Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

A VERSATILE PROGRAMMING MODEL FOR DYNAMIC TASK SCHEDULING ON CLUSTER COMPUTERS

by
Dejiang Jin

This dissertation studies the development of application programs for parallel and distributed computer systems, especially PC clusters. A methodology is proposed to increase the efficiency of code development, the productivity of programmers and enhance performance of executing the developed programs on PC clusters while facilitating improvement of scalability and code portability of these programs. A new programming model, named the Super-Programming Model (SPM), is created. Programs are developed assuming an instruction set architecture comprised of Super-Instructions (SIs). SPM models the target system as a large Virtual Machine (VM); VM contains functional units which are underlain with sub-computer systems and SIs are implemented with codes. When these functional units execute SIs, their codes will run on member computers to perform the corresponding operations.

This approach resembles the process of designing instruction sets for microprocessors but the VM employs much coarser instructions and data structures. SIs use Super-Data Blocks (SDBs) as their operands. Each SI is assigned to a single member computer and is indivisible (i.e., its implementation is not interrupted for I/O). SIs have predictable execution times because SDB sizes are limited by predefined thresholds. These qualities of SIs help dynamic load balancing. Employing software to implement instructions makes this approach more flexible. The developed programs fit to architectures of cluster systems better. SPM provides mechanisms, such as dynamic load balancing, to assure the efficient execution of programs. The vast majority of current programming models lack such mechanisms

for distributed environments that suffer from long communication latencies. Since SPM employs coarse-grain tasks, the overall management overhead is small. SDB access can often overlap the execution of other SIs; a cache system further decreases average memory latencies. Since all SDBs are virtual entities, with the runtime system support, they can be accessed in parallel and efficiently minimizes additional constraints to parallelism from underlying computer systems.

In this research, a reference implementation of VM has been developed. A performance estimation model is developed that takes these features into account. Finally, the definition of scalability for parallel/distributed processing is refined to represent a multi-dimensional entity. Sample cases are analyzed.

A VERSATILE PROGRAMMING MODEL FOR DYNAMIC TASK SCHEDULING ON CLUSTER COMPUTERS

by
Dejiang Jin

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Engineering

Department of Electrical and Computer Engineering

May 2005

Copyright © 2005 by Dejiang Jin

ALL RIGHTS RESERVED

APPROVAL PAGE

A VERSATILE PROGRAMMING MODEL FOR DYNAMIC TASK SCHEDULING ON CLUSTER COMPUTERS

Dejiang Jin

Dr. Sotirios G. Ziavras, Dissertation Advisor — Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Alexandros Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Sui-Hoi (Edwin) Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Roberto Rojas-Cessa, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Jie Hu, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Dejiang Jin

Degree: Doctor of Philosophy

Date: May 2005

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, USA 2005
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2000
- Master of Science in Materials Science,
  Wuhan University of Technology, Wuhan, P. R. China, 1991
- Bachelor of Science in Chemical Physics,
  University of Science & Technology of China, Hefei, P. R. China, 1986

Major: Computer Engineering

Presentations and Publications:

D. Jin and S.G. Ziavras, "Scalability: A Multidimensional Entity," *IEEE Transactions on Parallel and Distributed Systems*, April 2005 (Submitted).

D. Jin and S.G. Ziavras, "Modeling Distributed Data Representation and its Effect on Parallel Data Accesses," *Journal of Parallel and Distributed Computing, Special Issue on Design and Performance of Networks for Super-, Cluster-, and Grid-Computing*, 2004 (accepted).

D. Jin and S.G. Ziavras, "A Super-Programming Approach for Mining Association Rules in Parallel on PC Clusters," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, 9, p783-794, Sept. 2004.

D. Jin and S.G. Ziavras, "A Super-Programming Technique for Large Sparse Matrix Multiplication on PC Clusters," *IEICE Transactions on Information and Systems, Special Issue on Hardware/Software Support for High Performance Scientific and Engineering Computing*, Vol. E87-D, 7, p1774-1781, July 2004.

D. Jin and S.G. Ziavras, "Load Balancing on PC Clusters with the Super-Programming Model," *Workshop on Compile/Runtime Techniques for Parallel Computing (in conjunction with the International Conference on Parallel Processing-ICPP03)*, Kaohsiung, Taiwan, Oct. 6-9, 2003.

D. Jin and S.G. Ziavras, "A Super-Programming Technique for Large Sparse Matrix Multiplication on PC Clusters," *2nd Workshop on Hardware/Software Support for High performance Scientific and Engineering Computing (SHPSEC-03) [in conjunction with the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT-03)]*, New Orleans, Louisiana, Sep. 27-Oct. 1, 2003.

献给世界上最伟大的父母.
是他们一如既往的鼓励与支持使我能有勇气和力量不断
探索这无限奇妙的世界.

To the greatest parents in the world.
With their support, encourage and guide forever, I
could keep on exploring the wonderful world.

ACKNOWLEDGMENT

First of all, I would like to express my deepest appreciation to my advisor, Dr. Sotirios G. Ziavras, for his endless encouragement and support through my PhD study at NJIT. His motivation inspires me to move forward in my dissertation research. As my research supervisor, he not only provided countless resources, invaluable insight, and intuition, but also constantly gave me support, encouragement, and reassurance. Greatest thanks give to his enduring patience. I also would like to thank all my committee members, Dr. Alexandros Gerbessiotis, Dr. Sui-Hoi (Edwin) Hou, Dr. Roberto Rojas-Cessa and Dr. Jie Hu for their activel participation.

I also wish to give a special thank to Dr. David Perel for his technical guidance and assistance over the years. I wish to thank Mr. Dan Henderson and the Hashimoto family and fellowship program. Because of their generosity, I was granted a Hashimoto Fellowship for 2004/2005 academic year. Many of my fellow graduate students in the Computer Architecture Research Laboratory are deserving of recognition for their support.

Especially, I wish to thank my father, Limin Jin, and mother Wanzhu Sheng. It is impossible to successfully complete my research without their support in spirit and help in family matters.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 Research Motivations and Objectives 1

1.1.1 Problem Statement 1

1.1.2 Motivations 2

1.1.3 Objectives 3

1.2 Background in Programming Models and Computer Systems 5

1.2.1 Programming Models 5

1.2.2 Requirements of Programming Models 7

1.2.3 Evolution of Systems and Programming Models 8

1.3 Clustered Computer Systems 9

1.3.1 Cluster Technology and PC Clusters 9

1.3.2 More Characteristics of PC Cluster Systems 10

1.4 Background in Programming PC Cluster Systems 10

1.4.1 Programming Model Requirements for PC Cluster Systems . . 10

1.4.2 Existing Programming Models 12

1.4.3 Problems with Current Programming Models for PC Clusters . 17

1.5 Research Methodology 20

1.6 Outline of This Dissertation 21

2 SUPER-PROGRAMMING MODEL (SPM) 23

2.1 Overview of the Super-Programming Model 23

2.2 Super-Instruction Set Architecture 25

TABLE OF CONTENTS
(Continued)

Chapter	Page
2.2.1 Super-Instructions and Their Features	25
2.2.2 Functionality of Super-Instruction Sets	26
2.3 Programming with SPM	27
2.3.1 Super-Functions	27
2.3.2 Functionality of Super-Functions	27
2.4 Execution of Super-Programs	28
3 THE RUNTIME SUPPORT SYSTEM AND A REFERENCE IMPLEMENTATION	31
3.1 Structure of the Runtime Support System	31
3.1.1 Logical Structure of the Runtime Support System	32
3.1.2 Physical Structure of the Runtime Support System	33
3.2 Task Distribution System	35
3.2.1 Task Center	36
3.2.2 Task Pool	37
3.2.3 Task Agent	38
3.2.4 Reference Implementation	38
3.3 Task Execution System	41
3.4 Data Distribution System	43
3.4.1 Data Center	43
3.4.2 Data Agent	44
3.4.3 Reference Implementation	45
3.5 Runtime Management System	47

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4 EXAMPLES OF PROGRAM DEVELOPMENT UNDER SPM 49

4.1 Example 1: Mining Association Rules 49

4.1.1 Basic Concepts in Mining Association Rules 49

4.1.2 The Super-Data Blocks for Mining Association Rules 50

4.1.3 The Super-Instruction Set for Mining Association Rules 50

4.1.4 The Super-Program for Mining Association Rules 54

4.1.5 The Super-Functions for Mining Association Rules 54

4.2 Example 2: Matrix Multiplication (MM) 57

4.2.1 The Super-Instruction Set and Super-Data Blocks for MM . . . 57

4.2.2 The Super-Function for Matrix Multiplication 57

4.2.3 Scheduling Policies and Parallelization of the SF 58

5 PARALLEL EXECUTION OF SUPER-PROGRAMS AND RELEVANT OVERHEADS 61

5.1 Overview of Overheads 61

5.1.1 Overheads in Program Execution 61

5.1.2 Quantitative Definition of Overheads in Parallel Executing Programs 62

5.2 Overheads of the Runtime System of SPM's VM 64

5.3 Management Overhead 65

5.3.1 Source of Management Overhead in SPM Runtime Environment 65

5.3.2 Experiment Measure of Management Overhead 66

6 LOAD BALANCING 69

TABLE OF CONTENTS
(Continued)

Chapter **Page**

6.1 Overview of Load Balancing 69

6.1.1 Requirement for Load Balancing 69

6.1.2 The Types of Strategies for Load Balancing 71

6.2 Load Balancing for SPM 73

6.2.1 Mechanism of Automatically Load Balancing in SPM 73

6.2.2 A Model of Estimating the Imbalance Overhead for SPM . . . 75

6.3 Evaluation of the Load Balance Mechanism 78

6.3.1 Simulation of Automatically Load Balancing 78

6.3.2 Experiment of Mining Association Rule 81

6.3.3 Experiment of Sparse Matrix Multiplication 86

7 COMMUNICATION OVERHEAD 89

7.1 Communication Overheads for the SPM Runtime System 89

7.2 Technologies to Reduce Communication Overhead 90

7.2.1 Techniques to Reduce Type I Communication Overhead 90

7.2.2 Techniques to Reduce Type II Communication Overhead . . . 91

7.3 Model to Estimate the Communication Overhead of the SPM Runtime System 91

7.3.1 System Condition and a Statistical Estimation of the Communication Overhead 92

7.3.2 The Effect of Loading Remote Data in the Burst Access Mode 95

7.3.3 The Effect of the Local Cache 97

7.3.4 The Effect of Multicasting on the Communication Overhead . . 99

TABLE OF CONTENTS
(Continued)

Chapter **Page**

7.4 Evaluation of Communication Overheads in SPM 103

8 PARALLEL DATA VIRTUALIZATION AND DATA ACCESSES 111

8.1 Parallel Data Accesses for the Parallelization of Programs 111

8.2 Data Virtualization and Parallel Data Accesses 112

8.3 Evaluation of SPM in Accessing Parallel Data 112

8.4 Performance Analysis 121

9 SCALABILITY 125

9.1 Scalability and its Analysis 125

9.1.1 Demand on Scalability 125

9.1.2 Scalability as a Comprehensive Entity 127

9.1.3 Existing Scalability Definitions 128

9.1.4 More on Limitations of Current Approaches 130

9.2 New Quantitative Analysis of Scalability 132

9.2.1 Mathematical Definition of Scalability 132

9.2.2 Directionality of Scalability 134

9.2.3 Studying the Cost for Optimal Scale Up 135

9.3 Scaling up PC Clusters and Scalability Study under SPM 137

9.3.1 Techniques for Scaling up PC Clusters 137

9.3.2 Scaling Up Programs Developed under SPM 139

9.3.3 Scalability of SPs for Hierarchical PC Clusters 142

9.3.4 Scalability of an Optimally Configured Solution for a PC Cluster 144

TABLE OF CONTENTS
(Continued)

Chapter **Page**

9.4 Case Studies 146

9.4.1 Case 1 147

9.4.2 Case 2 149

9.4.3 Cases 3 and 4 151

9.4.4 Case 5 154

10 CONCLUSIONS 157

APPENDIX A PROOF OF THE OPTIMAL DIMENSION 161

BIBLIOGRAPHY 163

LIST OF TABLES

Table | Page

4.1 The SDBs for mining association rules 51

6.1 Name and properties of synthetic databases used. 81

6.2 Design parameters of SDBs and VM for mining association rules . . . 82

6.3 Summary of execution times and idle times of VM for executing the SP mining association rule. 83

7.1 A static task schedule for a heterogeneous PC cluster with 64 nodes . 104

8.1 Properties of the data matrices 114

8.2 Experimental configurations and data 114

8.3 Intrinsic degree of parallelism in the experimental SP 122

LIST OF FIGURES

Figure | Page

3.1 Logical structure of the virtual machine 32

3.2 Physical structure of the reference runtime support system 33

3.3 The components of a virtual node 34

3.4 The distributed structure of the IDU 35

3.5 The distributed structure of the data cache subsystem 35

3.6 The interface of the task agent 39

3.7 The interface of the task center 39

3.8 The sequence diagram for committing an SI 40

3.9 The sequence diagram for issuing an SI 40

3.10 The sequence diagram for executing an SI on an IEU 42

3.11 The interface of the data center 45

3.12 The interface of the data agent 46

4.1 The SP of mining association rules 54

4.2 The super-function for matrix multiplication 58

5.1 Relative management overhead in execution of the MM SP. 68

6.1 Simulation results of imbalance overhead of the MM SP 79

6.2 Percentage of average node idle time of the SP of mining association rules 85

6.3 Percentage of average node idle time of the SP of mining association rules (in the computing stage) 86

6.4 Comparison of the relative idle time for SPM and HPA. 87

6.5 Relative idle time of IEUs in execution of the MM SP (experiment) . . . 88

LIST OF FIGURES
(Continued)

Figure **Page**

7.1 Simulation results of communication overhead in a homogeneous environment . . . 107

7.2 Simulation results of communication overhead in a heterogeneous environment . . . 108

7.3 Comparison of communication overheads under different strategies. . . . 109

7.4 Communication overheads under different strategies in a homogeneous environment for different cache size. . . . 110

8.1 State diagram for logical data entities . . . 113

8.2 Relative performances of the three SPs using different data writing methods 115

8.3 The effect of the data access methods on the degree of parallelism 116

8.4 The local cache-miss ratio of data access . . . 117

8.5 The effect of multithreading on the execution time of the SPs . . . 118

8.6 The effect of the limit of threads per node on the actual degree of parallelism 119

8.7 Average time for synchronization of a sub-matrix block . . . 120

9.1 Scalability of a two-level cluster for matrix multiplication with fixed workload for the program and level-1 SIs . . . 148

9.2 Scalability of a two-level cluster for matrix multiplication with fixed overall workload but the workload of level-1 SIs increases with increases in the size of level-1 nodes . . . 150

9.3 Scalability of a two-level cluster for matrix multiplication in which the workload of level-1 SIs increases with the size of level-1 nodes and w_1 is constant . . . 152

9.4 Scalability of a two-level cluster for matrix multiplication where the w_1 increases with the number of subcluster nodes and w_2 is constant . . . 153

9.5 Scalability of a two-level cluster for matrix multiplication where w_1 and w_2 increase with the number of level-1 nodes and level-2 nodes, respectively . . . 155

CHAPTER 1

INTRODUCTION

1.1 Research Motivations and Objectives

The main objective of this dissertation is to describe an innovative programming model for developing parallel programs that can efficiently execute in parallel on distributed computer systems, mainly cluster computer systems.

1.1.1 Problem Statement

In our world people increasingly rely on computers to achieve their daily activities. Scientists often need powerful, scalable and easily accessible computer systems. Also business people often need powerful computers to as processing engines of their information systems in order to achieve management operations effectively or even use such systems to make strategic decisions. The demands for high performance and faster response drive the evolution of computer systems. Since sequential computer systems can hardly provide the required performance, users and computer system developers often focus on parallel and/or distributed computer systems [1,2]. Their focus drives the evolution of various facets of software development. One of the most important facets is programming models. Developing software comprises multiple levels [3–6]. Each level adopts a programming model to achieve its tasks. Programming models are designed based on system models described with abstract machines; there are some assumptions associated with abstract machines [7,8].

PC clusters are distributed systems used to run parallel programs. They have major advantages in potentially large raw capability, scalability and reasonable cost. However, abstract machines for existing programming models do not match well with

PC clusters. As a result performance degradation is common. PC clusters have long communication latencies than SMP servers or supercomputers [9, 10]. Therefore, the users cannot often obtain high performance by simply using existing parallel programming models. A new programming model for such systems is needed [11]. The model should hide the long communication latency of these distributed systems and should address the issue of load balancing very well for high performance. In addition, it should be easy to learn and use, and should be as much compatible in concepts as possible with current widely used programming models [12].

1.1.2 Motivations

The motivations of this research stem from the need to develop a new parallel programming model for efficient development of programs that can be executed on a wide class of parallel and distributed computer systems; this need is of paramount importance for emerging PC cluster systems. The satisfaction of the users is, of course, critical. For given hardware it can usually be measured for a piece of software with the total cost of ownership (TCO). This cost can be divided into two parts: the cost of obtaining the software, which ultimately depends on the development cost, and the cost of using the software, which ultimately depends on its performance and maintenance cost [12]. The general goal of any new parallel programming model should be to minimize the TCO of software developed under the programming model.

One of the objectives should be to obtain high performance since it is the most important factor in satisfying end users; it also affects directly users' cost in using the software. Programming models have much larger effect on the performance of programs executing in parallel compared to sequential programs. It stems from balancing the distributed workload, accessing remote data and various other overheads. Other motivations that can be addressed by an innovative programming

model are: 1) support dynamic load balancing for programs executing in parallel; 2) provide cache prefetching for large data sets so that the latency of communications can be hidden; 3) pack data objects and tasks coarsely, as needed, to reduce various overheads; and 4) support the portability of programs and the scalability of program-system pairs to satisfy end users [13]. This way, end users are free to update their cluster systems gradually on-demand without additional cost and constraints for migrating application programs. Also, the number of potential users for each program should often increase.

These research motivations are compatible with existing wide used models. From the view of software engineering, such models are easy to learn and use which are important factors in satisfying end users [14]. How easy it is to use a programming model can affect the speed and efficiency of software development. How easy it is to learn to apply the model can affect the availability of qualified developers. Both ultimately can affect the cost of software development and increase TCO. Therefore, we should be motivated to deploy a parallel programming model that is extremely similar with widely accepted sequential programming models; thus, any programmer with sequential programming experience could quickly learn to use it. As a result, a large number of programmers might shift to parallel program development for distributed systems.

1.1.3 Objectives

The main objective of this research is to develop an easy to use parallel programming model that can facilitate the development of high performance programs targeting parallel and/or distributed computer systems; the main focus is PC cluster systems. To achieve this main objective, our new programming model needs to:

- Facilitate task partitioning for parallel computation.

- Provide a runtime support system of the model to transparently distribute and schedule the execution of computing tasks for good workload balancing among the member computers in the cluster.

- Provide a standard interface to programmers to explicitly express, if desired, specific requirements for parallel scheduling and task distribution (mapping). The runtime support system should then be obliged to follow these explicit policies of task scheduling and distribution instead of the default policies.

- Facilitate the partitioning of application data

- Provide a mechanism for data provisioning so that the programmers should not consider data location. This requires that the long latency of remote data accesses should be hidden.

- Minimize all overheads corresponding to communication, data distribution and task management.

From the software engineering point of view, this model should:

- Be used to develop programs for parallel execution on both homogeneous and heterogeneous systems. Re-engineering the underlying cluster systems should not prevent from continuing using the application programs.

- Make the software development scalable. Programs should be able to run on various cluster systems of different sizes.

- Be portable. Software developed with this model could be ported to various cluster systems.

- Be very easy to learn. It should be a direct extension of widely accepted programming models.

1.2 Background in Programming Models and Computer Systems

1.2.1 Programming Models

Generally programming is the process of developing software to enforce a computer system to implement a required function; that means, it is the process of "customizing" computer systems for specific tasks. This is a complex process. To make this process of programming efficient, correct and easy, developers need appropriate system methodologies. Practice proves that layering the task into multiple layers of development is the most efficient methodology [5, 15]. In history, creating a new layer is a milestone in progress. For example, the appearance of high-level programming languages is such a great progress [16]. It makes programming more convenient and productive.

In this multilayered infrastructure, a computer system is viewed as multiple virtual/abstract computer systems or machines. Each machine is built on top of another. The abstract machine provides some abstract operations [5]. The most top layer is our application system.

Generally a programming model is a set of schemes for program organization, data and task decomposition, and execution arrangement. A programming model exists in a layered context and is associated with a set of abstract entities, namely a system model and an execution model. The underlying systems are modelled as sets of abstract entities described with a set of interfaces and programming is solely based on the behavior of the specified logical entities. Well-defined interfaces serve as a contract for the behavior, requirements and responsibilities of each part. That is where the programming model stands [17, 18]. It is an instrument with guidelines to build software in a layered context.

The system model is an abstract machine that provides an abstract description of the underlying virtual computer system. A programming model usually employs

the abstract machine to encapsulate the underlying support system and setup program semantics [19–21]. So, the architecture of the underlying system is hidden. The structure of software is simplified by expressing operations for the abstract machine. This makes program development easy and the programs are easily manageable. These abstract entities in programming models are used to describe various facets of software, the requirements and constraints of their executions, and the program's execution environment. These abstractions also specify the semantics of the languages that are used to build a higher layer program by specifying the requests for services of an underlying layer. They help developers to analyze the correctness and many other properties of programs or even analyze and estimate the performance of program execution. These abstractions may include data models and various mechanisms for data storage, information exchange and data manipulation. In some sense, any programming language corresponds to a model. To conclude, developing software with a programming model corresponds to just applying these abstractions to build software and to describe the desired logic of data manipulation.

The well-known von Neumann computer model represents computation on those sequential machines. It sets up a contract between the low-level computer hardware and the high level software. Even though technology and architectural ideas evolve rapidly, hardware designers still target efficient von Neumann machines without much concern for the programs that will be executed on them. On the other hand, software developers focus on programs that can be executed efficiently on this model, without explicit consideration of the hardware. The von Neumann model is versatile to enable diverse programs to run efficiently on sequential machines [18].

An execution model is the scheme used to execute software on the corresponding abstract machine. It specifies how operations in a program are executed. Combining an abstract machine and its execution model provides an instrument to describe

the process of state change in the corresponding computer system. Such a state machine can clearly define program semantics. This provides the basis for analyzing the correctness of programs. The execution model also can be used to estimate program performance. Programmers and/or the compiler can use it to optimize coded programs [22].

Since programming models provide an instrument to express operations of application programs in a highly abstract manner, programmers can focus on high-level application logic, but ignore the details of the computer system and its execution steps. The programming with such abstract operations can offload major work to lower layer system developers. So the employment of high-level programming models can reduce the complexity of programming [14]. This, in turn, can increase the efficiency of program development and increase productivity.

1.2.2 Requirements of Programming Models

Abstraction in programming models is essential. Programming models must be based on abstract machines, should completely encapsulate the underlying support systems and should be independent of system architectures [18,23]. Programs developed with such models will be easily ported. Developed software will be more stable and could survive the evolution of computer system hardware. To adhere to abstraction, the implementations of abstract entities in these models must be transparent to the users of the programming models.

For programming models to be usable, programmers must be able to express data structures and logical operations with the basic elements and constructors of the model. Only if the programming model provides adequate expressiveness, will it be possible to isolate high-level programming from low-level implementations of the underlying computer system. Otherwise, encapsulation in the programming model

of the underlying system is broken. It is also important for the programming model to get sufficient support from the underlying computer system. To be practically usable, the architecture should provide good support to implement the model.

1.2.3 Evolution of Systems and Programming Models

For higher performance and better usability, computer system architectures have been constantly evolving. One of the common trends in system evolution is the so called parallelization [24]. In some sense, almost all contemporary computer systems are parallel in nature. They have multiple functional units in a single processor [25, 26], multiple processors on a board, multiple computers in a box [27–29] or even multiple computer hosts in a network [9,30,31]. Another trend is distributed system. This evolution is driven by both improved design techniques and increased social needs. The appearance of the Internet, www services [32, 33] and grid computing [34, 35] are all results of this trend. During the evolution of computer systems, standardization plays a critical role. It gives developers of programs concise and clear environments. Programs become portable for wider use. PC cluster systems are a good example of using commercial the off-the-shelf (COTS) PCs as building blocks for cost-effective parallel and distributed computer systems [24].

Because of the recent evolution in computer system architectures that has broken away from those previous abstract machines for parallel computers and supercomputers , old programming models cannot encapsulate the new emerging features without loss in performance. In this case, both the system model and programming models also have to evolve. Thus, the development of a unifying model for parallel computation is required for general-purpose parallel and distributed computing [18].

1.3 Clustered Computer Systems

1.3.1 Cluster Technology and PC Clusters

Clustering of resources is a technique that groups identical or similar functional units together to form a larger system. A clustering technique can be used at various levels of a computer system for different components. The clustered components could be multiple ALUs in a processor [36], multiple processors in a computer [29,37], multiple disks in a storage system [38, 39], multiple computers in a large web server or a large computing facility [40–43].

The primary reasons of using a cluster technology are to build a large system with improved capabilities, reliability and availability using COTS components [44]. Clustering also provides a mechanism to improve the scalability of a system [2, 45] because users/administrators can expand the system easily by adding additional COTS components; for at least a certain size range, the capabilities of a clustered system can almost linearly increase by increasing the number of member components. Clustering provides a mechanism to improve system availability [46,47] and reliability [48–50]; member components can back each other up. Since member components are often similar and standardized the cost of development is low.

Computer clusters contain computers as the building blocks. The member computers, called nodes or hosts, could be SMP (Symmetric multiprocessors) servers, minicomputers, workstations or PCs (personal computers) [29, 51]. PC cluster systems (or PC clusters in short) employ common PCs normally running public domain software [31]. They are also connected via standard network devices and techniques.

The main reasons that have made PC clusters very popular are cost affordability and accessibility.

1.3.2 More Characteristics of PC Cluster Systems

A feature that differentiates PC clusters from other multi-computer systems is their network. A super-computer has a customized inter-processor communication network [15, 52, 53] and a customized data network that connects the processors to mass memory devices. When increasing the number of processors, the complexity of these networks increase super-linearly. However, a PC cluster has neither specialized inter-processor communication network nor special memory access networks [54, 55]. The communication between processors belonging to different member computers must go through network interface cards and the general-purpose network. Current practices have shown that simple and linearly scalable networks can be used to build very large computer systems, thus, improving scalability. Compared to massively parallel processor systems, PC clusters have very fat nodes. Each member node is an individual single- or multi-processor computer containing several memory modules and various storage disks. These PCs may possess locally installed programming libraries, administration software and utility programs; there may also exist many local system services on these member computers. These PCs even have auxiliary processors dedicated to network access. Also, nodes in PC clusters are fat autonomous systems where tasks are ultimately scheduled by the local operating system.

1.4 Background in Programming PC Cluster Systems

1.4.1 Programming Model Requirements for PC Cluster Systems

Generally, programming models are required to have some basic functionality. They should provide:

1) Instruments to decompose application data (to declare complex data)

2) Tools to define and delineate subtasks (statements) so that programmers can develop large application conveniently

3) Mechanisms to specify order constraints in executing the subtasks in application programs (for scheduling).

For distributed programming models, additional functionality is required corresponding to:

1) Partitioning application data into multiple groups and distributing them among the member computers

2) Providing a mechanism for data communication between distributed processes (i.e. a facility to remotely access distributed data).

3) Mapping application tasks into multiple groups of member computers so that all of these resources can be utilized.

4) Providing a mechanism to distribute tasks to member computers (either by distributing the existing tasks or launching more threads).

For high performance, the runtime support system of a parallel programming model should also provide:

1) Mechanisms to exploit the parallelism in programs while execution system can keep the aforementioned order constraints.

2) Workload balancing among multiple processors.

1.4.2 Existing Programming Models

Since the nature of PC clusters, which are both parallel and distributed, currently programmers employ various existing parallel programming models and/or distributed programming models to develop application programs. Most of these models were initially developed for some other types of systems. Distributed system programming models assume that processors and memories are spatially distributed in multiple autonomous computers [56]. Processors can only access their local memory. Programs consist of multiple pieces running on different computers. These pieces have their own instruction streams and data stored in the local memory. They co-operate with each other by communicating through a network. Most distributed system programming models stem from the client-server model [57]. In the client-server programming model, applications are split into client and server parts distributed to many computers. Clients invoke methods of server programs when they need to achieve pre-defined tasks; server programs execute predefined procedures when receiving requests and they then send the results to clients. After gotten a response, a client resumes its blocked thread. That is, the execution of threads crosses the boundary between processes and computers. RPC (Remote Procedure Call) is a typical client-server programming model. In distributed programming models, partitioning and delineating tasks is embedded in declaring and defining for remote accesses; remote procedure invocations provide mechanisms for both data and task distribution. Each call of a remote procedure just delivers a task to a server process. The passing of arguments and the return of results for remote procedures achieve the functionality of distributing data. In principle, distributed programming models emphasize the distribution of tasks. The roles of client and server can be dynamically determined within the context of programs progressing [58]. CORBA is a widely used distributed programming model [59, 60].

Logical threads in CORBA consist of executing methods on objects distributed to multiple computers. Through invoking remote methods, multiple distributed threads executing these methods are connected and they form these logical threads. DCOM is another example of a distributed programming model in the Windows world [61]. RMI is the Java version of a distributed programming model [62–64].

In a strict sense, client-server programming models are not parallel programming models, since an underlying program is blocked. Several distributed programming models do develop some basic features to support parallel execution, where clients also can invoke methods of servers asynchronously. Clients no longer need to block local execution threads. Servers can send back their responses via callback interfaces of the clients. The message-driven EJB in J2EE is such a good example that use asynchronous communication. However, these models still lack a sophisticated mechanism to schedule global tasks. For this reason, there are a number of efforts to enhance the parallel features of distributed programming models [63].

Parallel programming models have been studied extensively [65–68]. In these models application programs consist of multiple processes [15, 23, 56, 69] executing the same or different program codes simultaneously on multiple processors. Parallel programming models can roughly be classified into two types [56, 70]. The first one follows the SIMD (or SPMD) paradigm that exploits the data parallelism. The second one follows the MIMD paradigm that exploits the task parallelism. Based on the way of exchanging data between processes, parallel programming models are basically classified into two groups [71]. The first group is message-passing models where each process only has a local view of its own independent programming space [23, 72, 73]. It has no idea about the data structure used by peer processes. The other group is shared-memory models where the processes use a shared logical programming space [23, 74–76]. These programming models are independent of

the system architecture [23, 77], which can be encapsulated. The shared-memory programming models can be used for both shared-memory and message-passing architectures. Shared-memory models may need software distributed memory (SDM) systems [74,78] to encapsulate distributed architectures. On the other hand, message-passing programming models can also be used for shared-memory architectures by using the sharing memory mechanisms to emulate with message-passing channels [79]. These encapsulations usually introduce additional overheads [80].

Message-passing models are very common in parallel programming. In practice, programs are written in a sequential programming language, such as *C* or *Fortran*, and data exchanges employ calls to message-passing libraries. PVM (*Parallel Virtual Machine*) is a message-passing programming system that combines a set of computers to form a single, manageable virtual machine [81, 82]. PVM exploits both data and task parallelism. It has been ported to many different systems. MPI (*Message Passing Interface*) is another standard specification for message passing [83]. Following MPI, many libraries such as MPICH [84] and LAM [85] have been developed. The primary advantages that make MPI popular are its generality and portability. Using MPI, programmers can express any conceivable data distribution and be confident that the programs developed with MPI could run on any parallel platform.

Bulk Synchronous Parallel (BSP) is a special kind of a message-passing programming model [86]. In standard message-passing models, messages can be sent at any time and may be received out of order. Messages coming from different senders may interleave each other. Such a scheme makes programs prone to deadlocks and analyzing and predicting the performance of programs may be impossible. Instead of allowing this chaotic approach, BSP organizes the execution of application programs in a sequence of supersteps [18, 87]. Each superstep consists of three segments:

local computation, global communication and barrier synchronization. Only in the global communication stage, the distributed processes are allowed to exchange results generated in the previous computation and prepare the operands for the next computation stage. Thus, no process is stalled to wait for data during execution [86]. In the communication stage, many messages may be merged into large packages that make communications more efficient.

Shared-memory models also are widely used for their simplicity and easiness of programming. Shared-memory models can adopt data or task partitioning to allocate the workload. When applied to distributed systems, they usually employ an abstract data layer to encapsulate the distributed memory in the underlying machines [88]. Global array languages use shared-memory models with the SIMD (SPMD) paradigm to exploit data parallelism. ZPL [89], Co-Array Fortran [90], Unified Parallel C [91,92] and Titanium [93] are all in this style. OpenMP is another shared-memory parallel programming model [94]. It defines a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. Programmers use these directives to declare parallel constructs in sequential-like programs. OpenMP uses the fork-join parallel execution model. When a parallel construct is encountered, the executing thread creates a group of working threads and the creator becomes the master of the group. Each thread in the group executes in parallel the same code specified by the construct. At the end of the construct, all threads in the group execute a barrier. Although this fork-join model can be useful in solving a variety of problems, it is somewhat tailored for large array-based applications.

