$$\begin{aligned}E(Nij \to Ni-1 \cdot j-1) = 1 &- (1-E(Hij))(1-E(Vi \cdot j-1)) \\ &- (1-E(Vij))(1-E(Hi-1 \cdot j)),\end{aligned}$$

where $i \geq 1$ and $j \geq 1$ for uplink.

$$\begin{aligned}E(Ni-1 \cdot j-1 \to Nij) = 1 &- (1-E(Hi-1 \cdot j))(1-E(Vij)) \\ &- (1-E(Vi \cdot j-1))(1-E(Hij)),\end{aligned}$$

where $i \geq 1$ and $j \geq 1$ for downlink. □

From now on, we will just mention the downlink case for simplicity, as uplink and downlink are symmetric. Figure 2(a) shows virtual links, $S00$, $S01$, $S10$, and $S11$ created by the split-merge operation Here, the error rate can be replaced by the success probability by subtracting the former from 1.0. Based on the given and estimated error rate, namely, $E(Vij)$, $E(Hij)$, and $E(Sij)$, a cost matrix and success probability graph can be built. Then, Dijkstra's shortest path algorithm can find the best route having the lowest error rate (best success rate) for both graphs, after substituting the product of probabilities for the sum of link costs in each node expansion. Finally, the instance of virtual link, say Sij is to be expanded by the split-merge schedule to finalize a complete slot assignment [7]. In this example, Sij is replaced by $(Hi \cdot j+1, Vi+1 \cdot j+1)$ on the primary schedule and $(Vi+1 \cdot j, Hi+1 \cdot j+1)$ for the secondary schedule. However, the primary and secondary schedules can ne interchanged, if we do not consider power consumption. A horizontal link will be the primary route.

However, there are another virtual links to consider. We call $S00$, $S01$, $S10$, and $S1$ level 1 virtual links. With them, 4 level 2 links can be formed as shown in Figure 2(b). Their error rates can be estimated as in the level 1 virtual link with the path specification as follows:

$$\begin{aligned}&\text{T1} : N00 \to N12 = H01 \cdot S01 + V10 \cdot H11 \cdot H12 \\ &\text{T2} : N00 \to N12 = V10 \cdot S10 + H01 \cdot V11 \cdot V21 \\ &\text{T3} : N00 \to N12 = H11 \cdot S11 + V20 \cdot H21 \cdot H22 \\ &\text{T4} : N00 \to N12 = V11 \cdot S11 + V02 \cdot V12 \cdot V22\end{aligned}$$ □

We will describe the path reduction just for the $T1$ case, and the others can be reduced in the same way. In the following relation, $A \cdot B$ means B occurs just after A, so they cannot be changed. $A + B$ means A and B can occur simultaneously. The path schedule for $T1$ depends on the status of link $H01$. If it is good,

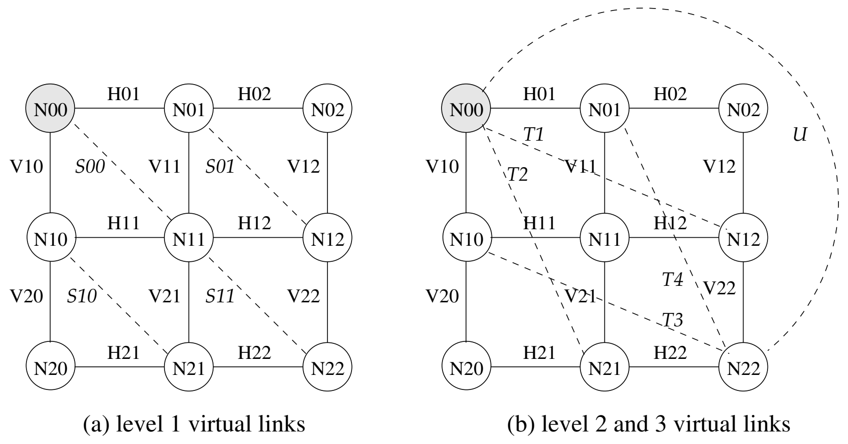


Fig. 2. Cost graph and substitutions

$N00$ sends a message to $N01$ via $H01$ first and the message takes $S01$ level 1 link. Otherwise, $V10 \cdot H11 \cdot H12$ is the delivery path. Namely,

$$\begin{aligned}
\mathrm{T1} &= N00 \rightarrow N12 \\
&= H01 \cdot S01 + V10 \cdot H11 \cdot H12 \\
&= H01 \cdot (H02 \cdot V12 + V11 \cdot H12) + V10 \cdot H11 \cdot H12 \\
&= H01 \cdot H02 \cdot V12 + H01 \cdot V11 \cdot H12 + V10 \cdot H11 \cdot H12
\end{aligned}$$

For the case the controller selects a level 2 virtual link as the route to the corresponding node, the controller also allocates time slots. Level 2 links commonly have 3 hops, that is, its transmission needs 3 time slots. According to the above reduction on the path, the scheduler can know which transmission must take place in each time slot from t1 to t2.

$$\begin{aligned}
t1(T1) &= (H01, V10) &&= (N00 \rightarrow N01, N00 \rightarrow N10) \\
t2(T1) &= (H02, V11, H11) &&= (N01 \rightarrow N02, N01 \rightarrow N11, N10 \rightarrow N11) \\
t3(T1) &= (V12, H12) &&= (N02 \rightarrow N12, N11 \rightarrow N12)
\end{aligned}$$

Finally, the schedule is decided as shown in the following table, which shows a partial time table from t to $t+2$. t is the first slot $t1(T1)$ will be assigned. In t, both $N00 \rightarrow N01$ and $N00 \rightarrow N10$ should take place. These two have the same sender, so the common sender must perform the split operation. As we chose the horizontal path first policy, $N00 \rightarrow N01$ is assigned to t as a primary path, while $N00 \rightarrow N10$ is assigned to $t^{'}$. But this order is not critical. $t+1$ have three transmissions. $N01$ appears twice as a sender, while $N11$ appears twice as a receiver. As $N01 \rightarrow N02$ and $N10 \rightarrow N11$ have different senders and receivers, they can be put to the same slot. So, $t+1$ has two entries, while $t+1^{'}$ has $N10 \rightarrow N11$. At $t+2$, $N11$ receives from two senders, so it must be a merge operation.

	sender	receiver
t	$N00$	$\rightarrow$ N01
t'	$N00$	$\rightarrow N10$
$t+1$	$N01$	$\rightarrow N02$
$t+1$	$N10$	$\rightarrow N11$
$t+1'$	$N01$	$\rightarrow N11$
$t+2$	$N02$	$\rightarrow N12$
$t+2'$	$N11$	$\rightarrow N12$

In addition, a level 3 virtual link, U, can be defined with level 2 links, and its reduction and slot allocation is the same as the case of level 2 links. Namely,

$$U = V10 \cdot T3 + H01 \cdot T4$$

5 Performance Measurement

This section measures and assesses the performance of the proposed routing scheme via simulation using SMPL which provides a discrete event scheduler [14]. The simulation considers 3 $\times$ 3 grid which consists of 9 nodes and 12 links, while the slot error rate of each link distributes randomly with an average value ranging from 0 to 0.2. We add virtual links after estimating their error rates. Only the downlink graph was considered for simplicity, as uplink and downlink communications are symmetric and calculated completely in the same way. Hence, in our model, the controller node transmits messages to each node one by one according to the slot assignment as described in the previous section. The experiment measures the delivery ratio of end-to-end messages for the proposed routing scheme and compares with the case in which the split-merge operation is placed as close to the source as possible. The reference scheme is named $Firstfit$.

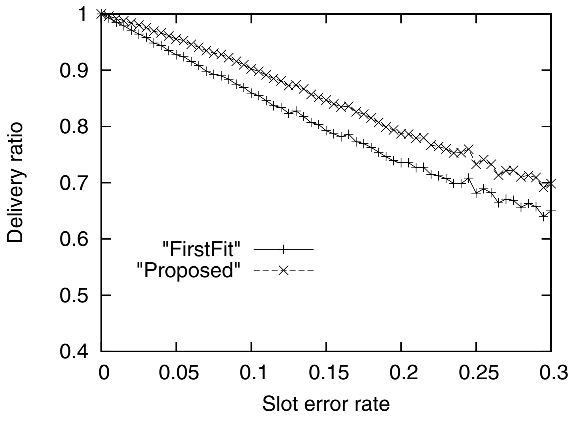


Fig. 3. Performance measurement result

Figure 3 exhibits the performance improvement according to the slot error rate. This experiment makes each link have the error rate according to the exponential distribution. Namely, the experiment generates 500 sets for each average slot error rate ranging from 0.0 to 0.3, and measure the message delivery ratio from the coordinator to all other nodes. As shown in this figure, the proposed routing scheme can enhance the delivery ratio by up to 5.21 %. The performance gap increases according to the increase of slot error rate. We are now intensively evaluating the performance of the proposed scheme according to the various parameters including error distribution, error model, message load, and so on. Particularly, when the difference of slot error rates in each link gets larger, more improvement can be expected.

6 Conclusion

This paper has designed and measured the performance of a routing scheme capable of efficiently handling the difference in the error rate of each link in the traffic light network which has a grid topology. The grid topology is easily found in most modern cities which have a Manhattan-style road network, and it is very easy to find the path having the same length. Using such primary and secondary paths, a split-merge operation can mask the channel error by making an intermediary node receive a message simultaneously in two directions. By this operation, the two 2-hop links between for two nodes in the diagonal of a rectangle can be considered to be a single virtual link. After estimating the error rate of the virtual links based on those of the surrounding links, the proposed scheme has employed the well-known shortest path algorithm to decide the best route and also to finalize the complete a slot assignment for the WirelessHART protocol. The performance measurement result obtained by simulation using a discrete event scheduler demonstrates that the proposed scheme can find the promising path, improving the delivery ratio for 3×3 grid by up to 5.21 %.

As future work, we are first planning to analyze the performance characteristics on robust communication, that is, how efficiently the proposed scheme can overcome link or node failures. Second, a slot assignment scheme is developed to integrate the primary and secondary path schedule, minimizing the length of a control loop.

References

1. US Depart of Transportation. Vehicle safety communication project-final report. Technical Report HS 810 591 (2006), `http://www-nrd.nhtsa.dot.gov/departments/nrd-12/pubs_rev.html`
2. IEC/PAS 62591: Industrial communication networks - Fieldbus specifications - WirelessHART communication network and communication profile (2008)
3. Yu, B., Gong, J., Xu, C.: Data aggregation and roadside unit placement for a VANET traffic information system. In: ACM VANET, pp. 49–57 (2008)

4. Jaap, S., Bechler, M., Wolf, L.: Evaluation of routing protocols for vehicular ad hoc networks in city traffic scenarios. In: Proceedings of the 5th International Conference on Intelligent Transportation Systems Telecommunications (2005)
5. Hart Communication Foundation, Why WirelessHART^{TM}? The Right Standard at the Right Time (2007), http://www.hartcomm2.org
6. Song, S., Han, S., Mok, A., Chen, D., Nixon, M., Lucas, M., Pratt, W.: WirelessHART: Applying wireless technology in real-time industrial process control. In: The 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 377–386 (2008)
7. Lee, J., Shin, I., Kim, C.: Design of a reliable traffic control system on city area based on a wireless network. In: Gervasi, O., Taniar, D., Murgante, B., Laganà, A., Mun, Y., Gavrilova, M.L. (eds.) ICCSA 2009. LNCS, vol. 5592, pp. 821–830. Springer, Heidelberg (2009)
8. Kodialam, M., Nandagopal, T.: Characterizing the capacity region in multi-radio multi-channel wireless mesh networks. In: ACM MobiCom, pp. 73–87 (2005)
9. Raniwala, A., Gopalan, K., Chieuh, T.: Centralized algorithms for multi-channel wireless mesh networks. ACM Mobile Computing and Communication Review (2004)
10. Cain, J., Billhartz, T., Foore, L., Althouse, E., Schlorff, J.: A link scheduling and ad hoc networking approach using directional antennas. In: Military Communications Conference, pp. 643–648 (2003)
11. Chang, N., Liu, M.: Optimal channel probing and transmission scheduling for opportunistic spectrum access. In: Proc. ACM international conference on Mobile computing and networking, pp. 27–38 (2007)
12. Alur, R., D'Innocenzo, A., Pappas, G., Weiss, G.: Modeling and analysis of multihop control networks. In: The 15th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 223–232 (2009)
13. Ramchandran, I., Roy, S.: Clear channel assessment in energy-constrained wideband wireless networks. IEEE Wireless Magazine, 70–78 (2007)
14. MacDougall, M.: Simulating Computer Systems: Techniques and Tools. MIT Press, Cambridge (1987)

Wireless Bluetooth Communications Combine with Secure Data Transmission Using ECDH and Conference Key Agreements

Hua-Yi Lin[1] and Tzu-Chiang Chiang[2]

[1] Department of Information Management, China University of Technology, Taiwan, R.O.C.
calvan.lin@msa.hinet.net
[2] Department of Information Management, Tunghai University, Taiwan, R.O.C.
steve312kimo@thu.edu.tw

Abstract. As the fast development of Bluetooth networks and wireless communications, the mobile devices share information with each other easier than ever before. However, the handy communication technology accompanies privacy and security issues. Nowadays, a Bluetooth adopts peer-to-peer and Frequency Hopping Spread Spectrum (FHSS) mechanisms to avoid data reveal, but the malicious attacks collect the transmission data of the relay station for a long period of time and then can break into the system. In this study, we take a Piconet as a cube, and transform a Scatternet into a cluster (*N*-cube) structure. Subsequently, this study exploits the Elliptic Curve Diffie-Hellman (ECDH) [1] and the Conference Key (CK) schemes to perform session key agreements and secure data transmissions. The proposed scheme only needs a small key length 160-bit to achieve compatible security levels on 1024-bit Diffee-Hellman (DH) [2], and each node uses few CPU, memory and bandwidth to complete security operations. As a result, the proposed fault-tolerant routing algorithm with secure data transmissions can perform rapidly and efficiently, and is quite suited for Bluetooth networks with limited resources.