There are also many shared-memory parallel-programming languages that adopt the MIMD style to exploit both data parallelism and task parallelism. *Cilk* is a typical example in this category [95,96]. Programs consist of multiple computation

tasks executed in individual threads in a shared common context. The programmer concentrates on structuring the program and leaves the responsibility of scheduling the computation to the runtime system [95, 97]. Cilk also uses the fork-join model [98, 99]. Programmers can split a part of the computation into an independent task that will execute in a separate thread. This achieves the functionality of task mapping.

Object-oriented programming methodologies extend the concept of shared-memory models to shared-object models [100,101] and combine parallel-programming with distributed- programming models. Under this kind of models, objects are programming entities. Programmers implement their programs in a global object space instead of global address space. All application data are encapsulated in objects and all computations are encapsulated in methods that can be applied to the objects; methods are called to manipulate data stored in objects. The primary responsibility of the programmers is to implement these methods and to specify when and which objects are called. During execution, these objects are distributed among multiple processes. They know each other by their name or ID rather than their address. The execution of programs may produce calls that may cross the boundary of processes by sending messages. The execution of programs is driven by events that trigger calls to methods.

Charm++ is a good example of an object-oriented parallel-programming model [102]. Charm++ uses a runtime library to let C++ objects communicate with each other efficiently. A program consists of a number of charm++ objects named chares. Chares send messages to each other by invoking these entry methods asynchronously. One of the important differences of charm++ with other distributed programming models is that objects float between processors; that means objects may be migrated when the runtime system wants to balance the workload among

the processors. Programmers are responsible in decomposing the computation into entry methods of chares and specifying when to create new chares or send messages to existing chares. The intelligent runtime system maps these chares to available processors and schedules the execution of methods to respond to messages [103]. Conceptually, the system maintains a "work-pool" consisting of seeds for new chares and messages for existing chares. The runtime system may pick up multiple items non-deterministically from this pool and execute them. Charm++ uses one-side asynchronous remote method invocation. Charm++ supports efficient dynamical load balancing by using object migration for irregular and dynamic applications; it can also deal with external factors that cause load imbalance.

1.4.3 Problems with Current Programming Models for PC Clusters

Most of current parallel programming models were initially developed for some special computer systems and then ported to other types of parallel computing platforms. So far no "native" programming model has been developed for PC clusters. Even though all parallel programming models can be ported to PC clusters, the cost of porting these models is usually a major sacrifice in overall performance because the architecture of PC clusters does not match well initial prototypes for these models. There are two fundamental features that prevent developers from applying very well existing programming models to PC clusters. The first one is the atomicity of nodes. Since local operating systems control the computing resources, estimating the execution time of partitioned tasks becomes more difficult and workload balancing becomes a bigger issue that impacts system performance. Another feature is the long communication latency associated with interconnection networks for PC clusters. Due to the lack of dedicated data networks, the latencies of data communications may be more than 1000 CPU cycles. This is much longer than

the latencies in systems with dedicated data exchange networks. With the improved frequencies of CPUs for member computers, the effect of these latencies will become even more preeminent.

The existing shared-memory programming models are too sensitive to data access time. All of these models depend on sequential programming languages that support ad hoc accesses of fine grain data. Instructions on individual processors are in a sequential stream, so if accessing the memory takes too long time the instruction using the data and all following instructions have to be stalled. Mature sequential programming models address this issue by attempting to hide the latency of accessing memory. But in PC clusters, the latencies are prohibitively high. Thus, it is difficult to hide latencies. Another problem of applying existing shared-memory programming models to PC clusters is related to the appropriate decomposition of tasks for load balancing.

In global array languages, the work of mapping tasks is statically done through code directives that partition the global arrays and map them to processors. The effectiveness of this approach depends on predicting accurately the workloads of processors. For PC clusters, the CPU time for individual tasks varies; many external factors can affect the execution time of a task. Thus, the approach of statically balancing the workload may not work very well. Other shared-memory programming models that exploit task parallelism, such as Cilk, dynamically map tasks to the processors. The runtime system can balance the workload dynamically by launching threads for new tasks on light processors or migrating active ones. This is a very attractive approach that is much better than static approaches. The problem of applying this approach to PC clusters is the cost of migrating active threads.

Message-passing models for PC clusters can partially address the issue of long latencies through multithreading and asynchronous communication techniques for

latency hiding. However the primary disadvantage of these models is that they require the programmers to manage all of the details for data distribution, parallel computation and inter-processor communication for each process. This is tedious and error-prone, and it often obscures logical computations. The tasks are statically associated with the program(s) at the time of coding and are mapped to processors at launching time. If the programmers do not include some adaptive logic in their code, the distribution of the workloads can only be statically balanced; as mentioned above, the statically approaches do not work very well for PC clusters. Adding some adaptive logic in the programs may do much better [104, 105]. The drawback of dynamically balancing the workload is that programmers must code the adaptive logic and embed it into the application logic. The message-passing model and corresponding runtime systems do not provide automatic mechanisms to do so. Incorporating dynamically load balancing techniques at the application level involves significant changes to the design and structure of applications. Actually, the logic of load balancing is incorporated in the data distribution logic. This approach makes communications even more complex and the process of developing large complex software becomes even more tedious, more error-prone and less productive. In the view of software engineering, mixing application logic with logic for load balancing makes it more difficult to reuse the logic for load balancing. Thus, programmers have to recodes the logic for load balancing for each application program.

The problems of applying object-oriented parallel programming models to PC clusters are still load balancing and long latencies for communications. Load balancing is bigger issue. It is a problem related to object distribution since computation code is associated with specific objects; these objects may be pinned to processors, may be duplicated among processors or may float among processors. For many distributed programming models, such as CORBA and J2EE, administrators

deploy these objects into some containers that reside on specific computers. These approaches are equivalent to static load balancing. In some situations, some objects may become hotspots. In the purpose of balancing the workload, floating these objects is more attractive. This necessitates migrating objects from one process context to another one when load is not balanced. But similarly to migrating active threads, the cost of migrating objects will rise. In complex heterogeneous environments, this cost may be prohibitively high. This may be the reason that most distributed programming models currently choose a static model of deployment.

1.5 Research Methodology

In this research, the author tries to port the von-Neumann computer model to the context of PC clusters. The proposed approach inherits all the advantages of existing programming models and the most widely accepted concepts. The new model is easy to comprehend and grasp and optimizes the spatial and temporal localities in program execution for high performance. There are two characteristics that support these changes. First, all program components are virtualized through appropriate software implementation. Second, coarse-grain data and complex domain dependent operations are employed. The former virtualization eliminates the limitations imposed by hardware implementations and makes it possible to change the size of these components in a very large scale. One of the most obvious example is instructions. Traditionally, a processor instruction adheres to its hardware implementation and can complete a simple logical or arithmetic operation. Also its operands are limited in size (such as 32bits or 64 bits). In contract, our approach for program development employs instruction that can deal with operands of various sizes.

This modular approach inherits huge volume of valuable assets by exploiting existing achievements. There are a lot of libraries to support various functions

and a lot of tools have been developed. All of them can be used directly for programming PC clusters without much effort to port them. Programming models and methodologies for program development for PCs are very mature. They can be employed directly for programming PC cluster. Also, the major of program developers are familiar with these models and methodologies. Thus, it will be a big advantage to exploit these successful achievements when developing code for PC clusters. To conclude, our proposed program design methodology will be based on sequential programming concepts through the development of appropriate instruction sets for applications. In addition, a runtime support environment will be developed for code porting, data sharing and exchanges and latency hiding

1.6 Outline of This Dissertation

Outline of Dissertation Following this introduction, Chapter 2 will give complete description of the proposed programming model - *Super-Programming Model.* In Chapter 3, a complete description of the runtime support system for our programming model is given, and a reference implementation and a set of relevant interfaces are presented. In Chapter 4, two examples from two different application domains, namely the business intelligence and scientific/engineering computation domains, are presented to illustrate how to develop programs targeting PC clusters for distributed processing. In Chapter 5, based on the execution model of SPs developed under SPM, the cost of execution is analyzed; various overheads are quantitatively defined and the management overhead is discussed as well. In Chapter 6, a mechanism for load balancing under SPM is discussed. Also, a model to estimate the imbalance overhead is formalized. The real effect of the imbalance overhead is evaluated with experiments. Our results are also compared with another approach for programming PC clusters. In Chapter 7, the communication overheads of SP execution are analyzed; the techniques that can be used to minimize factors

that can reduce the execution performance are discussed. In Chapter 8, a distributed data virtualization scheme that can benefit parallel program execution is presented. In Chapter 9, a new definition of scalability is presented and the scalability properties of various setups are analyzed. The chapter 10, summarize this research.

CHAPTER 2

SUPER-PROGRAMMING MODEL (SPM)

In sequential computation, the von Neumann unifying model sets up a standard for hardware and high-level software designs. Software developers target programs that can be executed efficiently under this model, without explicit consideration of the underlying hardware. Parallel programming also needs such standard bridge models [18, 19]. In this chapter, a parallel programming model is proposed for distributed computer systems.

2.1 Overview of the Super-Programming Model

In this dissertation the *super-programming model (SPM)* is proposed for developing parallel programs for distributed systems, such as PC clusters. The proposed model expends existing mature programming models for von-Neumann machines. The system model is basically modeled as a von-Neumann machine with multiple ALUs that can perform complex tasks. It replaces simple processor operations with much coarser tasks.

Developing software under the SPM consists of four layers. The lowest layer is development of the virtual machine (VM). It includes designing and implementing various components of the VM. The second layer is development of a super-instruction (SI) set architecture to customize the virtual machine. The third layer is development of reusable parallel program units called super-functions (SFs) that are implemented with super-instructions. The highest layer is development of application programs with super-functions and super-instructions

In SPM, targeted systems such as PC clusters are modeled as VMs with high-level architecture similar to a PC [106]. The VM contains a super-processor consisting of an instruction fetch/dispatch unit (IDU) and multiple instruction execution units (IEUs) that form the ALUs of the VM. The important difference between the VM and a PC is that the functional units (FUs) in the VM are much coarser and each FU contains a complete "microcomputer system." These FUs are implemented as a set of processes running on member nodes. Communication channels between these processes carry out the functionality of "buses" in the VM. The main memory of nodes can work as data and instruction caches. All these implementation processes of the FUs make up the runtime support system.

The processors in the VM can execute some predefined abstract operations to manipulate a set of "built-in" data types that may be specific to the application. These data types are called as *super-data blocks* (SDBs) and are much coarser than an integer or floating point number. These abstract operations are called *super-instructions* (SIs). Programming in SPM ultimately is just coding application programs using SDBs and SIs.

SDBs work as abstract operands for SIs. They input data and receive result data. An SDB may be expressed in different local formats on different nodes. At runtime these SDBs are incarnated and mapped onto data structures stored in underlying nodes. Implementers are free to incarnate SDBs with any data structure. As long as the SDB formats have been set up along with the SDB exchange protocols, nodes with different architectures can freely exchange SDBs. This feature makes it very easy to work with heterogeneous clusters.

2.2 Super-Instruction Set Architecture

The instruction set architecture (ISA) is most important user interface of a computer. It bridges hardware implementations with software systems. It is the basis of programming. SPM also uses the concept of ISA. The super-instruction set (SIS) architecture is the core of SPM that becomes the interface to high-level programming.

2.2.1 Super-Instructions and Their Features

SIs are abstract "built-in" operations of the VM. Since the FUs of the VM are implemented with software programs, these SIs are mapped at runtime to ordinary procedures and are executed in the context of IEUs. Since IEUs are supported by complete PC nodes, they may be very powerful. But, SIs have the following features similar to ISAs for ordinary processors:

1) SIs support atomic operations. Each SI can only be assigned to and be executed on a single IEU. Also there is no communication logic embedded in the body of an SI.

2) SIs are for abstract operations. Programs using SIs only care about the result of the executed SIs. Thus, system developers are free to implement SIs in any way and the runtime support system also is free to choose the most appropriate procedure in executing them.

3) The workload of an SI is predictable. Each SI has a set of known operands with a pre-defined size limitation. Thus, the workload of an SI has a quite accurate upper bound.

4) The data dependences are handled only at the beginning and end of an SI's execution. Once all operands are locally available, an SI can be executed without any interruption.

2.2.2 Functionality of Super-Instruction Sets

The chosen ISA in SPM is an instrument in customizing the clustered computer system. For each application domain, domain experts can develop an effective SI set architecture. Also, the SI set is expandable for each domain so that it matches the application's requirements. The SI set should provide all required basic operations. This is called completeness of the SI set. This makes it possible to express all applications in the domain with these SIs. The SI set encapsulates the underlying support system. This enhances portability and reusability of software components. Application programs are modelled as *super-programs* (SPs) that can be ultimately coded with SIs of the particular domain. Application programs just use SIs without a need to take care about the implementation of SIs.

SIs also provide a good basis for program porting. Since SIs are implemented with procedures, portability is enforced through standardizing SIs. As long as an efficient implementation exists for each SI on the nodes, SP code portability is guaranteed. The SIs also normalize operation workload. The workloads of SIs can be designed to have a similar upper bound. Thus, SIs can have expectable workloads. This provides a good basis for dynamically balancing the workload by appropriately scheduling these SIs. The multiple IEUs in the VM can be easily balanced through dynamically scheduling SIs to available processing units at run time. When the degree of parallelism in an SP is much larger than the number of nodes in the cluster, any node has little chance to be idle. Good load balancing becomes more feasible by focusing on scheduling at this coarser SI level.

2.3 Programming with SPM

Programming with SPM inherently is the process of developing programs with SIs for a specified application domain. For completeness, a high-level programming language and a corresponding compiler may be needed. However, for illustrative purposes in this dissertation, this process is simplified and embedded in the code used to implement the VM.

2.3.1 Super-Functions

To facilitate ease of developing applications, SPM adopts a structured programming style. Higher-level reusable programming units called *Super-Functions* (SFs) are developed. SFs are coded with SIs and/or other SFs. They may combine many SIs to form higher level abstract operations. SFs are "binary" executables for the VM. While executing SFs, the IDU fetches SIs in SFs and dispatches them to the IEUs to execute. More than holding a set of SIs, SFs also include order constraints in issuing SIs. By cooperating with the VM, SFs control how to issue the SIs in order and how to exploit parallelism.

2.3.2 Functionality of Super-Functions

Super-functions play a critical role in SPM. First, they provide a high-level abstract layer. It makes SPs to be more concise when expressed with SFs. These SFs can also be reused in either the same SP or different SPs as library components. Therefore, these SFs can be developed once and be used in many places. This can increase productivity and reduce the development time. Thus, ultimately they can reduce the cost of program development. Second, SFs play a critical role in the parallelization of SPs. As mentioned above, SPM targets parallel and distributed

systems and the VM provides the mechanism to execute multiple SIs simultaneously. Thus super-instruction level parallelism (sILP) is supported, while SIs are sequential procedures. This means that programmers and/or developing tools must deal with the parallelization when coding SPs. To parallelize the execution of SPs, the VM must extract parallelism among SIs when executing a SP. Since SFs include all SIs and also hold information about dependences among the SIs, these SFs, can cooperatively give the information that the VM need to schedule SIs, i.e., SFs are responsive of helping the VM to make execution of SPs parallel.

From the programming point of view, SFs help in expressing the parallelism in algorithms. SFs can co-operate with the runtime support system of the VM to dynamically check for dependencies between SIs. This makes dynamic load balancing a responsibility for the runtime system. In addition to implicitly expressing dependencies in code, developer of SFs can add some logic to specify all kinds of constraints for executions of SIs in parallel and/or explicitly express dependencies between these SIs.

2.4 Execution of Super-Programs

Execution models are used to describe the behavior of programs during execution. Here the execution model for SPs is described. While executing an SP, the IEUs of the VM are the workhorses. When the computing resources in an IEU are available, the IEU requires the IDU to allocate more SIs to it. For a new SI, the IEU selects an implementation of the SI and loads its executable into the context of the IEU. In the process of localization, the IEU also loads the operand SDBs for the SI and localizes these SDBs by mapping them to local data structures in the IEU context. After the localization process, the IEU executes the local version of the SI as an independent working thread. After it finishes the execution of an SI, the IEU notifies the IDU

to commit the SI and begins another cycle. If there is no available SI, the IDU will deny the request of the IEU. In this case, the IEU will repeat its request later.

At runtime, the VM has a priority task queue that virtually holds all tasks (SFs/SIs) that need to be executed. SFs in fact contain their own queues of SIs. SFs feed their tasks to the IDU of the VM when the latter looks for more SIs to satisfy the request of IEUs. If the fed task is an SI, the SI is assigned directly to the IEU that requests an SI. If the fed task is another SF, the SF is incarnated and activated. The activated SF is added to the task queue of the IDU to serve as a source of SIs. Generally, an entire SP may be treated as a large SF. it is added into the task queue of the VM when the system start to execute the SP.

In the execution model, the IDU controls all active SFs in its task queue; it can be a priority structure based on the properties of SFs. When an IEU asks for more SIs, the IDU turns to the active SFs in its queue to find an available SI to be issued. The order of SFs to feed the IDU is based on the priority of the SFs. A low priority SF can feed an SI to the IDU only if all higher priority SFs do not have an available SI to issue. Once the IDU gets notice from an IEU to commit an SI, the IDU sends a feedback to the corresponding SF. SFs can use this feedback information to adjust their status. When the IDU requests an SF to feed more SIs, it also gives the SF some information about the requester. SFs can use the information to optimize their behavior of feeding SIs so that they can maximize parallelism and maximize resource utilization. Activated SFs keep themselves active until all tasks they hold have been executed and committed. After that, the IDU also commits the execution of the SF similar to SIs.

Although an SF holds a set of SIs, the tasks that make up an SF are not necessary static. It is not necessary to determine the number and type of SIs to be actually executed at development time. It is even not necessary to determine when

the SF is to be activated. Like the number of instruction that a loop structure fill into instruction stream is determined at runtime. Some SIs/SFs can be created on the fly and then they are added in task queue. This feature makes SFs more flexible and powerful.

CHAPTER 3

THE RUNTIME SUPPORT SYSTEM AND A REFERENCE IMPLEMENTATION

All programs developed under the SPM are executed on a virtual machine that is built on the top of a distributed multi-computer system, such as a PC cluster system. The virtual machine is implemented with software programs to perform operations specified in application programs. The software programs include two parts. The first part is the runtime support system of the virtual machine. It is the core of the implementation of the virtual machine. The runtime support system provides general mechanisms to support the execution of programs developed in the SPM. It does not depend on any application domain. The other part is the implementations of the super-instruction set that specifies the basic operations the virtual machine can perform. It depends on application domains and then can be used to customize the virtual machine. In this chapter, through describing a reference runtime support system, the structure of runtime support system of SPM is described. Some important interfaces are defined.

3.1 Structure of the Runtime Support System

The runtime support system (RTS) for SPM is a collection of programs that make up the core part of the VM implementation. It provides a high-level environment to execute SPs. The structure of RTS can be described in a high-level logical structure and a low-level physical structure.

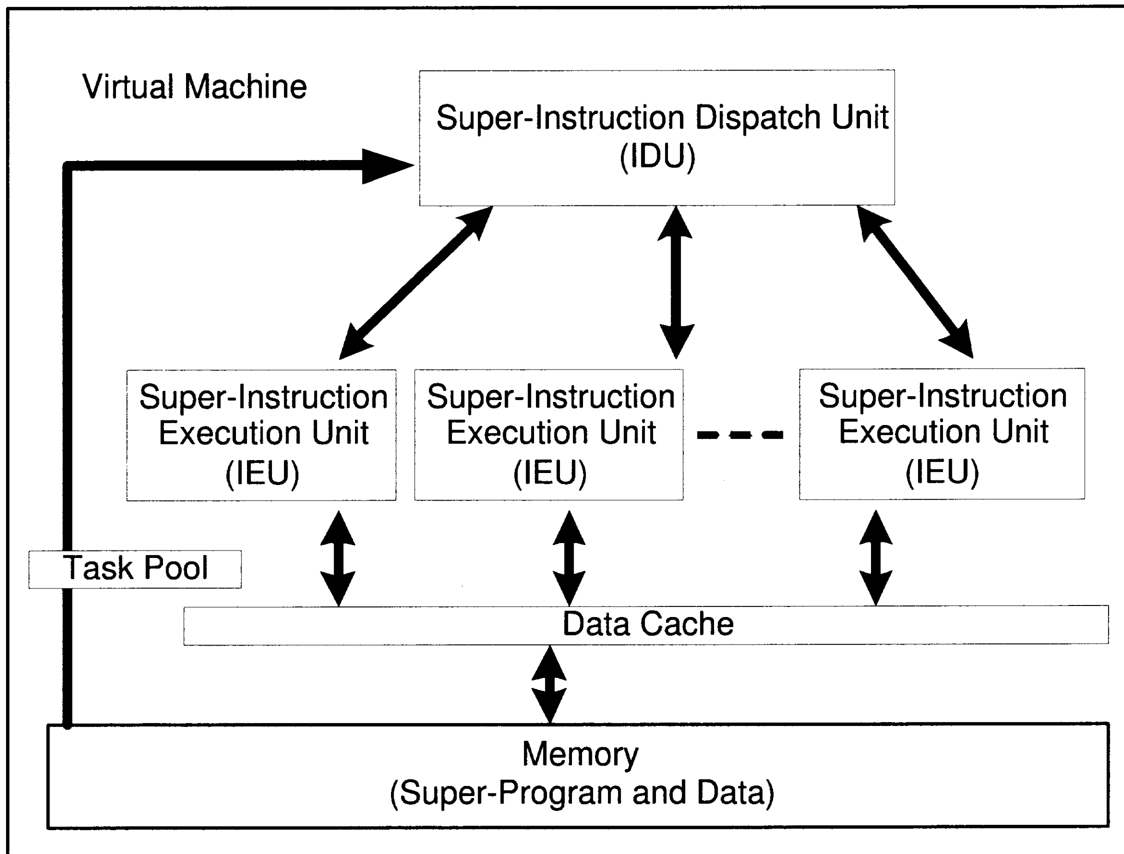


Figure 3.1 Logical structure of the virtual machine

3.1.1 Logical Structure of the Runtime Support System

Similar to the architecture of a PC, the logical structure of the VM consists of system memories, an instruction dispatch unit (IDU), instruction execution units (IEUs) and a data cache subsystem. Fig. 3.1 illustrates this logical structure. At this level, the IDU controls the execution of SPs. It fetches SIs from the task queue and distributes them to IEUs. The IEUs select the most appropriate local implementation of the SI to execute.

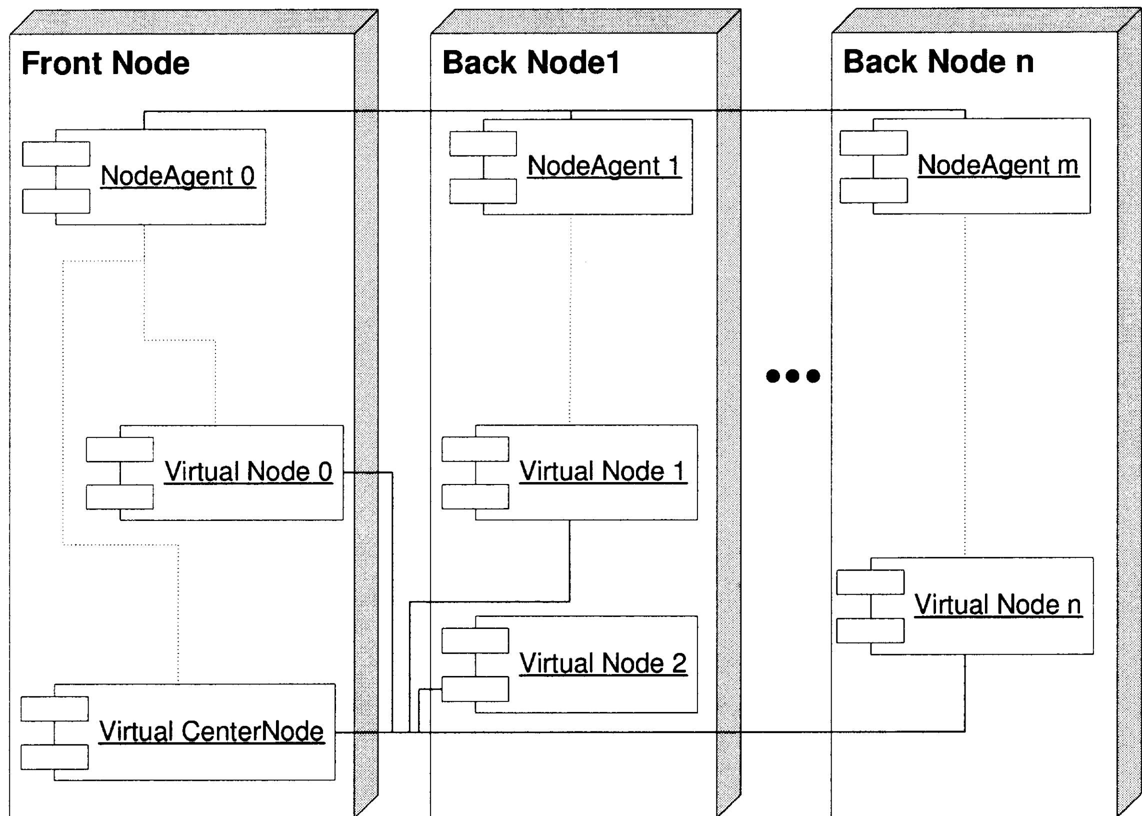


Figure 3.2 Physical structure of the reference runtime support system (deployment of component processes)

3.1.2 Physical Structure of the Runtime Support System

Physically RTS consists of multiple processes executing asynchronously. They cooperate with each other to support the execution of SPs. Each process runs on a member node of the PC cluster. Figure 3.2 illustrates the overall physical structure of a reference RTS.

These RTS processes can be classified into three types: management processes, virtual nodes (VNs) and the virtual center. Management processes are used to boot up the VM and set up communication channels. The VNs provide low-level runtime environments that host multiple threads to perform functions on the virtual units

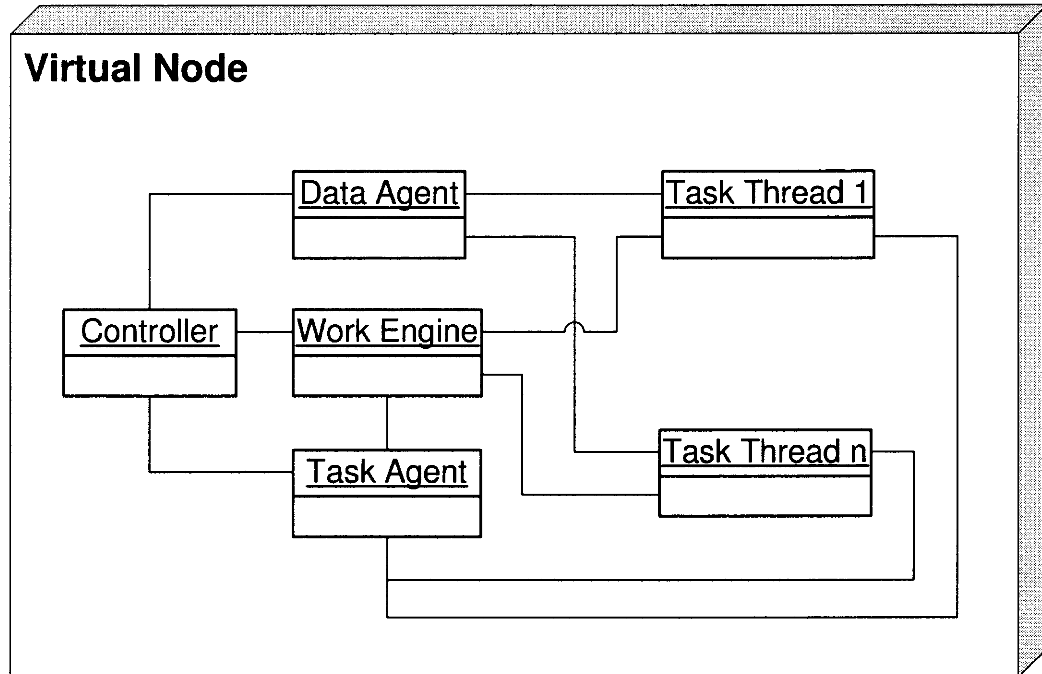


Figure 3.3 The components of a virtual node

and hold all the resources allocated for these threads. This structure of threads is shown in Figure 3.3. The virtual center is a special node process that holds a task center and a data center.

The IDU is a virtual functional unit, which is a set of threads distributed and embedded in the VNs. Its structure is shown in Fig. 3.4. The memory and data caches also are virtual components. The memory of the VM represents all the external storage. The data cache is a temporary storage of limited size, which is faster than memory. Physically it may map to the distributed memory of nodes in the VN. Part in a VN becomes the level-1 cache of an IEU and also the level-2 cache of another IEU in the VM. This distributed structure is shown in Figure 3.5.

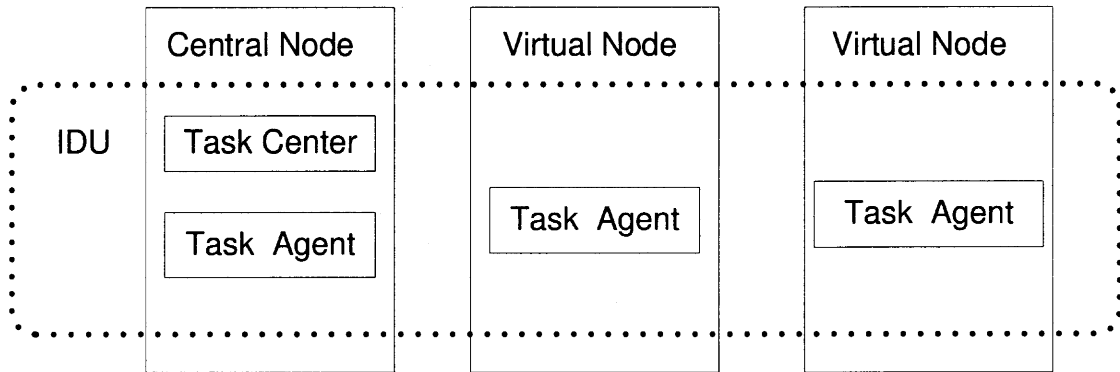


Figure 3.4 The distributed structure of the IDU

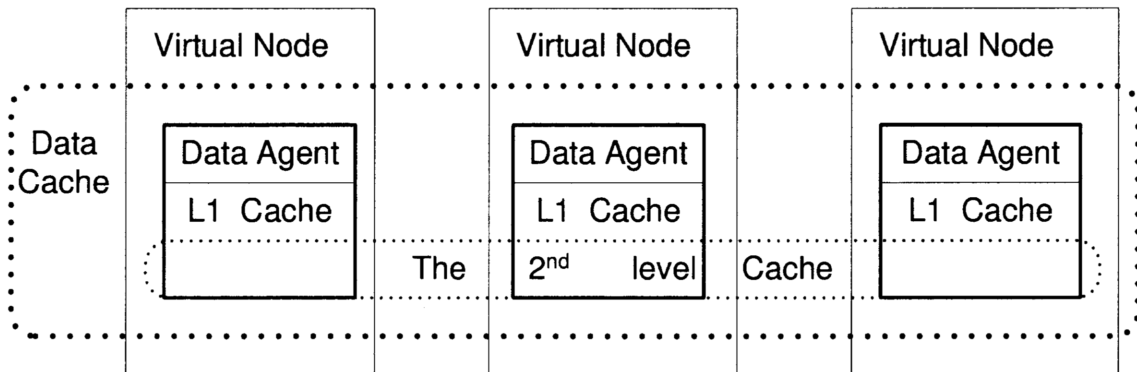


Figure 3.5 The distributed structure of the data cache subsystem

3.2 Task Distribution System

The task distribution (sub)system is one of most important parts of the runtime support system. For executing programs on a distributed computer system, task partitioning and distribution are essential. For programs developed under SPM, tasks are partitioned into standard units - SIs. Task distribution is achieved by dispatching individual SIs in programs. This process is performed dynamically at runtime. Therefore, the runtime support system includes a task distribution system.

In the VM of SPM, the task distribution system is implemented with the SI dispatch unit (IDU). The IDU dispatches SIs to available processors in correct time order. Specifically, the functionalities of the IDU include:

1) Fetching SIs in SFs;

2) Checking data dependencies among issued SIs;

3) Dispatching each time the SI with the highest priority which is ready to be issued;

4) Handling all issues related to SI issuing and commitment;

In the VM, the IDU is a passive unit. It is triggered to work only by external actors. These triggering actors may be IEUs that request more SIs or the commitment of an SI. They may also be users submitting super-programs with request to add their initial SFs to the task pool in the IDU. The IDU is actually implemented as a set of task agents being part of the runtime system processes running on member computers. These agents include the task agent and the task center.

3.2.1 Task Center

The task center in the core of the IDU controls the task distribution. It determines where and when these tasks are to be distributed. Each SP is a special SF that holds all SFs/SIs implementing the corresponding application program under the SPM. When a new SP arrives the task center adds corresponding SF, to the task pool. When an IEU asks for an additional SIs, the task center gets the SI with the highest priority from the task pool after checking for SI dependency. If there exists such an SI that is ready to be issued, the task center dispatches the SI to the requesting IEU

for execution. When it gets message for the commitment of an SI, the task center updates its status appropriately in order to deal with SI interdependencies.

3.2.2 Task Pool

The task pool is a queue structure that helps the task center to find a correct task to issue. Logically, the task pool holds all SIs that must be executed but have not yet been issued. It puts SIs in the queue based on the priority properties of SIs/SFs. The priority of SIs in SFs/SPs is normally implied from data dependencies while programmers may assign a relative priority to different SFs/SIs between which there is not data dependency.

The task pool does not need to hold each all the available SIs. It only holds the SIs that the system currently knows should be definitely executed. During the execution of SPs, new SIs can be added to the task pool based on task activation. This approach makes the task pool small and more efficient.

For higher efficiency, the task pool can also contain SFs. If the task with the highest priority is an SF when a new SI must be issued, the IDU will ask the system to provide an SI from the active SF. An SI from the SF's task pool can be chosen or even can an SI can be created to respond the SI request.

Data dependencies are adhered. A low priority SF can be used to feed an SI for scheduling only if all the SFs with higher priority cannot provide an SI due to constraints from data dependencies. The system will keep all activated SFs are kept active until all their tasks have committed their results. After that, the IDU also commits the SFs as SIs.

SFs are programming units. Using them in the task pool creates a new mechanism for interaction between the runtime system and programs. This makes

the runtime system more flexible. Actually when the IDU gets a notice to commit an SI, the task center can "inform" the corresponding SF. This feedbacks information can be used to adjust the SF status. When the IDU requests for an SF to forward more SIs, it can also provide information that can be used in an effort to maximize parallelism.

3.2.3 Task Agent

A task agent is the local representative of the IDU for the task distribution system. Task agents are physically distributed among multiple computers but they reside in virtual nodes. All task agents in the virtual machine cooperate each other to perform the tasks of the IDUs Each virtual node process includes a task agent. SFs and all other parts of the runtime system interact with the IDU through the local task agent.

3.2.4 Reference Implementation

The entire reference system is implemented in the Java programming language. Therefore, when designing the reference runtime system, the components are specified with java interfaces.

In the reference implementation, the task agents are specified with the interface TaskAgent shown in Figure 3.6. It also serves as the main interface of the IDU. Although it could be implemented in a distributed manner, in the reference implementation the task center is not distributed. It resides in a special virtual node - the virtual center node process. It is specified with two interfaces. The first one, TaskCenter, provides the interface for all task agents to request SIs and commit SIs. The second one, LocalTaskCenter, provides the interface to add SFs in the task pool. It is only available to local components in the virtual center node. To check

```
public interface TaskAgent extends TaskLocalizer{
    public void addLocalTask(Task task);
    void addTaskScheduler(TaskGroup task);
    public IncarnatedTask getNextTask();
    public void commit(IncarnatedTask task);
    public void setTaskCenter(TaskCenter center);
}
```

Figure 3.6 The interface of the task agent

```
public interface TaskCenter {
    Task getTask(TaskRequest request);
    void commit(TaskID taskID, TaskCommitMsg feedback);
}
public interface LocalTaskCenter extends TaskCenter{
    void addScheduler(TaskGroup task);
}
public interface TaskIssueCtrl {
    boolean canIssue(Task task);
    void commit(Task task);
}
```

Figure 3.7 The interface of the task center

data dependencies in SFs before responding to the task center, the task center also exposes another callback interface, TaskIssueCtrl. They are all shown in Figure 3.7.

The interactive processes to commit and issue an SI are illustrated in the sequence diagrams of Figures 3.8 and 3.9, respectively.

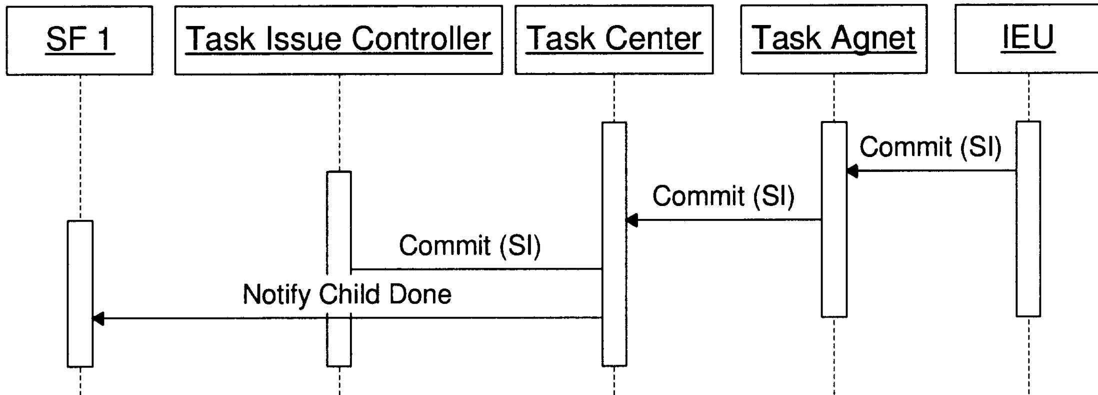


Figure 3.8 The sequence diagram for committing an SI

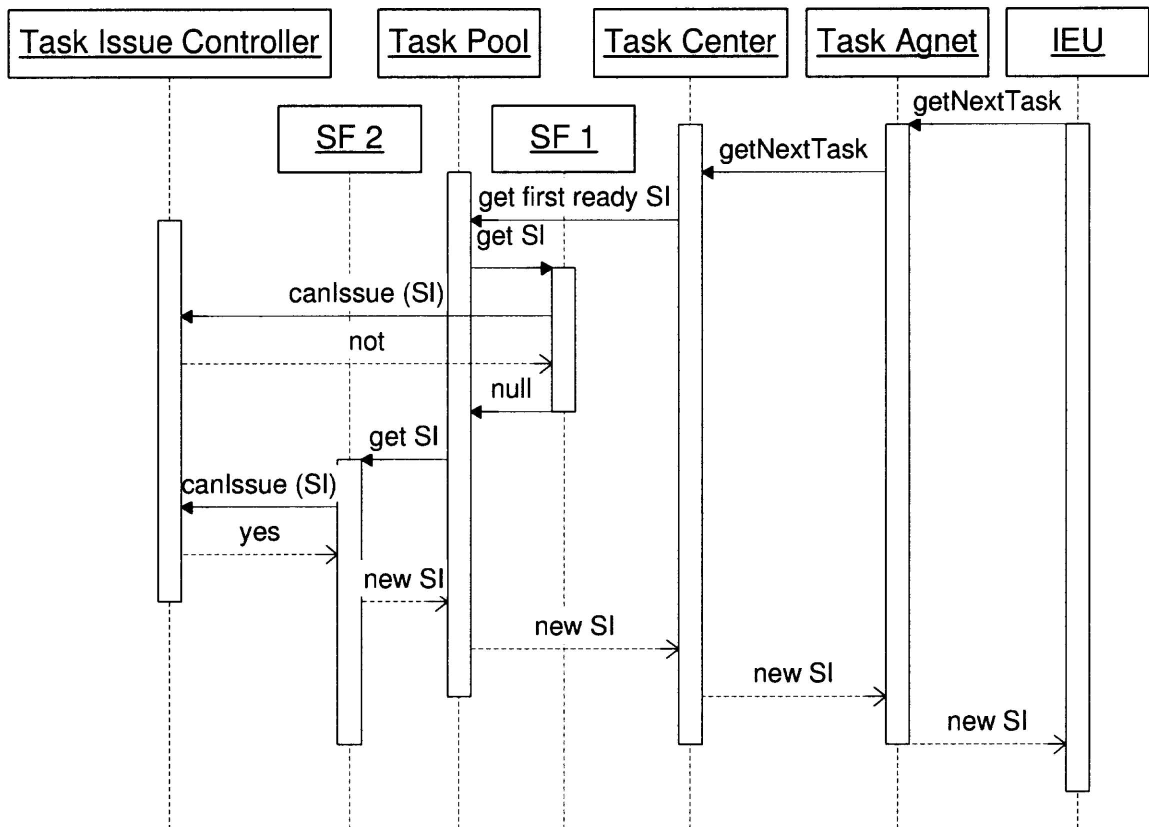


Figure 3.9 The sequence diagram for issuing an SI

3.3 Task Execution System

Executing super-programs on a VM under SPM is performed through scheduling and executing all SIs of the SPs. The task execution system in the VM is part of the runtime system that executes SIs. The task execution system consists of all IEUs in the VM. Although the entire task execution system is distributed, an IEU is not. This property isolates the development of SIs from the parallelization of SPs. In this context, the implementation of individual SIs can be completely sequential. The task execution system is the work engine of the entire VM. It drives the VM to execute super-programs by pulling SIs. When an IEU could load more tasks, it actively asks the IDU of the VM for more SIs. Once it gets a new SI, the IEU will decode the SI and load a local procedure to implement the SI. It also asks the data cache system to provide localized data for operands and then executes the local procedure. After finishing, it puts the results in the data system and notifies the IDU for committing of the SI. It then repeats this procedure by asking more tasks

In the reference implementation of the runtime system, an IEU consists of a task localizer, a controller (called "work engine") thread and multiple work threads. All these components reside in the same virtual node process. The interaction sequence among these components and the task agent is shown in Figure 3.10.

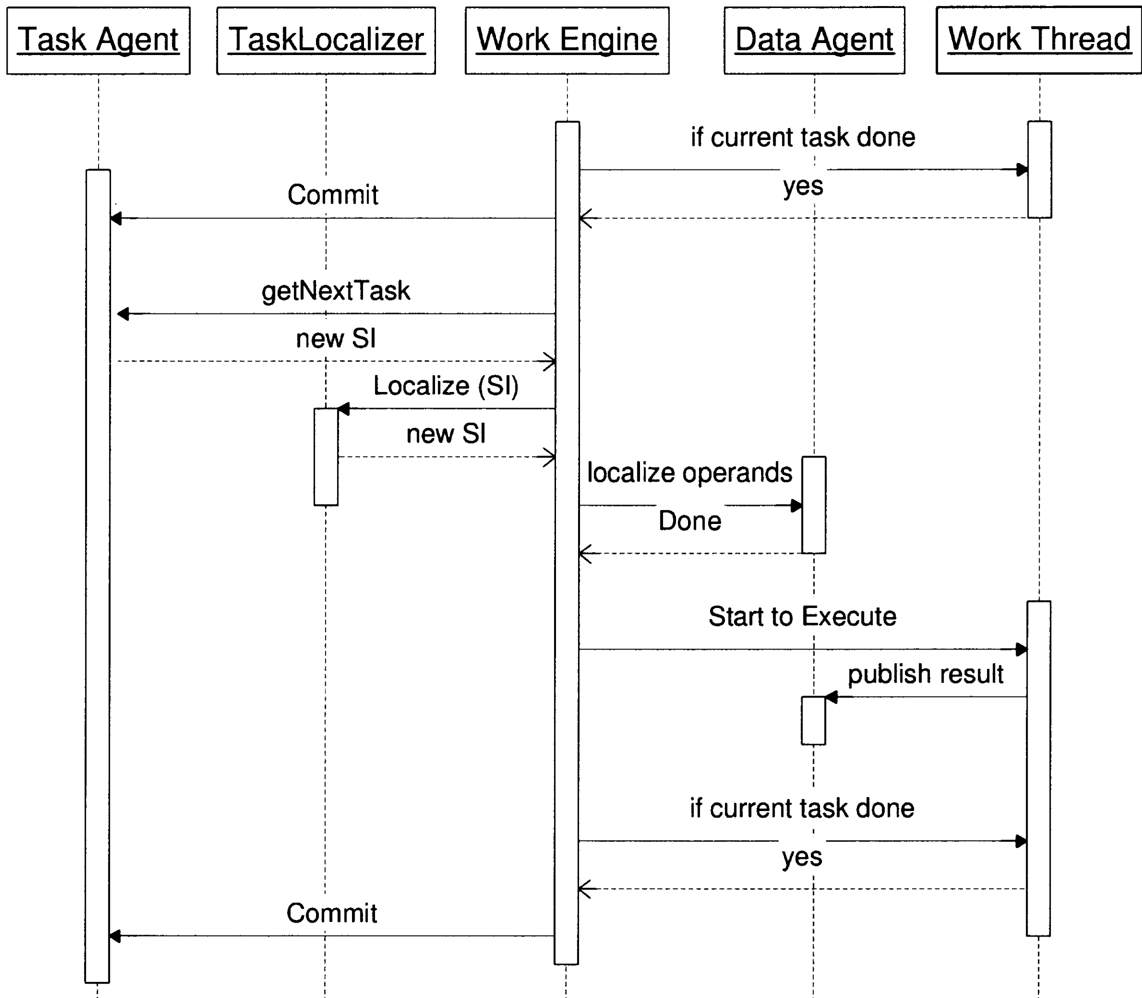


Figure 3.10 The sequence diagram for executing an SI on an IEU

3.4 Data Distribution System

The data distribution system is another part of the runtime support system. Data partitioning and distribution for executing programs on a distributed computer system are essential. For programs in SPM, data are partitioned into standard units -SDBs. These SDBs are dynamically distributed among virtual nodes at runtime. The data distribution system provides the required data to IEUs so that distributed tasks can be executed. The basic functionality of the system is to efficiently deliver data to requesting virtual nodes.

In the VM of SPM, the data distribution system is implemented with a virtual data cache subsystem that achieves the task of distributing data by loading them on demand. This subsystem includes two levels of cache. The first level is the local cache. The second level is a distributed cache. The entire system is backed up by an external mass storage system. All data that cannot be held in the cache are stored in the external system. When an IEU need data, if it is cached locally the data is immediately available; if it is not cached locally but is stored in the distributed cache (i.e., it is cached on another virtual node), the data is loaded from the remote virtual node; otherwise, the data is loaded from the external storage system. Physically the data cache system consists of a data center and multiple data agents.

3.4.1 Data Center

The data center is the coordinator of the data distribution system for operations involving multiple virtual nodes. It also helps distributed task agents to find where they can get a particular SDB for local clients. The data center holds critical meta-information for the data distribution subsystem and all SDBs. The meta-information of an SDB includes the data-ID of the SDB, the location of clients which are currently

using the SDB, the type of SDB usage (e.g., read, write), and so on. At any moment, any other components of the data distribution system can turn to the data center to find any their peers or the status of any specified SDB.

The data center is also a mediator for accessing data. It controls SDB access for SIs. The task center of the virtual machine always cooperates with the data center. Before the task center issues an SI, through the task issue controller, it consults the data center. When the SI is issued, it notifies the data center to update the status of corresponding SDBs. After the SI commits, the task center also notifies the data center to update the status of the relevant SDBs.

3.4.2 Data Agent

In the data distribution system of the virtual machine, there are multiple data agents which are distributed among multiple computers. Actually they reside in virtual nodes where each virtual node hosts a data agent. The data agent is the local representative of the data distribution system. The data agent is the local data provider for a virtual node. All SIs executed on the local IEU delegate the task of finding their operands to the data agent. With the help of the data center, the data agent loads/updates the most recent version of the data (SDBs) into the local memory from any possible locations. The data agent also translates data formats. When it loads data for its local clients, it always translates the data into the most appropriate local format.

All data agents in the virtual machine form a data cache system. The data agent in a virtual node works as the local data cache level that holds the most recently used SDBs. When a local client (SI) needs a particular SDB, if the most recent version of the data is in the cache the data agent can respond to the client

```
public interface DataCenter {
    String getPersistRoot(EntityID callerId);
    Ref getMulticastChannelRef();
    DataRef resolveData( njit.ece.spm.DataID did);
    DataRef notifyHostFull( njit.ece.spm.EntityID callerId);
    void synchronizeDistributedDone(DataID did);
    void publish(DataID did, EntityID host);
    void destroy(DataID did);
}
```

Figure 3.11 The interface of the data center

quickly. All data agents in the virtual machine also cooperate to form the second level, distributed cache. When a local client (SI) needs a particular SDB, if the data is not cached locally or the cached data is outdated, but another data agent has an updated copy of the SDB, the former data agent can load data from the latter data agent remotely. When no data agent has cached the SDB, the local data agent will load the data from the external storage system. When an SDB is kicked out of the cache system, the last data agent that holds the SDB will update the SDB in the external storage system.

3.4.3 Reference Implementation

In the reference implementation, the data agents and the data center are specified with the interface DataCenter and DataAgent, respectively. They are shown in Figure 3.11 and 3.12.

```
public interface DataAgent {
    int UNKNOWN = 0;
    int HOST = 1;
    int CACHE = 2;
    int PERSIST = 3;
    int LOCALWRITE = 4;
    DataModel getData(DataID did);
    void setData(DataModel data, DataID did);
    DataModel getDataTemplate(DataID did) ;
    DataModel getPartialData(DataID did);
    void setPartialData(DataModel data, EntityID nid);
    void updatePartialData(DataModel data, EntityID nid);
    Object checkHost(DataID did, Notice notice);
    void sendNotice(DataID did, Notice notice);
    void destroy(DataID did);
    void persist(DataRef ref) ;
    int host(DataRef ref, DataRef persistRef);
    void invalid(DataID did);
    void synchronizeDistributed(DataID did, EntityID[] peerIDs);
    void grasp(DataID did);
    DataModel graspWithModel(DataID did);
}
```

Figure 3.12 The interface of the data agent

3.5 Runtime Management System

The runtime management system is an auxiliary part of the runtime support system. The functionality of this system is to generally manage various components in the virtual machine. The system should provide for:

1) A bootstrap mechanism for all distributed components in the virtual machine to be able to locate each other.

2) Instruments to dynamically reconfigure the structure of the virtual machine by mapping its resources to different parts of the underlying computers.

In the reference implementation, the runtime management system consists of a set of node agent processes and control threads of virtual nodes. On each computer, a system level service process named node agent is launched before any virtual node process is launched. All node agents form a distributed service. Through a multicasting mechanism they first find each other and setup other communication channels for multicast or unicast communications among them. After that each agent becomes the local representative of the service system. They back each other up by exchanging information on demand.

One of the functionalities of a node agent is to manage all the virtual nodes running on the same computer and help them find other virtual nodes running on other remote computers. This requires that all virtual nodes at launch time register their local node agent. They send some basic information to the node agent, such as the name and end points of their primary communication channels. Then the node agent assigns a unique ID to register the virtual node and publishes this information to the world. A node agent achieves publishing by broadcasting this information to all node agents.

Another functionality of a node agent is to help the local virtual nodes to find other virtual nodes running on other remote computers. A node agent holds all the information of the local virtual nodes and also caches some information related to remote virtual nodes. When a node needs to locate another virtual node by its logical name, the local node agent first checks its cache. If it cannot resolve the request, it can turn to other node agents. At least one node agent is guaranteed to resolve the request. The node agent is also a local instrument to central management tools (that is, not included in a virtual machine). It can perform commands such as launch a new virtual node process or remove a virtual node process from the virtual machine.

CHAPTER 4

EXAMPLES OF PROGRAM DEVELOPMENT UNDER SPM

In this chapter, the process of applying SPM to develop application programs for PC clusters is described with two examples: mining association rules and matrix multiplication.

4.1 Example 1: Mining Association Rules

Mining association rules is a computation intensive problem in business intelligence [107,108]. Many parallel algorithms have been developed [109,110] on many parallel computing platforms for that problem [111–113]. Thus, it is a good candidate for SPM implementation [106] [114].

4.1.1 Basic Concepts in Mining Association Rules

Let $I = \{a_1, a_2, a_3, \cdots, a_m\}$ be a *set of items* and $DB = \langle T_1, T_2, T_3, \cdots, T_n \rangle$ be a *transactions* database with items in I. A *pattern* is a set of items in I. The number of items in a pattern is called the *length of the pattern.* Patterns of length k are sometimes called *k-item patterns.* The *support s(A)* of a pattern A is defined as the number of transactions in DB containing A. Thus, $s(A) = |\{T | T \in DB, A \subseteq T\}|$.

A pattern A is a *frequent pattern* (or a *frequent set*) if A's support is not less than a predefined *minimum support threshold* s_{min}.

A rule is an expression of the form $R : X \Longrightarrow_{s(R),\alpha(R)} Y$, where X and Y are exclusive patterns of I ($X \cap Y = \varnothing$). X and Y are called the *pre-pattern* and

post-pattern of R, respectively. s(R) and $\alpha(R)$ are the support and confidence of the rule R, respectively. The *support s(R)* of rule R is defined as the support s(Z) of the joint pattern $Z = X \bigcup Y$. The *confidence* $\alpha(R)$ of rule R is defined as $s(Z)/s(X)$. An association rule is a rule with support not less than a minimum threshold. Given a transactions database, a minimum support threshold s_{min} and a minimum confidence threshold α_{min}, the problem of finding the complete set of association rules $M =\{$ R: $X \Longrightarrow_{s(R),(R)} Y | s(R) \geqslant s_{min}$ and $\alpha(R) \geqslant \alpha_{min}$ $\}$ is called the *association rules mining problem.* Also, given a transactions database and a minimum support threshold s_{min}, the problem of finding the complete set of frequent patterns is called the *frequent patterns mining problem.*

A number of relevant algorithms are frequent-pattern based [115, 116]. They first solve the frequent patterns mining problem and then check the confidence of all candidate association rules which are built with frequent sets and their subsets. Since the supports of the pre-pattern X and post-pattern Y of a rule are not less than the support of their joint pattern Z ($X \subset Z, Y = Z - X$), if Z is a frequent pattern then X and Y also must be frequent patterns. A rule can only be found between a frequent pattern Z and its subset (X).

4.1.2 The Super-Data Blocks for Mining Association Rules

To mine association rules, a set of SDBs and SIs are designed. The SDB types are shown in Table 4.1.

4.1.3 The Super-Instruction Set for Mining Association Rules

The sample SI set developed to mine association rules is:

Table 4.1 The SDBs for mining association rules

SDB types	Description
BlockOfItems	A list of sorted distinct items that cover a continuous, exclusive partition
BlockOfTransaction	A list of transaction data. Each transaction is a sorted list of items
BlockOfJoinResult	A list of sorted candidates of frequent patterns
BlockOfCandidates	A list of sorted candidates of frequent patterns that have passed screening
BlockOfFrequentSet	A list of sorted frequent patterns and their support
BlockOfRules	A limited number of association rules

1) **LoadDataBlock** (in **DS**, in **s**, out **rawId**): gets a block of data outside of the system. "DS" is the reference to the external data source; "s" is the maximum size of the fetched data; "rawId" is the global ID of a BlockOfTransaction block to hold incoming data.

2) **CountItemSupport** (in **rawBlockId**, in/out **itemsId**): extracts all distinct items and counts the number of times each item appears in a block of raw transactions (identified by "rawBlockId"); with the help of a global mapping object (identified by "itemsId"), it also merges results into respective global data blocks.

3) **ShrinkItemBlock** (in/out **itemsBlockId**, int **minSupport**): This SI prunes items in a BlockOfItems block (identified by "itemsBlockId") by removing all items with counts less than the minimum support "minSupport."

4) **GetFrequentItemsBlock** (in **items**, out **fItemBlockId**, out **fSetBlockId**, in/out **fMapId**): collects all data contained in a list of pruned blocks of items, creates an SDB (identified by "fItemBlockId") of frequent items and an SDB (identified by "fSetBlockId") of 1-length frequent patterns, and then sends appropriate information to the global mapping object (identified by "fMapId"). "items" is the list of global IDs of pruned blocks of items.

5) **ShrinkTransactionBlock**(in **rawId**, out **itemsId**): This SI shrinks a block of transactions based on a block of frequent items. It removes all non-frequent items from every transaction. The parameter "rawId" identifies an SDB of transactions. The parameter "itemsId" identifies an SDB of frequent Items.

6) **MergeTransactionBlock** (in/out **list**): This SI merges a list of pruned SDBs of transactions. The parameter "list" is a list of global IDs of pruned blocks of transactions. The first ID in the list is also used as the ID of the generated data block.

7) **GenxxxCandidatesFamilyBlock** (in **itemSet**, in **frequentBlockId**, in **frequentMapId**, out **candsBlockId**): joins an n-length frequent pattern with all n-length patterns that follow and have the same prefix in the SDB identified by "frequentBlockId"; it generates a "BlockOfJoinResult" type SDB of (n+1)-length candidate patterns that is identified by "candsBlockId." This SI has multiple versions. "xxx" may be "Large", "Middle" or "Small" based on the format of the first parameter "itemSet" that is the first set of n-length patterns. The parameter "frequentMapId" is the ID of the global mapping object that maps the partitions of n-length frequent patterns to IDs of SDBs.

8) **FilterCandidates** (in / out **candsBlockId**, in **frequentBlockId**, in **frequentMapId**): screens candidate patterns in an SDB identified by

"candsBlockId" with an SDB of n-length frequent patterns identified by "frequentBlockId." The parameter "frequentMapId" is the ID of the global mapping object that maps the partitions of n-length frequent patterns to IDs of SDBs.

9) **CountBlockCandidatesSupport** (in/out **candsBlockId**, in **transBlockId**): counts the partial support of the candidate patterns in a candidate block identified by "candsBlockId" in an SDB of transactions identified by "transBlockId".

10) **PruneCandidatesBlock**(in/out **candBlockId**, in **minSupport**): prunes a block of candidates identified by "candBlockId" by removing candidates with support less than the threshold "minSupport".

11) **GetFrequentSetBlock** (in **list**, out **fid**, in/out **fMapId**): merges a list of pruned SDBs of candidate patterns, generates a permanent SDB of frequent patterns, sends the range information to the global mapping object and publishes the block in the global space. The parameter "list" is a list of IDs of pruned SDBs. The parameter "fid" is the ID of a generated SDB of frequent patterns. The parameter "fMapId" is the ID of a global mapping object that maps a range of partitions of frequent patterns to an SDB of frequent patterns.

12) **CheckConfidenceInBlock** (in **preFSetsBlockId**, in **postFSetsBlockId**, in / out **rulesBlockId**, in **minConfidence**): extracts association rules by checking frequent patterns in an SDB with their sub-patterns in another SDB against a threshold "minConfidence." The parameter "preFSetsBlockId" is the ID of a data block that includes the (n-i)-length sub-patterns, where i is an integer. The parameter "postFSetsBlockId" is the ID of the SDB that includes n-length frequent patterns. The parameter "rulesBlockId" is the ID of an SDB of rules.

13) **StoreResult** (in **rulesId**, out **des**): stores a list of generated rules in the external storage. The parameter "rulesId" is the ID of an SDB of rules. The parameter "des" is the reference to a destination in the external storage.

4.1.4 The Super-Program for Mining Association Rules

The overall algorithm of the SP adopts the Apriori algorithm described in [116]. The SP is shown in Fig. 4.1. Initial, gen_frequentSet, and find_rules are separate SFs and gen_candidate includes two sequentially issued SFs (join and filter):

P_1 = `initial(`DS, s_{min}`) //DS is the external data source;`

`//if P1 is available and` $P_1 \neq \varnothing$ `continue`

`For (`$k = 2$; $P_{k-1} \neq \varnothing$ `; k++) do begin`

C_k = `gen_candidate(`P_{k-1}`) //include join and filter.`

`//When` C_{k-1} `is available:  issue the next SF`

P_k = `gen_frequentSet (`C_k, s_{min} `)`

`//When` P_{k-1} `is avaible:  issue the next SF and`

`//simultaneously start the next iteration`

R_k = `find_rules(` $\bigcup_{j<k} P_j$, P_k `)`

`End`

`Answer =` $\bigcup R_k$

Figure 4.1 The SP of mining association rules

4.1.5 The Super-Functions for Mining Association Rules

1. The "initial" SF

"initial()" finds the value domain of the items, identifies all distinct items and stores them in a sorted list of SDBs; it also counts them and prunes these blocks. It then generates a list of data blocks of 1-length frequent patterns. In the SP, the pseudo code of the implementation is:

```
while (there is more data){ LoadDataBlock (DS, s, rawBlockId_i);}
parallel do for all rawBlockId_i{
    CountItemSupport(rawBlockId_i, itemsId);
}
parallel do for all blocks in item list {ShrinkItemBlock(itemsBlock_i, s);}
parallel do for all blocks list_i{
    GetFrequentItemsBlock (list_i, fItemBlock_i, fSetBlock_i, fMap);
}
parallel do for all blocks in transactions list rawBlockId_i {
    while (there are more itemsBlock_i blocks that must be checked){
        ShrinkTransactionBlock(rawBlockId_i, itemsBlock_i);
}   }
```

2. The "gen_candidate" SF

The pseudo codes of the implementation of this SF are:

```
parallel do all data blocks of frequentBlock_i {
  parallel do all same and following data blocks frequentBlock_j{ //i <= j
    if (can be joint) {
        GenxxxCandidatesFamilyBlock (itemSet in frequentBlock_i,
          frequentBlock_j, frequentMapId, candsBlock_m)
        //version of the SI used is determined at runtime based on data
        while (candsBlock_m still need check by frequentBlock_p) {
```

```
        FilterCandidates (candsBlock_m, frequentBlock_p, frequentMapId)
} } } }
```

3. The "gen_frequentSet" SF

The pseudo codes of the implementation of this SF are:

```
parallel do for all data blocks of k-length candidate patterns candsBlock_m{
    parallel do for all blocks in transactions list transBlockId_i {
        CountBlockCandidatesSupport (candsBlock_m, transBlockId_i)
    }
    PruneCandidatesBlock(candsBlock_m, S_min)
}
parallel do for all partition list_i {
    GetFrequentSetBlock (list_i, fid_i, fMapId);
}
```

4. The "find_rules" SF

The implementation is:

```
parallel do for each data model of k-length frequent patterns
    block frequentBlockp {
    parallel do for each SDB of frequent patterns block frequentBlock_m {
        if (there are more SDBs that include sub-patterns needed
            to be checked){
            CheckConfidenceInBlock (frequentBlock_m, frequentBlock_p,
                rulesBlock_i, minConfidence)
}   }   }
```

4.2 Example 2: Matrix Multiplication (MM)

Matrix multiplication (MM) is a widely used operation in scientific and engineering computing [117, 118]. In this section, our approach of MM in SPM is presented [119, 120].

4.2.1 The Super-Instruction Set and Super-Data Blocks for MM

Following the requirements of SPM, a set of SDBs is defined for MM that includes only one SDB corresponding to a *sub-matrix block*. The sub-matrix block is an $n \times m$ matrix, where both m and n are not larger than a predefined parameter k. This kind of SDB encapsulates all types of matrices. They can be dense or sparse. The sub-matrix block not only includes the original values of the contained elements, but also the matrix features, such as the sparsity of the block, in the form of metadata. Arbitrary large matrices in applications can be partitioned and expressed as matrices with SDBs as elements. The sparsity and type of SDB elements may vary.

To support MM, a set of SIs is defined. Besides load and store SIs, it includes only one extra SI namely Multiply (A, B, C), where A, B and C are sub-matrix blocks. This SI performs $C = C + A \times B$. SIs include all operand references including the data ID and meta-data of the SDBs. Thus, the IEU executing the SI always knows the type of sub-matrix block and can select an appropriate implementation to deal with this kind of matrices to achieve high performance. Therefore, at the SI level one SI may be sufficient to handle all kind of matrices.

4.2.2 The Super-Function for Matrix Multiplication

To support the multiplication of arbitrarily large matrices, an SF of MM is defined that performs $C = A \times B$, where A, B and C are arbitrarily large. For the sake of

simplicity, we assume A, B, and C are evenly partitioned into N x N matrices (i.e., X=$\{X_{i,j}\}$, for X = A, B or C and i, j = $1, \cdots, N$, where the elements $X_{i,j}$ are sub-matrix blocks). All SIs updating the same sub-matrix block are grouped into a task group ($SI_{i,j,k}$, for $k = 1, 2, \cdots, N$ form group $G_{i,j}$).

The SF adopting the classical MM algorithm is illustrated in Fig. 4.2. It invokes the multiply SI N^3 times. Each SI instance $SI_{i,j,k}$ performs the following operation: $C_{i,j} = A_{i,k} \cdot B_{k,j} + C_{i,j}$, for all i, j, k = $1, 2, \cdots, N$. The all-in-one algorithm fits all kinds of matrices without heavy performance penalty because the VM works in the asynchronous mode issuing and executing SIs. An MM SI with an {0} can finish its execution when it finds an operand is a zero block. The check can even be performed at SI issue stage on IDU. And when an IEU quickly finishes its current SI, it can request another task to exploit it resource without need to wait for finishing of those slow IEUs.

```
C_{i,j} = {0} // for any i and j;
While (not done){
    SI_{i,j,k} = selectNextSI( IEU information);
    // issue SI_{i,j,k} to perform C_{i,j} = A_{i,k} × B_{k,j} + C_{i,j}
}
```

Figure 4.2 The super-function for matrix multiplication

4.2.3 Scheduling Policies and Parallelization of the SF

Under the SPM model, parallelization of SPs is achieved through the cooperation of the IDU and SFs in issuing SIs. For MM, the following policies were designed for

scheduling. Except for the synchronous policy, all other policies are asynchronous. The effect of them is discussed later.

1) Synchronous Policy (syn)

This policy imitates parallel computation on a VM with N^2 nodes in the style of BSP. Under this policy, SIs are issued in the synchronous mode. Each IEU in each superstep gets just one SI. More specifically in the k-th superstep the $SI_{i,j,k}$ is assigned to the $(i \times N + j)$-th IEU for each i and j. No more SIs are issued until all IEUs finish their work and the next superstep begins.