Keywords: Bluetooth, FHSS, Piconet, Scatternet, ECDH, CK, DH.

1 Introduction

The Bluetooth adopts the 2.4GHz frequency band using Frequency Hopping Spread Spectrum (FHSS) for changing frequency 1600 times per second, and the 2.4GHz is divided into 79 channels[3]. For avoiding interruptions and protecting data transmissions, each hope chooses a specific channel, and then jumps into the other channel after 400ms as shown in Fig. 1. In a Bluetooth infrastructure, a Piconet consists of a master device and 7 clients, and several Piconets construct a Scatternet. The communication of Bluetooth devices allocated on distinct Piconets exploits a gateway node to assist in data transmissions. Therefore, this study considers a Piconet as a special case of a cluster. The master device servers as a cluster head, and the Scatternet consists of several clusters. A well Scatternet has to consider the routing efficiency and secure data transmissions. Nowadays, the security in Bluetooth networks only employs a simple Peer-to-Peer password authentication and FHSS technologies to protect data transmissions. However, the identification and the security key are transmitted in plain texts. Thus, malicious intruders can easily launch attacks such as Bluejacking, Bluesnarfing, Backdoor attack, Cabir worm, and L2CAP connection re-setter. Additionally, since 802.11 and Bluetooth use the same frequency on 2.4GHz, the neighboring devices possibly interfere with each other. Therefore, it is an important issue to provide a reliable and secure communication protocol.

The Bluetooth communication has limited CPU, Memory, Power and bandwidth. Many asymmetric key mechanisms although provide rather secure functions, however, they need a lot of computing power and are not suited for Bluetooth networks. This study proposes a secure data transmission protocol associated with ECDH which can save resources and achieve secure data transmissions in Bluetooth networks. Additionally, we propose a fault-tolerant path-finding protocol which only needs O(n) time complexity to find a routing path in Bluetooth networks.

The rest of this study is structured as follows. Section 2 introduces the Piconet and Scatternet fault-tolerant routing protocols. Section 3 describes the proposed secure data transmission scheme. Section 4 presents the security analyses of the proposed scheme. Section 5 depicts the computing evaluation.

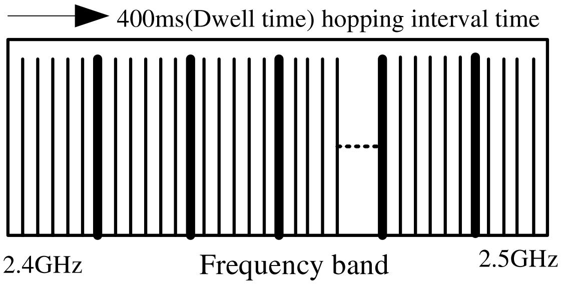


Fig. 1. Frequency Hopping Spread Spectrum (FHSS)

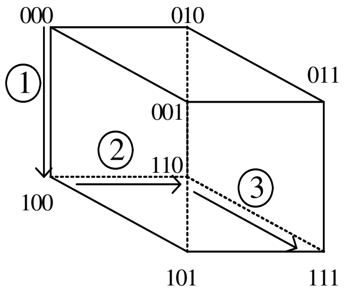


Fig. 2. A Piconet routing protocol

2 Piconet and Scatternet Routing Protocols

Since the Piconet infrastructure is similar to a cluster or a cube, this study considers the fault-tolerant capacity of a routing path, and proposes a multiple path routing protocol. First, we divide the entire Bluetooth network into several cubes, and give each node on the cube a coordinate (x, y, z), where x, y, and z belong to 0 or 1. For example, if a node allocated on (0, 0, 0) wants to find a routing path to (1, 1, 1) as shown in Fig. 2. Initially, according to the routing algorithm, it performs (111 XOR 000) operations to determine the next passing node. The XOR operation result shows that the different bits indicate the possible routing paths. After 3 phases, this algorithm can determine a routing path.

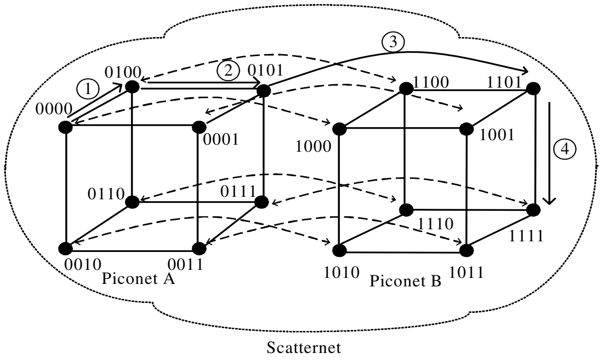


Fig. 3. A Scatternet routing protocol

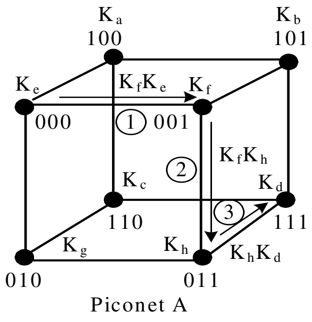


Fig. 4. Piconet secure data transmission protocols

Similarly, in a Scatternet structure as shown in Fig. 3. We extend the communication nodes to 16 nodes, and give each node a coordinate (w, x, y, z) consisting of 4 bits. The routing algorithm only needs 4 phases to determine the routing path to the target node. Consequently, this study applies this routing algorithm on a N-cube structure (with k=2^n node, the maxima distance between two nodes is $\log_2 k = n$), and realizes that the Scatternet only takes n phases to find the destination node. The proposed algorithm only needs simple XOR operations, and the time complexity of the path-finding is O(n), where the n indicates the dimension of the cube. The routing algorithm pseudo code is as follows:

Phase 1. The coordinate of the destination node XOR the coordinate of the current node→ result.

Phase 2. Compare the result with the coordinate of the current node, the first different bit is the first choice path, and the second (third, fourth) different bit is the second (third, fourth) choice path.

Phase 3. Move forward to the next node, and record the coordinate on the routing path.

Phase 4. Repeat the above steps, until find the destination node.

If source node (0000) wants to find the destination node (1111), the brief routing algorithm is presented as follows:

(1111 XOR 0000) → 1111 (choose 1000).
(1111 XOR 1000) → 0111 (choose 1100).
(1111 XOR 1100) → 0011 (choose 1110).
(1111 XOR 1110) → 0001 (choose 1111).
(1111XOR 1111) → 0000 (find the target node).

3 Secure Data Transmission Protocols

In this study, we consider that the Bluetooth network has limited resource. Therefore, the asymmetric key and public key infrastructures although provide excellent security functions, but they need a huge amount of computing powers and resources, and therefore are not suited for Bluetooth networks. This study adopts the CK and ECDH schemes to solve the problem of constrained resources in Bluetooth networks. In the CK scheme, the entire system exploits a master key to en/decrypt transmitted data, and effectively improves the efficiency of security operations. However, in the initial phase, the CK scheme needs more complicated key agreement operations than the ECDH scheme. Oppositely, the ECDH scheme does not need to compute the master key. But, ECDH has to calculate a session key between each pair of nodes which want to perform secure data transmissions. However, only few addition and multiplication operations have to be performed in ECDH. In other words, the CK and ECDH schemes have different characters, but they are efficient for secure data transmissions. This study presents the secure data transmission scheme for CK and ECDH in the following sessions.

3.1 Conference Key Agreement

Since each Bluetooth device has a unique make address (MAC), thus we employ the MAC address (6 bytes) as the identification ID_i, and adopt that T. Hwang et al.[4,5] proposed conference key algorithms to calculate the conference key K. First of all, in a Piconet, the head node chooses a common parameter (N, q, r, s, t, and $n=qr$, $st=1$ mod L, where L=lcm(q-1, r-1). And then we calculate $B=(b_1,b_2,\ldots,b_n)$, $1 \leqq b_i \leqq L$-1. Let $H=(h_1,h_2,\ldots h_n)=(h^{b1}$ mod N, h^{b2} mod N,... h^{bn} mod N), where h is a root of GF(q) and GF(r). Subsequently, each member U_i of the Piconet applies for private keys Z_i and Y_i from the head node using $ID_{i.}$ Since each Bluetooth device ID_i is the verification basis, and $ID_i=(X_{i1},X_{i2},\ldots X_{ij})$, $X_{ij} \in \{0,1\}$, $1 \leqq j \leqq 48$. After head node receiving $ID_{i,}$ it performs a one way hash function $f_1(ID_i)=(Y_{i1},Y_{i2},\ldots,Y_{ij})$, $Y_{in} \in \{0,1\}$, $1 \leqq j \leqq n$, and then head node calculates the private key $Z_i=(ID_i)^d$ mod N for each node, where $1 \leqq i \leqq 8$.

When the head node calculates H_i for each member (U_1~U_8), then chooses a random number r, $C_1=h^{sr}$ mod N, and $C_2=Z_M h^{f2(t,C1)r}$ mod N, where f_2 is a one way function with two parameters and t denotes current time and M denotes the serial number of the conference members (1~8). Subsequently, $K_{Mi}=(H_i)^{sr}$ mod N, then head node chooses a conference key from (1~N-1). According to K_{Mi}, the head node can construct a Larange Interpolation as follows:

$$A(X) = \sum_{i=1}^{7} (K + ID_i) \prod_{j=1, j \neq i}^{7} \frac{(x - KM_j)}{(KM_i - KM_j)} \bmod N \; = a_6x^6 + a_5x^5 + \ldots + a_1x + a_0M \; .$$

Subsequently, the head node broadcasts $(C_1,C_2,a_0,a_1,\ldots,a_6,t)$ to the other members. Anyone receives it, and verifies the correct of $\frac{(C_2)^s}{(C_1)^{f_2(t_1,C_1)}} = ID_M \pmod N$ to insure that the head node is the dealer, and then calculates $K_{M_i} = (C_1)^{Y_i} \bmod N = g^{rsY_i} \bmod N$. Consequently, the legitimate U_i can obtain the conference key K from $A(X)$, where

$A(K_{Mi})=a_6K_{Mi}+\ldots+a_1K_{Mi}+a_0$ mod $N = K + ID_i$ mod N, and $K = K+ ID_i - ID_i$(mod N). In other words, each member has the same conference key K, and can perform the secure data transmission using K.

3.2 Piconet and Scatternet Secure Data Transmission Protocols Using Conference Key

In a Piconet, each node obtains the same conference key, if the sender K_e wants to deliver data to the receiver K_d, as shown in Fig. 4. The detailed secure data transmissions using the CK scheme is as follows:

$K_e \rightarrow K_f$
$\mathrm{EK}_K[K_e|M_e|\mathrm{HMAC}(M_e)]$

Initially, K_e employs the conference key K to calculate the message authentication code of M_e as HMAC(M_e). Subsequently, K_e uses the conference key K to encrypt the passing node ID, transmitted data M_e and HMAC(M_e), and then delivers the encrypted result to the following node K_f.

$K_f \rightarrow K_h$
$\mathrm{EK}_K[K_fK_e|(M_f||M_e)|\mathrm{HMAC}(M_f||M_e)]$

As K_f receives the encrypted data, K_f exploits K to decrypt the received data, and verifies the integrity of HMAC(M_e). Subsequently, K_f adds its own ID into the routing path; accumulates the transmitted data M_f with M_e, and calculates the message authenticate code HMAC($M_f||M_e$). Consequently, K_f encrypts the entire data using the conference key K, and sends them to the next node K_h.

$K_h \rightarrow K_d$
$\mathrm{EK}_K[K_hK_fK_e|(M_h||M_f||M_e)|\mathrm{HMAC}(M_h||M_f||M_e)]$

Upon K_h receiving the encrypted data, K_h decrypts the encrypted data using the conference key K; verifies the integrity of HMAC($M_f||M_e$); adds its own ID into the routing path $K_hK_fK_e$, and accumulates M_h with $M_f||M_e$. Subsequently, K_h calculates the HMAC($M_h||M_f||M_e$); encrypts the entire data using the conference key K, and sends them to the following node K_d. Eventually, the destination node K_d receives the encrypted data; then decrypts them using the conference key K, and verifies HMAC($M_h||M_f||M_e$). During the secure data transmissions, if any modifications occur, the following node immediately detects the tampered data.

Initially, the proposed CK scheme takes much longer time to calculate the conference key K, since the members of a Piconet has to obtain the same conference key K. After then, each node uses K to en/decrypt the transmitted data, and thus improves the efficiency and avoids calculating the session key for each pair of nodes.

In a Scatternet, as shown in Fig. 5, the entire system has to calculate the same conference key K for all members, and thus takes much longer time than a single Piconet. Therefore, we propose an efficient scheme named ECDH key agreement which can decrease a lot of time to synchronize the conference key K for the entire system. We present the ECDH scheme in the following section.

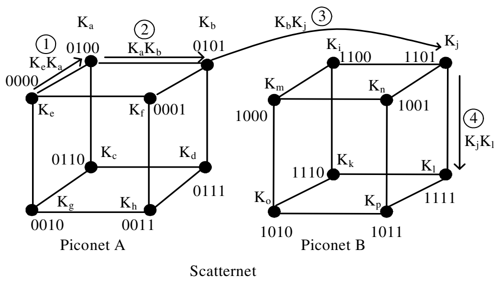


Fig. 5. Scatternet secure data transmission

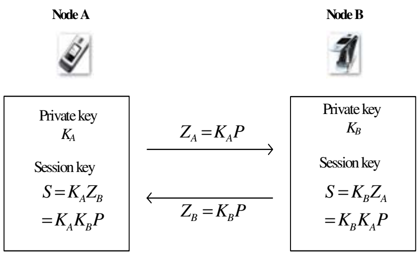


Fig. 6. ECDH key agreement

3.3 ECDH Key Agreements

So far, since the ECDH mechanisms have better performance than Diffee-Hellman (DH), This mechanism only uses a shorter key length 160-bit to achieve compatible security levels on RSA or Diffee-Hellman(DH) 1024-bit [2,6], and therefore this study adopts the ECDH method on wireless Bluetooth networks to improve the efficiency of secure data transmissions.

Consider the case in ECDH, where sensor node *A* wants to establish a shared key with node *B*, as shown in Fig. 6. The public parameters (a prime *p*, a base point *P* as a generator in Diffe-Hellman, coefficients *a* and *b*, elliptic curve $y^2=x^3+ax+b$) must first be set. Additionally, each party must have an appropriate key pair for elliptic curve cryptography, comprising a ECC private key *K* (a randomly selected integer) and a public key *Z* (where $Z = KP$). Let a node key pair of *A* denote (K_A, Z_A), and a node key pair of *B* denote (K_B, Z_B). Each party must have the other party's public key. Node *A* calculates $Z_A = K_AP$, while node *B* calculates $Z_B= K_BP$. Both parties calculate the shared key *R* as $R = K_AZ_B = K_AK_B\text{P} = K_BK_A\ \text{P} = K_BZ_A$. The protocol is secure because it reveals nothing (except public keys, which are not secret), and because no party can calculate the private key of the other unless it can solve the elliptic curve Discrete Logarithm Problem (DLP).

The ECDH scheme provides rapid and efficient operations with limited resources. Therefore, a node requires only addition and multiplication operations without performing exponent operations in DH. Consequently, this study employs the ECDH key agreement to achieve secure data transmissions in Bluetooth networks.

3.3.1 ECDH Secure Data Transmission Protocols for Piconet

In a Piconet, as shown in Fig. 4, the sender K_e wants to deliver data to the destination node K_d. According to the proposed routing protocol, this study can find one of the routing paths ($K_e \rightarrow K_f \rightarrow K_h \rightarrow K_d$). Subsequently, this study exploits the ECDH scheme to perform secure data transmissions along the routing path. The detailed processes are as follows:

$K_e \rightarrow K_f$
$\text{EK}_{KeKfP}[K_e|M_e|\text{HMAC}(M_e)]$

Initially, K_e and K_f calculate the session key K_eK_fP for both using the ECDH scheme. Subsequently, K_e employs K_eK_fP to calculate message authentication code HMAC(M_e); then K_e encrypts the passing node *ID*, transmitted data M_e and HMAC(M_e) using K_eK_fP, and then sends the entire encrypted data to the next node K_f.

$K_f \rightarrow K_h$
$EK_{KfKhP}[K_fK_e|(M_f||M_e)|HMAC(M_f||M_e)]$

Upon K_f receiving the transmitted data, K_f uses the session key K_eK_fP to decrypt the encrypted data, and adopts K_eK_fP to verify the integrity of HMAC(M_e). Subsequently, K_f adds its own *ID* to the routing path; accumulates message M_f with M_e, and exploits K_fK_hP to calculate the HMAC($M_f||M_e$). Eventually, K_f encrypts the entire data using the session key K_fK_hP, and deliveries them to the following node K_h.

$K_h \rightarrow K_d$
$EK_{KhKdP}[K_hK_fK_e|(M_h||M_f||M_e)|HMAC(M_h||M_f||M_e)]$

After receiving the transmitted data, K_h uses the session key K_fK_hP to decrypts them, and verifies the integrity of HMAC($M_f||M_e$). Subsequently, K_h adds its own *ID* into the routing path K_fK_e; accumulates M_h with $M_f||M_e$, and uses K_hK_dP to calculate HMAC($M_h||M_f||M_e$). Consequently, K_h encrypts the entire data using the session key K_hK_dP, and sends the encrypted data to the next node K_d. Upon receiving the encrypted data, K_d decrypts the encrypts data using the session key K_hK_dP, and verifies the integrity of HMAC($M_h||M_f||M_e$).

This proposed mechanism is plain and straightforward with well performance and fault-tolerant routing functions. This system exploits a small key length with 160-bit to achieve compatible security levels on 1024-bit for DH or RSA, but only needs few CPU, memory and bandwidth. The proposed scheme is highly suited for Bluetooth networks with constrained resources.

3.3.2 ECDH Secure Data Transmission Protocols for Scatternet

As shown in Fig.5, K_e wants to send data to the destination node K_l, if one of routing paths is $K_e \rightarrow K_a \rightarrow K_b \rightarrow K_j \rightarrow K_l$, the detailed secure data transmission processes are as follows:

$K_e \rightarrow K_a$
$EK_{KeKaP}[K_e|M_e|HMAC(M_e)]$

Initially, K_e and K_a use the ECDH scheme to calculate the session key K_eK_aP, and then K_e calculates the message authentication code HMAC(M_e) using K_eK_aP. Subsequently, K_e exploits K_eK_aP to encrypt the passing node *ID*, transmitted data M_e and HMAC(M_e), and then sends the entire encrypted data to the following node K_a.

$K_a \rightarrow K_b$
$EK_{KaKbP}[K_aK_e|(M_a||M_e)|HMAC(M_a||M_e)]$

Upon receiving the encrypted data, K_a employs the session key K_eK_aP to decrypt the encrypted data, and verifies the integrity of HMAC(M_e) using K_eK_aP. Subsequently, K_a adds its own *ID* into the routing path; accumulates M_a with M_e, and uses K_aK_bP to calculate HMAC($M_a||M_e$). Eventually, K_a encrypts the entire data using K_aK_bP, and sends them to the next node K_b.

$K_b \rightarrow K_j$
$\mathrm{EK}_{KbKjP}[K_bK_aK_e|(M_b||M_a||M_e)|\mathrm{HMAC}(M_b||M_a||M_e)]$

After receiving data, K_b decrypts the received data using the session key K_aK_bP; verifies the integrity of HMAC($M_a||M_e$). Subsequently, K_b adds its own *ID* into the routing path K_aK_e; accumulates M_b with $M_a||M_e$; calculates HMAC($M_b||M_a||M_e$); encrypts the entire data using K_bK_jP, and then sends them to the following node K_j. Upon K_j receiving the encrypted data, K_j decrypts the encrypted data using K_bK_jP; verifies the integrity of HMAC($M_b||M_a||M_e$), and then send them to the next node K_j.

$K_j \rightarrow K_l$
$\mathrm{EK}_{KjKlP}[K_jK_bK_aK_e|(M_j||M_b||M_a||M_e)|\mathrm{HMAC}(M_j||M_b||M_a||M_e)]$

Repeating the similar operations, the destination node K_l receives the encrypts data; then decrypts the encrypted data using K_jK_lP, and verifies the integrity of HMAC($M_j||M_b||M_a||M_e$). If any changes take place during the transmissions, the receiving node detects the modifications immediately by verifying the HMAC.

4 Security Analyses

This section provides several security analyses for the proposed mechanisms, and evaluates the performance of the methods.

(1) Multiple routing path protocols for Piconet and Scatternet networks

In a Piconet, each node has multiple routing paths. Once the dedicated routing path collapses, nodes exploit the routing algorithm to search a spare routing path, and then the system regains normality. Similarly, the Scatternet (n=16, k=4 or above) employs the same algorithm to search a spare path. This mechanism is rapid and efficient for fault-tolerant routing in peer-to-peer networks.

(2) Confidentiality and authentication

During the data transmission, this study exploits the session key to encrypt the transmitted data. Only the node with the same session key can decrypt the encrypted data. The other nodes are not aware of the session key, and therefore can not decrypt the encrypted data. Thus, the scheme can insure that the data transmission is confidential and authentic.

(3) Data Integrity and accuracy

This study employs message authentication code (HMAC) to verify the integrity of transmitted data. During the transmission, each node calculates HMAC, and the receiver verifies the integrity of HMAC. Since HMAC is an irreversible operation, given a random number y, no ways can compute x such that $H(x)=y$. Moreover, when $a \neq b$, then $H(a) \neq H(b)$. Therefore, if any nodes modify the transmitted data during transmissions, the receiver detects the unmatched HMAC instantly and recognizes the tampered data.

(4)The performance of secure data encryption and encryption

Since the limited resource of Bluetooth networks, the asymmetric key and Public Key Infrastructure (PKI) need a lot of resources, and thus are not suited for Bluetooth devices. In this study, we propose a CK scheme. Initially, each node takes much longer time to calculate a common conference key. After that, each node employs the conference key to efficiently perform the secure data transmission. Oppositely, in initial, ECDH scheme doesn't need to calculate the common key for each node, and therefore ECDH saves the key synchronization time. However, ECDH has to calculate the session key for each pair of nodes in every secure data transmission, and thus takes much longer time during transmission. However, the two proposed schemes only conserve a session key, and exploit a hash function such as HMAC-160 or RIPEMD-160 to verify data integrity. Consequently, they can reduce the operational resources and are highly suited for Bluetooth networks.

5 Computing Evaluation

Since the time complexity of the routing algorithm relies on the dimension of the Piconet or Scatternet. In this study, the Piconet adopts a 3-dimension cube and the Scatternet adopts a 4-dimension cube. Therefore, a Piconet only needs three phases to determine a routing path, and a Scatternet needs four phases to decide a routing path. The proposed scheme is rapid and efficient, and takes O(n) time complexity, where n denotes the dimension of the cube. Additionally, the proposed method provides fault-tolerant and multiple path-finding functions. Even the dedicated routing path collapses and the system can rapidly find a replacement path performing the routing algorithm.

Consider the performance of secure data transmissions, the experiments are implemented on a MICAII-like mote production JN-5121 with 32-bit 16 MHz CPU, 96KB RAM and 64KB ROM. Several ECDH operations are performed on the JN-5121 platform. The mote computes a point multiplication in 0.18s on SECP-160 curve [7] and 5.2s to establish a session key on ECDH in the prime field. Additionally, we evaluate the routing algorithm and the secure data transmission on a Piconet and a Scatternet.

Figure 7 demonstrates the searching time of a routing path for the Piconet and Scatternet. From the experimental results, we can conduct that the scheme needs few XOR operations and takes O(n) time complexity to find a routing path form the source node K_a (0000) to the destination node K_h (1111).

Figure 8 demonstrates the comparison of three kinds of schemes (DH, ECDH and CK) on secure data transmission time in a Piconet using the session key scheme. As the increasing nodes, the experimental results show that ECDH outperforms the other two schemes on the compatible security level.

Figure 9 depicts the secure data transmission time in a Scatternet. This study adopts DH, ECDH and CK schemes to calculate the secure data transmission time between two Piconets. During the transmission, since the CK scheme has to calculate the conference key in advance, therefore takes much longer time than the other two schemes. After that, the entire system uses the same conference key to perform secure data transmissions, and thus saves a lot of key agreement time. As the increasing nodes, the experimental results demonstrate that ECDH and CK schemes outperform DH.

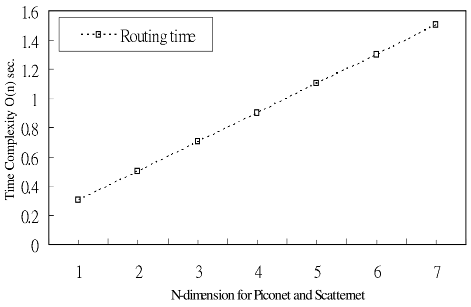


Fig. 7. The time complexity of finding a routing path for the Piconet and Scatternet

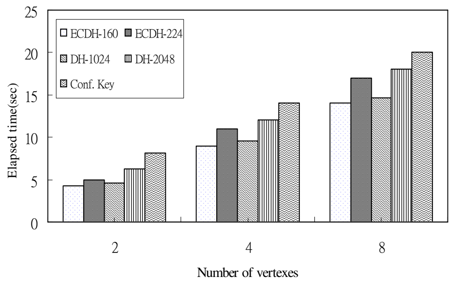


Fig. 8. The secure data transmission time of DH, ECDH and CK schemes in a Piconet

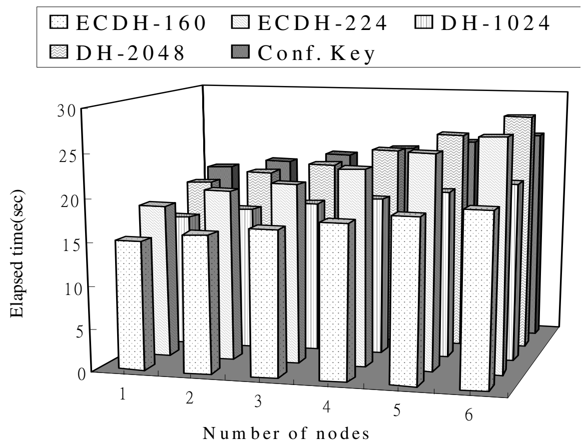


Fig. 9. The secure data transmission time of DH, ECDH and CK schemes in a Scatternet

6 Conclusions

In the future, Bluetooth technologies are the main stream of short distance communications, and are gradually integrated into wireless networks. However, the transmitted data are exposed to a public network, and thus are easy to be sniffed. Therefore, secure Bluetooth data transmissions are an important issue.