2) Static Scheduling Policy (S)

In this scheme, task groups are statically associated with IEUs. Determining the task group for each IEU uses the optimal algorithm described in [121]. When an IEU requests an SI, the SF feed it with an SI from the task group associated with it. If all its task groups are done, no SI will be issued to the IEU.

3) Random (Dynamic) Scheduling Policy (R)

Under this policy, the task groups are dynamically allocated. Each IEU is associated with a current task group. When an IEU requests an SI, the SF feeds it with an SI from its current task group; if its current task group is empty and there exist unassociated task groups, the SF randomly picks up such a task group and associates it to the IEU and issues an SI from the group. This process continues until all task groups have been associated with IEUs. After that, the static policy (S) is applied.

4) Smart Scheduling Policy with Random Seed (SR)

This scheduling policy is basically similar to the Random Scheduling Policy. The difference is that only the first task group associated with an IEU is randomly

selected, and the following tasks are selected following this rule: the new task group should reuse sub-matrix block operands of previous SIs as more as possible.

5) Smart Scheduling Policy with Static Seed (SS)

This policy is very similar to the above SR policy except that the first task group assigned to IEUs is statically selected as in the static scheduling policy.

CHAPTER 5

PARALLEL EXECUTION OF SUPER-PROGRAMS AND RELEVANT OVERHEADS

The execution of programs developed under high level programming models may suffer from some runtime overheads. In this chapter, the overheads of executing SPs are analyzed and one of the overheads, management overhead, is estimated.

5.1 Overview of Overheads

Generally, overheads are any additional costs not directly related to the application that are the result of used to developing or executing programs. Although there are many different meanings of cost, herein all overheads are related to actual execution overhead.

5.1.1 Overheads in Program Execution

Overheads can involve various system resources. Since the execution time of programs has closer relationship to the utilization of processors, in most situations overheads correspond to additional CPU times. Overheads are the price we pay in applying programming models to develop software. Although these models make software development more efficient, the corresponding runtime support systems consume system resources and introduce overheads. Every layer in the layered infrastructure of software development may introduce overheads. For example, when developing applications in Java for multi-user computers, Java VM and the local operating system may introduce overheads. For this reason, in evaluating new

programming models we need to study the overhead introduced by them and separate them from overheads imposed by the underlying system.

5.1.2 Quantitative Definition of Overheads in Parallel Executing Programs

Let us first give quantitative definitions of some important concepts. The ideal (inherent) performance P_{ideal} of a computer is expressed in MIPS or MFLOPS. For a non-parallel system, its ideal performance can be measured directly by executing an optimized program sequentially on the computer. For a cluster system, the ideal performance can be reasonable defined as the sum of the performance of all member computers; i.e.,

$$P_{ideal} = \sum P_i \tag{5.1}$$

where P_i is the ideal performance of the i-th node in the cluster system.

Since the computing capacity R of a computer system depends on time and its performance,

$$R = P_{ideal} \times T = (\sum P_i) \cdot T = \sum (P_i T) = \sum R_i \tag{5.2}$$

where Ri is the computing capacity of the i-th member computer in the system, The workload W of a task is now defined as

$$W = R = P_{ideal} T \tag{5.3}$$

where T is its ideal execution time.

For a specified task, its workload may vary for different computers. Its inherent workload W_{ideal} is the minimum workload of the task. In practice, it may be measures as the workload of the optimized program executing on a sequential computer. For this reason, W_{ideal} is accumulative. I.e., The W_{ideal} of a task is equal to the sum of inherent workload of all sub-tasks. And the W_{ideal} can be defined reasonably as produce of the ideal performance of a reference sequential computer with the execution time of an optimized program. I.e.,

$$W_{ideal} = P_{ideal-ref} \times T_{s-ref} \tag{5.4}$$

The workload W of a program that execute on a parallel computer system, usually, is definitely larger than the inherent workload W_{ideal} of the task. The difference between W and W_{ideal} is the so called overhead of the program. For parallel computer system programs, generally, overheads of execution of programs mainly come from 3 factors: data communication, nodes idle and management. Thus, the total workload should be:

$$W = W_{ideal} + W_{idle} + W_{comm} + W_{man} \tag{5.5}$$

where W_{idle} is wasted computing resource; W_{comm} and W_{man} are overhead of communication and managing sub tasks, called as absolute imbalance, communication, and management overheads of the problem respectively. For easy to compare, overheads are express in term of relative amount of computation resource consumed in each parts against the total workload of the program. I.e.,

$$\begin{aligned} w_{idle} &= \frac{W_{idle}}{W_{ideal}} \\ w_{comm} &= \frac{W_{comm}}{W_{ideal}} \\ w_{man} &= \frac{W_{man}}{W_{ideal}} \end{aligned} \tag{5.6}$$

where w_{idle}, w_{comm}, and w_{man} are (relative) imbalance, communication, and management overheads respectively. The efficiency of execution of a program on a computer can be defined as the ratio of its inherent workload vs the actual workload. I.e., the efficiency is

$$E = \frac{W_{ideal}}{W} = \frac{1}{1 + w_{ideal} + w_{comm} + w_{man}} \tag{5.7}$$

5.2 Overheads of the Runtime System of SPM's VM

In SPM, the VM is underlain with the PC cluster. All processors are encapsulated in virtual FUs such as IDU, and IEUs. Super-programs run on the system through IDU scheduling and distributing SIs to IEUs and IEU executing these SIs. The total consummation of computing resource is the sum of CPU time to execute the runtime processes held all these virtual functional units. So we can count the processor consummation of execution of IEUs and IDU.

As mentioned in Section 2.4, during executing an SP, the IDU in the VM continually fetch SIs from SFs and assign these SIs to IEUs, and then IEUs perform these tasks (SIs) assigned to them. Since the IDU only perform controlling task required by SPM, execution of the IDU is management overhead completely. Before an IEU begin to execute the SIs assigned to it, the operands of the SIs must be local available. Since the SIs are distributed dynamically, all the SIs and the operands have to be distributed at runtime. I.e., IEUs may need remotely load these data. Therefore they suffer communication overhead. Besides these overheads, SPs executing on PC clusters may also suffer from the problem of load imbalance. When programs are lack of parallelism, IDU cannot always satisfy requests of IEUs for SIs. In this case, some IEU will be idle and its computing resource may be wasted. This has exactly same effect as others overhead that makes the execution time of programs

increase. Therefore it is treated as another overhead universally–imbalance overhead. Summarily, the total overheads F is

$$F = M + C + I \tag{5.8}$$

where M, C and I represent the management overhead, communication overhead and imbalance overhead, respectively when executing SPs.

5.3 Management Overhead

5.3.1 Source of Management Overhead in SPM Runtime Environment

Management overhead is CPU consummation to conduct the management activities that are the execution of the IDU and the interaction between the IDU and SFs. It includes activating SFs, maintaining tasks pools, scheduling SIs and notifying relevant parts for commit of SIs.

The absolute management overhead M directly depends on the number of SIs to be managed that in turn depends on the size of problems and these grain parameters of SIs. i.e:

$$M = M(n_{SI}) \tag{5.9}$$

where n_{SI} is the number of SIs to be executed to solve a specified problem. For a specified SP, larger the application problem is, more SIs are managed and heavier the overhead. For a particular problem, finer the grains of SIs are defined, more SIs are managed and heavier the overhead is. For generally evaluating, the relative (management) overhead may be more important. In practice, it can express in the fraction of management workload vs the total workload of SPs. Using to represent

the relative overhead, it is

$$\mu = \frac{M}{R} \tag{5.10}$$

where, instead of ideal workload of the problem, R is the actual total workload. If temporary ignore others kinds of overheads in this analysis, R will be

$$R \approx M + W_{ideal} \approx M + n_{SI} t_{SI} P \tag{5.11}$$

where n_{SI} is the number of SIs in a SP, t_{SI} is the execution time of an SI on a member computer, and P is the inherent performance of a member computer. Since both M and R depend on n_{SI}, the ratio μwould have weaker dependency on the number of SIs n_{SI}. It means the relative overhead would have weaker dependency on problem size. Thus, the relative management overhead is better in use to evaluate the SPM and compare it with other programming. Equation (5.8) and (5.10)) show that M does not explicitly depend on the average time to execute an SI while R linearly increase with the average execution time t_{SI} of an SI. This implies that the relative management overhead reduces while the grains of SIs increase. Thus, in the view of point of reducing overheads, SPM is better to be applied to computer systems with coarse clustering components such as PC cluster than to systems with fine parallel components.

5.3.2 Experiment Measure of Management Overhead

To verify the management overhead of the runtime system is affordable, an experiment was setup. In the experiment, two groups of programs are developed. One group is the SPs for the MM described in chapter 4; the other is programs that simulate these SPs and the RTS. Both of them implement management system in same way. And each pair of the SP and corresponding simulation program

implements exactly same scheduling policies. For this reason, two programs should suffer same management overhead. Since the overhead of the simulation program should be less than its total workload, the latter should be upper bound of absolute management overhead of the experimental SPs and the ratio of execution time of the simulation program vs execution time of the SP should be an upper bound of relative management overhead. The experimental results are shown in Fig. 5.1. Most executions of the simulation program for selected data sets take about 60-200ms. And the execution time of the SP for corresponding data set take about 150-300 seconds while the net execution time of an SI is about a few ms for multiplication of a pair of 256×256 sparse matrices with 5% non-zero elements. The relative management overhead is less than 1% of the total execution time. Figure 5.1 shows the upper bound on the relative management overhead, which is expressed as the ratio of the execution times between simulations and experiments. This result implies that for such coarse-grained system the management overhead is affordable.

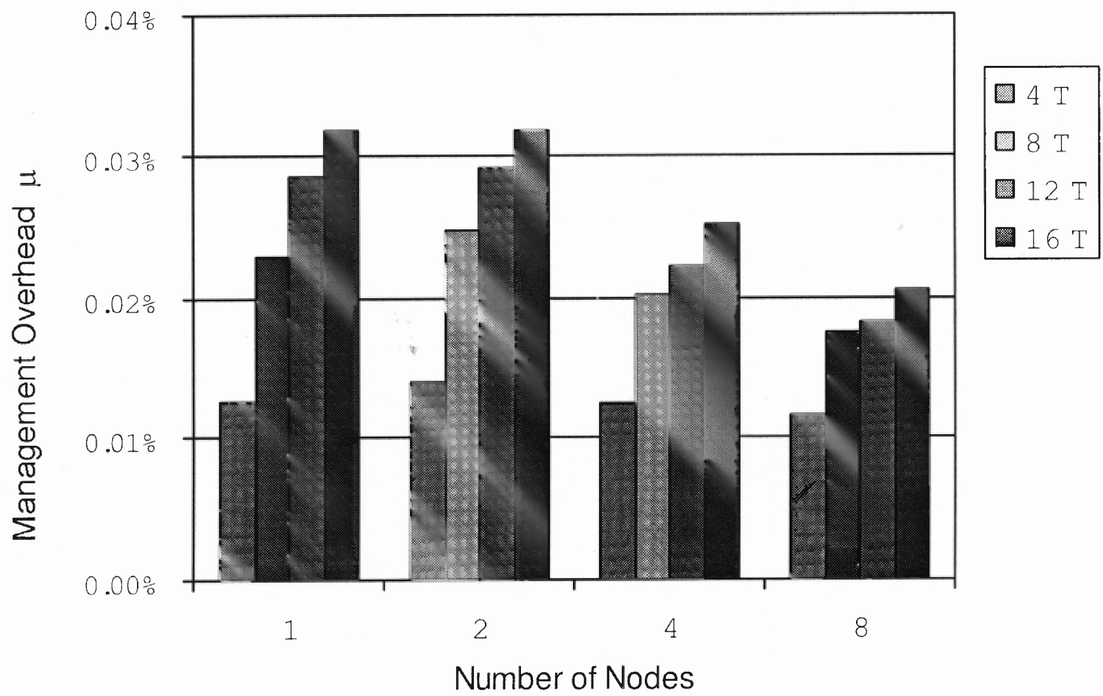


Figure 5.1 Relative management overhead in execution of the MM SP.

CHAPTER 6

LOAD BALANCING

Load balancing focus on the efficient utilization of resources. In this chapter, the load balance mechanism under SPM is discussed. In SPM, we pay special attention to minimizing the associated overhead.

6.1 Overview of Load Balancing

Load balancing is the effort to execute a program in parallel so that the computation work of the entire program is fairly distributed among all system resources, especially processors. Then, the resources are utilized efficiently and higher performance can be achieved.

6.1.1 Requirement for Load Balancing

Load balancing is fundamental in parallelizing programs for a high performance [41, 105]. During the execution of a program in parallel, the computation tasks (no matter with what kind of granularity level) must be partitioned and distributed among the processors of the computer system. If the distribution of these tasks is imbalanced, the overloaded processor must execute more computation tasks. This means they have to take longer time to perform the task assigned to them. Then the execution of the entire program will be extended because the execution of the program finishes only after all of its partitioned tasks have finished. Well balancing the distribution of tasks can help to improve performance of executing the program in parallel. Most programs executed on parallel computer system do real care about

execution performance, although some programs, especially some human-computer interactive programs, do not care execution time very much. Therefore Load balancing is essential to execute programs on parallel computer systems.

Load balancing is just technologies that work in the context for parallelization of programs. The executing programs in parallel just exploit the parallelism of the programs and the parallelism of programs may exist in multiple levels. There are the finest instruction level parallelism, the procedure level parallelism and the coarsest job level parallelism and so on. In general, the execution of a program may involve a combination of multiple levels. But most programming models involve only one or two level of these parallelisms. Different programming models work at different grain level to exploit different level parallelism by distributing tasks at the specified grain level. Load balancing for programs under these programming model try to balance the distribution of these tasks at the particular level. The tasks at different levels have different characteristics about their execution and communication. The strategies of load balancing for these programming models, therefore, should base on the features of task at the particular grain levels of the programming models.

For PC cluster systems, programs usually exploit procedure level or job level parallelisms, the workload of programs are partitioned into coarse-grained functional-oriented tasks instead of fine-grained native operation of processors. These tasks are far more irregular than the basic instructions of the hardware processors. And these tasks usually do not occupy entire computer resource completely but the local operating system schedules them to use the resource on the computers. For such systems, well balancing the workloads means that the member computers in clusters are neither overloaded nor underloaded. During execution of the entire programs, they have a little chance to be idle while others are busy.

6.1.2 The Types of Strategies for Load Balancing

General load balance strategies can be classified into two categories. The first one is the static load balance strategies. The other one is the dynamic load balance strategies. Static load balance strategies are applied before the execution of programs. They try to optimize the task partition at compile time so that processors get even computation tasks but they do not adjust the task distribution at runtime. In the other hand, dynamic approaches perform some load balancing operation at runtime. I.e., at runtime, the loads of processors are adjusted continuously based on the execution condition of distributed processors.

Static strategies are based on predicting the workload and been used in many scientific computing program [118]. Strategies of statically load balancing work well for predictable environments such as the cases that the application problems have regular sub-tasks and the parallel platforms targeting these programs have dedicated resource so that the workload of each (sub)task are predictable. (SIMD computers are a good example of such environment). In the case, the basic tasks distributed are some very simple operations that are executed with dedicated hardware. The entire processes of execution of the tasks are predictable. Therefore the compilers can create correct task distribution for targeted processing elements. Static load balancing may not introduce any other runtime overhead.

Strategies of dynamically load balancing fit to non-predictable environments better. In these situations, sine the execution time of each distributed tasks that executes on specified computing resource is not predictable, compilers lose their base to optimize the task distribution. The distribution of statically partitioned task may be imbalanced. Then static load balancing cannot work very well. It may increase the execution time of entire programs and reduce the performance. Thus, dynamically balancing load at execution time is high desirable.

PC cluster systems are typical environment of non-predictable computation platforms. For PC clusters, programs are usually parallelized at the procedure level. The workloads of programs are partitioned into coarse-grain function-oriented tasks that are usually scheduled by the local operating system. The execution times of these tasks are far more irregular than the basic instructions. The one reason of generating the derivation of execution time is that subtasks themselves may have irregular workload. Workloads may not be proportional to the data and the data distribution may be skewed. The other reason is that the availability and performance of computing resources. That could happen when these computing resources are not dedicated, which are shared among many programs. All of these factors can make a static estimation inaccurate [122]. In this situation, static load balancing does not work well. Therefore, for PC clusters, dynamically balancing workload is essential.

Dynamically load balancing can be implemented in two different ways. One is by re-distributing tasks. The other is using task queues. The first approach partition all tasks to all member processors. And then adjust task distribution by migrating tasks between the distributed nodes when loads are imbalanced. Adopting the second approach, the system only distributes a few workloads to each nodes and keeps remained tasks in task pools; it delays to make the decision of assigning tasks to particular nodes until it get more real time information of execution conditions on each node. Once a task is assigned to a processor, the task may not be migrated again. From the view of point of developer, there are two kinds of approaches to implement the logics of dynamically balancing workload. One is embedding logics in application programs. The other is employing high-level runtime system to provide some load balance mechanism. Adopting the first approach, application programmers implement some adapting algorithms so that the tasks of the application program can be repartitioned and immigrated among computers in periods or based on demands.

Adopting the second approach, system developers implement the mechanism of load balancing; application programmers just exploit the mechanism. They follow requirements of runtime system to fill all tasks of the parallelized programs into the task queue. In this case, the most efforts involve a one-time job and are included in the system development of runtime support systems for specified programming models.

6.2 Load Balancing for SPM

SPM adopts the dynamic load balance approach. It requires the runtime support systems of the virtual machines provide a system level mechanism for load balancing automatically. It balances workload with a task queue at SI level. All application programs developed under the programming model can exploit the system mechanism to achieve load balancing. The workloads of all application programs developed under the programming model can be balanced without additional effort in development, as long as SPs are well parallelized, (i.e., the degrees of parallelism in SFs/SPs are high enough).

SPM adopts the dynamic load balance approach. It provides a system level mechanism for load balancing.

6.2.1 Mechanism of Automatically Load Balancing in SPM

SPM adopts a dynamic load balancing approach without task migration. It achieves the goal of balancing workload well through providing system level mechanism. This mechanism for load balancing is embedded in the mechanism of mapping and scheduling tasks. Instead of pre-allocating all tasks to member computers based on the any kind of estimation of workload of partitioned tasks and performance of member processors in the multi-computer system, the runtime system continuously

distribute SIs to multiple IEUs that run on the distributed computers based on the availability of free computation resource.

The mechanism of automatic load balancing is based on a protocol between the IDU and IEUs. This protocol specifies that:

1) The task distribution is a successive process during the execution of SPs. The IDU continuously controls the flow of tasks distributed to IEUs;
2) Each task assigned to IEU is an SI, which always has a limited workload.
3) The IDU assigns an SI to an IEU only when the IEU request more tasks.
4) All IEUs are general-purpose. They can execute any assigned SI. The IDU only considers availability of IEUs, availability of SIs, and dependences between SIs.
5) An IEU requests more tasks immediately when the computation resource is available to execute one more SI. All IEUs work in an asynchronous mode.
6) An IEU requests more tasks only if it has enough resources to execute additional SIs and promises to complete the SI in a predetermined time.
7) If possible, the IDU always satisfies the task requests of the IEUs. This means that if there is any SI in task pools that are ready to issue (i.e., it does not depend on all currently executing SIs), the IDU always dispatches an SI to the IEU that requests more tasks.

With this contract between the IDU and IEUs, the workload can be well balanced among all member nodes in clusters. Under SPM, SIs are well designed task units. They have predetermined upper bound of workload. IEUs know how many computing resource is needed to execute an SI. The item 1), dynamically distributing tasks at SI level, assures the flexibility of the mechanism. And combination of the

item 2), 3) and 6) guarantees no nodes to be overloaded. The combination of the item 5) and 7) make nodes less possible to be starved. The member nodes that can execute SIs assigned to them more quickly than others will be assigned more work, i.e., the faster nodes obtain more SIs. The system can be unbalanced only if SPs do not have enough parallelism. For a good program, such a situation usually appears when the last SI before a synchronization operation has been assigned. However, SIs have limited workload. So, this situation does not last long time. Therefore, if SPs are well parallelized, good load balancing is achieved automatically.

To facilitate simplicity and efficiency in practice, SIs are not required to have restrict identical workloads. A small to medium diversity should be allowed among the workloads of SIs. The asynchronous work-mode assures the diversity not increase the chance that IEUs are idle. A scheduling policy using a producer-consumer protocol will balance the workload well for an application with abundant parallelism.

6.2.2 A Model of Estimating the Imbalance Overhead for SPM

The purpose of load balance is preventing some clustering computers from being under-loaded while others are overloaded. Since the workloads are relative. The under-loaded member computers ultimately always become idle while the overloaded member computers are busy. Thus the performance of a load balance strategy can be measured with the chance that the member processors /computers are idle. The model of the load balance should be based on task partition structure. In SPM super-programs have the layered structures. At high level, SPs that are developed by application programs who focus on application logics and pay less attention on parallelization and load balance are compose in SFs. And at middle level, these SFs are compose in SIs and developed by domain experts and system developers who focus on parallelization of these SFs and load balance so that the parallelism of underlain

hardware resource can be exploited efficiently. At the low level SIs as atomic tasks are implemented as normal sequential procedures. For this structure, when develop a basic model to describe the behavior of the mechanism to automatically balance workloads, we can assume that SFs, which themselves are executed on VM in parallel, are executed one by one in sequences but these SFs in an SP always are ended with synchronization operations.

Assume that the number of IEUs, i.e. the parallelism of underlain resource is n_p, the average number of SIs in an SF is n_i that are much greater than np. in the model, the diversity of workload of SIs and the diversity of available capacity of underlying processors resource are considered together. Both are represented with a single measurement – the execution time of SIs. Also assume the average resource that is consumed to execute an SI is I that include the management overhead to issue and commit the SI and the communication overhead to handle communication. Then for an SP with n SF super function, there is total n periods that during them system is lack of parallelism and workload are distributed imbalance, (i.e., some IEU have to be idle). Each of them is corresponding to an end of an SF. After the last SI of an SF, an IEU no long can get any more SI when it finish its current task until entire SF has done. During such an imbalance period, $n_p - 1$ IEUs can become idle. The average idle time of an IEU can roughly be estimated as half time of the execution of thee last SI of the SF. Therefore the wasted computation resource is

$$D = (n_p - 1) \cdot c \cdot I \tag{6.1}$$

where c is a constant $(0 < c < 1)$. The value of c depends on profile of performance distribution of IEUs in the system. While the net accumulated time the SF consume

to execute its SIs (include overhead other than imbalance overheads) is

$$A = n_i \cdot I \tag{6.2}$$

And the total consumed computation resource - IEU time is

$$R = A + D = [n_i + c \cdot (n_p - 1)] \cdot I \tag{6.3}$$

Therefore the percentage of wasted processor time, the chance of an IEU being idle is

$$\delta(n_p) = \frac{D}{R} = \frac{(n_p - 1)}{(n_i/c) + n_p - 1} \tag{6.4}$$

$$= \frac{c \cdot n_p}{n_i} \qquad when 1 \ll n_p \ll n_i \tag{6.4 a}$$

This means the effect of load balance directly relevant to parallelism of SFs.

Although application developers who use SFs as high level task components may pay less attention on parallelization and load balance, it is not means the multiple SFs in an SP absolutely must be executed sequence. In the case, the parallelism is determined by runtime system of the VM. If the VM find there is no dependency between successive SFs, it can also issue the both SFs in the sequence in the SP. It can execute all SIs in these issued SFs in parallel but the SFs issued later have lower priority to issue its SIs. When the SF with higher priority are lack of parallelism, the SIs in the low priority SFs can satisfy the request of IEUs and then the chance that an IEU become idle is reduced. The exist of parallelism among SFs equivalent to increase the average number n_i of SIs in an SF. Therefore exploiting parallelism at SFs level still is important. It is especially critical for

non-scientific/engineering applications that have not so large number of parallelism in every SF.

6.3 Evaluation of the Load Balance Mechanism

To examine the effect of our load balancing strategy employed in SPM, a couple of experiments were conducted, which include running the sample SPs described in Chapter 4 on PC clusters and simulating the executions of these SPs. During the execution, the utilization of IEUs is monitored and the actual idle time of the IEUs is measured.

6.3.1 Simulation of Automatically Load Balancing

The first evaluation of the effect of the proposed automatic mechanism on load balance was simulation of the execution of super-programs. It was performed with a simulation program for the MM problem. In the simulations, the number of PCs in the cluster is chosen as 1, 2, 4, 8, 16, 32 and 64. The input/output matrices contain 32×32 blocks. Based on the results of our pilot experiments, we assume that the workload of SIs is distributed uniformly and is between 0 and $20000\mu s$. Both the sender and receiver take $250\mu s$ to load a sub-matrix block remotely. In a heterogeneous environment, the relative peak performance of nodes with an odd index or even index is 3 and 1, respectively. All simulations were repeated 16 times for different data sets. Except for the "syn" policy that was used only with multicast operations in a homogeneous environment, all combinations of scheduling policies and environments were used. Simulation results of the imbalance overheads are shown in the Fig. 6.1.

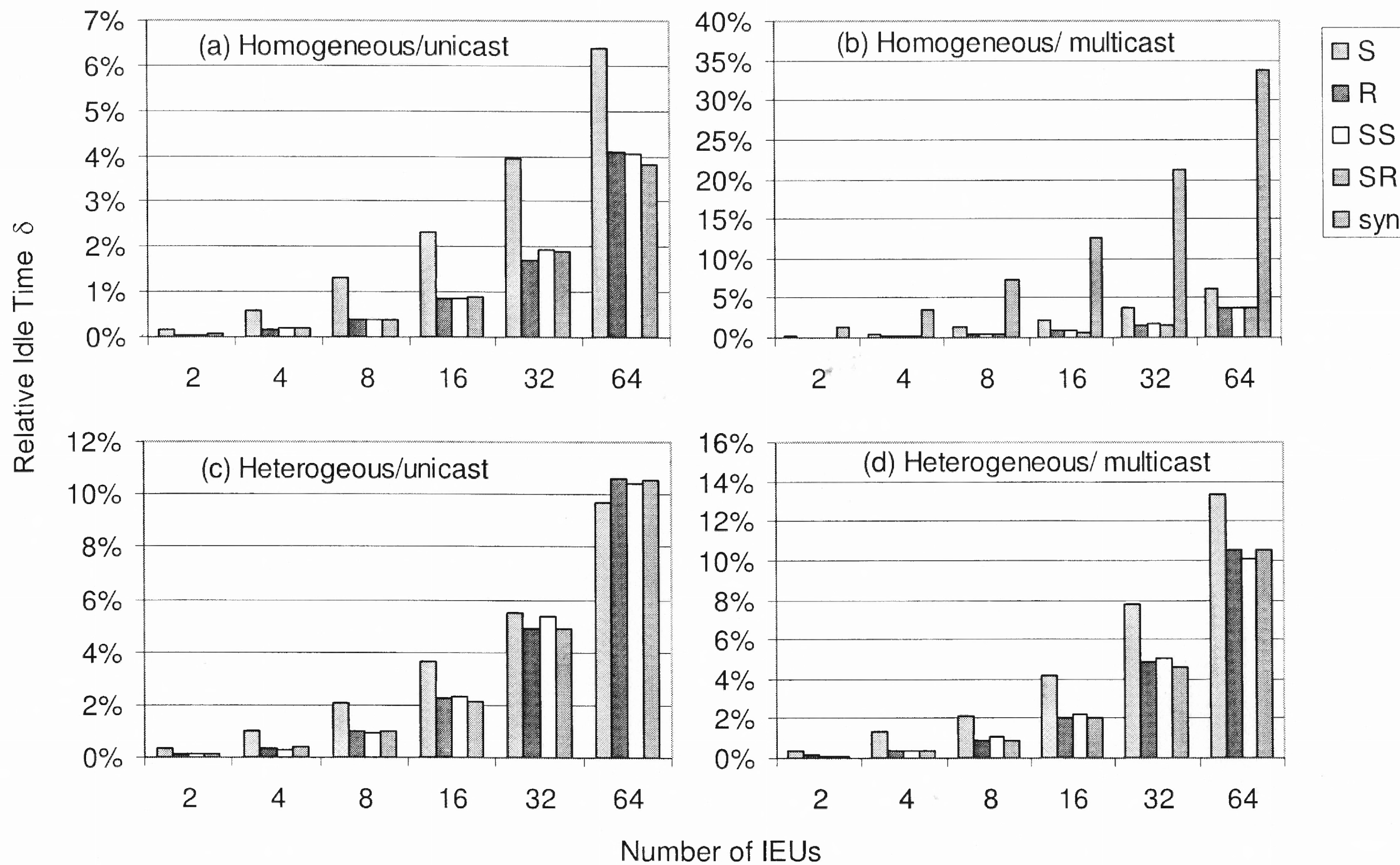


Figure 6.1 Simulation results of imbalance overhead of the MM SP

From Fig. 6.1(b), we can see that the synchronous policy introduces significantly higher imbalance overhead. This is because SIs, unlike instructions in microprocessors, have very irregular execution times. Thus, for high performance it is more favorable for PC clusters to issue SIs asynchronously. Also, dynamic scheduling algorithms with multicasting have less imbalance overhead than static scheduling algorithms independent of unicasting or multicasting for data communications in homogeneous [(a) and (b)] or heterogeneous environments [(c) and (d)]. The difference in the imbalance overhead among different (dynamic) scheduling strategies, however, is very small. The difference in the imbalance overhead between the static and dynamic schedules depends on the average number of tasks executed on a node. The experiments give similar results (Fig. 6.5). More tasks a node executes (i.e. fewer nodes in the cluster), the larger the difference. Thus, for sparse MM, in the case where there exists diverse workload between updating different sub-matrix blocks, especially for a large sparse MM, the asynchronous programming models are more appropriate than synchronous programming models. For large problems, dynamic strategies have an advantage in reducing the imbalance overhead.

Comparing the results [(a) and (b)] in a homogeneous environment with their counterparts [(c) and (d)] in a heterogeneous environment in Fig. 6.1, we can see that the imbalance overhead increases for the latter. This is because in a heterogeneous environment the nodes have different peak performance. If two nodes begin to execute a new SI at same time, the slower one will finish its execution later. Therefore slower nodes in a cluster have better chance to run the last completed SI of an SF. Other nodes with higher peak performance may be idle at the same time, thus wasting more computing capability. The increase in dynamic scheduling policies is larger. This makes the advantage of dynamic scheduling policies to diminish.

Table 6.1 Name and properties of synthetic databases used.

Name of Database	The number of transactions	The average number of items	The average length of the maximal pattern
T25.I10.D3K	3,000	25	10
T25.I10.D10K	10,000	25	10
T25.I10.D30K	30,000	25	10
T25.I10.D100K	100,000	25	10
T25.I10.D500K	500,000	25	10

6.3.2 Experiment of Mining Association Rule

The second evaluation of the effect of the proposed automatic mechanism on load balance was experiment of executing the SPs for mining association rules. All the experiments were performed on a PC cluster with six nodes. Each node contains two AMD Athlon processors running at 1.2 GHz. Each node has 1GB of main memory, a 64K Level-1 cache and a 256K Level-2 cache. All the nodes are connected via an Ethernet switch. Each link has 100Mbps bandwidth. All the PCs run Red Hat 7.3. All nodes have the same view of the file system by sharing files via an NFS file server.

The input data for the experiment was a set of synthetic databases generated using the program provided in [123] with sizes from 2MB to 400MB. These generated databases and some parameters are listed in Table 6.1. These database are named as T25.I10.D3K, T25.I10.D10K, T25.I10.D30K, T25.I10.D100K, and T25.I10.D500K. The number following the letter "T", "I" or "D," is the average number of items per transaction, the average length of the maximal pattern, and the number of transactions, respectively. The number of distinct items N is 1000; the number of patterns is 10000.

Table 6.2 Design parameters of SDBs and VM's runtime for mining association rules

Parameter Name	Value
Maximum size of block of items	1000
Maximum size of block of transactions	1000
Maximum size of block of candidate patterns	1000
Maximum size of block of rules	1000
Maximum number of hosted data blocks	10000
Maximum number of cached data blocks	10000

A few special functions were added in the code that implements the runtime environment; they collect information at runtime about the utilization of individual PC nodes. When no SI is executed on a node, the node is considered to be idle. The information includes the total time elapsed, the time for the execution of sequential code, the total time for the execution of SIs in parallel and the total idle time. During the experiments, each member computer launches a runtime environment process. The VM caches SDBs as much as possible by processes run on the host. The runtime environment also caches recently used SDBs. When an IEU needs some data that are not cached locally, it first tries to find them in its peers. The values of the relevant parameters of VM and SDBs are listed in Table 6.2. Number of records stored in each SDB is up to 1000. The size of the data cache on each virtual node is 10000. Each database is stored in a single file. Thus, loading them is a sequential process.