This paper proposes a rapid, efficient and fault-tolerant mechanism for secure data transmissions on Bluetooth networks. The proposed routing algorithm only needs O(n) time complexity to find a fault-tolerant routing path in a Piconet and a Scatternet. Additionally, we employ CK and ECDH schemes to secure the data transmissions on Bluetooth networks. The mechanism merely needs a small key length 160-bit to achieve compatible security levels on 1024-bit DH or RSA, and the mechanism is highly suited or Bluetooth networks with constrained resources.

Additionally, we adopt session key to perform secure data transmissions in Bluetooth networks, and we exploit a gateway node to assist in securing data transmissions in a Scatternet. Eventually, the simulation results indicate that the proposed ECDH scheme outperforms CK and DH schemes. Consequently, the proposed scheme consumes few CPU, memory, bandwidth and resources, and therefore is highly suited for Bluetooth networks.

Acknowledgements

This work was supported by the National Science Council of Republic of China under grant NSC-99-2218-E-029-002.

References

1. Diffe, W., Hellman, M.: New Directions in Cryptography. IEEE Transactions on Information Theory (November 1976)
2. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks, pp. 245–256 (2008)
3. Lee, G., Park, S.C.: Bluetooth security implementation based on software oriented hardware-software partition. In: IEEE International Conference on Communications, May 2005, pp. 2070–2074 (2005)
4. Hwang, T., Chen, J.L.: Indentity-Based Conference Key Broadcast System. IEE Proc.-Computer Digital Technical 141(1), 57–60 (1994)
5. Laih, C.S., Harn, L., Chang, C.C.: Contemporary Cryptography and It's Applications, pp. 20-2–20-7. Flag Publisher (2003)
6. Eschenauer, L., Gligor, V.D.: A Key-management Scheme for Distributed Sensor Networks. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, November 2002, pp. 41–47 (2002)
7. Szczechowiak, P., Oliveira, L.B., Scott, M., Collier, M., Dahab, R.: NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008)
8. Chang, C.T., Chang, C.Y., Sheu, J.P.: BlueCube: Constructing a hypercube parallel computing and communication environment over Bluetooth radio systems. Journal of Parallel and Distributed Computing 66(10), 1243–1258 (2006)

Robust Multicast Scheme for Wireless Process Control on Traffic Light Networks[*]

Junghoon Lee[1], Gyung-Leen Park[1], Seong-Baeg Kim[2], Min-Jae Kang[3], and Mikyung Kang[4,**]

[1] Dept. of Computer Science and Statistics, [2] Dept. of Computer Education,
[3] Dept. of Electronic Engineering
Jeju National University, 690-756, Jeju Do, Republic of Korea
[4] University of Southern California - Information Sciences Institute, VA22203, USA
{jhlee,glpark,sbkim,minjk}@jejunu.ac.kr, mkkang@east.isi.edu

Abstract. This paper designs and analyzes the performance of an efficient and robust multicast scheme for the grid-style wireless network built upon the WirelessHART protocol, aiming at improving the reliability of wireless process control without extending the slot assignment schedule. The proposed scheme first makes the controller node select the transmission order to its downstream neighbors according to the current channel condition returned from the clear channel assessment . Next, receiver nodes having two predecessors listen first from the primary sender and then switches to the secondary sender when the message doesn't arrive within the predefined time bound. The simulation result obtained from the discrete event scheduler shows that the proposed scheme can improve the multicast delivery ratio by up to 35.6 %, compared with the multicast tree-based scheme having the same slot schedule, maintaining the consecutive message losses below 8.

1 Introduction

Based on matured wireless communication technologies, vehicular telematics networks extend the global network coverage to a driver while he or she is driving on the road [1]. In this network, the fast moving vehicles are the communication entities. Thus, not only the connection between them is extremely instable but also multihop connections cannot last for an interval enough to provide a reasonable quality service. To overcome this problem, static elements, capable

[*] 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-2010-(C1090-1011-0009)).
Following are also results of a study on the "Human Resource Development Center for Economic Region Leading Industry" Project, supported by the Ministry of Education, Science & Technology(MEST) and the National Research Foundation of Korea(NRF).
[**] Corresponding author.

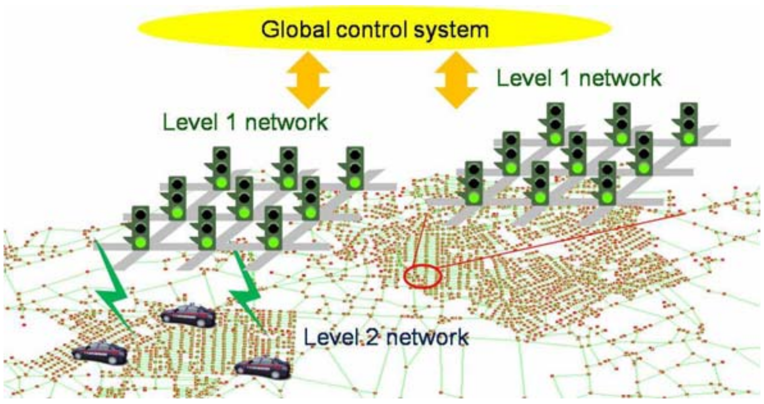


Fig. 1. control architecture

of providing the access point to moving vehicles, can be installed in appropriate locations on the road network. Correspondingly, each component communicates in two-level hierarchy in vehicular networks as shown in Figure 1[2]. Level 1 connection is established between static nodes and level 2 between a static node and moving vehicles. Level 1 communication creates a kind of wireless mesh networks which can be built based on commonly available protocols such as IEEE 802.11, Zigbee, and the like [3]. Based on this network, many vehicular applications can be developed and serviced.

Traffic lights are found in every street, and we think that they can desirably install a wireless communication interface as they have sufficient and reliable power provision and their locations are highly secure. On such a traffic light network, it is possible to implement a monitor-and-control application when the network includes sensors and actuators [4]. Practically, a lot of traffic-related or environmental devices such as speed detectors, pollution meters, and traffic signal controllers, can be added to the vehicular network. Moreover, vehicles can also carry a sensor device and report the collected sensor data to a static node. The message of process control applications is necessarily time critical and how to schedule messages is the main concern. To meet this requirement, the network protocol must provide predictable network access, while a message schedule is determined based on the routing policy, also considering the underlying network topology. Particularly, when traffic lights are placed in each intersection of a Manhattan-style road network, the traffic light network has grid topology.

Meanwhile, the WirelessHART standard provides a robust wireless protocol for various process control applications [5]. First of all, the protocol standard exploits the slot-based access scheme to guarantee a predictable message delivery, and each slot is assigned to the appropriate (sender, receiver) pair. In addition, for the sake of overcoming the transient instability of wireless channels, a special emphasis is put on reliability by mesh networking, channel hopping, and time-synchronized messaging. This protocol has been implemented and is about to be released to the market [6]. Moreover, it can design a split-merge operation to mask channel errors as well as an efficient routing scheme to find the path that is most likely to successfully deliver the message [7].

For the timely control action, the network must deliver the sensor and control message not only reliably but also timely, or as fast as possible. Thus, every message transmission schedule, or slot assignment, is decided in priori. Moreover, one of the most useful communication primitives is the multicast or broadcast. Conceptually, the multicast mechanism ranges from the message relay along the BFS (Breadth First Search) tree to message flooding. Between these two extremes, there are many optimization schemes to compromise reliability and the number of messages. On the other hand, the split-merge operation, which will be explained later, can also improve the reliability and timely delivery of multicast messages. In this regard, this paper is to design and evaluate the performance of an efficient multicast protocol, where some nodes receive a message from two possible senders, for process control messages in the grid-style traffic network.

The paper is organized as follows: After defining the problem in Section 1, Section 2 introduces the background of this paper focusing on the WirelessHART protocol and multicast schemes. Section 3 designs a multicast scheme exploiting the split-merge operation during the message propagation along the multicast tree. The performance measurement result is discussed in Section 4, and finally Section 5 summarizes and concludes this paper.

2 Background and Related Work

The WirelessHART standard is defined over the IEEE 802.15.4 GHz radioband physical link, allowing up to 16 frequency channels spaced by 5 MHz guard band [8]. The link layer provides deterministic slot-based access on top of the time synchronization primitives carried out continuously during the whole network operation time. According to the specification, the size of a single time slot is 10 ms, and a central controller node coordinates routing and communication schedules to meet the robustness requirement of industrial applications. According to the routing schedule, each slot is assigned to a (sender, receiver) pair. For more reliable communication, each sender performs CCA (Clear Channel Assessment) [9] before the transmission. However, how to react when a channel returns bad CCA condition is not yet defined in the current standard.

The split-merge operation, defined on top of the WirelessHART protocol, enables a sender to try another channel if the CCA result of the channel on the primary schedule is not clear. For the destination node of each connection, a controller may reserve two alternative paths having sufficient number of common nodes. Two paths split at some nodes and meet again afterwards. When two paths split, a node can select the path according to the CCA result in a single slot by switching to the channel associated with the secondary route. When two paths merge, the node can receive from either of two possible senders by a timed switch operation. Besides, the WirelessHART protocol can be reinforced by many useful performance enhancement schemes such as the virtual-link routing scheme that combines split-merge links and estimates the corresponding error rate to apply the shortest path algorithm [7].

As for the multicast mechanism in wireless communication, the IEEE 802.11 standard specifies multicast frames by which a node sends a message to all reachable nodes using the basic service set data rate [10]. However, the data rate is limited by the poorest channel between the sender and each receiver, so the data rate is generally much slower than the unicast data rate. Moreover, considering the directional antenna [11], it looks more reasonable to transmit a single multicast packet by multiple unicast packets. For this purpose, a node relays the multicast message according to the prebuilt multicast tree or simply floods to the every nearby downstream node [12]. The first scheme suffers from the packet loss, especially when the loss happens at the high-level node in the multicast tree, while the second scheme cannot avoid so-called multicast storms. Additionally, gossiping is proposed to relieve problems of the flooding scheme. Instead of sending to all of its neighbors, a node randomly chooses some of its receivers. Other schemes consider geographic location, generally partial routing information, and others [13]. Anyway, in the slot-based MAC, the multicast message must be scheduled and assigned to the appropriate slots, and more messages need more time slots [14,15].

3 Routing and Scheduling Scheme

3.1 System Model

The traffic lights form a grid network in modern cities, as a traffic light node is placed at each crossing of the Manhattan-style road network, as shown in Figure 2(a) [16]. Each node can exchange messages directly with its vertical and horizontal neighbors. Two nodes in the diagonal of a rectangle do not have a direct connection, as there may be obstacles like a tall building that blocks the wireless signal propagation. In this network, the central controller is assumed to be located at the fringe of a rectangular area, for this architecture makes the determination of the communication schedule simple and systematic. In Figure 2(a), $N00$ is the controller node of a 3×3 grid. In addition, it must be noted that any grid network can be transformed into this network by partition and rotation. In the example of Figure 2(b), four 4×4 grids are generated and each of them can be mapped to a grid shown in Figure 2(a), regardless of the grid dimension.

In most control applications, the controller initiates a multicast process periodically, so the route and corresponding slot allocation can be decided in priori. How to route a message or how to build a multicast tree is not our concern and any scheme can be integrated into our transmission model [10]. Each node-to-node transmission is assigned to a slot, so the one-hop message relay can succeed only if the channel between two nodes remains good during the slot time. In addition, an acknowledgment scheme can be employed between the receiver and

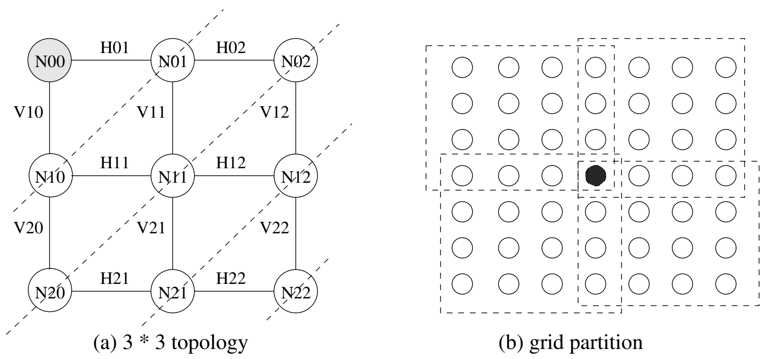


Fig. 2. Traffic light network

sender in the single hop message exchange, while the the retransmission or duplicated transmission can be integrated into the slot schedule. The controller can retransmit to the node which has missed the message after rebuilding a multicast tree connecting such nodes. However, the retransmission mechanism, be it for single-hop or system-wide, is out of scope of this paper.

3.2 Multicast Scheme

This section begins with the basic multicast scheme based on the BFS multicast tree. We call this BFS multicast in short. Practically, every node is the receiver for the multicast from the controller. Here, each node has its predecessor except the controller. A node can receive the message only if its predecessor receives successfully. The message failure can do more harm, when the node is in the upper level of the multicast tree. Figure 3(a) is the example of a multicast tree built based on the BFS traverse order on the 3 $\times$ 3 grid network shown in Figure 2(a). Based on this tree, a slot schedule can be decided as shown in Figure 4(a), where the receiver order corresponds to the BFS order. That is, 1-hop neighbors of $N00$ receive in the first 2 slots. Then, 2-hop nodes, namely, $N20$, $N11$, and $N02$, receives from the 1-hop nodes, and so on. $N22$ is most likely to fail in receiving the multicast message. In this schedule, if time slot 1 is not good, $N10$, $N20$, $N21$, and $N22$ cannot receive the message. Most multicast tree-based schemes suffer from this invulnerability. However, in case the channel condition is good for the most time, just 8 times slots can complete an entire multicast.

To improve the multicast performance on this transmission schedule, it is necessary to enhance each node-to-node delivery ratio, especially for the nodes close to the controller. Accordingly, the controller employs a split operation in the first two slots. Not sending a message one by one to each of two receivers as in the BFS multicast, the controller executes CCA, which needs just 8 bit time according to the standard, and dynamically select the receiver having good condition. Namely, in the first slot, after performing CCAs to $N01$ and $N10$,

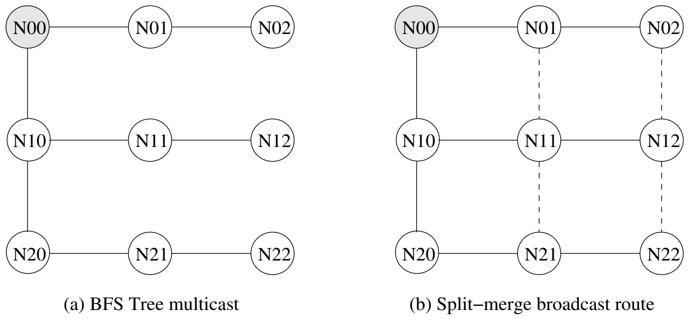


Fig. 3. multicast trees

the controller sends its multicast message to the one which returns clear channel condition. In the second slot, the controller sends to the other one. For this operation to work, $N01$ and $N10$ must wake up and listen during the first two slots. The slot schedule shown in Figure 4(b) has two entries, namely, $N00 \rightarrow N10$ and $N00 \rightarrow N01$, in the first two rows. These rows can be switched according to the CCA result.

In addition, some nodes can receive from two possible predecessors, which correspond to two upstream neighbors. For example, assume that $N10$ and $N01$ have received for the multicast message successfully. Then, $N11$ can receive either of them. Figure 3(b) illustrates this situation. $N11$ has two receive links, one denoted by a solid line to $N10$ and the other by a dotted line to $N01$. It means that $N10$ is the primary sender and $N01$ is the secondary sender. As a result, in Figure 4(b), slot 3 has two transmissions of $N10 \rightarrow N11$ and $N01 \rightarrow N11$. In this slot, any one of $N10$ and $N01$ having a valid message send to $N11$. They must use different frequency channel, while the secondary sender must begin after a predefined delay to give a margin for the receiver to change the listening channel. On the other hand, $N11$ first tries to receive through $N10$, as it is the primary route. If the packet arrives, it receives as scheduled. Otherwise, namely, the message doesn't arrive until the first $TsRxOffset$, the node switches channel bound to the secondary sender.

The message cannot reach a node when its two predecessors cannot sends a message, resulting in the improvement of delivery ratio of the message multicast. In this way, $(n-1) \times (n-1)$ nodes out of $n \times n$ can benefit from the improved message relay in the overall multicast. The proposed scheme, which will be called SM multicast named after split-merge, does not extend the length of the slot assignment schedule. Redundant transmission and channel selection can be done just within a slot. If we are to allocate slots for flooding schemes, the schedule length inevitably grows too much. Instead, SM multicast can carry out another message multicast within the same time interval. More reduction is possible by overlapping transmissions across the consecutive multicasts. For an extreme example time slot 8 of the current multicast and the slot 1 of the next multicast can run concurrently.

slot	action	action	merge
1	N00 –> N10	N00 –> N10	split
2	N00 –> N01	N00 –> N01	
3	N10 –> N20	N10 –> N11	N01 –> N11
4	N10 –> N11	N10 –> N20	
5	N01 –> N02	N01 –> N02	
6	N20 –> N21	N11 –> N21	N20 –> N21
7	N02 –> N12	N02 –> N12	N11 –> N12
8	N21 –> N22	N21 –> N22	N12 –> N22

(b) BFS tree schedule (b) split–merge schedule

Fig. 4. multicast schedule

4 Performance Measurement

This section measures the performance of the proposed multicast scheme via simulation using SMPL which provides abundant functions and libraries for discrete event scheduling, while combined easily with the commonly used compilers such as *gcc* and *Visual C++* [17]. Only the downlink graph was considered for simplicity, as uplink and downlink communications are symmetric and calculated completely in the same way. In addition, the slot error rate is set to be the same for all links to give focus on the performance of the multicast scheme. It must be mentioned that the routing scheme can find a better route to cope with the case the slot error rate is different link by link. The slot error rate depends on the data size and the channel error rate distribution. Here, we employ Guilbert-Elliot error model [18], which is quite simple, but can easily set the average error rate we want for the experiment.

The first experiment measures how many nodes can receive a multicast message from the controller, according to the grid dimension. We change the dimension from 3 to 15, 500 multicasts are generated for each dimension, and the percentage of nodes receiving the message was averaged. The slot error rate is set to 0.01. As shown in Figure 5, the delivery ratio for the BFS multicast drops sharply according to the increase of the grid dimension, reaching 50 % when the dimension is 15. As contrast, SM multicast is hardly affected. The message failure recovery at the high-level multicast tree can keep low the overall failure. The SM curve stays above 0.95 for all dimension ranges, indicating that 1 out of 20 nodes receive the multicast message from the controller. It seems that additional error recovery will not be necessary. Actually, SM has an effect of reducing the slot error rate to almost its square.

The second experiment measures the delivery ratio according to the slot error rate for the 4×4 grid. We change the slot error rate from 0 to 0.3. The wireless channel experiences diverse slot error rates. Figure 6 plots the measurement result. As shown in the graph, the performance gap gets larger according to the increase of the slot error rate, reaching 35.6 % at the error rate of 0.18. After that point, the influence of severe error rate exceeds the effect of error recovery.

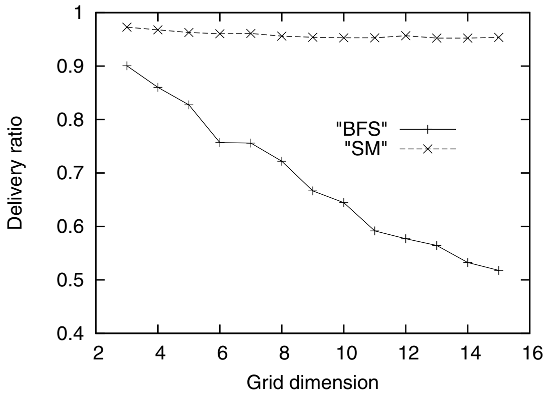


Fig. 5. Effect of grid dimension

In addition, the proposed scheme is not so much affected by the slot error rate until 0.1, then it begins to decrease a little bit sharply. As contrast, the delivery ratio of the BFS scheme drops by the significant amount from the first.

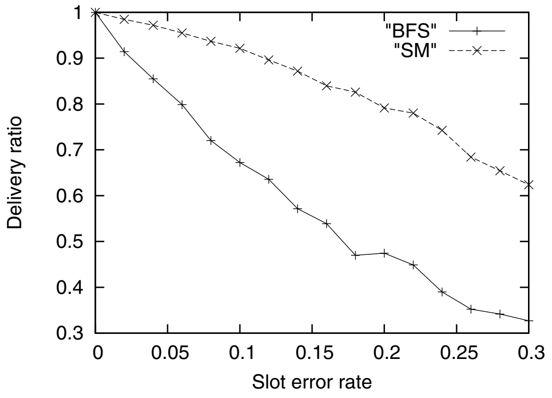


Fig. 6. Effect of slot error rate

In the process control system, the same command can repeat for times to mask the communication error in the previous message multicast or make the actuator continue its operation. In this situation, consecutive message losses can lead to undesirable malfunction. So, the third experiment measures the maximum of consecutive multicast errors according to the slot error rate. In this experiment, the slot error rate has a value of 0.0 through 0.3. and the grid dimension is set to 5. Figure 7 plots the maximum of consecutive message losses for $N44$ which is farthest away from the controller and has the lowest receive ratio. As can be inferred from the figure, the node doesn't miss more than 8 times in a row in

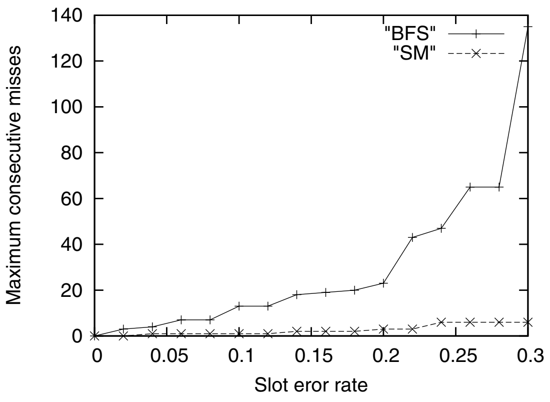


Fig. 7. Maximum consecutive loss

the SM multicast even when the slot error rate is 0.3. As contrast, the node can miss more than hundred times consecutively in BFS multicast. This experiment demonstrates that our scheme can work very reliably for the process control application.

5 Conclusions

This paper designs and analyzes the performance of an efficient and robust multicast scheme for the grid-style wireless network built upon the WirelessHART protocol. The proposed scheme exploits the split-merge operation which provides a channel switch, or receive frequency change, within a slot, aiming at improving the reliability of wireless process control without extending the slot assignment schedule. With the modification of split operation, the controller node selects the transmission order to its downstream neighbors according to the current channel condition returned from the CCA result specified in the WirelessHART standard. In addition, using the merge operation, the nodes having two predecessors listen first from the primary sender and then switches to the secondary sender when the message doesn't arrive within the predefined time bound. The simulation result obtained from SMPL shows that the proposed scheme can improve the multicast delivery ratio by up to 35.6 %, compared with the BFS multicast, makes the delivery ratio stay above 95 % for all given grid dimension range, and finally maintains the consecutive message losses below 8.

As future work, we are planning to find how to build the multicast tree and corresponding slot assignment schedule which can meet the specific application requirement such as maximizing the delivery ratio, minimizing the schedule length, isolating the failed node, and the like.

References

1. Lee, J., Park, G., Kim, H., Yang, Y., Kim, P., Kim, S.: A telematics service system based on the Linux cluster. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4490, pp. 660–667. Springer, Heidelberg (2007)
2. Bucciol, P., Li, F.Y., Fragoulis, N., Vandoni, L.: ADHOCSYS: Robust and service-oriented wireless mesh networks to bridge the digital divide. In: IEEE Globecom Workshops, pp. 1–5 (2007)
3. Gislason, D.: ZIGBEE Wireless Networking. Newnes (2008)
4. IEC/PAS 62591: Industrial communication networks - Fieldbus specifications - WirelessHART communication network and communication profile (2008)
5. Hart Communication Foundation, Why WirelessHART^{TM}? The Right Standard at the Right Time (2007), http://www.hartcomm2.org
6. Han, S., Song, J., Zhu, X., Mok, A.K., Chen, D., Nixon, M., Pratt, W., Gondhalekar, V.: Wi-HTest: Compliance test suite for diagnosing devices in real-time WirelessHART network. In: The 15th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 327–336 (2009)
7. Lee, J., Song, H., Mok, A.K.: Design of a reliable communication system for grid-style traffic control networks. In: Accepted at the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (2010)
8. Song, S., Han, S., Mok, A.K., Chen, D., Nixon, M., Lucas, M., Pratt, W.: WirelessHART: Applying wireless technology in real-time industrial process control. In: The 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 377–386 (2008)
9. Ramchandran, I., Roy, S.: Clear channel assessment in energy-constrained wideband wireless networks. IEEE Wireless Magazine, 70–78 (2007)
10. Chakeres, I., Koundinya, C., Aggarwal, P.: Fast, efficient, and robust multicast in wireless mesh networks. In: 5th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks, pp. 19–26 (2008)
11. Dai, H., Ng, K., Wu, M.: An overview of MAC protocols with directional antennas in wireless ad hoc networks. In: Proc. International Conference on Computing in the Global Information Technology, pp. 84–91 (2006)
12. Penttinen, A.: Efficient multicast tree algorithm for ad hoc networks. In: IEEE MASS, pp. 519–521 (2004)
13. Yen, W., Chen, C., Yang, C.: Single gossiping with directional flooding routing protocol in wireless sensor networks. In: 3rd IEEE Conference on Industrial Electronics and Applications, pp. 1604–1609 (2008)
14. Huang, S., Wna, P., Jia, X., Du, H., Shang, W.: Minimum-latency multicast scheduling in wireless ad hoc networks. In: IEEE INFOCOM, pp. 733–739 (2007)
15. Wan, P., Huang, S., Wang, L., Wan, Z., Jia, X.: Minimum latency aggregation scheduling in multihop wireless networks. In: MobiHoc, pp. 185–193 (2009)
16. Jaap, S., Bechler, M., Wolf, L.: Evaluation of routing protocols for vehicular ad hoc networks in city traffic scenarios. In: Proceedings of the 5th International Conference on Intelligent Transportation Systems Telecommunications (2005)
17. MacDougall, M.: Simulating Computer Systems: Techniques and Tools. MIT Press, Cambridge (1987)
18. Bai, H., Atiquzzaman, M.: Error modeling schemes for fading channels in wireless communications: A survey. IEEE Communications Surveys, 2–9 (2003)

A Note-Based Randomized and Distributed Protocol for Detecting Node Replication Attacks in Wireless Sensor Networks

Xiangshan Meng, Kai Lin, and Keqiu Li*

Department of Computer Science and Engineering
Dalian University of Technology
No2, Linggong Road, Dalian 116023, China
keqiu@dlut.edu.cn

Abstract. Wireless sensor networks are often deployed in hostile environments and sensor nodes are lack hardware support for resistance; thus, leaving them vulnerable to several kinds of threats and attacks. While most of threats and attacks can be prevented by using cryptographic approaches provided by key management, such as eavesdropping, intrusion and node compromising. Unfortunately node replication attacks can still go undetectable. In node replication attacks, an attacker compromising a node, uses its secret cryptographic key materials to populate the network with several clones of it. Several node replication detect protocols were proposed. However, centralized protocols have a single point of failure, local protocols fail to detect distributed replications, and distributed protocols need nodes know their exact geographic locations. This paper proposes a note-based protocol for detecting node replication attacks, which introduces no significant overhead on the resource-constrained sensors. The proposed protocol needn't the geographic locations of nodes as well. Moreover, further analysis shows that it achieves a much higher probability of detecting replications.