Table 6.3 Summary of execution and idle time of VM for executing the SP for mining association rules. (In milliseconds)

	Number of nodes	1	2	3	4	5	6
	Average idle time	191	3556	5101	7612.75	8881	10572
T25.I10.D3K	Idle time in computing*	191	1266.3	3799	6126	6441	7837
MinSupport = 0.6%	Total time	71170	63209	53217	54884	53295	51285
	Computing time	68757	59901	50775	52406	49483	47183
	Average idle time	440	5478	8178	9030	11049	11196
T25.I10.D10K	Idle time in computing*	440	2716	3864	4334	5400	5737
MinSupport = 0.6%	Total time	146062	103904	97505	85210	88830	86648
	Computing time	139947	96999	89417	77384	80003	78460
	Average idle time	217	9340	14090	14580	18544	17745
T25.I10.D30K	Idle time in computing*	217	2617	4397	3977	5714	4820
MinSupport = 0.6%	Total time	376079	231358	211133	158794	162268	157674
	Computing time	361751	214551	192959	141123	142221	138286

* excluding the sequential data load stage

Table 6.3 Summary of execution and idle time of VM for executing the SP for mining association rules. (Continue)

	Number of nodes	1	2	3	4	5	6
T25.I10.D100K MinSupport = 0.6%	Average idle time	306	19484	32092	37448	37305	49967
	Idle time in computing*	306	2696	8016	3438	8070	13158
	Total time	1234613	701748	515564	547617	377532	431875
	Computing time	1189756	656980	467411	487153	328808	372980
T25.I10.D500K MinSupport = 0.6%	Average idle time	1912	85293	134191	144945	182876	196851
	Idle time in computing*	1912	53622	78463	32467	90942	91268
	Total time	6883521	4825186	3081771	3263902	2325782	2585250
	Computing time	6652822	4571818	2747404	2664023	1866108	2078452

* excluding the sequential data load stage

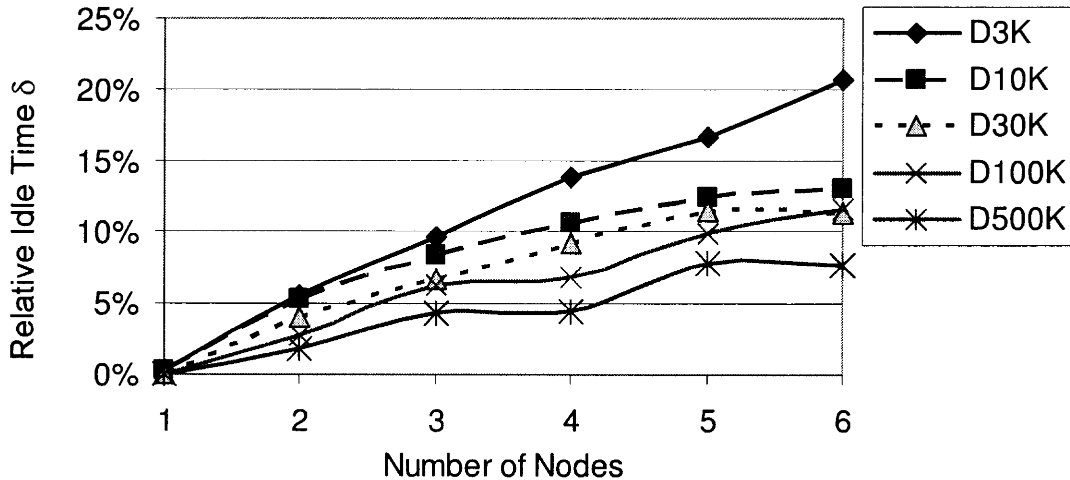


Figure 6.2 Percentage of average node idle time as a function of the total number of nodes (in execution of the SP of mining association rules)

Table 6.3 presents a summary of the total program running time, actual computation time, total idle time of nodes during actual computation and average idle time of nodes for each mining problem in our experiments. The times of computation (in parallel) were presented because the initial part of the super-program loads data sequentially. The average percentage of the idle time of IEUs during the execution of the entire SP and the parallel computing stage are shown in the Fig. 6.2 and 6.3, respectively.

Experimental results show that the percentage of idle time is not more than 8% if the problem size is large enough. We have also examined the effect of the SDB size on the SP for various SDB sizes. The results show that SDBs of size 1000 are a good choice for small- to medium-sized datasets. For very large data sets, the best SDB size may be larger but the improvement in performance is less than 10%. For the sake of comparing the effectiveness of load balancing, another SP was developed that implemented the HPA (Hash Partitioned Apriori) algorithm of [109].

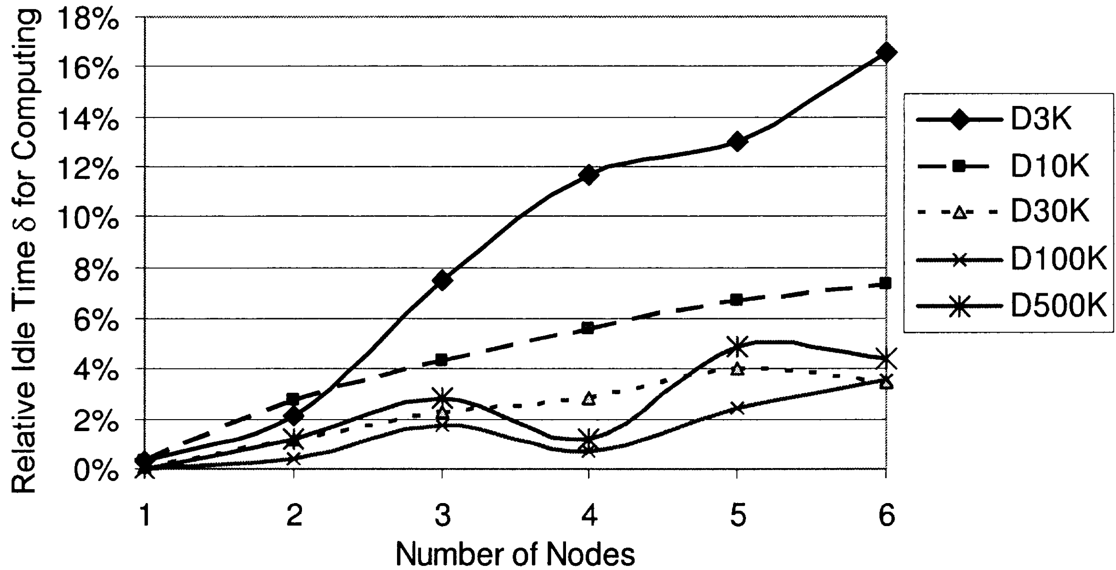


Figure 6.3 Percentage of average node idle time as a function of the total number of nodes (in the computing stage of the SP of mining association rules)

The results shown in Fig. 6.4 indicate that the load balancing mechanism employed in SPM is very competitive.

6.3.3 Experiment of Sparse Matrix Multiplication

The second evaluation of the effect of the proposed automatic mechanism on load balance was experiment of executing the SPs for mining association rules. All the experiments were performed on a PC cluster with eight dual-processor nodes. The random/dynamic scheduling approach was followed. Each node is equipped with two Athlon processors running at 1.2 GHz, 1GB of main memory, a 64K Level-1 cache and a 256K Level-2 cache. All the nodes are connected through a switch that forms an Ethernet LAN. Each link has 100Mbps bandwidth. All the PCs run Red Hat 9.0 and share the same view of the file system for files via an NFS file server.

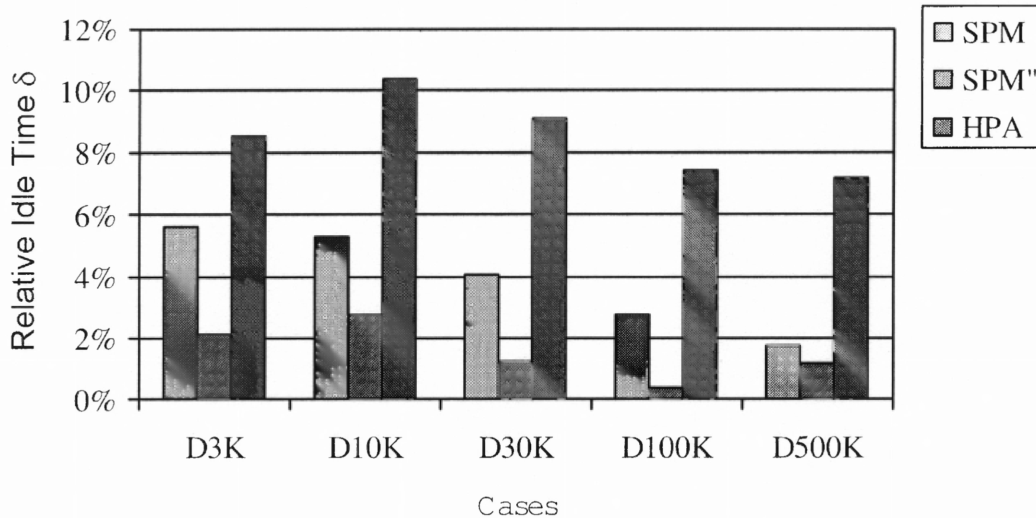


Figure 6.4 Comparison of the relative idle time for SPM and HPA.

A set of synthetic sparse matrices of size 8192×8192 were used; they are completely irregular with 5% non-zero elements. These matrices were partitioned into 32×32 sub-matrices of size 256×256. The super-program was developed manually using the SF described in Section 4.2. We implemented a runtime environment to support our SPM. In our runtime environment, there is a virtual IEU on each PC node. Recently used SDBs are cached in nodes. The runtime environment and the SIs are implemented in the Java language. To hide the long latency of loading data remotely, an IEU receives multiple SIs. They are executed in separate threads and are scheduled by the local operating system when their operands are locally available.

Experimental results of the imbalance overhead are shown in Fig. 6.5.

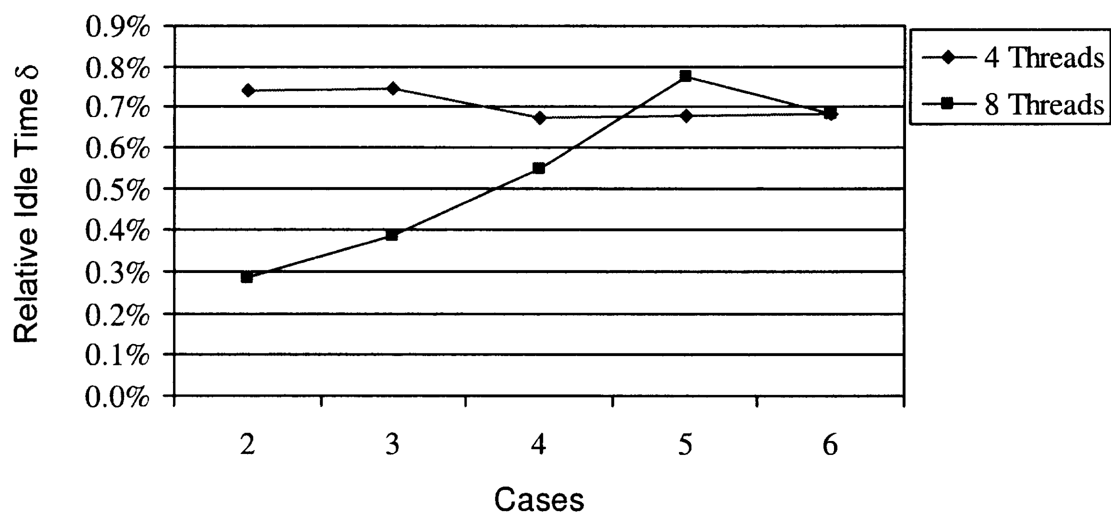


Figure 6.5 Relative idle time of IEUs in execution of the MM SP (experiment)

CHAPTER 7

COMMUNICATION OVERHEAD

Communication overhead is another part of overheads of programs developed in SPM. It is hidden in the services of the runtime system of SPM. In this chapter the communication overhead introduced by the runtime system are studied. The primary reasons of introducing the communication overheads are analyzed and the technologies to decrease the communication overhead are discussed and evaluated.

7.1 Communication Overheads for the SPM Runtime System

Communications on PC clusters are expensive and have few differences from communications for other types of parallel systems. The member nodes are fat PCs that contain many resources rather than simple processors. Besides the CPU, each node also has one or more network interface cards (NICs) containing auxiliary processors dedicated to communications among member computers. Therefore, the CPUs on these computers communicate with each other indirectly and many low level communication operations can be offloaded to the NICs.

Because of these existing auxiliary resources and the differences in the communication mechanism, the communication overheads on PC cluster systems are rather large. Since application programs execute on CPUs, not on NICs, the execution time of application programs depends heavily on the utilization of CPUs. Therefore, only the time consumed by CPUs is counted as overhead. Although the processors on NICs cannot be utilized to perform computations, they do help to reduce the communication overheads. They perform many low level communication operations simultaneously, while the CPUs perform computations dictated by the

application programs. In this situation, the communication time does not count as overhead. For PC clusters, the real communication overhead involves the CPU two cases: 1) consumed CPU time to support communication and 2) CPU waste time waiting for communications to complete. The first kind of communication overhead is instinct to communications. It includes CPU interaction with NICs and data formatting. The second kind of communication overhead comes from a mismatch of the data provisioning capacity with the computation capacity of the processor. When an SI is assigned to an IEU, if during the operand loading for the SI the IEU has no other task, then the CPU has to wait.

7.2 Technologies to Reduce Communication Overhead

7.2.1 Techniques to Reduce Type I Communication Overhead

The first type of communication overhead depends on the amount of communication operations that, in turn, depends on the application problems themselves, the algorithms adopted to develop application programs, the task distribution techniques, and the data access techniques. In SPM, RTS cannot change the first two factors; it cannot eliminate these overheads. The techniques used to reduce this kind of overhead need to decrease the demand for network communications. Because the communication patterns depend on the distribution of tasks and data, by controlling the ways of distributing tasks and data, the RTS may change the types and amounts of communication.

The primary instrument to reduce the first type of overheads is to translate some network communications to intra-process communications. Through a cooperation of the IDU and SFs, we can provide "communication-friendly" SIs. Assigning SIs that use the same data to the same IEU (if possible) makes communications between these SIs local, thus eliminating these overheads.

7.2.2 Techniques to Reduce Type II Communication Overhead

The second type of communication overhead is due to communication latencies. Before an SI is executed, its operands must be loaded into the local process. The corresponding member computer has to wait. A technique that can used to reduce this type of overhead is multithreading. Since an IEU has lots of resources, it can request multiple SIs that form a virtual pipeline and overlap the process of loading operands remotely for an SI with the execution of SIs having their operands available. The local operating system may schedule the execution of these SIs without a restricted order. Since no all SIs need to load data remotely, the operating system makes this arrangement more efficient. In this scheme, matching the time of loading data with the execution time of SIs is critical. Actually this criticality gave us the initial motivation to develop SPM with coarse grain operands. Cache techniques also help to reduce this type of communication overhead. Besides reducing the demand for communication, they also increase the ratio of ready SIs vs. SIs waiting for operands and make the overlapping of data loads with SI execution easier.

7.3 Model to Estimate the Communication Overhead of the SPM Runtime System

In this section, a theoretical model is developed to estimate the communication overhead of SPs. In this model, the execution of an SI consists of loading operand data and running a local routine. Assume that with probability pr the operands of an SI need to be loaded remotely; it actually depends on the data distribution among the nodes. Otherwise, the operand data are loaded locally.

7.3.1 System Condition and a Statistical Estimation of the Communication Overhead

We first build a simple statistical model that estimates the communication overhead based on the description above. This model assumes that the net CPU time of running the local procedure implementation of an SI is T_0. If the SI loads its operands locally, the runtime system consumes no time in preparing the operands of the SI. But if it loads operands remotely, the SI needs to wait to acquire the communication channel, to use the network resources to transfer data, and to use the CPU to control the communication and handle the transferred data. During the time waiting for the channel, it does not consume any resources. Usually the SIs running on different nodes compete for network resources to access remote data. The operations of loading data remotely affect each other. However, in this model, we assume that these SIs are executed independently. All the effects on each other are simply modeled as a waiting period. This is a system constant. The wait time depends on the number of nodes and it weakly depends on the average communication load when the system is not congested. Assume that the wait time of the SI that needs to load operands remotely is T_{delay}, the time using the network resources for the SI is T_{load} and the time using the CPU to handle the communication is $T_{load-cpu}$. The average span time to execute an SI is

$$T_{SI} = T_0 + p_r \times (T_{delay} + T_{load} + T_{load-cpu}) = T_{SI-cpu} + T_{SI-comm} + T_{SI-wait} \quad (7.1)$$

where T_0 is the actual time to execute the SI without overheads, $T_{SI-cpu} = T_0 + p_r \times T_{load-cpu}$, $T_{SI-comm} = p_r \times T_{load}$ and $T_{SI-wait} = p_r \times T_{delay}$

For a simple schedule, assume that each time only a single SI is assigned to a node. The efficiency of executing SIs is then $E_0 = T_0/T_{SI}$. Since the execution of

SIs breaks into three stages and each of them uses different resources, it is possible to assign more than one SI to a node and arrange their execution in a pipelined manner. This can increase the utilization of the communication resources and the CPU. Assume that at any moment there are n_T SIs on a node ($n_T > 2$) and that they are in an (n_T)-stage pseudo pipeline. In the first $n_T - 2$ s tages, the SIs just wait. The execution time of each stage is $T'_{SI-wait} = T_{SI-wait}/(n_T - 2)$. In the second to the last stage, the SIs take time TSI-comm to use the communication resources to load data. In the last stage, the SIs take time T_{SI-cpu} to run their local procedures. In this pipeline, the average time of executing an SI is the execution time of the longest stage, assuming that all stages are used all the time. It means that the effective execution time of each SI is

$$T'_{SI} = max\{T_{SI-cpu}, T_{SI-comm}, T'_{SI-wait}\} \tag{7.2}$$

The stage which determines the effective execution time of an SI depends on the system condition. If $T_{SI-cpu} > T_{SI-comm}$ and $T_{SI-cpu} > T'_{SI-wait}$, then $T'_{SI} = T_{SI-cpu}$. The execution time of the SI is determined by the CPU activities, which is in the *CPU bound* condition. If $T_{SI-comm} > T_{SI-cpu}$ and $T_{SI-comm} > T'_{SI-wait}$, then $T'_{SI} = T_{SI-comm}$. Then the execution time of the SI is determined by the consumption of communication resources. The system is in the *communication bound* condition. If $T'_{SI-wait} > T_{SI-cpu}$ and $T'_{SI-wait} > T_{SI-comm}$, then $T'_{SI} = T'_{SI-wait}$. The execution time of the SI is determined by the wait time. The system is in the *delay bound* condition. In the delay bound condition, neither recourses for computation nor recourses for communication can be utilized efficiently. This situation, however, can be avoided because $T'_{SI-wait} = T_{SI-wait}/(n_{SI} - 2)$ and $T_{SI-wait}$ only weakly depends on the average communication load. Thus designers could always prevent the system

from executing SIs in the delay bound condition by appropriately increasing n_{SI}. So, the *delay bound* condition is no longer discussed here.

Under the CPU bound condition, the effective execution time is $T_{SI}^{'} = T_{SI-comm}$. The efficiency of executing SIs is

$$E_{cpu} = \frac{T_0}{T_{SI-cpu}} = \frac{T_0}{(T_0 + p_r \times T_{load-cpu})} \tag{7.3}$$

And the relative communication overhead (under the CPU bound condition) is

$$w_{comm} = \frac{T_{SI-cpu} - T_0}{T_0} = \frac{p_r \times T_{load-cpu}}{T_0} \tag{7.4}$$

In the communication bound condition, the effective execution time is $T_{SI}^{'} = T_{SI-comm}$. The efficiency is

$$E_{cpu} = \frac{T_0}{T_{SI-comm}} = \frac{T_0}{(T_0 + p_r \times T_{load-cpu})} < E_{cpu} \tag{7.5}$$

And the relative communication overhead (under the communication bound condition)

$$w_{comm}^{'} = \frac{T_{SI-comm} - T_0}{T_0} = \frac{p_r \times T_{load-cpu}}{T_0} > w_{comm} \tag{7.6}$$

It is obvious that in the communication bound condition $T_{SI-comm} > T_{SI-cpu}$, the communication overhead is heavier, and the CPU resources cannot be utilized efficiently. Under the CPU bound condition, the system has the lightest communication overhead. Thus, we should always tune the design parameters so that the system works under this condition. In the following sections, if no explicitly stated, we assume system working under this condition.

The criterion of running in the CPU bound condition is $T_0 + p_r \cdot T_{load-cpu} > T_{SI-comm}$. If $T_0 > T_{SI-comm}$, then a super-program always works under the CPU bound condition no matter how low pr is. We call this case the *absolute CPU bound condition.* Similarly, if $T_0 + T_{load-cpu} < T_{SI-comm}$, a super-program always works under the communication bound condition. This is called the *absolute communication bound condition.* A CPU(communication) bound condition that is not absolute is called a *statistical CPU (communication) bound condition.*

7.3.2 The Effect of Loading Remote Data in the Burst Access Mode

In the previous model, we have always used the statistical average data as the property of each SI. It implies that the SIs that need to load data remotely are uniformly distributed. This may not be the case. To study the effect of the distribution of SIs that need to load data remotely, this section studies an extended model. In this model, the series of SIs executed on a node are form several groups. Each group includes two parts. In the first part, all SIs need to load data remotely; all SIs in the second part never need this. We assume that the number of SIs in each group is N_{SI} and $p_r N_{SI}$ is their number in the first part.

If $p_r N_{SI} > n_{SI}$, then at a particular moment all SIs on a node need to load data remotely. This is called a *burst condition (of remote loading data).* Otherwise, the node always has some SIs that do not need to load data remotely. If all SIs present in a node at the same time have no inter-dependencies, then these SIs could always fill the time period during which others wait for data. This case is called a *stable condition (of remote loading data).* In this stable condition, using statistical average data to represent each SI is valid.

Let us now discuss the burst case. Under the absolutely CPU bound condition, the total execution time is

$$T = p_r N_{SI}(T_0 + T_{load-cpu}) + (1 - p_r)N_{SI}T_0 = N_{SI}T + p_r N_{SI} T_{load-cpu} \tag{7.7}$$

The effective execution time of each SI is

$$T'_{SI} = \frac{T}{N_{SI}} = T_0 + p_r \cdot T_{load-cpu} \tag{7.8}$$

The efficiency is

$$E_{cpu} = \frac{N_{SI}T_0}{T} = \frac{T_0}{T'_{SI-cpu}} = \frac{T_0}{(T_0 + p_r T_{load-cpu})} \tag{7.9}$$

It is the same as that for the statistical description in Equation 7.3.

If $T_{SI-cpu} = T_0 + T_{load-cpu} < T_{SI-comm}$, then the total execution time is $T = p_r N_{SI} T_{load} + (1 - p_r) N_{SI} T_0$. The effective execution time of each SI is

$$\begin{aligned} T'_{SI} = \frac{T}{T_{SI}} &= p_r T_{load} + (1 - p_r)T_0 = T_{SI-comm} + (1 - p_r)T_0 \\ &> T_{SI-comm} \end{aligned} \tag{7.10}$$

The efficiency is

$$E = \frac{T_0}{T_{SI-comm} + (1 - p_r)T_0} < \frac{T_0}{T_{SI-comm}} \tag{7.11}$$

Thus, under the burst condition, the statistical description is valid for the absolute CPU bound condition. But in all the cases of the statistical CPU (communication) bound condition and absolute communication bound condition, the efficiency is lower than the expected value described in Equation 7.5 and the communication overhead is heavier.

7.3.3 The Effect of the Local Cache

Now, the factor of the data cache is added in the model. From Equation (7.4), we know that the communication overhead depends on the probability pr that SIs load operands remotely. If remotely loaded data are cached locally, they can be reused by many SIs. The capacity of the local cache for nodes can directly affect pr. Thus, it has a strong effect on the communication overhead.

Let us assume that the number of nodes in the cluster is n and the distribution of the original location of data is not correlated with the distribution of nodes on which the SIs are run. Then, the probability of referring to data that originally reside in a remote node is $p_r^{'} = (n-1)/n$. Assume that due to the local cache, the average times reusing the data loaded remotely is r. This is called the *data reuse rate.* Then, the probability with which SIs load operands remotely is

$$p_r = \frac{p_r^{'}}{r} = \frac{(n-1)}{n \times r} \tag{7.12}$$

Thus, the relative communication overhead is

$$w_{comm} = \frac{(n-1)}{n \times r} \times \frac{T_{load-cpu}}{T_0} \tag{7.13}$$

Let us now distinguish among three cases.

Case 1: Without the local cache, any SI that refers to data in a remote node has to load the operands remotely. The data reuse rate is r = 1 for any scheduling policy. Thus, the relative communication overhead is

$$w_{comm} = (1 - \frac{1}{n}) \times \frac{T_{load-cpu}}{T_0} \tag{7.14}$$

Case 2: Now we assume that every node can fully cache all the data it needs. Each node then loads any remote data only once and it can be held indefinitely. For the synchronous and static scheduling strategies of the matrix multiplication SPs, following the assumptions described in Section 4.2 each node needs to refer to q rows of distinct sub-matrix blocks of A and q columns of distinct sub-matrix blocks of B ($q = N/p$ and $n = p^2$). There is N blocks in each row/column. The total number of SIs executed by a node is $n_{SI} = N \cdot q \cdot q = N^3/n$. The number of block pairs is $N \cdot q$. The data reuse rate is r = q. The probability of loading data remotely is

$$p_r = \frac{n_{load}}{n_{SI}} = \frac{(1 - \frac{1}{n})}{q} = \frac{(1 - \frac{1}{n})}{N/p} = \frac{(1 - \frac{1}{n}) \times n^{1/2}}{N} \tag{7.15}$$

$T_{load-cpu}$ is the CPU time consumed for an SI that loads a pair of sub-matrix blocks. Then, the relative communication overhead is

$$w_{comm-syn}(= w_{comm-static}) = \frac{p_r \cdot T_{load-cpu}}{T_0} = \frac{(1 - \frac{1}{n}) \cdot n^{1/2}}{N} \times \frac{T_{load-cpu}}{T_0} \tag{7.16}$$

For the dynamic scheduling strategies, the sub-matrix blocks of the result C are dynamically assigned to nodes. The blocks assigned to a node are not longer located in a single rectangle but are dispersed. The data reuse rate r will then drop. Assume that the expected number of distinct rows/columns in them is q'. Then, $n_{load} = (1 - 1/n) \cdot N \cdot q'$ and

$$p_r = \frac{n_{load}}{n_{SI}} = \frac{(n-1) \cdot q'}{N^2} \tag{7.17}$$

For a random dynamic scheduling policy, when $n \ll N$ then $q^{'}$ converges to a constant N. When $N \ll n < N^2$, then $q^{'}$ converges to N^2/n. Thus,

$$p_r = \frac{n_{load}}{n_{SI}} = \frac{(n-1)\cdot q^{'}}{N^2} = \begin{cases} \frac{(n-1)}{N} & when \quad n \ll N \\ (n-1)n & when \quad N \ll n \ll N^2 \end{cases} \qquad (7.18)$$

For the smart dynamic scheduling policies, the system tries to align the sub-matrix blocks in the same row or column, if possible. Thus, the expected number of relevant rows/columns should be lower than that for the random dynamic scheduling policy. So there is a lower pr and w_{comm}. This is the reason for proposing the strategy.

Case 3: When the system provides some local cache but it cannot use a full cache to hold all the remote data, then the data reuse rate will drop from the above expected value. The value will depend on the cache policy and the implementation details of scheduling.

7.3.4 The Effect of Multicasting on the Communication Overhead

In the above discussion we have assumed that data exchanges in a unicast manner. Remote data are loaded on demand. This is a simple imitation of operand fetching in CPUs. As the member nodes in clusters (PCs) have more powerful high-level communication mechanisms, they can also broadcast and multicast. This section discusses the effect of using these channels for data communication.

The main benefit of broadcast/multicast to many nodes comes from merging multiple unicast communication operations into a single multicast operation. When multiple SIs almost simultaneously need to load remote data, a multicast of the data may reduce the communication cost and decrease the overall communication overhead.

Now let us modify the above model to analyze the effect of data multicast. In the above model the effective cost of a unicast load is $T_{load-cpu}$. Actually, it includes two parts. One is the CPU time of the sender $T_{send-cpu}$ and the other is the CPU time of the receiver $T_{receive-cpu}$. A broadcast/multicast has more than one receiver. Thus, its cost will be greater than that of a unicast operation. Assume a multicast is performed for a group of nm nodes. A broadcast is a special multicast where the group includes all nodes in the system. The total cost of a multicast is

$$C_{multi} = T_{send-cpu} + n_m \cdot T_{receive-cpu} = T_{load-cpu} + (n_m - 1) \cdot T_{receive-cpu} \qquad (7.19)$$

We assume that when the first member node needs the remote data, the data will be multicast to a group. All member nodes will receive the data. If we consider a combination of the multicast and the local cache, we can further assume that the data are cached and aged based on the local cache policy. For a node i in the group (i $= 1$ to n_m), the data reuse rate is r_i. The total data reuse rate of the system is $r = \sum r_i$. Then, the effective communication cost of an SI that refers to remote data is $c_{multi} = C_{multi}/r$. The relative communication overhead is

$$w_{multi-comm} = \frac{(1 - \frac{1}{n})}{r} \times \frac{T_{load-cpu} + (n_m - 1) \cdot T_{receive-cpu}}{T_0} \qquad (7.20)$$

It is clear that most receiver nodes receive data not based on their own demands, except for the node that requests the multicast. Thus, this approach may reduce the reuse rate of data for these nodes. The communication overhead will depend on the synchronous characteristics of the consumed data. Let us now distinguish between two cases.

Case 1: If there is no cache, then $r_i = 0$ or 1. If nu nodes in the group consume the data synchronously and others do not, then $r = n_u$ and

$$w_{multi-comm} = \frac{(1 - \frac{1}{n})}{n_u} \times \frac{T_{load-cpu} + (n_m - 1) \cdot T_{receive-cpu}}{T_0} \quad (7.21)$$

Comparing Equation (7.21) and (7.13), we have

$$w_{multi-comm} \leq w_{comm}$$

if and only if

$$n_u \geq 1 + (n_m - 1) \times \frac{T_{receive-cpu}}{T_{load-cpu}} \quad (7.22)$$

For the synchronous strategy of the matrix multiplication SPs described in Section 4.2, $n_u = n_m = p = n^{1/2}$. Also,

$$w_{multi-comm} = (1 - \frac{1}{n}) \cdot (T_{send-cpu}/n_u + T_{receive-cpu})/T_0$$

and

$$w_{comm} - w_{multi-comm} = (1 - \frac{1}{n}) \cdot (1 - \frac{1}{n_u}) \cdot T_{send-cpu}/T_0 \approx (T_{send-cpu}/T_0)$$

Thus, the relative communication overhead drops to $T_{send-cpu}/T_0$. For the asynchronous strategy of the same SPs, $n_u = 1$ and $n_m = p = n^{1/2}$. Also,

$$w_{comm} - w_{multi-comm} = -(1 - \frac{1}{n}) \cdot (n_m - 1) \cdot (T_{receive-cpu}/T_0)$$

Thus, multicasting increases the relative communication overhead by about $n_m \cdot T_{receive-cpu}/T_0$. This may be prohibitively high so that multicasting can not be used in practice.

Case 2: With a cache, once the received data can be used in the future no matter who requested the data. The longer received data is cached, the higher the benefit. Receiving data before it is needed is equivalent to data pre-fetching. The key in reducing the communication overhead is to choose a group for the multicast. For a static scheduling strategy, it is easy. For the synchronous strategy described above, the nodes which process the blocks of C on the same rows form groups to multicast sub-matrix blocks of A; the nodes which process the blocks of C on the same columns form groups to multicast sub-matrix blocks of B. The effect is $n_u = n_m = p = n^{1/2}$ and r_i = q = N/p for i = 1 to n_m. Thus, $r = \sum r_i = n_u \cdot r_i = pq = N$ and

$$w_{multi-comm} = \frac{(1 - \frac{1}{n})}{n_u} \times \frac{T_{load-cpu} + (n^{1/2} - 1) \cdot T_{receive-cpu}}{T_0} \qquad (7.23)$$

These results show that the choice of the communication mechanism is not completely orthogonal with other design factors. For the efficient use of multicast, a synchronous strategy and a cache should complement each other. If nodes provide the capacity to fully cache all data, the designer can choose any one of them arbitrarily. In another case, if nodes do not provide the capability to cache data, the broadcasted data have to be consumed in a synchronous fashion. All SIs which use the same broadcast/multicast data should be issued at the same time to different nodes. This adds a strong limitation to scheduling and sacrifices valuable independency of SIs, so the design space of scheduling polices is shrunk. Under this condition, only the synchronous policy among the above five scheduling strategies of our matrix multiplication example can be used. Since both synchronization and the cache scheme have external cost, synchronization may increase the imbalance overhead. The cache increases the system cost. The requirement of reducing these costs form constraints in the design. These constraints make the design space become 3-dimensional and non-orthogonal.

It should be indicated that besides directly affecting the consumed CPU time for handling data communication, multicasting data also has an indirect effect on the overall performance of the executed programs. Multicast, similar to cache, may increase the data reuse rate because it may decrease the number of remotely loaded data which further reduces the total demand for network communication. Although under the CPU bound condition reducing the communication load may not affect the system performance, it may decrease $T_{SI-comm}$ and $T'_{SI-wait}$. The system may work better under the CPU bound condition compared to shitching to the communication bound condition. When the load of an Ethernet LAN is close to its saturation capacity, the efficiency of the communication network is sensitive to the communication load. In this case, reducing the communication load is really beneficial.