1 Introduction

A Wireless Sensor Network (WSN) is a large collection of sensors with limited resources. Cost considerations make it impractical to use shielding that could fight against pressure, voltage, and temperature changes [1,2,3]. Because of the lack of resources, sensors are easy to get exhausted. Sensors are often deployed in harsh environments where they are vulnerable to be captured and compromised by an adversary. An adversary can compromise a single node, use some approaches to access the sensor's internal state, get the credentials of the node and other information. A serious consequence is that the adversary can replicate it indefinitely, and then insert the replicas at strategic locations within the network. The replicas make the network vulnerable to a large of attacks, such as DoS attacks, injecting false data, subverting data aggregation and so on.

Different protocols were proposed for detecting node replication attacks. Two reference protocols are centralized protocols and local protocols. Centralized protocols

* Corresponding author.

typically rely on a centralized base station [6]. They require all of the nodes in the network to send a list of their neighbors' claimed location to the base station. These protocols have a single point of failure like all centralized schemes. Local protocols [7,8,9] depend primarily on a neighborhood voting mechanism, and they can't detect replications that are distributed in the network. Several distributed protocols [4,5,16] were proposed to detect node replication attacks. They all require nodes in the network know their exact geographic locations. These protocols require every node to transfer its location information to a set of nodes called witness that can examine the conflict. However, not every WSN application is location aware, which limits their application in some scenarios.

In this paper, we analyze the related node replication detect protocols, then we present a note-based protocol NRDP for detecting node replication attacks. Extensive simulations show that NRDP is more energy efficient, and it detects node replication attacks with much higher probability than existing protocols.

The remainder of the paper is organized as follows: In section 2, we overview some related works on detection of node replication attacks. In section 3, we describe our network model and adversary model. In section 4, we describe our proposed protocol NRDP, and we give a detailed theoretical efficiency and security analysis in section 5. In section 6, we show some experimental results on NRDP and compare it with other distributed protocols. We conclude our work in section 7.

2 Related Work

A centralized protocol [6] relies on a centralized base station (BS). Each node sends a list of its neighbors and their claimed locations to the BS. The BS can then examine every neighbor list to look for replicated nodes. Finally the base station can revoke the replicated nodes by flooding the network with an authenticated revocation message. This solution has a single point of failure, and it requires a high communication cost. Further, nodes close to the BS will exhaust their power earlier than others because of funneling effect. Local protocol is also a kind of solution for detecting replication attacks. A voting mechanism is used on a node's neighbors in [6,7,8,9]. The neighbors can reach a consensus on the legitimacy of a given node. But those protocols fail to detect replicas two or more hops away from each other.

Several distributed detect protocols were proposed for detecting node replication attacks. We adopt some notations in [17]. In these protocols, every node broadcasts its ID and location to one-hop neighbors. We call this message a **claim** and the node that broadcasts a claim a **claimer node**. Upon receiving a claim message, each neighbor with probability p_f forwards the claim message to a set of nodes called **witnesses**. A neighbor node which forwards a claim, we call it a **reporter node**. If a witness node receives two or more claim messages containing the same ID but different locations, the witness node detects a replication attack.

The first distributed node replication detect protocol was proposed in [4]. Two distributed protocols were proposed: Randomized Multicast (RM) and Line Select Multicast (LSM). RM protocol propagates claim message to randomly selected witness nodes. When a claimer node broadcasts its location claim, each of its neighbors with probability p_f propagates the claim to a set of randomly selected witness nodes. According to the Birthday Paradox, at least one node is likely to receive conflicting location claims of a particular node. Each neighbor needs to send $O(\sqrt{n})$ messages. The notation n is the number of sensors in the network.

LSM protocol behavior is similar to RM but introduces a modification that achieves a noticeable improvement in terms of detection probability and communication cost. When a node broadcasts its location claim, every neighbor forwards this claim with probability p_f. If a neighbor forwards the claim, it randomly selects a fixed number g witness nodes, and sends the signed claim to all the g nodes. The number of witness nodes g can be much smaller than in RM. Every node that is routing the claim message must to check the signature of the claim, then store the signed claim, and check for coherence with the other location claims stored within the same detect iteration. So, the forwarding nodes are also witness nodes of the claimer node. Node replication is likely detected by the nodes on the intersection of two route paths that originate from different locations by the same ID.

Two distributed replication detect protocols SDC and P-MPC were proposed in [16]. The network is considered to be a geographic grid in the study. In the SDC protocol, a geographic hash function is used to uniquely and randomly map a node's identity to one of the cells in the grid. The location claim message is forwarded to the mapping cell. When the first copy of the location claim arrives at the destination cell, the location claim is flooded within the cell. The nodes in the cell randomly become witness nodes. In P-MPC, to increase the reliability to a large amount of replication nodes, a node's identity is mapped to several cells in the grid. So, the candidate witness nodes for one node are nodes of several cells.

An efficient, distributed protocol RED was proposed in [5]. Different from RM and LSM, all reporter nodes of a particular claimer node α would choose the same g witness nodes for α, while in RM and LSM, each reporter node randomly determines a set of witness nodes. In RED protocol, the witness nodes' locations are determined by the claimer node ID and the seed $rand$. A trusted entity broadcasts a seed to the whole network in each detect iteration. Because the seed changes in every detect iteration, so the attacker cannot anticipate the witness nodes. As described above, each neighbor node of a claimer node with probability p_f becomes reporter node and forwards the claim message to g witness nodes. The larger p_f is, the higher the success detect rate is, and a claimer node tends to have more reporter nodes. But if there are more than one reporter node for a claimer node, the corresponding witness nodes will receive the same claim message several times. Actually, only one claim message is necessary per claimer node for the witness nodes.

3 System Model

3.1 Network Model

We consider a sensor network with a large number of nodes distributed over a wide area. The nodes in the network are relatively stationary. Except we don't need the locations of nodes in the network, all the assumptions are the same with other node replication detect protocols in [4,5,16]. Different from other replicated node detect protocols, we don't require nodes know theirs geographic locations. In order to obtain the locations of nodes, several beacon nodes equipped with GPS devices are needed, or a location discovery protocol such as [10] needs to be executed in the network. It will lead to a significant communication cost and a hardware cost. We also assume that the network is loosely time synchronized. This can be achieved by schemes proposed in [4,11,18]. Each node is assigned a unique identity and a pair of identity-based public and private keys by a trust authority (e.g., BS). In identity-based signature protocols like [12], the public key is usually a hash on its unique identity, and the private key is generated by signing its public key with a master secret held only by the BS. To generate a new identity-based private and public key pair, cooperation of the BS is required. Because they cannot generate the key pair corresponding to the identities, the adversaries cannot create sensors with new identities. We also assume that message forwarding is not affected by dropping or wormhole attacks. These threats can be addressed by protocols proposed in [13,14,15].

3.2 Adversary Model

We assume that the adversary has the ability to surreptitiously compromise a limited number of legitimate nodes. Once a node is compromised, all the secret keys, data, and code stored in it are exposed to the adversary. The adversary could then clone the node by loading the node's cryptographic information onto multiple replicated sensor nodes, and then insert the replicas in different locations within the network. The replicated nodes can establish communications with their neighbors using the stolen cryptographic keys. These replicated nodes can be the basis for various attacks.

4 The NRDP Protocol

4.1 Overview

In this section we propose a note-based, randomized and distributed protocol NRDP for detecting node replication attacks. There are three key words in NRDP: **claimer node**, **reporter node** and **witness node**. We call a node which broadcasts a claim message a claimer node. Neighbor node which forwards a claim message is a reporter node. And we call the destination node of a claim message a witness node.

In related protocols, each neighbor node of a claimer node with a fixed probability p_f becomes reporter node and forwards the location claim for the claimer node. To get a related high replica detect rate, a large p_f is required, p_f and neighbor degree d determine that there are always more than one reporter node for a claimer node. In this case, several identical claims are forwarded to a witness node. However, only one claim is necessary per claimer node for the witness node. For example, if d = 10, to detect a singe replication of node α with probability greater than 95%, p_f has to be greater than 0.308. In this case, there are about 3 reporter nodes for a claimer node in average. Two reporter nodes are redundant.

In our protocol, a claimer node α randomly specifies a reporter node γ from its neighbor nodes and requests a signature note from γ. After α gets the signature note from γ, it tells its neighbors the ID of γ by broadcasting the note along with its claim in one-hop neighborhood. The claim containing a sub neighbor list of α. The note makes sure that a claimer node cannot specify a nonexistent ID of its neighbor nodes as its reporter node. Upon receiving the corresponding claim, using a pseudo-random function γ chooses g witness nodes for α. Then node γ certainly forwards the claim to α's witness nodes, no matter γ is a valid node or not. This is because we assume that message dropping attacks can be addressed by the surrounding nodes. If a witness node receives two claim messages containing a same ID but different neighbor lists in one detect iteration, it detects a replication attack. The two signature claim messages become evidence to trigger a revocation of the replicated node.

4.2 Details of Protocol

In the beginning of NRDP, it is a **neighbor discovery period**. Each node in the network broadcasts a message within its one-hop neighbors. After neighbor-discovery period, each node in the network gets a neighbor list.

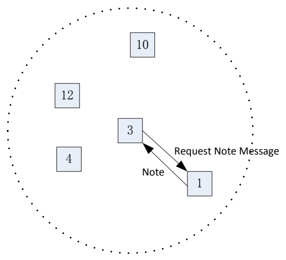
Algorithm 1. NRDP Algorithm

```
1  rand ← GlobalBroadcastDevice;
2  Set time-out $\Delta t$;
3  Neighbor node r ← OneNeighborOf(a);
4  $a \to r$:   $< ID_a, ID_r$, RequestNote, $(ID_a, R, time_a, K_a^-(H(ID_a, R, time_a))>$;
5  while $\Delta t$ not elapsed and ReceiveMessage(M) do
6      if IsRequestNote(M) then
7          $<$-, -, -, $(ID_x, R, time_x, SignedRqst_x)> \leftarrow M$;
8          $a \to x : < ID_a, ID_x$, Note, $(ID_a, N, time_a, K_a^-(H(ID_a, N, time_a))>$;
9      if IsNote(M) then
10         $<$-, -, -, $(ID_x, N, time_x, SignedNote_x)> \leftarrow M$;
11         $a \to$ NeighborsOf(a): $<ID_a$, NeighborsOf(a), IsClaim, $(ID_a, list_a, time_a, ID_x,$
               $SignedNote_x, K_a^-(H(ID_a, list_a, time_a))>$;
12     if IsClaim(M) then
13         $<$-, -, -, $(ID_x, list_x, time_x, ID_r, Note_r, SignedClaim_x)> \leftarrow M$;
14         if Equal$(ID_r, ID_a)$ then
15             CheckJobs();
16             $WitnessesIDSet_x \leftarrow$ PseudoRand$(ID_x$, rand, g);
17             for each $ID_{dst} \in WitnessesIDSet_x$ do
18                 $a \to ID_{dst}$: $< ID_a, ID_{des}$, ForwardedClaim, $(ID_x, list_x, time_x)$,
                       $SignedClaim_x>$;
19         else
20             with probability $p_c$ CheckJobs();
21     if IsForwardedClaim(M) then
22         $<$-, -, -, $(ID_x, list_x, time_x, SignedClaim_x)> \leftarrow M$;
23         CheckingJobs();
24         if IsNotPresent(Memory, $ID_x$) then
25             Add(Memory, $ID_x, list_x, SignedClaim_x$);
26         else
27             $<ID_{x'}, list_{x'}, SignedClaim_{x'}> \leftarrow$ LookUpID(Memory, $ID_x$);
28             if IsNotCoherent$(list_x, list_{x'})$ then
29                 $a \to BS$: $< ID_x, list_x, list_{x'}, SignedClaim_x, SignedClaim_{x'}>$;
30 Clear Memory;
```

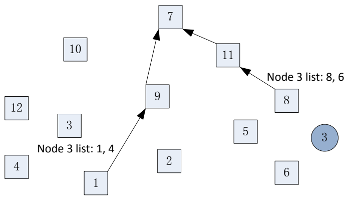
The pseudo code for every node is described in Algorithm 1. The **replication detect period** starts when the neighbor discovery period ends. Replication detect period consist of two steps. We call the first step **request-note step**, and the second step **send-claim step**. In request-note step, node α randomly chooses a node γ from its neighbor list as its **reporter node**, and then sends a **request-note message** to the reporter node (Algorithm 1, line 3, 4). Upon receiving α's request-note message, node γ replies with a signature note message which containing a **note** (Algorithm 1, line 6, 7, 8). The parameter *time* is fresh time of the note. Nodes in the network use it to identify the validity of a note received in different iterations. Note is an evidence to prove that the reporter node of a claimer node is existing and valid. In the send-claim step, every node generates a **claim message**, which includes a signed sub neighbor list and a note got from the corresponding reporter node. The parameter $list$ in the claim message is an ID list, which consists of q α's neighbor node IDs. And the reporter node γ must be in the list. Each node α then broadcasts the claim message in one-hop neighbors (Algorithm 1, line 11). When the reporter node receives corresponding claim message, it first verifies the signature and the time fresh of the note containing in the claim message. Further, the reporter node verifies that the $list$ in the claim message containing its ID. If all the verifications succeed, using a pseudo-random function $PseudoRand$ the reporter node calculates g witness nodes for the claimer node (Algorithm 1, line 16). This function takes in input the ID of the claimer node, that is the first argument of the claim message, the current $rand$ value, and the number g of witness nodes that have to be generated. In order to make the witness nodes of a certain node change in different iterations, we adopt the method in [5]: A trusted entity broadcasts a seed $rand$ to the network before each detect iteration start. By doing this, we can prevent the adversary from anticipating the witness nodes in a given protocol iteration. The reporter node analyzes the claim message, then generates a **forwarded claim message**, and forwards the forwarded claim message to all the g witness nodes (Algorithm 1, line 17). The forwarded claim message just contains the sub neighbor list signed by claimer node, without note.

When a node receives a claim message, it first checks whether it is the corresponding reporter node (Algorithm 1, line 12, 13, 14). If it is the reporter node of the claimer node, it checks the signature, the fresh of the note and the $list$ in the claim message. If it is not the reporter node, with probability p_c it does the checking jobs as the reporter node does. It is necessary for non-reporter node neighbors to do the checking jobs with probability p_c. By doing this, we can prevent a claimer node from specifying a non-existing neighbor node as its reporter node. Each node in the network has to specify an actual neighbor node as its reporter node, or it will be detected as a replicated node by its neighbor nodes.

Each witness node that receives a forwarded claim message, verifies the signature and time fresh firstly. Then, it compares the claim to each previously stored claim. If it is the first time received claim containing ID_α, then it simply stores the claim (Algorithm 1, line 24, 25). If a claim from ID_α has been received, the witness checks whether the claimed neighbor list is same with the stored claim. If a conflict is found, the witness detects a node replication attack. Then the witness triggers a revocation procedure for ID_α. Actually, because there is always only one reporter node for a claimer node, if the



(a) Request Note Step



(b) Send Claim Step

Fig. 1. The Two Steps of NRDP Protocol

claimer node is a valid node, its corresponding witness nodes would never receive more than one forwarded claim message from the claimer node. Therefore, once a witness node receives two claims containing the same ID in one detect iteration, it detects a replication attack. The two signature claims become evidence to trigger the revocation of the replicated node. The witness node forwards both claims to the base station (Algorithm 1, line 28, 29). The base station will broadcast a signature message within the network to revoke the replicated node.

The NRDP protocol runs as illustrated in Figure 1. Assume that the adversary compromises node 3 and clones it one time. These two nodes are placed in two different network locations, as illustrated in Figure 1b. Node 3 and the replica establish connections with surrounding nodes using the same key materials. Node 3 and its replica node have the same ID but different neighbor nodes. During a NRDP detect iteration, node 3 and the replica would broadcast the same identity 3, but different sub neighbor lists. In the request-note step, node 3 chooses node 1 from its neighbor nodes as its reporter node, and then requests a signature note from node 1, as showed in Figure 1a. In the same way, the replicated node of node 3 also gets a signature note from its reporter node 8. In the send-claim step, each node generates a claim message which contains a sub neighbor list and a note, broadcasts it in one-hop neighbors. The reporter node of the claimer node have to be in the list. As showed in Figure 1b, node 3's sub neighbor list is $< list_1 : 1, 4 >$, and the sub neighbor list of the replica is $< list_2 : 8, 6 >$. When a reporter node receives corresponding claim message, it calculates g witness nodes for the claimer node, generates a forwarded claim message and then sends to the witness nodes. When $g = 1$, using the pseudo-random function both node 1 and 8 would select the same witness node 7 for identity 3. Therefore, witness node 7 will receive two non-coherent claims from identity 3. So the witness node detects a replication attack. Then the witness node 7 forwards the two conflicting signature claims to the BS. BS will revoke the compromise node by broadcasting the identity of the replicated node in the whole network.

In order to be undetected by NRDP protocol in the network, replicated nodes with the same identity may try to claim a same sub neighbor list. For example, as showed in Figure 1b, the replica of node 3 may try to claim a false sub neighbor list $< list_2^{'} : 1, 4 >$ as same as node 3' claim. Please note that, when a node broadcasts a claim message, it has to show a valid note from the corresponding reporter node to its neighbors. Because the replica of node 3 can't get the valid note corresponding to $list_2^{'}$ from node 1. It would be detected as a replicated node by its neighbor nodes. In this situation, we detect a replication attack while pay for no propagation message overhead.

5 Protocol Analysis

5.1 Efficiency Analysis

There are three metrics to evaluate the efficiency of replication detect protocol: communication cost, storage cost and computation cost. The main computation cost of NRDP is the signature check. We measure the communication cost by the average number of the packets sent and received while propagating the claim message per node. The storage cost is measured by the average number of copies of the claim message stored in a sensor node. And the computation cost is measured by the average times of signature check per node. According to [4], in a network randomly deployed on a unit square, the average distance between any two randomly chosen nodes is approximately $0.521\sqrt{n} \approx \frac{\sqrt{n}}{2}$. So it is easy to compute out the communication cost of NRDP, RED, SDC and P-MPC. We show the summary of protocol cost for per node in Table 1, where d is the average neighbor degree, g is the number of witness nodes selected by each reporter node, p_f is the probability a neighbor node becomes a reporter node, and p_c is the probability non-reporter neighbor node does the checking jobs when receives a claim message. In SDC and P-MPC, the notation s is the number of nodes in a cell, and w is the number of the witness nodes that store the claim from a claimer node.

Table 1. Summary of Protocol Cost per Node

Protocol	Communication	Storage	Signature
NRDP	$O(\sqrt{n})$	$O(g)$	$O(p_c \cdot d) + O(g)$
RED	$O(p_f \cdot d \cdot \sqrt{n})$	$O(g)$	$O(p_f \cdot g \cdot d)$
SDC	$O(p_f \cdot d \cdot \sqrt{n}) + O(s)$	$O(w)$	$O(d)$
P-MPC	$O(p_f \cdot d \cdot \sqrt{n}) + O(s)$	$O(w)$	$O(d)$

5.2 Security Analysis

Let the adversary capture a node α, clone it t - 1 times, and insert the replicas at t - 1 locations: $l_1, l_2, l_3, ..., l_{t-1}$. We denote the replicated node at location l_i as α_i. Thus, there are t sensor nodes in the network having the same ID but in different locations. We would like to determine the probability of a successful detection using our NRDP protocol.

After the adversary has deployed the replicas in the network, each replica node α_i would establish a connection with the surrounding nodes. So, each node α_i gets a new, actual neighbor list, denoted as nb_list_i. Node α's neighbor list is denoted as nb_list. For all the $t-1$ replicated nodes, there are two choices when the replication detect period starts. That is:

1. Node α_i claims a sub neighbor list from its actual neighbor list nb_list_i.
2. Node α_i tries to claim the same neighbor list as node α. Node α_i claims a sub neighbor list from node α's neighbor list nb_list.

In the situation 1, the corresponding witness nodes would receive two or more conflicting claims from the same ID with different neighbor lists. The witness nodes certainly detect a replication attack. In the situation 2, when a claimer node α_i specifies a reporter node γ from α's neighbor list nb_list, then tells its actual neighbors the reporter node γ's ID, the claimer node α_i has to show a fresh note signed by γ to its neighbors. Please note that, node γ is a neighbor node of α, and γ is not an actual neighbor node of α_i. Therefore, the replicated node α_i cannot get a valid note which is fresh and signed by γ. In this situation, no claim message is propagated to any witness nodes. We pay for no propagating communication cost for detecting the replicas.

A replicated node either is detected by its neighbors, or is detected by its witness nodes. If a replica is not detected by its neighbors, NRDP has a 100% detect rate in theory, when at least two conflicting claims arrive at a corresponding witness node.

5.3 Resilience against Node Compromise

In the discussion of our replication detect protocol, we have assumed that each node has at least one legitimate witness node. Without this assumption, all the replication detect protocols including ours would fail to detect node replication. If the adversary compromises all of node α's g witness nodes in one detect iteration, then he can create several replicas of α without fear of detection. However, even the adversary can compromise all the witness nodes of α in one detect iteration, he trends to be failed in next detect iteration. This is because that the witness nodes of α in next detect iteration would be different. Here we assume the adversary has the ability to compromise t nodes in one detect iteration. We would like to determine the probability p_r that the adversary can

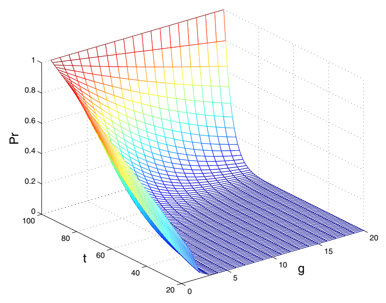


Fig. 2. p_r under different g and t when $n = 100$

compromise all the g witness nodes storing the location claim of a given identity. Assuming that the adversary has compromised t nodes in a network of n nodes. p_r can be calculated as follows:

$$p_r = \frac{C_{n-g}^{t-g}}{C_n^t} = \frac{t(t-1)(t-2)(t-3)\cdots(t-g+1)}{n(n-1)(n-2)(n-3)\cdots(n-g+1)} \tag{1}$$

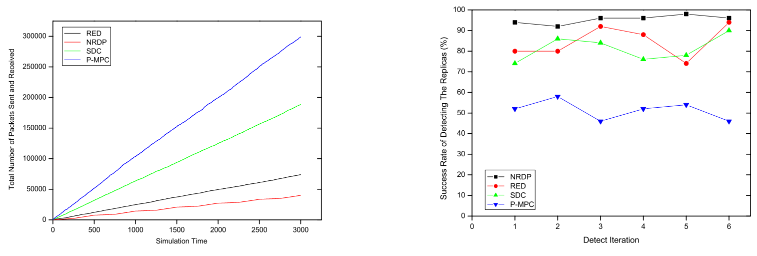
When the network size is 100, we plot the probability that an adversary controls all the witness nodes of a given node in Figure 2. In particular, when g = 15 and t = 60, p_r is only 2.1×10^{-4}. Even if g is chosen a relative small value, e.g 4, the adversary still needs to compromise 57 nodes out of 100 nodes in the network to achieve a success rate of 10%. Figure 2 shows that, when g is chosen appropriately, p_r is negligible, even if the adversary can compromise a large number of nodes in the network.

6 Simulations

In this section, we show that NRDP is low overhead, high replication attack detect probability by simulation. We compare NRDP with RED, SDC, P-MPC in our simulation. Simulation result shows that NRDP outperforms other detect protocols in both communication cost and detect probability.

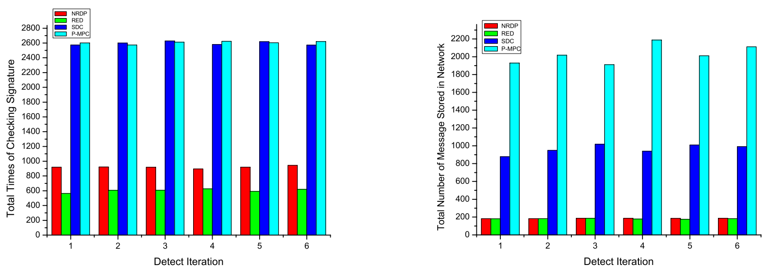
We randomly deploy 200 nodes at a uniform 400×400 square. We assume a unit-disc bidirectional communication model. And we adjust the communication range, to make sure the average neighbor degree d is 10. We fix the size of sub neighbor list q in claim message to 2 in our simulation. We use the total number of packets sent and received in the network as the metric of communication cost. In our simulation we set the number of witness nodes for a claimer node g = 1 for all protocols. We set the average number of reporter nodes per claimer node to 3, it makes RED, SDC and P-MPC all give a detect probability lower than 95% in theory to detect a single replication. Please note that our NRDP has a 100% detect probability in theory. Protocols for detecting node replication attacks are actually independent of the routing protocols used in the network, so we consider the same routing protocol for a fair comparison.

We first compare the communication cost and the detect probability. We deploy 10 single replicas in the network, and set the time of each detect iteration to 500s. We



(a) Communication Cost (b) Probability of Detection

Fig. 3. Communication Cost and Probability of Detection



(a) Signature Cost (b) Storage Cost

Fig. 4. Signature Cost and Storage Cost

assume that a replicated node would not trigger any revocation procedure. Figure 3a indicates the total number of packets sent and received of the entire network along with the simulation time. And Figure 3b indicates the corresponding detect probability. Each detect iteration consists of two steps: request-note step and send-claim step. Therefore, we can see from Figure 3a that the line in each detect iteration is composed by two segment lines. And the two segment lines have different slopes. Figure 3a also indicates that our protocol has much lower communication overhead that all other protocols. We can see from Figure 3b that NRDP gives a detect rate lower than 100%. This is because replicated nodes would not trigger a revocation procedure, and a claim message may be lost in the route path to the corresponding witness node. NRDP achieves a relative higher detect probability than other protocols.

We further compare the signature and storage overhead in our simulation. We show the simulation results in Figure 4. We can see that NRDP and RED have a much lower signature and storage overhead than SDC and P-MPC. Comparing to RED, NRDP has a higher signature overhead. These extra signature overhead is generated by the note checking jobs. The storage overhead of NRDP is nearly the same with RED.