7.4 Evaluation of Communication Overheads in SPM

The effect of using the above techniques to reduce the communication overheads was evaluated by running a set of simulation programs that simulates the execution of MM SPs described in Chapter 4. During the simulation, the "simulator" performs the task of the IDU for scheduling SIs and keeps running until all SIs have been executed and all nodes are idle. The system maintains an abstract time progress and various overheads are counted based on the SIs assigned to IEUs. The workload of tasks and the static schedule were created manually in advance. Table 7.1 shows an example of static schedule for tasks on a heterogeneous PC cluster with 64 nodes. The numbers in the left column and top row are the indices of sub-matrix blocks to be calculated. The other numbers are the indices of the nodes that tasks are assigned to. For the SS strategy, the initial task assigned to each node is chosen to be the same as that for the static strategy.

Table 7.1 A static task schedule for a heterogeneous PC cluster with 64 nodes

	0-5	6-7	8-13	14-15	16-21	22-23	24-29	30-31
0-3	0	1	16	17	32	33	48	49
4-7	2	3	18	19	34	35	50	51
8-11	4	5	20	21	36	37	52	53
12-15	6	7	22	23	38	39	54	55
16-19	8	9	24	25	40	41	56	57
20-23	10	11	26	27	42	43	58	59
24-27	12	13	28	29	44	45	60	61
28-32	14	15	30	31	46	47	62	63

Both unicast and multicast communications were simulated. For multicasts, the membership of nodes in the multicast group is determined statically as follows for all strategies. In our simulation, the sub-matrix blocks of A and B are cached separately. The unit of cached data is a row of A and a column of B. The size of the local cache is the number of rows of A and columns of B cached. The size of the cache is chosen to be 1 to 32. The size of 32 means that every node can fully cache all data it needs.

Simulations to count the communication overhead of MM SPs executing on homogeneous and heterogeneous clusters with a full cache are shown in Fig. 7.1 and 7.2, respectively. The comparison of the communication overheads for different communication schemes and cache sizes are shown in Fig. 7.3 and 7.4 respectively. In these figures, Syn, S, R, SS and SR represent the synchronous policy, the static scheduling policy, the random scheduling policy, the smart scheduling policy with a static initial assignment and the smart scheduling policy with a random initial assignment, respectively. HU/HM represents the case of using unicast/multicast

communication in a homogeneous environment; RU/RM represents the case of using unicast/multicast communication in a heterogeneous environment. The relative communication overhead χ is expressed in the percentage of CPU time used in communications.

From the figure 7.1 it is observed that Syn always has the lowest communication overhead. S using multicasts has almost the same level of communication overhead in most cases, except for a cluster with only two nodes. However, in a heterogeneous cluster multicasting loses its advantage. S using **unicasts** has the lowest communication overhead while SS using unicasts has a little higher. When the number of nodes is more than 16, the overhead for SS using unicasts is even less than that for the static strategy using multicasts. This means that in heterogeneous environments, unicast communications reduce the communication overhead. The significant difference in communication overheads between SS and R indicates that for dynamic scheduling strategies appropriate optimization is necessary.

From Fig. 7.4 we can find that the communication overhead decreases when increasing the cache size, especially when the cache size is small so that an IEU cannot cache all SDBs needed. In the non-full cache cases, the results show that unlike the full cache case, both the S and SS strategies using unicasts have lower communication overhead than the corresponding strategies using multicasts. This indicates that, even in a homogeneous environment unicasts are better than multicasts. SS using unicast communication has the lowest communication overhead. The other obvious feature is that when increasing the cache size, the changes in the communication overhead of SS are more significant than those for the S strategy. No matter if using unicasts or multicasts, the communication overhead for the SS strategy drops faster when increasing the cache size. This indicates that SS is more flexible. It can utilize the cache space very well. Compared with the result in Fig. 7.1, in this case, SS

is a better choice. For large embedded MM problems, in the general case there is no-sufficient cache space available. Combining the advantages of SS and uncasts is the best choice.

To Summary, since the SS is the best choice in many cases, utilizing an IDU-SF cooperation really helps to reduce the communication overheads. Also, the utilization of cache techniques is really useful in reducing communication overheads even further.

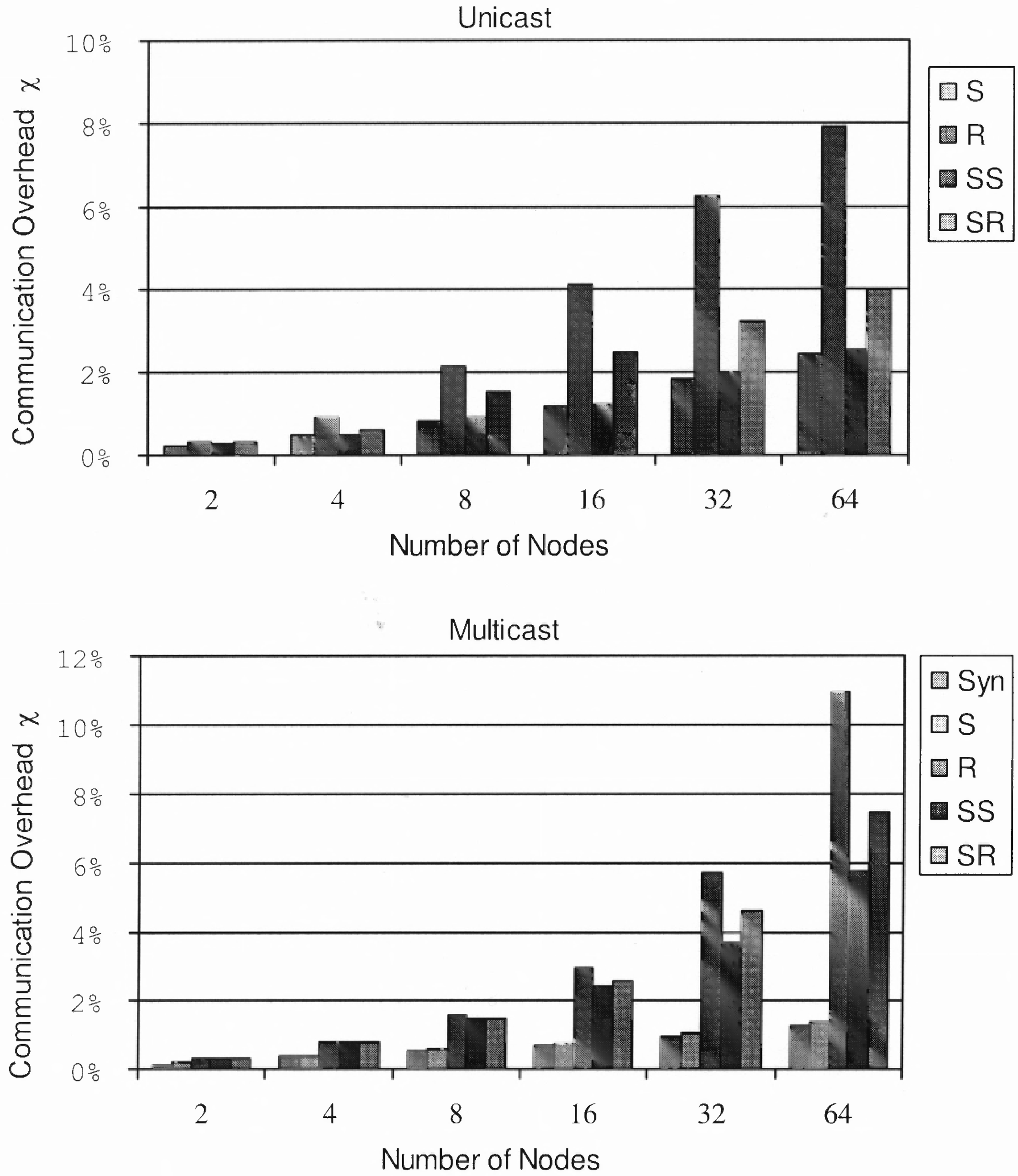


Figure 7.1 Simulation results of communication overhead in a homogeneous environment

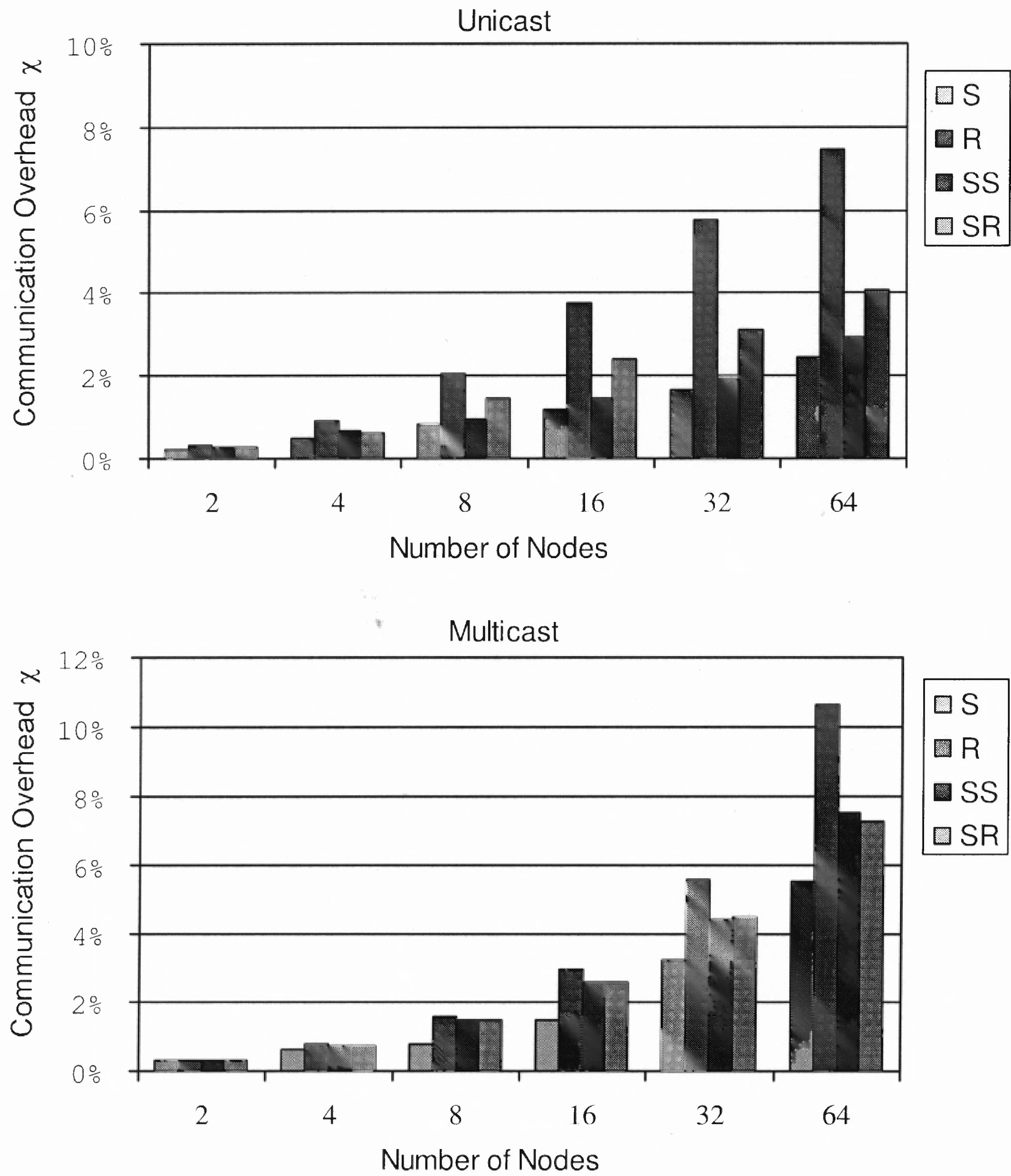


Figure 7.2 Simulation results of communication overhead in a heterogeneous environment

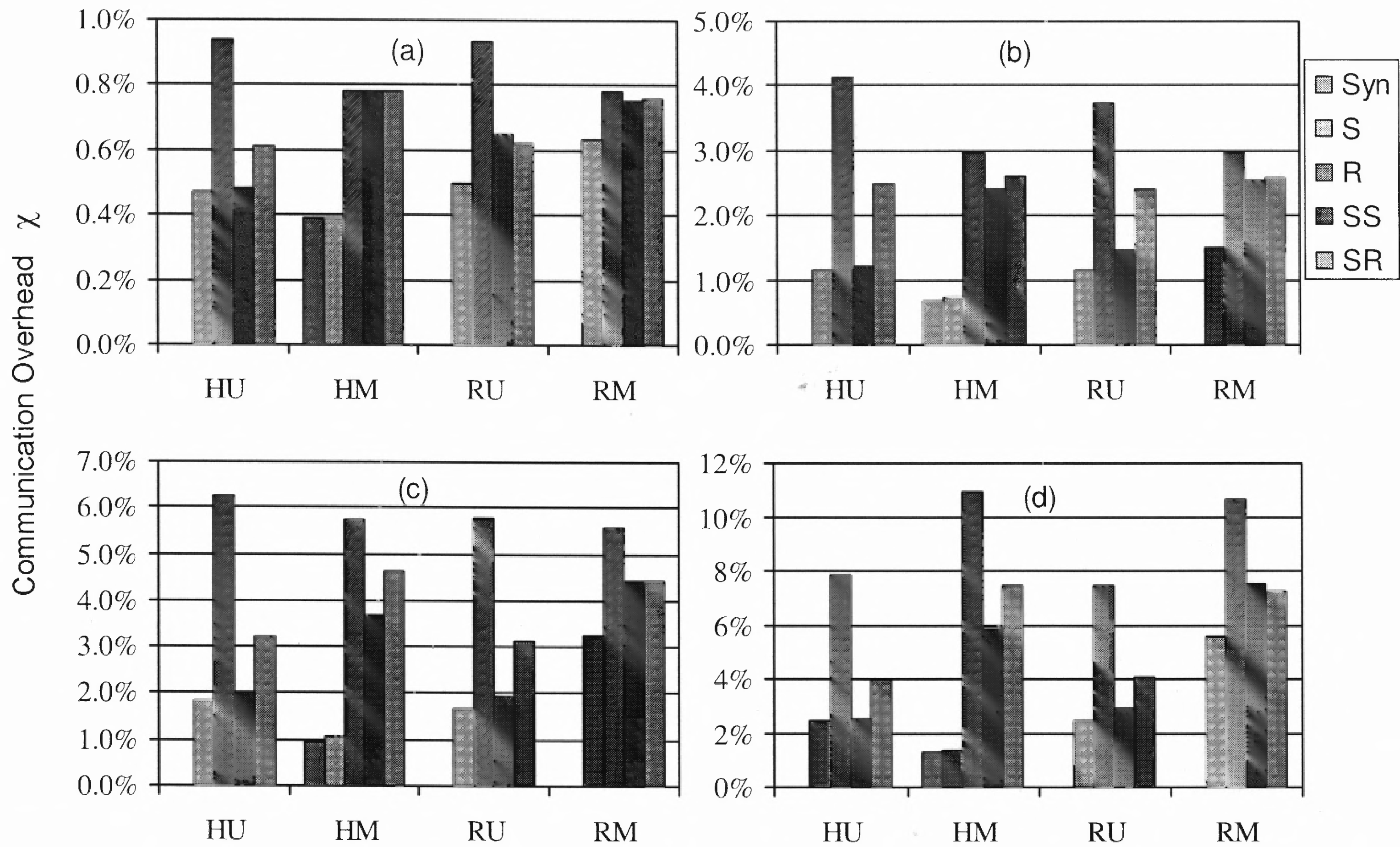


Figure 7.3 Comparison of communication overheads under different strategies. Number of Nodes is (a) 4 (b) 16 (c) 32 (d) 64

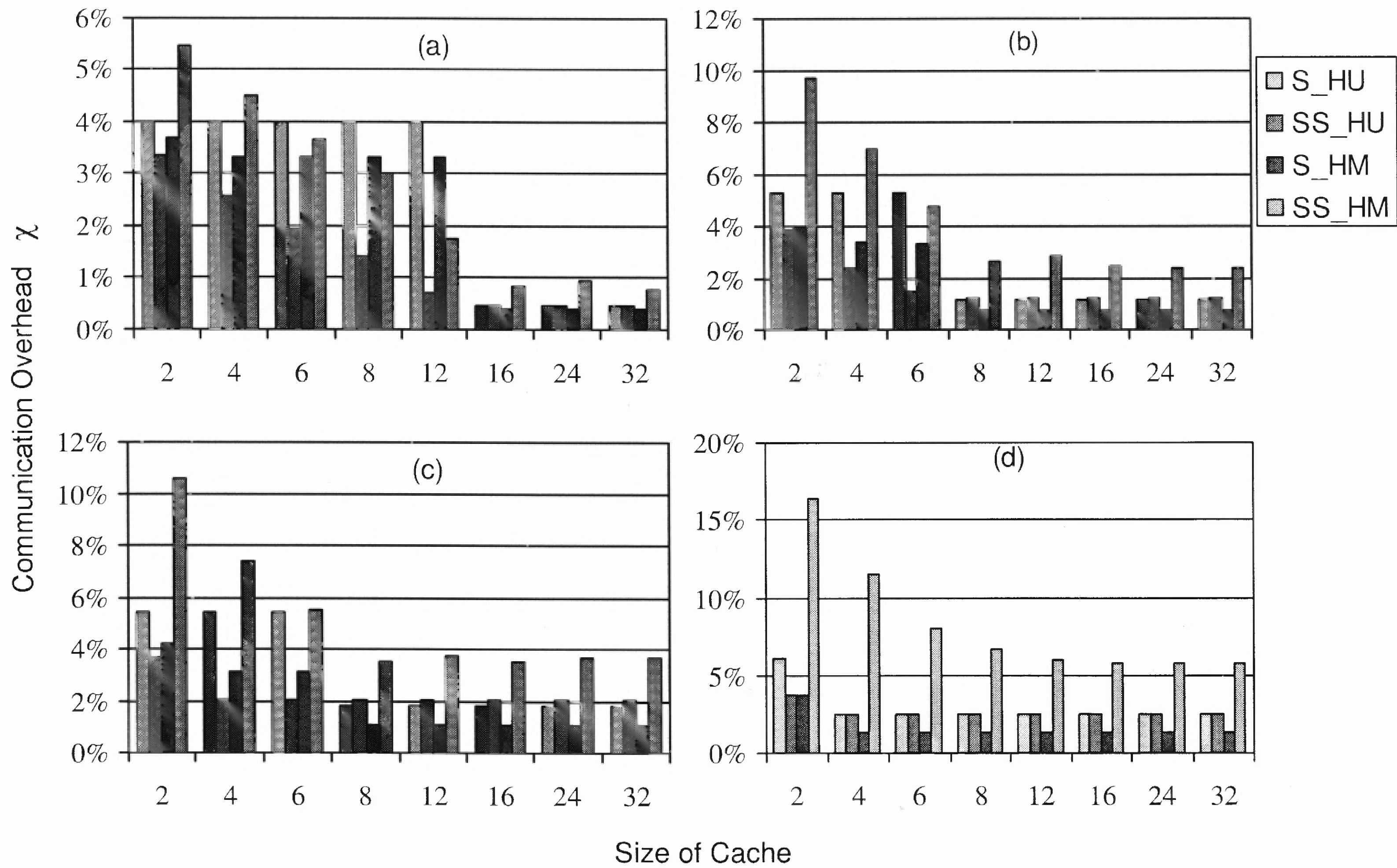


Figure 7.4 Communication overheads under different strategies in a homogeneous environment for different cache size. Number of Nodes is (a) 4 (b) 16 (c) 32 (d) 64

CHAPTER 8

PARALLEL DATA VIRTUALIZATION AND DATA ACCESSES

The parallelism in programs can benefit from support for parallel data accesses. In this chapter, the means for data representation in SPM are described and their impact on parallel data accesses is discussed.

8.1 Parallel Data Accesses for the Parallelization of Programs

Data dependences are one of the most important factors that affect performance. Some data dependences are associated with intrinsic properties of the problem and the algorithm used to solve the problem (i.e., they are true dependences). Others are fake data dependences introduced by the programming model or the implementation of the underlying computer system. WAW (write after write) and WAR (write after read) in stream pipelining of executing instructions are two well known examples of such restrictions [15]; they stall subsequent instructions and may destroy temporal parallelism. Alleviating these restrictions in data is the main responsibility of the designers of programming models and the developers of computer systems.

For parallel programs, this problem appears in multithreading when accessing shared data. Data sharing among multiple processes/threads is essential in parallel programs [70]. For most of the parallel programs, multiple threads may need to access the same data at the same or different times. If multiple threads need to access the same data simultaneously but the underlying system storing the data does not support such concurrent accesses, some threads must be stalled [124]; i.e., the way of representing data in the system may restrict the parallelization of programs

8.2 Data Virtualization and Parallel Data Accesses

SPM addresses this issue through the virtualization of application data [125]. In SPM, SDBs are only data entities used by the programmers of SPs. They are pure logical entities. They exist in a global logical space for each application and are managed by RTS. At runtime, RTS provides local data representations of SDBs for executing SIs. The representations of an SDB may be a set of incarnated data objects and/or external data files stored in external storage instead of one incarnated object. These incarnated data objects are distributed among IEUs. Accessing an SDB is achieved through interfaces of its incarnated data objects either locally or remotely. Therefore, concurrent accesses can be supported easily.

For data coherence, data accesses are controlled by the runtime support system. We define a set of data states. RTS maintains the states of all SDBs and grants access privileges for particular data based on its current state. A state diagram for data is shown in Fig. 8.1. The change of state is triggered by various events, such as an SI's issue or commitment. The difference between *updates* and *distributed-updates* is that the system guarantees the latter access a local incarnated object. In SPM, besides reads and writes, SDBs can be accessed simultaneously for *remote-update* and *distributed-update* operations. Programmers can easily develop parallel programs.

8.3 Evaluation of SPM in Accessing Parallel Data

To demonstrate the effect of our approach, a set of experiments were set up. The MM SPs described in Chapter 4 were chosen. To study the effect of distributed data representation and parallel data accesses on the performance, several versions of the SP were created. The difference among them lies only in the way that SIs update blocks in the result. In the version "SimpleUpdate," SIs only use "write" to access resulting SDBs. In the version "RemoteUpdate," SIs only use

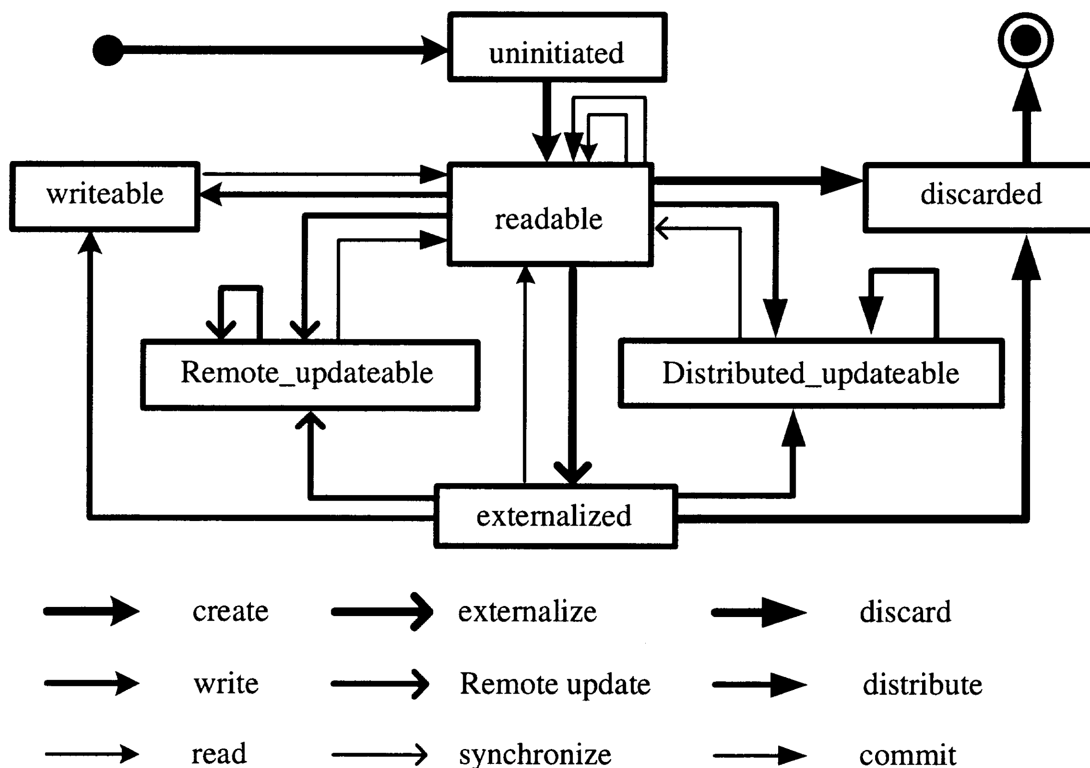


Figure 8.1 State diagram for logical data entities

"update" to access resulting SDBs. In the version "DistributeUpdate," SIs only use "distributed_update" to access resulting SDBs. In the version "Mixture," SIs use "write" until the last SDB in the resulting matrix begins to be computed; then the reminding SIs use "distributed_update." Two scheduling policies R and SS, described in Chapter 4, were used for distributing SIs. R was used for all SPs; SS was used only for "SimpleUpdate" and "Mixture."

In the experiments, a set of synthetic sparse matrices with 12.5% non-zero elements were used. The properties of the matrixes are listed in Table 8.1. The SP versions and the parameters of the input matrices are listed in Table 8.2. In Table 8.2 S, R, D and M represent the "SimpleUpdate," "RemoteUpdate,"

Table 8.1 Properties of the data matrices

Matrix name	Sizes	Number of block	SDB size
M11	4096 × 4096	32 × 32	128 × 128
M12	1024 × 16384	8 × 128	128 × 128
M13	16384 × 1024	128 × 8	128 × 128
M22	4096 × 4096	16 × 16	256 × 256
M23	1024 × 16384	4 × 64	256 × 256
M24	16384 × 1024	64 × 4	256 × 256
M25	2048 × 32768	8 × 128	256 × 256
M26	32768 × 2048	128 × 8	256 × 256

Table 8.2 Experimental configurations and data

Case Name	Operands	Threads limit	SP version
256r1	M25 × M26	1, 4, 6, 8	S, R, D, M, G, N
256r2	M23 × M24	6	S, R, D
128r	M12 × M13	6	S, R, D
256s	M22 × M22	6	S, R, D
128s	M11 × M11	6	S, R, D

"DistributedUpdate" and "Mixture" SPs using the basic policy (R), respectively. G and N represent "SimpleUpdate" and "Mixture" SPs using the SS policy, respectively. The operands refer to the matrices listed in Table 8.1.

The performance of SPs relative to "SimpleUpdate" is shown in Fig. 8.2. The average number of threads is shown in Fig. 8.3. It represents the degree of parallelism in the SPs.

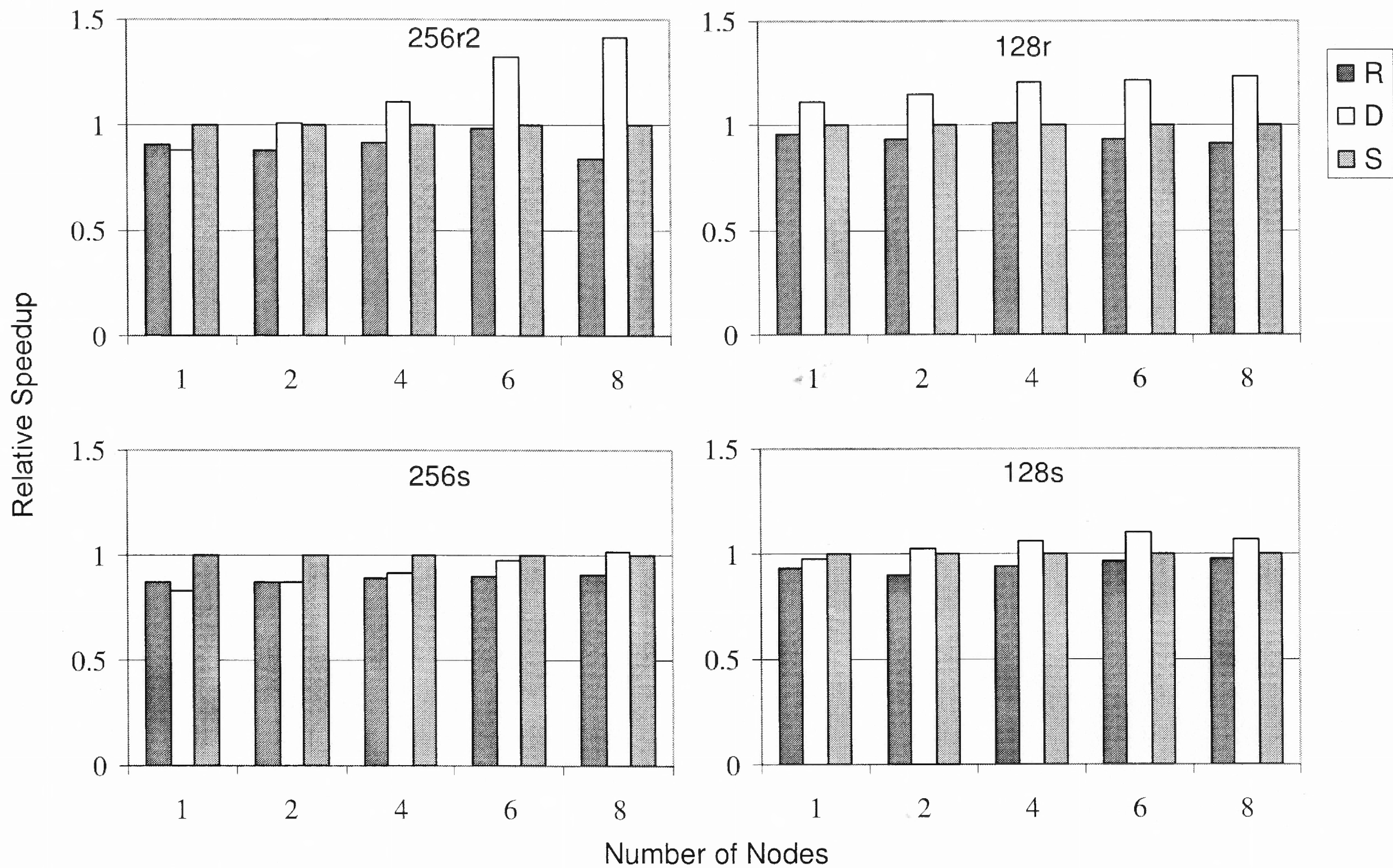


Figure 8.2 Relative performances of the three SPs using different data writing methods