7 Conclusion

In this paper, we propose a note based, randomized, distributed protocol NRDP for detecting node replication attacks. The previous distributed detect protocols [4,5,16] all require nodes know their geographic locations. The first contribution of our study is that NRDP can detect replication attacks without location information. Further more, we achieve a much higher detect probability while cost a lower communication overhead. Both our theoretical analysis and simulation result show that our scheme is more efficient than previous methods in terms of communication cost and detect probability.

Acknowledgment

This work is supported by NSFC under grant nos of 90718030, 60973116, and 90818002.

References

1. Dyer, J., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S.W., Weingart, S.: Building the IBM 4758 Secure Coprocessor. IEEE Computer (2001)
2. Smith, S.W., Weingart, S.: Building a high performance, programmable secure coprocessor. Computer Networks, Special Issue on Computer Network Security (April 1999)
3. Weingart, S.: Physical security devices for computer subsystems: A survey of attacks and defenses. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 88–95. Springer, Heidelberg (2000)
4. Parno, B., Perrig, A., Gligor, V.: Distributed detection of node replication attacks in sensor networks. In: Proceedings of The 2005 IEEE Symposim on Security and Privacy, pp. 49–63 (2005)
5. Conti, M., Di Pietro, R., Mancini, L.V., Mei, A.: A Randomized, Efficient, and Distributed Protocol for the Detection of Node Replication Attacks in Wireless Sensor Networks. In: MobiHoc 2007, September 9-14 (2007)
6. Eschenauer, L., Gligor, V.: A key-management scheme for distributed sensor networks. In: Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), pp. 41–47 (2002)
7. Chan, H., Perrig, A., Song, D.: Random key predistribution schemes for sensor networks. In: Proceedings of 2003 IEEE Symposium on Security and Privacy, pp. 197–213 (2003)
8. Douceur, J.R.: The sybil attack. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2001), pp. 251–260. Springer, Heidelberg (2002)
9. Newsome, J., Shi, E., Song, D., Perrig, A.: The sybil attack in sensor networks: analysis & de-fenses. In: Proceedings of ACM IPSN 2004, pp. 259–268 (2004)
10. Caruso, A., Urpi, A., Chessa, S., De, S.: Gps-free coordinate assignment and routing in wireless sensor networks. In: Proceedings of IEEE INFOCOM 2005, pp. 150–160 (2005)
11. Elson, J., Girod, L., Estrin, D.: Fine-grained network time synchronization using reference broadcasts. SIGOPS Operating Systems Review 36(SI), 147–163 (2002)
12. Hess, F.: Efficient identity based signature schemes based on pairings. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 310–324. Springer, Heidelberg (2003)
13. Marti, S., Giuli, T.J., Lai, K., Baker, M.: Mitigating routing misbehavior in mobile ad hoc networks. In: ACM OBICOM, pp. 255–265 (2000)
14. Wang, G., Zhang, W., Cao, G., Porta, T.: On supporting distributed collaboration in sensor networks. In: IEEE MILCOM (2003)
15. Deb, B., Bhatnagar, S., Nath, B.: Reinform: Reliable information forwarding using multiple paths in sensor networks. In: 28th IEEE LCN, p. 406 (2003)
16. Zhu, B., Addada, V.G.K., Setia, S., Jajodia, S., Roy, S.: Efficient distributed detection of node replication attacks in sensor networks. In: IEEE ACSAC, pp. 257–267 (2007)
17. Sei, Y., Honiden, S.: Reporter node determination of replicated node detection in wireless sensor networks. In: Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication (2009)
18. Elson, J., Estrin, D.: Time synchronization for wireless sensor networks. In: Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS 2001), pp. 1965–1970 (2001)

Author Index

Abbes, Heithem I-287
Alvarez-Bermejo, Jose Antonio I-454
An, Hong II-32
Aracil, Joan II-454
Araújo, Guido I-499
Arul, Joseph M. I-205
Awwad, Ahmad II-1

Bahig, Hazem M. II-391
Bai, Yuebin I-363, I-391
Bai, Yuein I-324
Baran, Jonathan II-79
Bassiri, Maisam Mansub II-422
Behki, Akshay Kumar I-476
Belabbas, Yagoubi II-112
Bellatreche, Ladjel I-124
Benkrid, Soumia I-124
Bhat, Srivatsa S. I-476
Bossard, Antoine I-511
Brock, Michael II-254
Butelle, Franck I-287
Buyya, Rajkumar I-13, I-266, I-351

Calheiros, Rodrigo N. I-13
Cao, Qi II-235
Cao, Qian II-308
Cérin, Christophe I-287
Chai, Ian I-266
Château, Frédéric II-281
Cha, Youngjoon II-136
Chedid, Fouad B. I-44
Chen, Bo I-79
Chen, Bo-Han II-90
Chen, Ching-Wei II-22
Chen, Gang I-193
Cheng, Tangpei II-413
Chen, Pan I-149
Chen, Quan-Jie II-338
Chen, Shih-Chang I-278
Chen, Tai-Lung I-278
Chen, Tianzhou I-136, II-11
Chen, Tsung-Yun I-205
Chen, Yan II-318
Chiang, Tzu-Chiang I-538
Chikkannan, Eswaran I-266

Cho, Hyeonjoong II-42
Cho, Hyuk I-32
Chung, Hua-Yuan I-205
Chung, Yongwha II-42
Church, Philip II-188
Chu, Tianshu I-174, I-404, I-441
Chu, Wanming I-54
Cong, Ming II-32
Crespo, Alfons II-454
Cuzzocrea, Alfredo I-124

Dai, Jian I-441
Dai, Kui I-149
Domínguez, Carlos II-454
Duato, José II-444
Duncan, Ralph II-52

Emeliyanenko, Pavel I-427
Ercan, Tuncay II-198

Fang, Weiwei I-441
Fang, Yi-Chiun II-166
Fujita, Satoshi II-235

Gong, Chunye I-416
Gong, Zhenghu I-416
Goscinski, Andrzej II-188, II-225, II-254
Guo, Yupeng II-289

Haddad, Bassam II-1
Hai, Ying-Chi II-68
Han, Xiaoming I-113
Hassan, Houcine II-444, II-454
Hayes, Donald II-79
He, Haohu II-308
Hobbs, Michael II-225
Hsieh, Chih-Wei II-297
Hsu, Chih-Hsuan I-299
Hsu, Ching-Hsien I-278
Hsu, Yarsun II-166
Huang, Libo I-226
Huang, Po-Jung II-338
Huang, Tian-Liang II-338
Huang, Xiang II-308
Hu, Changjun II-308

Hung, Chia-Lung II-381
Hu, Wen-Jen II-90, II-121
Hwang, Guan-Jie I-205
Hwang, Wen-Jyi II-381

Inoue, Hirotaka II-146
Iwasaki, Tatsuya II-264
Iwasawa, Nagateru II-264

Jain, Praveen I-476
Jemni, Mohamed II-328
Jiang, Guanjun I-136, II-11
Jiang, Haitao I-79
Jianzhong, Zhang II-244
Jingdong, Xu II-244
Ji-shun, Kuang II-434
Ji, Xiaohui II-413
Jou, Yue-Dar II-275
Ju, Lihan II-11
Jungck, Peder II-52

Kai, Pan II-244
Kaneko, Keiichi I-511, II-264
Kang, Mikyung I-528, I-549
Kang, Min-Jae I-549
Kayed, Ahmad II-1
Kestener, Pierre II-281
Hariprasad, K. I-476
Kim, Dongwan II-348
Kim, Hansoo I-520
Kim, HyungSeok I-520
Kim, Jae-Jin I-520
Kim, Jong-Myon I-487
Kim, Mihye II-348
Kim, Seong-Baeg I-549
Kim, Seong Baeg I-528
Kim, Seongjai II-136
Kojima, Kazumine II-100
Kouki, Samia II-328
Kuo, Chin-Fu II-68
Kuo, Sheng-Hsiu II-297
Kwon, Bongjune I-32

Ladhari, Talel II-328
Lai, Kuan-Chou II-338
Lee, Cheng-Ta II-131
Lee, Eunji II-42
Lee, Hyeongok II-348
Lee, Jonghyun I-520
Lee, Junghoon I-528, I-549
Lee, Liang-Teh II-22
Lee, Shin-Tsung II-22
Lee, Sungju II-42
Lee, You-Jen I-205
Lee, Young Choon I-381
Lefèvre, Christophe II-188
Li, Bo I-404
Li, Dandan II-413
Li, Hui-Ya II-381
Li, Keqiu I-559
Li, Kuan-Ching II-338
Lim, Sang Boem I-520
Lin, Chih-Hao II-121
Lin, Cho-Chin I-299
Lin, Chun Yuan II-178
Lin, Fu-Jiun I-205
Ling, Zhang II-434
Lin, Hua-Yi I-538
Lin, Kai I-559
Lin, Reui-Kuo II-297
Lin, Te-Wei I-91
Lin, Xiaola I-163
Lin, Yeong-Sung II-131
Li, Shigang II-308
Liu, Bing I-215
Liu, Dong II-318
Liu, Feng I-193
Liu, Jie I-416
Liu, Jingwei I-136
Liu, Xiaoguang I-102, I-215, I-236, II-289
Liu, Yi I-441
Li, Wang II-218
Li, Wing-Ning II-79
Li, Yamin I-54
Luan, Zhongzhi I-174, I-404
Luo, Cheng I-324, I-363

Ma, Jianliang I-136, II-11
March, José Luis II-444
Matsumae, Susumu I-186
Mehnert-Spahn, John I-254
Meng, Xiangshan I-559
Meriem, Meddeber II-112
Mihailescu, Marian I-337
Misbahuddin, Syed I-313
Miyata, Takafumi II-401
Mohan, Madhav I-476
Mourelle, Luiza de Macedo II-156
Muthuvelu, Nithiapidary I-266

Nedjah, Nadia II-156
Nguyen, Man I-463
Nicácio, Daniel I-499
Noor, Fazal I-313

Okamoto, Ken II-358
Ok, MinHwan I-246

Pan, Sung Bum II-42
Park, Gyung-Leen I-549
Pathan, Al-Sakib Khan II-208
Peng, Shietung I-54, I-511
Petit, Salvador II-444
Phan, Hien I-463
Porter, Cameron II-79
Prabhu, Vishnumurthy I-476

Qian, Depei I-174, I-404, I-441
Qin, Jin I-416
Qin, Tingting I-215, II-235
Qu, Haiping I-113

Raghavendra, Prakash I-476
Ranjan, Rajiv I-13
Rao, Jinli I-149
Raposo, Sérgio de Souza II-156
Ren, Yongqing II-32
Roca-Piera, Javier I-454
Ross, Kenneth II-52
Ryoo, Rina I-520

Sahuquillo, Julio II-444
Saika, Yohei II-358
Sakib, Md. Sabbir Rahman II-208
Salehi, Mohsen Amini I-351
Santana Farias, Marcos II-156
Saquib, Nazmus II-208
Schoettner, Michael I-254
Schweiger, Tom II-79
Shahhoseini, Hadi Shahriar II-422
Sheng, Yonghong I-65
Sheu, Wen-Hann II-297
Shieh, Jong-Jiann II-368
Shih, Kuei-Chung II-178
Shi, Qingsong II-11
Soh, Ben I-463
Song, Xiaoyu I-193
Song, Yongzhi I-215
Sugimoto, Kouki II-358
Sui, Julei I-236
Sun, Chao-Ming II-275
Sun, Tao II-32

Taheri, Javid I-381
Tang, Chuan Yi II-178
Tang, Minghua I-163
Tang, Xingsheng II-11
Tan, Qingping I-193
Teo, Yong Meng I-337
Teyssier, Romain II-281
Thapngam, Theerasak I-1
Thejus, V.M. I-476
Tong, Jiancong I-236
Tseng, Chien-Kai II-166

Wang, Chen I-381
Wang, Chung-Ho I-91
Wang, Dongsheng I-65
Wang, Gang I-102, I-215, I-236, II-289
Wang, Hui I-174
Wang, Qun II-413
Wang, Rui I-174
Wang, Xiuwen I-113
Wang, Yaobin II-32
Wang, Yongjian I-404
Wang, Zhiying I-226
Watanabe, Tatsuro II-264
Wei, Qiying II-235
Wei, Su I-1
Wei, Xin I-391
Wong, Adam II-188
Woo, Jung-Hun I-520
Wu, Dan I-149

Xie, Jing I-416
Xiong, Naixue II-318
Xu, Cong I-363
Xu, Dan I-65
Xu, Lu I-113
Xu, Yun I-79

Yamamoto, Yusaku II-401
Yang, Chao-Tung II-90, II-121
Yang, Hailong I-404
Yang, Jiaoyun I-79
Yang, Laurence T. II-318
Yan-xiang, He II-218
Yıldız, Mehmet II-198
Yoshida, Makoto II-100
Yuan, Hui I-136
Yu, Kun-Ming II-178

Yuntao, Yu II-244
Yu, Shui I-1
Yu, You-Fu II-338

Zhang, Fan II-289
Zhang, Huiyong I-363
Zhang, Jiangang I-113
Zhang, Liang I-324, I-363, I-391
Zhang, Ning II-318
Zhang, Shao-Liang II-401
Zhang, Yuyuan II-318
Zhao, Xin II-289
Zhao, Zhenhai I-215
zhi-qiang, You II-434
Zhong, Ming-Yuan II-368
Zhou, Bing Bing I-381
Zhou, Jianfu I-102
Zhou, Jiayi II-178
Zhou, Wanlei I-1
Zhu, Danfeng I-174
Zomaya, Albert Y. I-381
Zou, Xuecheng I-149