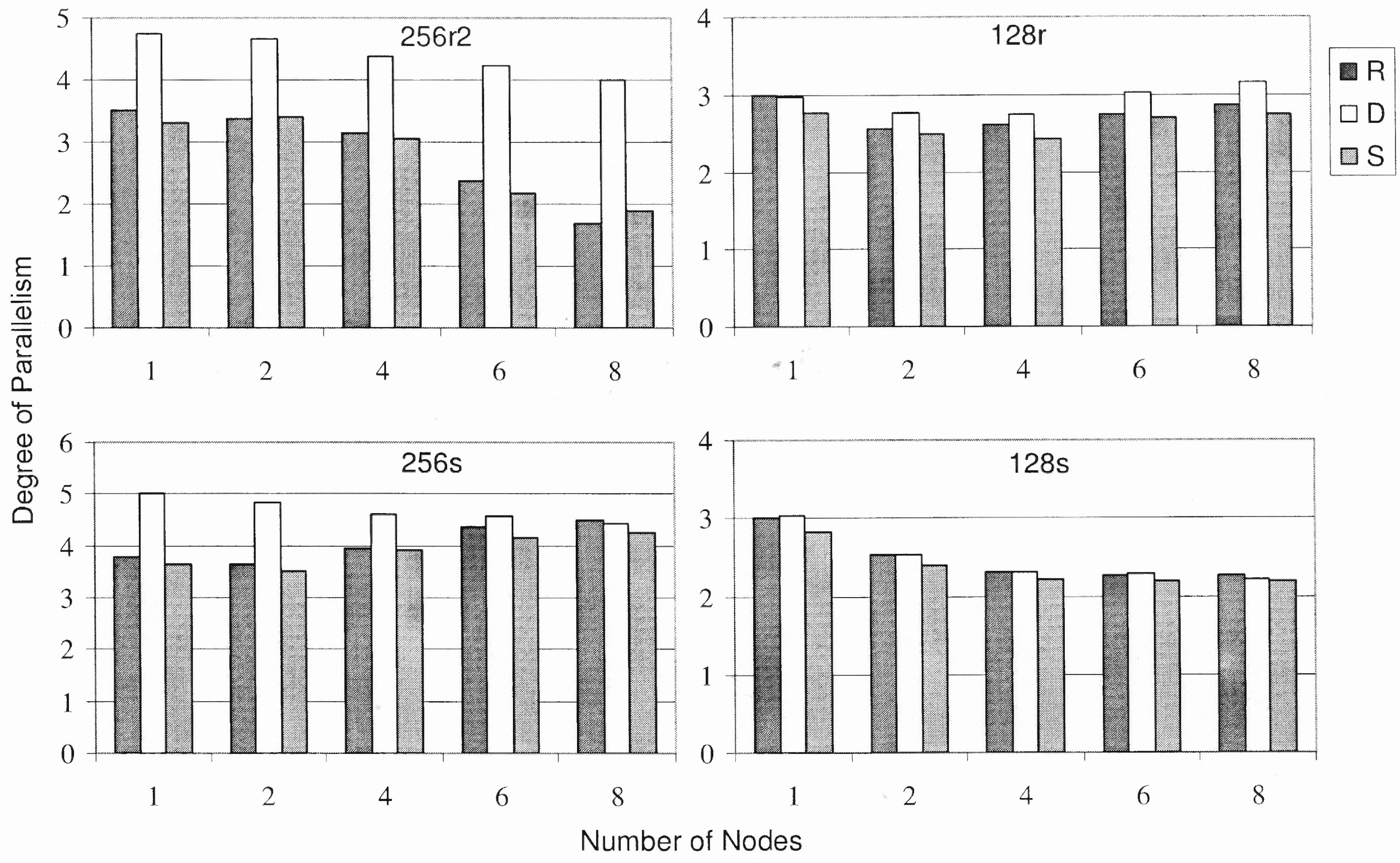


Figure 8.3 The effect of the data access methods on the degree of parallelism

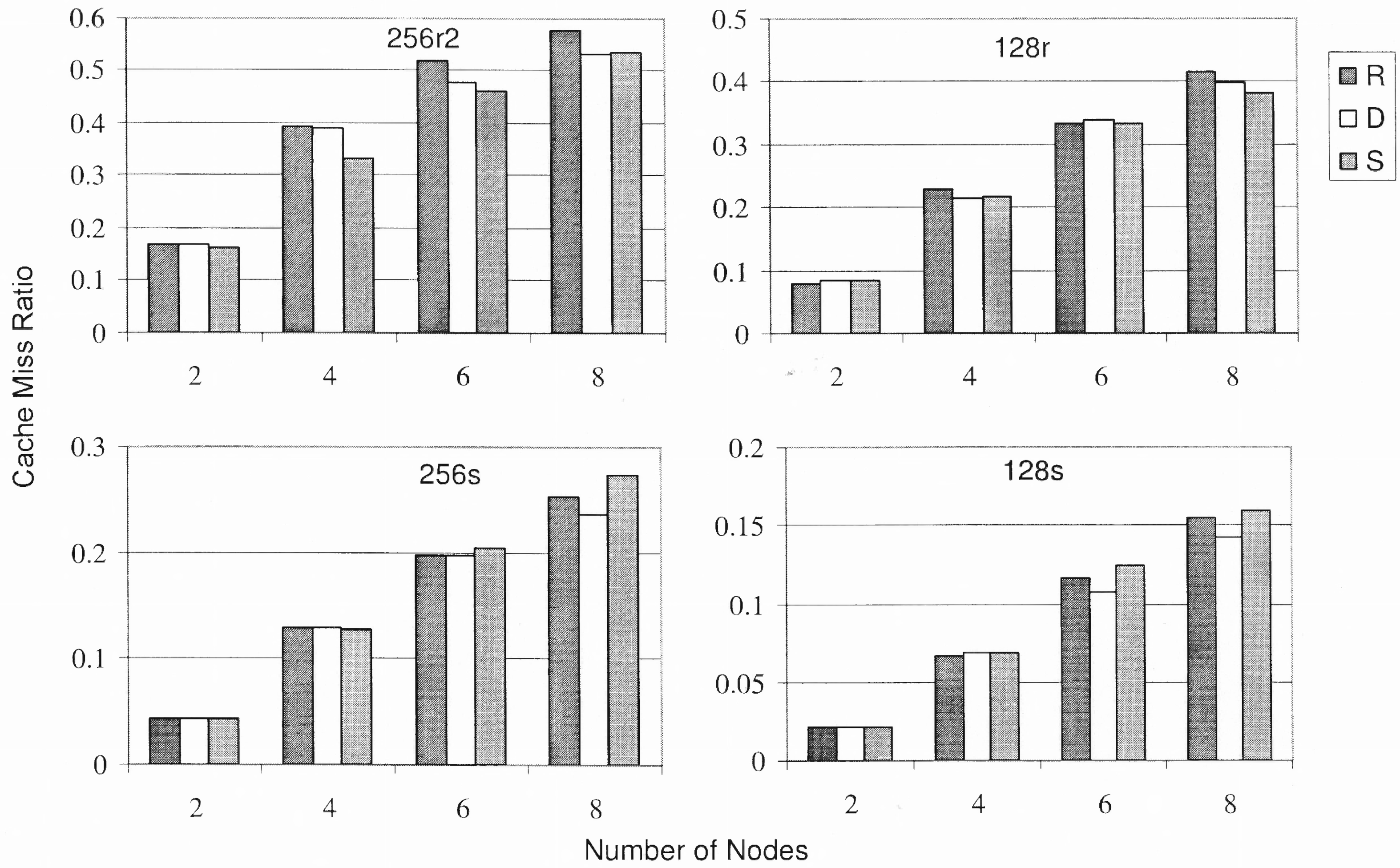


Figure 8.4 The local cache-miss ratio of data access

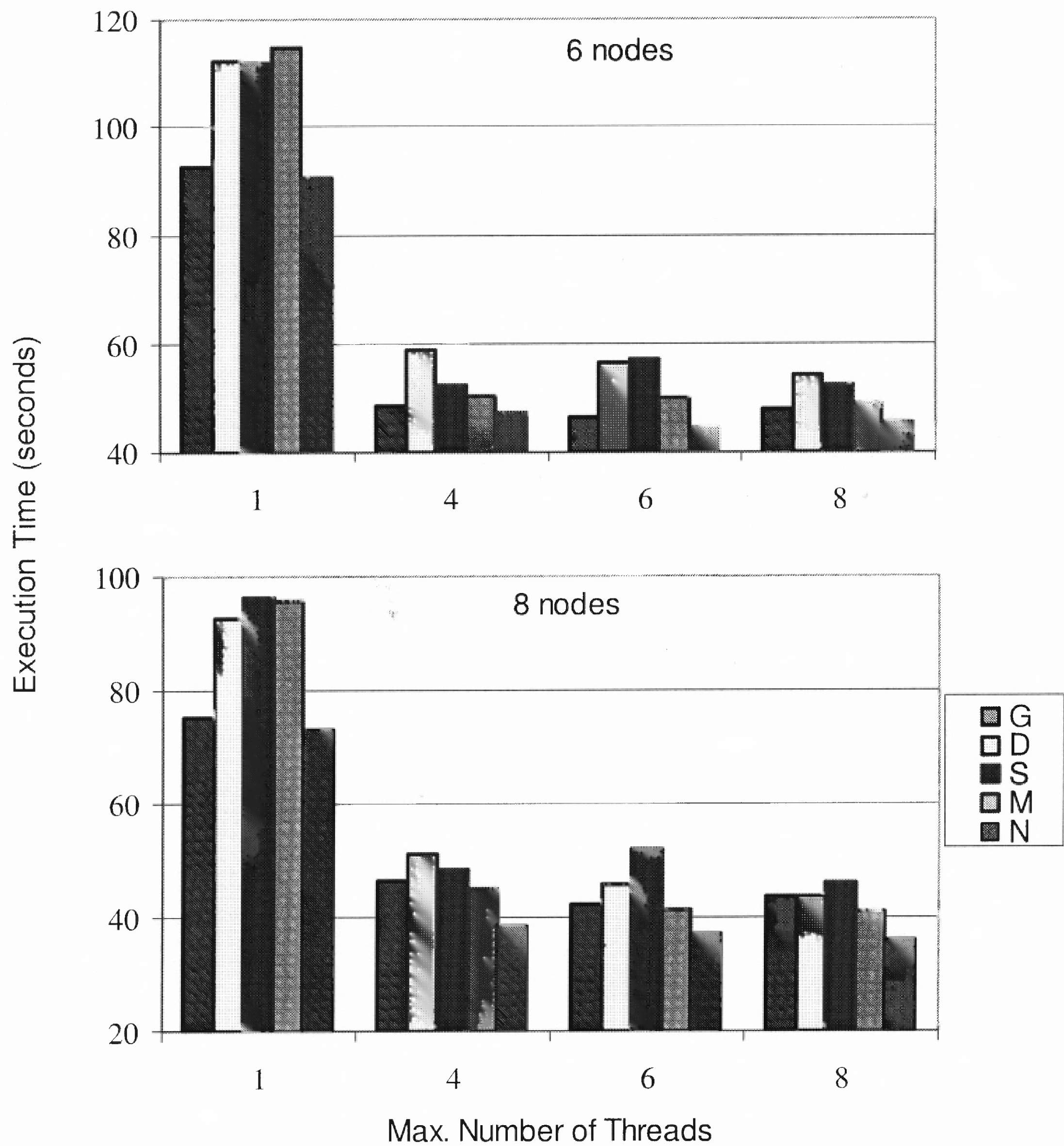


Figure 8.5 The effect of multithreading on the execution time of the SPs for 256r1

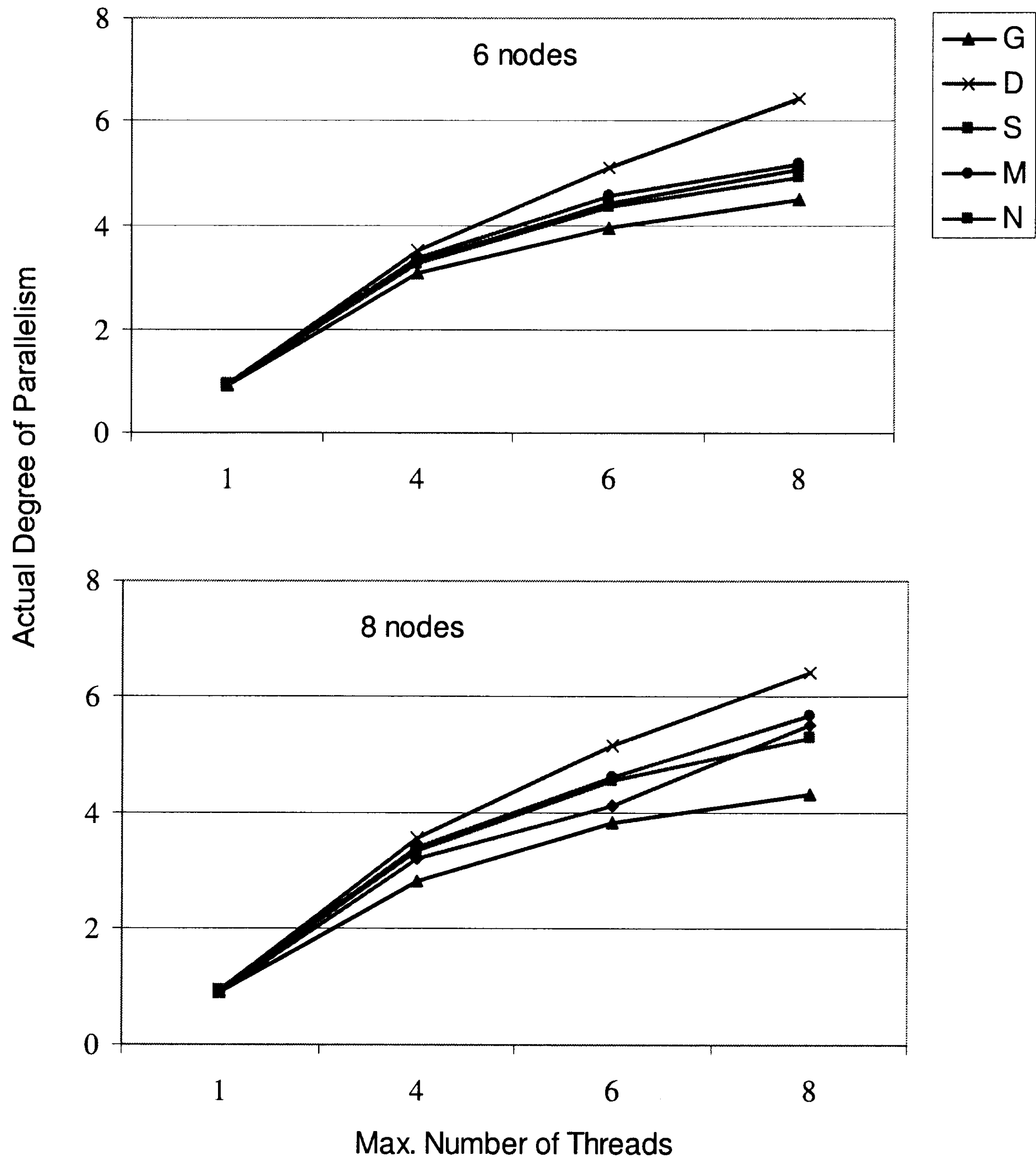


Figure 8.6 The effect of the limit of threads per node on the actual degree of parallelism for 256r1

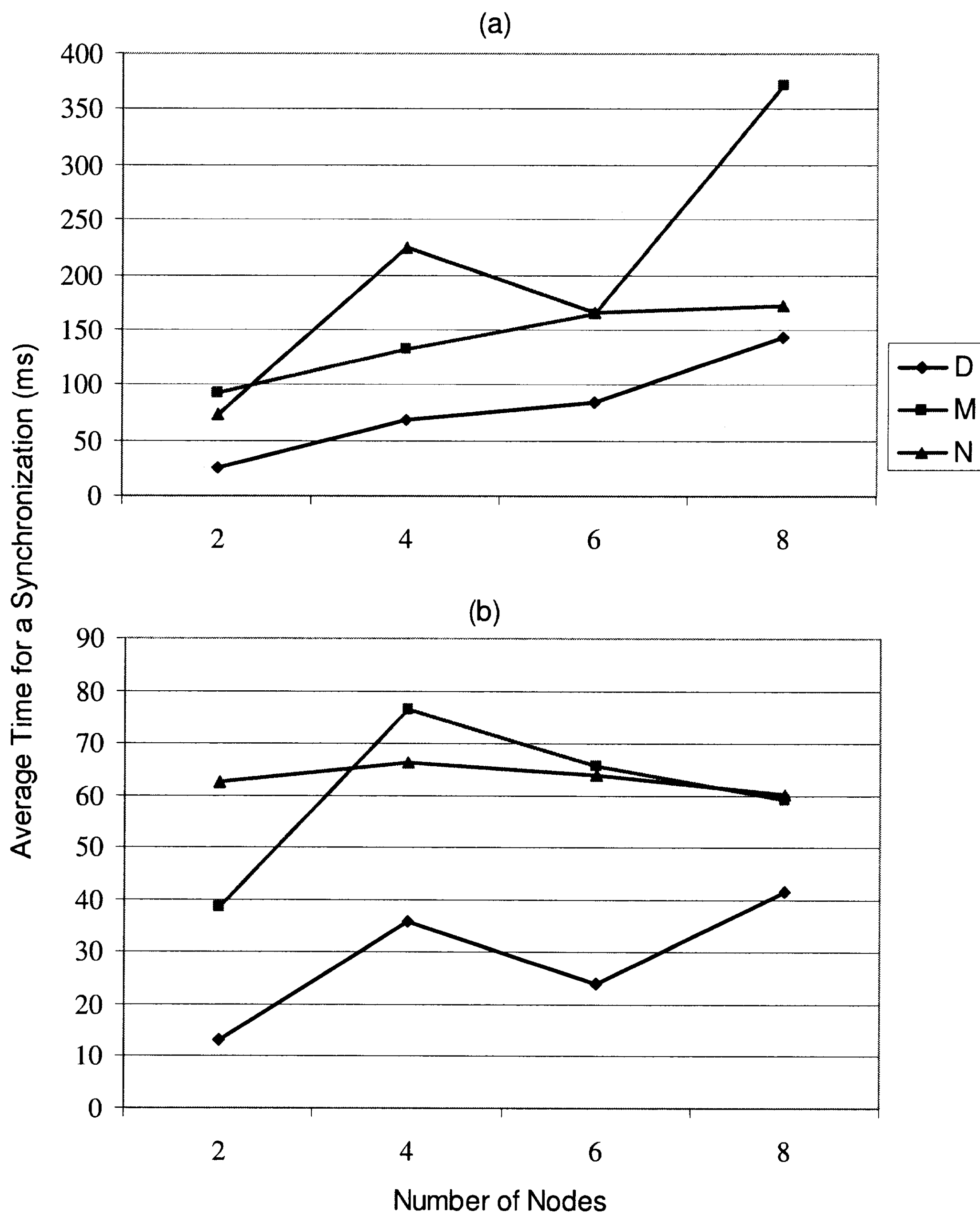


Figure 8.7 Average time for synchronization of a sub-matrix block (a) 256×256 blocks (b) 128×128 blocks

The local cache-miss ratios during the execution of SPs are shown in Fig. 8.4.

In the experiments, 1, 2, 4, 6 or 8 nodes were employed. The maximum number of SIs assigned simultaneously to an IEU is six with an exception for the case of 256r1 that changes the limit from one to eight to confirm that this choice is reasonable. The average net CPU time to execute an SI that multiplies a pair of SDBs was measured with a separate pilot program; it is 1.0ms and 18.5ms for SDBs of size 128×128 and 256×256 (with 12.5% non-zero elements), respectively.

For the case of 256r1, the number of threads is changed from one to eight. The execution time of these SPs is shown in Fig. 8.5 and the actual degree of parallelism is shown in Fig. 8.6. The average time for synchronization with an SDB under various conditions is shown in Fig. 8.7.

8.4 Performance Analysis

From Fig. 8.2 and Fig. 8.3, we can see that there is correlation among the data access method, the degree of parallelism and the performance of SPs. The long communication latency has an adverse effect on the performance in PC clusters. SPM uses multithreading to hide this latency and employs coarse-grain tasks (i.e., SIs) to decrease its effect [106, 119]. Since efficient multithreading requires enough parallelism, we must investigate the factors that affect it. Although assigning six SIs concurrently is allowed, the actual number of concurrent threads, shown in Fig. 8.3, is significantly lower. In the 128s and 128r cases, it is less than 3. One reason is distributing an SI needs some time. Another reason relates to the way that SPs exploit the intrinsic parallelism in the problem. In block-wise matrix multiplication, there are two types of parallelism. The first is inter-block parallelism ($P_{external}$) appearing when all SDBs in the resulting matrix are computed in parallel. The second is intra-block parallelism ($P_{internal}$) involving the multiplication of pairs of

Table 8.3 Intrinsic degree of parallelism in the experimental SP

Experiments	Inter-block parallelism	Intra-block parallelism
256r1	16	64
256r2	64	128
128r	64	128
256s	256	16
128s	1024	32

sub-matrix blocks for a resulting SDB. For $n \times k$ and $k \times m$ SDBs, these degrees of parallelism are equal to $n \cdot m$ and k, respectively. The $P_{external}$ and $P_{internal}$ parallelism in each experiment are shown in Table 8.3.

Because of its exclusive writes, "SimpleUpdate" can only exploit inter-block parallelism while "RemoteUpdate" and "DistributedUpdate" can exploit both types of parallelism since SIs separate their computations from updating the SDBs. In the 256s and 128s cases, the total $P_{external}$ is 256 and 1024, respectively. For 8 nodes, the theoretical limit on $P_{external}$ per node is 32 and 128, respectively; these numbers are much larger than 5 (the upper bound on the actual parallelism, is shown in Fig. 8.3). The actual parallelism drops because of the overhead in delivering SIs. This explains the small difference in actual parallelism among SPs. In contrast, in the 256r2 case the theoretical $P_{external}$ is only 16. When the number of nodes equals 4, 6 and 8, the $P_{external}$ per node is only 4, 2.66 and 2, respectively. However, by combining $P_{external}$ with $P_{internal}$, in the D SP, the parallelism increases substantially (see Fig. 8.3). This also explains why the actual parallelism drops significantly in S while decreases slightly in D with increases in the number of nodes.

We can now investigate the effect of parallelism on performance. SPM allows data communications only at the beginning and at the end of an SI's execution [119]. Multithreading overlaps the pre-fetching of remote operands with the execution of previous SIs, and also result storing/updating with the execution of new SIs. Assume that the latency to get a remote SDB is t_r; the probability to need a remote access is r; the average number of input operands per SI is n_0; and the net CPU time to execute an SI is t_e. If the actual degree of parallelism is larger than $p_c = (r \times n_0 \times t_r)/t_e$, then the communication latency can probably be hidden.

Based on experiments with SDBs of size 128 × 128, the value of p_c is 1.41, 5.46, 9.86 and 12.97 with 2, 4, 6 and 8 nodes, respectively. Thus, these SDBs have fine granularity and the long communication latency cannot be hidden well. The system works under the delay bound condition [119]. For SDBs of size 256 × 256, the value of p_c is 0.57, 1.58, 2.18 and 2.60 with 2, 4, 6 and 8 nodes, respectively. In these cases the communication latency can be completely hidden. The system works under the CPU bound condition [119]. For this reason, the cases of 256s and 256r2 yield better performance than 128s and 128r, respectively. For matrix multiplication with same sized matrices, the former take less than half the time (case 256s versus 128s and case 256r2 versus 128r). In the case of 256r2 the difference in the actual degree of parallelism between the SPs is significant. The actual parallelism in "DistributedUpdate" is greater than 4 (see Fig. 8.3) and greater than or close to the critical value p_c. The actual parallelism in "SimpleUpdate" is lower; in fact, it is lower than pc for 6 or 8 nodes. This explains the result in Fig. 8.2.

Based on this analysis, increasing the actual parallelism can improve performance up to a point under the delay bound condition. Further increasing parallelism would increase the number of SIs competing for the local CPUs and would extend the lifespan of SIs. This behavior is verified in Fig. 8.5 and 8.6. From

Fig. 8.6, we can see that the actual degree of parallelism increases by increasing the maximum number of threads per node. The execution time of the SPs, however, decreases significantly only when the maximum number of threads increases from 1 to 4. Further increasing the maximum number of threads results in insignificant execution time decreases.

Although "distributed_updates" increase paralleli, they require synchronization operations. Their overhead is insignificant compared to the gains. The available parallelism may vary during execution. In our experiments, the intrinsic $P_{external}$ decreases gradually as execution progresses. The inter-block parallelism is not enough only close to the end of the execution. Thus, expensive "distributed update" operations are only needed during the latter part of the SP's execution. We could then combine "write" with "distributed update" operations to get the benefit of the latter with a reduced overhead. Fig. 8.5 testifies to this effect. To conclude, good performance is obtained by combining multithreading with the distributed representation of application data.

CHAPTER 9

SCALABILITY

Scalability has become an attribute of paramount importance for computer systems used in business, scientific and engineering applications. Although scalability has been widely discussed, especially for pure parallel computer systems, there is not really a widely-accepted definition of scalability. In fact, the term "scalable" is so much abused that it has become a marketing tool for computer vendors independent of the system's technical qualifications. Since the primary technical purpose of scalability is to show users how well the system can work for larger problems with increases in its size, we define, a refined multi-dimensional definition of scalability quantitatively [126]. We also apply this definition to PC clusters, a rather difficult subject due to their long communication latencies. Since scalability does not solely depend on the system architecture but also on the application programs and their actual management by the run-time environment, for the sake of illustration we evaluate scalability for programs developed under the super-programming model (SPM) [106,119,125].

9.1 Scalability and its Analysis

9.1.1 Demand on Scalability

Generally scalability relates to the possibility to build larger systems to address larger problems without significant performance degradation due to communication increased and other latencies in the parallel or distributed environment. Scalability has become a critical attribute of computer systems and/or software solutions in various application domains for economical and technical reasons. Different users

may have different expectations from computer systems, such as their computing capability. The particular user's requirements may also be subject to change with time. For example, a business user may initially need a small computer system to satisfy basic requirements and then may demand a more powerful system for larger business tasks. Actually many servers increase their user base daily and their application datasets become ever larger. In practice, it is neither possible to design and build a computer system for each user, nor to replace computer systems very often. Therefore, the solution to address demand diversity is to make systems adaptable and scalable. That is, keeping the basic features of the computer systems, more resources could be added easily to satisfy higher demands. This way, a computer system could satisfy many users or applications for relative long period of time.

The primary objective of scalability is to characterize computer systems in a way that can convince users that they can still work well with larger workloads. Intuitively, scalability is also an attribute of a scheme, design, architecture or solution for a device or system as it pertains to solve large problems. When the users have an overall solution (e.g., a pair of a computer system and an algorithm) to solve a particular problem or provide a particular service for a certain problem size, they usually want to know how well 1) the solution works for larger problems (i.e., if the results are produced quickly enough or an acceptable service level is possible). If the predicted results cannot reach the desired goal (in execution time or service quality), they may also want to know 2) if they can keep the software solution but increase the system resources to reach another desired performance level. In the latter case, they may even want to know 3) how many resources must be added to achieve this goal and the associated cost. To answer these questions, scalability studies should produce quantitative metrics to describe their solution. If the answer to question 2 is yes, the solution can be still called "scalable." When comparing different solutions

that solve the same enlarged problem, the solution that requests fewer resources has better scalability.

In general, scalability measures the capability of a solution to maintain or increase the performance when the problem size increases and/or the size of the computer system increases. For this reason, relationships between the performance and the problem size, and/or the system size [127–129]. Researchers have used various performance metrics and techniques in their definition of scalability expressed in the form of a single index [130–134]. Most often, scalability is associated with parallel computer systems. None of the aforementioned definitions can be appropriate for all computer architectures because of narrow focus on a few parameters and their limited search space. We strongly believe that scalability is a comprehensive attribute that can hardly be expressed by a single index.

9.1.2 Scalability as a Comprehensive Entity

As mentioned earlier, scalability is an attribute of a particular problem solution in association with a computer system. Neither the properties of the architecture nor the properties of the algorithm can exclusively determine scalability. Solutions adopting the same application algorithm may have different scalabilities. It is possible that the implementation of the algorithm on one system is scalable but another implementation of the same algorithm on another system with different architecture is not scalable. Claiming that an algorithm is scalable may only mean that there is an architecture on which the solution of this algorithm is scalable. In theory, the architecture may be an ideal computer model, such as the PRAM [135]. Similarly, we cannot consider only a computer system or architecture without mentioning an algorithm or a problem. Sometimes the system can be expanded rather easily to potentially accommodate larger problems. There is a working window where

the implementation of some algorithms on an expandable architecture often improves performance [136, 137]. This is also true for PC clusters.

From the quantitative analysis point of view, scalability depends on many factors that characterize both the architecture and the algorithm. For example, to determine the scalability of matrix multiplication, one may need the size of the matrices, their type and their sparsity; for a distributed multi-computer system, one may need the number of processors, the size of the memories attached to the processors, and the number and type of communication channels interconnecting these processors along with their bandwidth. Current practices in scalability analysis do not provide a direct way to take into account large numbers of parameters reflecting the detailed features of the application algorithm and the target computer system.

9.1.3 Existing Scalability Definitions

The various definitions of scalability come from similar motivations [13]. A widely used definition employs the asymptotic speedup metric S(p, n), where p is the number of processors and n is the problem size [132]. Speedup is defined as the ratio of the serial run time $T_s(n)$ of the best algorithm that solves the given problem of size n to the time T(p, n) taken by the selected parallel algorithm for the same problem with p processors. Formally,

$$S(p,n) = \frac{T_s(n)}{T(p,n)} \tag{9.1}$$

for a problem of fixed size. However, this definition may ignore significant non-primary terms related to overhead; this asymptotic behavior study may not be practical then in real program runs. The definition should actually use the exact order Θ notation. The simplest definition of scalability uses the system efficiency [69]

which is

$$Sc = \frac{S(p,n)}{p} \tag{9.2}$$

The authors then introduce the iso-efficiency concept, for an algorithm-system pair. It states that the efficiency should be fixed in scalable problem-system pairs independent of their sizes. For a fixed $Sc > 0$, we can then approximate the execution time when the system size increases:

$$T(p,n) = \frac{T_s(n)}{Sc \times p} \tag{9.3}$$

Since all overheads are reflected into the parallel execution time T(p, n), we could find how many processors should be added into the system to decrease the execution time to a desirable level:

$$p = \frac{T_s(n)}{Sc \times T(p,n)} \tag{9.4}$$

This definition has the problem that users do not always know in advance the optimal efficiency for a given pair.

Another definition of scalability for a given architecture-algorithm pair uses the ratio of the asymptotic speedup of the algorithm on this architecture to its asymptotic speedup $S_I(p,n)$ on an ideal machine (such as the PRAM) with the same number of processors [132]:

$$Sc = \frac{S(p,n)}{S_I(p,n)} = \frac{T_{pI}(p,n)}{T_p(p,n)} \tag{9.5}$$

This definition tries to decouple the architecture from the algorithm in order to compare with the best possible performance. Thus, it uses the time complexity of the algorithm on the selected ideal machine as a reference point. This approach is not practical for a given system that has to be "modified" appropriately to get the

best performance. In fact, end users cannot even estimate the number of required processors for good scalability based on the reference data. References [129, 135] focus exclusively on problems that are scalable in nature primarily because of data parallelism. Such problems appear in scientific applications (such as weather prediction) where the size of the input data set can be increased "indefinitely" to produce higher accuracy in the solution. Business and many other applications, however, do not often follow this pattern or sometimes users are interested in finding out how performance being improved with limited resource improvements that focus on specific aspects of the computer system.

9.1.4 More on Limitations of Current Approaches

Scalability analysis relies on a chosen theoretical methodology to analyze performance for various problem and system sizes. However, current methods of scalability analysis have major limitations because there are inadequacies in their definition of scalability as well as the analysis approaches taken by them. A major disadvantage comes from adopting asymptotic terms in the definition of scalability. Two problems show up. The first one is practical since the users are forced to care about the behavior when the system size varies in a tremendous range. However, most of architectures work well for given problems only when the system size changes are of limited range. An asymptotic analysis of scalability may miss too much important information in practical cases. Performance behavior due to changes in the system size within a specific practical range may be very different from that of asymptotic changes. The second problem stems from the fact that such definition may be meaningless since system performance normally deteriorates rapidly due to intolerable latencies in humongous systems.

The second disadvantage is that these definitions measure the system size based exclusively on the number of processors. They assume that the performance of programs depends primarily on the number of processors and less on other resources in the system. Under this assumption, several scalability analyses only account for the overhead of communication [138]. However, this is not sufficient. As discussed earlier, the performance depends on both the problem and system sizes. To completely represent the system size may require multiple parameters rather than just the number of processors. Besides processors, a system may be increased in size by installing several additional resources (e.g., memory) to improve its performance. Increasing the amount of available memory can help to improve the scalability of memory-bound problems such as sorting of very large data sets. Ignoring these other types of resources may have adverse effect in performance estimation. Although scalability analysis exclusively for memory-bound problem has been pursued, an integrated approach that incorporates many choices for system changes has not been attempted. In general, these approaches are incomplete.

To make scalability analysis practical, one usually builds an execution model depending on some assumptions. A widely accepted implicit assumption is that the execution time of all basic processor operations is constant and does not depend on the overall system size. Based on this assumption, the sequential execution time of a program can be estimated by its total number of basic operations. It is often called the workload $W_{ideal}(n)$ of the problem. The cost of solving a problem on a p-processor system can be defined as $p \times T_p(n)$; it is the maximum number of operations in the parallel execution time, which is expressed in time units for basic operations. During execution, the p processors perform useful operations to solve the problem and operations resulting in overhead. If $W_{overhead}(p, n)$ represents all

the operates corresponding to overhead, including the idle time of processors, then

$$p \times T_p(n) = (W_{ideal}(n) + W_{overhead}(p, n)) \times t_0 \quad (9.6)$$

where t_0 is the execution time of a basic operation. If $W_{overhead}(p, n)$ can be estimated, then the execution time $T_p(n)$ can be approximated and the system size can be derived to solve a problem of given size in a specified amount of time. Most scalability analyses focus on the overhead function $W_{overhead}(p, n)$, only estimating the amount of additional operations due to communications. A drawback is that this estimation does not count idle times. Also, it is difficult to estimate this function when the system size changes in a wide range and/or non-processor resources in the system also increase.

9.2 New Quantitative Analysis of Scalability

9.2.1 Mathematical Definition of Scalability

An application problem in this study can be characterized by two sets of attributes. One relates to the methodology (i.e., the chosen algorithm). The other is a set of parameters that relates to the characteristics of the input (such as the dimensionality and sparsity of matrices in matrix multiplication). The latter parameters are imply collectively referred to as the size of the problem; they can be denoted with a vector $\mathbf{P}(p_1, p_2, \ldots, p_j, \ldots)$ in the problem space $\mathscr{P}$. Each point in this space represents a particular size for the problem.

A computer system also can be described with two sets of attributes. The first set of qualitative attributes identifies the system's specific architecture. For example, a PC cluster consists of a set of autonomous PCs which are connected to each other via a standard local network. The type/class of nodes and the type of network should

be included in this set. The other is a set of quantitative attributes $\mathscr{Q}$ which specify the system's detailed characteristics. For example, a PC cluster can be characterized by its number of processors, the total amount of memory per node and the bandwidth of the network. Each member value quantifies the system in a specific aspect. The specific system condition can be identified with a vector $\mathbf{R}(r_1, r_2, \ldots, r_i, \ldots)$ in the multi-dimensional system space $\mathscr{R}$. Among the set of quantitative attributes there is a sub-set of attributes which are associated actually with the system's size (e.g., number of nodes). Therefore, the size of the system can be projected as a vector $\mathbf{R}(r_1, r_2, \ldots, r_i, \ldots)$ into a sub-space of $\mathscr{R}$. Each point in this space represents a particular instance of the computer architecture which can be quantified.

Using the cross-product space $\mathscr{N}$ of $\mathscr{R}$ and $\mathscr{P}$ (or, equivalently, the cross-product space $\mathscr{M}$ of $\mathscr{Q}$ and $\mathscr{P}$) for our values domain, we can study scalability. The vector $\mathbf{N}(r_1, r_2, \ldots, r_i, \ldots; p_1, p_2, \ldots, p_j, \ldots)$ in space $\mathscr{N}$ denotes an implementation case of running the program of specific size on the specified system with specific quantitative attributes. For each such implementation case, there is an associated performance metric.

$$\mathbf{U} = \mathbf{U}(\mathbf{R},\ \mathbf{P}) \tag{9.7}$$

This function should completely profile the performance of the algorithm on the given instance of the computer architecture. The metric U could be defined as any quantitative index that can measure the degree of user satisfaction. For example, the chosen metric could be throughput for service programs. For most computational programs, the metric could be the execution "speed" defined as 1/T, where T = T($\mathbf{R}$, $\mathbf{P}$) is the execution time of the program.

The scalability Sc at a particular point ($\mathbf{R}$, $\mathbf{P}$) can now be defined as the directional derivative of the considered metric along a particular directly in the new

space $\mathscr{N}$. That is:

$$Sc(\mathbf{V}, \mathbf{R}, \mathbf{P}) = \frac{(U(\mathbf{N} + \triangle\mathbf{N}) - U(\mathbf{N}))}{\delta} \tag{9.8}$$

where δ is a small scalar, $\triangle\mathbf{N} = \delta\mathbf{V}$ and $\mathbf{V}$ is the unit vector in the specified direction along which the system resources $\mathbf{R}$ increases while the problem sizes $\mathbf{P}$ is constant or increase simultaneously in the space $\mathscr{N}$. Of course, $\mathbf{N}$ represents the pair ($\mathbf{R}$, $\mathbf{P}$).

When $Sc(\mathbf{V}, \mathbf{R}, \mathbf{P}) > 0$, we can say that the system-problem pair ($\mathbf{R}$, $\mathbf{P}$) is locally scalable in the direction $+\triangle\mathbf{N}$ of the chosen dimension. Based on this definition of scalability, it is easy to conclude that the result does not depend on the selection of any reference case.

9.2.2 Directionality of Scalability

This definition of scalability is generic for a single dimension. Besides the chosen computer architecture and algorithm pair, the scalability also depends on the way selected to increase the system size and/or problem size; i.e., scalability is directional. Different analysts may have different considerations and may get different conclusions regarding the scalability of a solution. Scalability analysis in all directions is needed to completely characterize a solution. Scalability analysis in a particular dimension will give a solution under a set of specific constraints. Under some constraints, scalability analysis may be reduced to existing analyses discussed earlier. By selecting 1/T as the metric of user satisfaction and normalizing to a reference case that solves the problem on an optimized sequential computer, we get

$$U = \frac{1/T_p(\mathbf{R}, \mathbf{P})}{1/T_s(P)} = \frac{T_s(P)}{T_p(R, P)} \tag{9.9}$$

which is simply the speedup; $T_s(\mathbf{P})$ and $T_\mathbf{p}(\mathbf{R}, \mathbf{P})$ represent the sequential and parallel execution time, respectively. Under the constraints that the system increases its size by exclusively adding more processors (a practical example is a multiprocessor computer with centrally shared memory) while the problem size stays fixed, the obtained scalability corresponds to the first definition. Similarly, if the problem size increases linearly with the number of processors and all other system parameters are kept constant, we study the scalability according to the definition of scaled speedup [135].

Since PC cluster systems normally increase in size by adding more member computers, the numbers of many and diverse system resources (including processors, memory, NICs, etc.) increase simultaneously. Therefore, it is more reasonable to analyze the scalability along the direction in which the relevant parameters of the associated resources increase simultaneously and linearly.

9.2.3 Studying the Cost for Optimal Scale Up

As mentioned above, the primary purpose of scalability analysis is to ultimately help users to decide if a system can be scaled up graciously to solve a larger problem. Quantitative analysis can help to decide how exactly to scale up a system to improve a measure, such as performance. However users usually have a comprehensive cost metric for system updates as well. Some updates may be scalable but their cost may be prohibitively high. The cost of an update can be represented by

$$cost = \mathbf{V_P} \cdot \mathbf{V_U} = \sum_i (Price_i \cdot \triangle r_i) \tag{9.10}$$

where $\mathbf{V_P} = (Price_1, Price_2, \ldots, Price_i, \ldots)$ is a vector of prices for the considered resources and $\mathbf{V_U} = \triangle\mathbf{R} = (\triangle r_1, \triangle r_2, \ldots, \triangle r_i, \ldots)$ is a vector of resource changes

(in arbitrary units); $\triangle r_i = (r_i - r_i^o) \geqslant 0$. We can assume, for the sake of simplicity, that if r_i expresses the resource amount in linear scale, then $Price_i$ gives the regular price of the i-th type of resource; if r_i expresses the resource amount in logarithmic scale, then $Price_i$ gives the price when doubling the i-th type of resource. An optimization problem is to update various types of resources for maximum scalability benefits under a given total cost increase. The users may conveniently define the efficiency of updates for system-problem pairs as

$$\begin{aligned} E = \frac{U}{cost} &= \frac{U(\mathbf{N^o} + \triangle \mathbf{N}) - \mathbf{U}(\mathbf{N^o})}{\sum_i (Price_i \cdot \triangle r_i)} \\ &= \frac{U(\mathbf{R^o} + \triangle \mathbf{R}, \mathbf{P^o} + \triangle \mathbf{P}) - \mathbf{U}(\mathbf{N^o}, \mathbf{P^o})}{\sum_i (Price_i \cdot \triangle r_i)} \end{aligned} \tag{9.11}$$

where $\mathbf{R}^o(r_1^o, r_2^o, \ldots, r_i^o, \ldots)$ is a reference system configuration, $\mathbf{P}^o(p_1^o, p_2^o, \ldots, p_j^o, \ldots)$ is a reference problem size, $\mathbf{R}(r_1, r_2, \ldots, r_i, \ldots)$ is the size of a scaled up system and $\mathbf{P}(p_1, p_2, \ldots, p_j, \ldots)$ is the size of the problem under study. In this case, the optimality problem is to find the direction in which the efficiency of the update is maximal, i.e., find the direction corresponding to the largest possible value of U for a given update cost.

Based on field theory, in a range enclosing a reference system configuration $\mathbf{R}^o$ the increase in U achieved by moving the system state from $\mathbf{R}^o$ to R along the vector V_U will be

$$\triangle U = \mathbf{grad(U)} \cdot \mathbf{V_U} \tag{9.12}$$

where $\mathbf{grad(U)}$ is the gradient of U [139]. Thus, the scalability in the i-th dimension is

$$Sc_{i,\triangle P} = \frac{U(\mathbf{R^o} + \triangle \mathbf{R}, \mathbf{P^o} + \triangle \mathbf{r_i}) - \mathbf{U}(\mathbf{N^o}, \mathbf{P^o})}{\triangle r_i} \tag{9.13}$$

Based on Equation (9.11), the efficiency of the system update is

$$E = \frac{\mathbf{grad(U)} \cdot \mathbf{V_U}}{\mathbf{V_P} \cdot \mathbf{V_U}} \tag{9.14}$$

We can prove that if $(Sc_{k,\triangle P}/Price_k) > (Sc_{i,\triangle P}/Price_i)$ for all $i \neq k$, then increasing the resources in the k-th dimension yields the most efficient solution. We can also prove that if there are two or more dimensions that maximize $Sc_{i,\triangle P}/Price_i$, then a combination of resource updates in a subspace could yield the same efficiency.

Proof: The proof is included in the Appendix. □

9.3 Scaling up PC Clusters and Scalability Study under SPM

9.3.1 Techniques for Scaling up PC Clusters

Scaling up a system is the basis for users to improve performance so that they can solve a larger problem and/or solve the same problem more quickly. PC clusters can be scaled up in multiple ways that can be roughly classified into two categories. One is increasing the number of computer nodes in the cluster. The other is scaling up the member nodes by improving their capabilities.

The first technique is straightforward and essential in scaling up PC clusters, as long as the new nodes can work seamlessly in the cluster. This way, multiple types of resources in a PC cluster can be increased simultaneously. Besides increasing the number of processors, the total memory in the cluster also increases. The number of nodes in the cluster may still be a good parameter to describe the size of the system. However, newly installed PCs do not need to be identical to previous PCs. They may have more processors, more memory, may be equipped with more powerful processors or faster memory, etc. This approach may result in a heterogeneous cluster. Then, the number of nodes may not be a good parameter to denote the system size.

The second category of techniques scales up a PC cluster by increasing the amount of resources in its member nodes or improving their capabilities. These techniques may be applied in many different directions. One can increase the number of processors in individual nodes, the memory in the nodes individually or simultaneously, etc. Such approaches can improve a PC cluster to help end users to solve larger problems in the requested time. The idea of scaling up clusters by modifying member nodes can have a larger meaning. People may treat each member computer in the PC cluster as a logic subsystem. They can then scale up subsystems by replacing member computers with entire PC clusters. This way, a PC cluster will become a hierarchical structure of many levels providing many opportunities.

Denoting the size of a hierarchical computer cluster via multiple parameters is reasonable and makes scalability analysis easier to handle. For example, the size of a homogeneous PC cluster with a k-level hierarchical structure may be denoted with the data vector $\mathbf{R}\{r_1, r_2, \ldots, r_j, \ldots, r_k, r_c, r_m\}$; r_1 is a parameter for the number n_1 of nodes at the top most (first) level, r_2 is a parameter for the number n_2 of nodes in a subcluster that serves as a node in the first level structure; similarly, $r_j (1 < j \leqslant k)$ is a parameter for the number n_j of nodes in a subcluster that serves as a node in the (j-1)-th level structure. We select a logarithmic expression for these parameters, i.e., $r_j = \log n_j$. For each k-level system, the k-th level subclusters consist of atomic PCs (i.e., the member nodes of these k-th level subclusters are individual PCs). r_c is a parameter for the number of processors in the PC and the r_m is a parameter for the memory size in each PC. There may be more parameters needed to characterize the size of last-level atomic computers; we ignore them here for the sake of simplicity.

This hierarchical approach provides significant flexibility in building PC clusters, thus improving their scalability features. For example, assuming a uniform

cluster with four levels and 32 nodes per subsystem (i.e., $r_1 = r_2 = r_3 = r_4 = 5$) the resulting system is huge consisting of about one million or 2^{20} PCs.

9.3.2 Scaling Up Programs Developed under SPM

Scaling up a PC cluster is just an instrument to help users solve larger problems. However, they should care about the performance of their software solution on the scaled up system as well. The factors that affect the scalability of a program under a given programming model constitute two groups. One group is related to the scalability of the program on the given architecture; the other is related to the scalability of the implementation in terms of basic operations. The following example illustrates the difference between these two groups. Assume the problem of multiplying two dense matrices of size $M \times M$ on p "processors." If these processors can collectively perform multiplication and addition of two dense matrices of size $m_2 \times m_2$ directly (i.e., in constant time), where $m_2 \leqslant M$ and, then the complexity of the program is $W(m_1) \approx const \times m_1^3$, where $m_1 = M/m_2$. Considering the overhead in terms of additional basic operations $W_o(p, m_1)$, the execution time of the program running on this computer system will be

$$T(p, f, m_1, m_2) = C(p, m_1)t(f, m_2) = \frac{W(m_1) + W_o(p, m_1)}{p} t(f, m_2) \qquad (9.15)$$

where f is a performance metric of the processors (e.g., the MIPS rate) and $t(f, m_2)$ is the average time to perform a basic operation on these processors. When users need to multiply larger matrices (i.e., with larger M) but keep the total execution time at the same level, they can either increase the number p of processors to decrease the first factor $C(p, m_1)$ or increase the computation performance f of the processors to decrease $t(f, m_2)$. The effect of these techniques depends on the real values of m_1, m_2, f and p. $C(p, m_1)$ characterizes the scalability of the program itself

while $t(f, m_2)$, on the other hand, characterizes the scalability of basic computer operations.

Few of the existing programming models mention the effect of implementing of basic operations on the overall scalability of the solution. The primary reason is that most of these programming models do not provide a mechanism to scale up the set of supported basic operations. They use directly the instruction set of the processor to derive their basic operations. However, the problem size that can be solved directly with the instruction set of a processor is usually fixed. For example, COTS (commercial off-the-shelf) processors for PCs only support the multiplication of a pair of scalar variables (i.e., in terms of dimensions they often implement only directly the multiplication of matrices of size 1×1). Besides this, processors are normally impossible to scale up since their resources cannot be increased after they have been manufactured.

Our super-programming model (SPM), however, can fully exploit the scalability of basic operations [106, 119, 125]. This is because the basic operations under SPM are coarse-level super-instructions. SPM integrates both message passing and shared memory. Under SPM an effective instruction-set architecture (ISA) is to be developed for each application domain [119]. Frequently used operations in that domain should belong to this ISA. The *Super-Instructions (SIs)* in the ISA are to be developed efficiently in the form of program functions for individual PCs in the cluster. The operand sizes (i.e., function parameter sizes) for SIs are limited by predefined thresholds. Application programs are modeled as *Super-Programs (SPs)* coded with SIs. Under SPM the parallel system is modeled as a virtual machine (VM) which is composed of a single super-processor that includes an SI fetch/dispatch unit (IDU) and multiple SI execution units (IEUs). For PC clusters, an IEU is a process running on a member node that provides a context to execute SIs. SIs are

dynamically assigned to IEUs based on a producer-consumer protocol. The IDU assigns SIs, from a list of SIs ready to execute, to IEUs as soon as the latter become available. The super-processor can handle a set of "build-in" data types that form opprands for SIs; they are called *Super-Data Blocks (SDBs).* All application data are stored in SDBs which can be accessed via appropriate interfaces. The runtime support system provides local data representation for SDBs. Each SDB can be incarnated into a set of objects distributed throughout the cluster. The SPM runtime support system controls the incarnated objects during their lifecycle based on their usage [125]. Thus, the logic of scheduling and distributing tasks can be decoupled from the actual data distribution and the impact of the latter on workload balancing is reduced dramatically. The aforementioned set of incarnated objects represents a coherent data entity. They cooperate with each other with the mediation of the runtime system. Such multiple distributed representations can serve efficiently the demand for data of multiple SIs. Our approach increases the number of SIs executing in parallel by minimizing thread stalling. The reader is encouraged to read [119,125] for more details.

The size of the problem solved by an SI depends heavily on its SDB operands that have configurable size. Also, SIs execute on the IEU virtual functional unit which can be implemented with scalable hardware/software systems such as symmetric multiprocessors or even PC clusters. In this situation, exploiting the scalability of basic operations may make the programs more scalable. Let us review the previous example of matrix multiplication. When the size M of the input matrices doubles while m_2 is kept fixed, the complexity $W(m_1)$ of the program will increase eight times and the overhead $W_o(p, m_1)$ may increase more. In this case, increasing the resource- processors by a factor of eight cannot keep the total execution time unchanged. In our pilot experiment with sparse matrix multiplication, the increase

in the overhead is typically greater than 20%. However, keeping m1 (i.e., the number of the blocks the matrix is divided into) and the number of IEUs fixed will not necessarily change the complexity of the super-program and the overhead. This increase in the problem size can be handled without a time penalty at this level by increasing the problem size for a basic operation (i.e., m_2) and scaling up the system by increasing the resources in each IEU by a factor of eight. If $t(f, m_2)$ increases by less than 20% when m_2 doubles and the individual node performance f increases eight times, this approach will be better than the previous one. This example shows that scaling up the SI provides a powerful instrument to improve the overall scalability.

9.3.3 Scalability of SPs for Hierarchical PC Clusters

SPM matches very well the architecture of PC clusters with multi-level structure. SIs are all implemented with procedures executing on member nodes in the cluster. When a member node itself is comprised of another lower-level cluster, these SIs can be implemented under SPM; i.e. they are similar to super-functions (SFs) that can be coded with lower-level SIs. Such a hierarchical structure can be extended until the lowest level is an atomic PC, as discussed earlier.

Finding the optimal parameters for SIs at various levels of the hierarchical cluster involves multiple solution subspaces. Each subspace involves problem and resource sizes; for the sake of simplicity, we assume here a homogeneous cluster where the latter is simply the number of member nodes. Based on our earlier discussion, the total execution time is

$$T(r_1, r_2, \ldots, rk, p_1, p_2, \ldots, pk, p_{k+1}) = (\prod_i C_i(r_i, p_i)) \cdot t(p_{k+1}) \qquad (9.16)$$

where r_i is the parameter for the resource size of the i-th level cluster, p_1 is the problem size in terms of operands for first-level SIs, $C_1(r_1, p_1)$ is the complexity of the problem including all associated overheads in number of first-level SIs, p_i (for i = 2, ..., k) is the problem size for SIs in (i-1)-th level clusters, $C_i(r_i, p_i)$ (for i = 2, ..., k) is the average complexity including all associated overheads of (i-1)-th level SIs in number of i-th level SIs, p_{k+1} is the problem size for k-th level SIs and $t(p_{k+1})$ is the average execution time of k-th level SIs on atomic PCs. ri may be expressed as $\log_2 n_i$, where n_i is the number of member nodes in an i-th level cluster. The total number of atomic PCs in the cluster is $n = 2^r$, where $r = r_1 + r_2 + \ldots + r_k$. The overall problem size in the number of basic instructions for atomic PCs is $p = p_1 \times p_2 \times \ldots \times p_k \times p_{k+1}$. Increasing the total number of atomic PCs m times can be achieved by increasing the number of nodes m times at any level. Similarly, increasing the problem size pi at the i-th level q times also increases the overall problem size q times. If we adopt $U(R, P) = \log(1/T(R, P))$ as the performance metric (all logarithms here have the base 2) and $r = \log n$ as the resource metric, then according to Equation (9.13) the scalability of the (i-1)-th level SIs is represented by

$$Sc_{i,\triangle P} = \frac{\log(\frac{C_i(r_i^o, p_i^o)}{C_i(r_i, p_i)})}{\triangle r_i} \tag{9.17}$$

It only depends on the algorithm adopted for the i-th level.

From the results in Section 9.2.3 we know that the dimension with the maximum value for $Sc_{i,\triangle P}/Price_i$ is the most efficient to update. (see Equation (9.14)). This will be referred to as the edge-scalability of the solution.

9.3.4 Scalability of an Optimally Configured Solution for a PC Cluster

Now let us discuss some important properties of scalability for a well configured solution. For the sake of simplicity, we assume that the overall problem size is constant and the performance metric is the logarithm of the speedup. For $\triangle r_i = 1$, which means that $\log n_i - \log n_i^o = 1$, we have $n_i = 2n_i^o$ and

$$Sc_{i,\triangle P} = \log(\frac{C_i(r_i^o, p_i^o)}{C_i(r_i^o + 1, p_i)}) \tag{9.18}$$

Therefore, the scalability can be expressed with logarithmic execution time decreases when the resources double.

A multi-level PC cluster can be easily reconfigured and the programs developed under SPM can also fit the reconfigured system by adjusting the parameters for SIs. The basic reason is that scaling up levels requires the same "raw materials"-PCs. An optimal configuration provides maximum performance under a given amount of resources. The first property of an optimally configured cluster is that the scalabilities of all its non-leaf levels are identical. The fundamental reason is liquidity of resources. Reconfiguring the system then simply requires reducing the resources in one level to scale up another level. This is proved as follows. If $Sc_{i,\triangle P} > Sc_{j,\triangle P}$ (for $i, j < k$), we can always reduce the number of nodes in each j-th level cluster while increasing the number of nodes in each i-th level cluster. For this adjustment

$$\triangle U = U(R^o + R, P^o + P) - U(R^o, P^o) = Sc_{i,\triangle P}\triangle r_i + Sc_{j,\triangle P}\triangle r_j \tag{9.19}$$

where $\triangle r_i = -\triangle r_j > 0$; therefore, $\triangle U = (Sc_{i,\triangle P} - Sc_{j,\triangle P})\triangle r_i > 0$. It means that the current configuration is not optimal if the scalabilities of non-leaf levels are not same. The resources at the lowest level are not interchangeable with resources at other levels since leaf nodes are atomic PCs. Normally we can neither break

single PCs to build two or more PCs nor merge the resources of multiple PCs into a more powerful PC. However, since a PC usually has many resources, the problem size for SIs executing on a PC can vary in a wide range. Users can reconfigure the task grains between atomic PCs and upper level clusters to pursue maximum performance. Therefore, the lowest level should have the same scalability with upper levels; it can be obtained through appropriate tasks assignment. This means that a well configured PC cluster should have the same scalability at all the levels; it should be close to the edge-scalability of the cluster (as defined earlier).

Another property of an optimally configured cluster is that its edge-scalability should not be less than the scalability Sc0 of the initial cluster when a PC is replaced with a two-node cluster; that is,

$$\log(\frac{C_i(r_i^o, p_i^o)}{C_i(1 + r_i^o, p_i)}) \geqslant Sc_0 \tag{9.20}$$

where $Sc_0 = \log Sp_0$ and Sp_0 is the speedup of a two-node cluster.

Equation (9.20) should hold for all $r_i^o = 0, 1, 2, \ldots, r_{i-1}$, if Sc_i is to decrease monotonically. Then,

$$\log(\frac{C_i(0, p_i)}{C_i(r_i, p_i)}) = \log[\frac{C_i(0, p_i)}{C_i(1, pi)} \cdot \frac{C_i(1, p_i)}{C_i(2, pi)} \cdot \ldots \cdot \frac{C_i(r_i - 1, p_i)}{C_i(r_i, pi)}]$$

and , therefore $\log(\frac{C_i(0,p_i)}{C_i(r_i,p_i)}) \geqslant r_i Sc_0$. Then, based on Equation (9.15) the overall speedup of the solution is

$$\begin{aligned} Speedup &= \log(\frac{T(0, 0, \ldots, 0, p_1, p_2, \ldots, p_k, p_{k+1})}{T(r_1, r_2, \ldots, r_k, p_1, p_2, \ldots, p_k, p_{k+1})}) \\ &= \textstyle\prod_{i=1}^{k}(C_i(0, p_i)/C_i(r_i, p_i)) \\ &= \exp\,(\textstyle\sum_{i=1}^{k} r_i) \cdot Sc_0 \\ &= (Sp_0)^r \end{aligned} \tag{9.21}$$

where $r = \sum_{i=1}^{k} r_i = \log n$, n is the total number of PCs and Sp_0 is the average speedup of a two-node cluster for the SIs. The efficiency of the solution is then

$$E = \frac{Speedup}{n} \geqslant (\frac{Sp_0}{2})^r \tag{9.22}$$

or

$$\log E \geqslant (Sc_0 - 1) \times \log n \tag{9.23}$$

9.4 Case Studies

To showcase our proposed scalability analysis approach, let us take a look at a few cases for matrix multiplication. We assume a two-level cluster system represented with the pair of parameters (r_1, r_2) that denote size, as previously. Also, the input matrices are assumed partitioned into primary submatrices (i.e., level-1 SDBs) which are partitioned further into basic (i..e, level-2) SDBs. Multiplying a pair of matrices is implemented with primary or level-1 SIs. Each primary SI performs multiplications/additions of a pair of level-1 SDBs on a level-1 node in the cluster; these SDBs are produced by the multiplication of submatrices at level-2 of size $m_2 \times m_2$. A level-1 node is itself a cluster (i.e., a level-2 cluster) that consists of many atomic PCs. Each primary SI is implemented with level-2 SIs. Multiplying a single pair of level-2 SDBs can be performed by a single level-2 SI on an atomic PC. We assume that the size of these SDBs is constant and the workload of each level-2 SI is constant. Also, the average execution time of level-2 SIs is T_2.

The primary super-program and level-1 SIs at both levels can be developed under SPM. Therefore, the SI execution times can be estimated. Assume that the workload of a primary SI is expressed in number w_2 of level-2 SIs. The average execution time of a primary SI is then

$$T_1 = C_2(r_2, m_2) \times T_2 = \frac{w_2 + W_{o2}(n_2, w_2)}{n_2} \times T_2 \tag{9.24}$$

where W_{o2} represents the overhead. The total execution time of the super-programs for matrix multiplication is

$$T = C_1(r_1, m_1) \times T_1 = \frac{w_1 + W_{o1}(n_1, w_1)}{n_1} \times T_1 \tag{9.25}$$

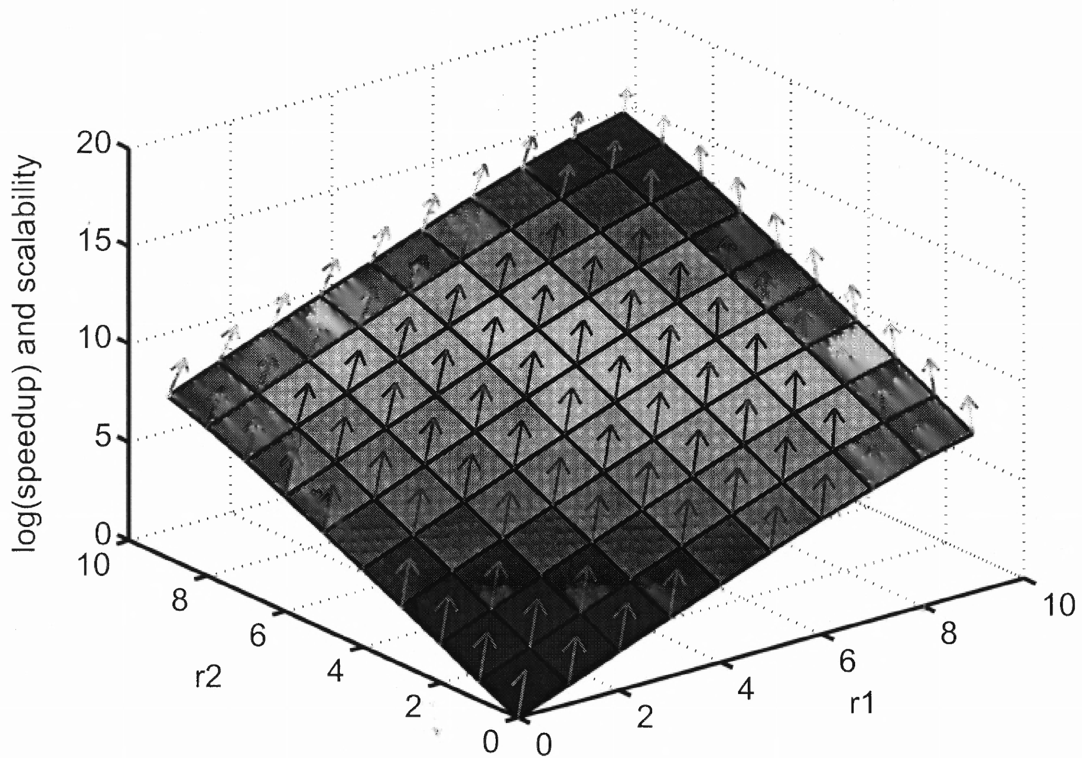
where w_1 is the workload of the program expressed in number of primary level-1 SIs and Wo1 is the overhead at this level. Under SPM, the most important overhead is due to workload imbalance since the primary objective is to minimize the idle time of cluster nodes through multithreading. This overhead can be expressed simply as

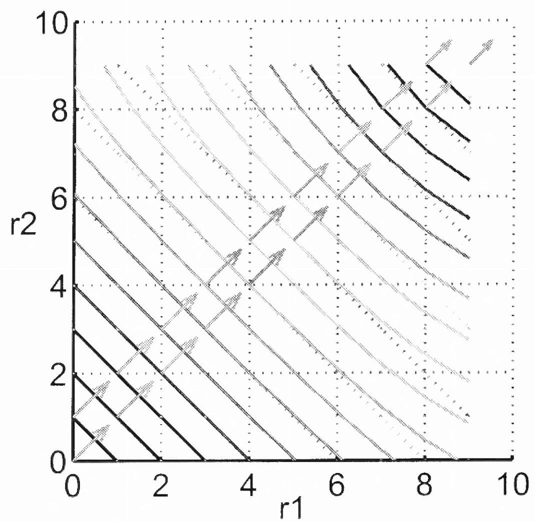
$$W_{oi} = c_i \times (n_i - 1) \qquad \text{for } \ i = 1, 2 \tag{9.26}$$

where $c_i = b_i - a_i$; b_i is the maximum number of SIs assigned simultaneously by IDU to a node to execute with multithreading and ai is the average number of SIs per node that are executing on the other $n_i - 1$ nodes when the last set of SIs are assigned to a node. $c_i T_i$ is the expected average idle time of the other $n_i - 1$ nodes. Five case studies are presented in the following subsections.

9.4.1 Case 1

Assume that the workload of the program in number of level-1 SIs is constant (i.e., m_1 and w_1 are constant) and the workload of primary (i.e., level-1) SIs is also constant (i.e., m_2 and w_2 are constant) when the system is scaled up. The result of scalability for matrix multiplication solutions is shown in Figure 9.1. In all of our simulations, the number $n_1 = 2^{r_1}$ (level-1 nodes) and the number $n_2 = 2^{r_2}$ (level-2 nodes in each subcluster) vary independently from 1 to 512; also $c_1 = 0.9$, $c_2 = 0.7$. And in this simulation case, $w_1 = 512$ and $w_2 = 512$.





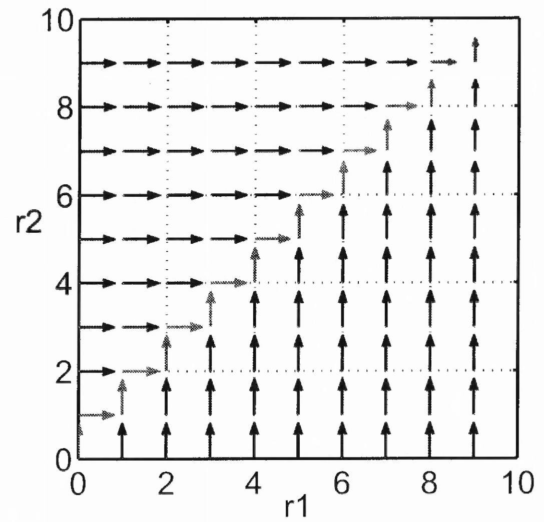
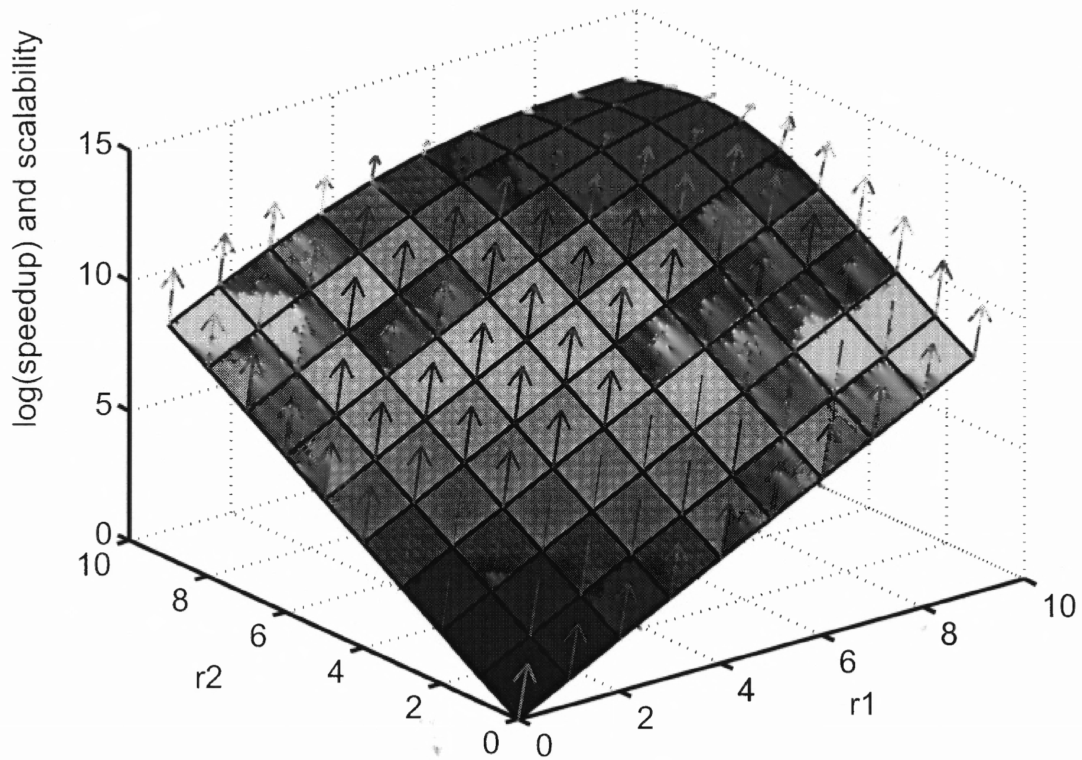


Figure 9.1 Scalability of a two-level cluster for matrix multiplication with fixed workload for the program and level-1 SIs

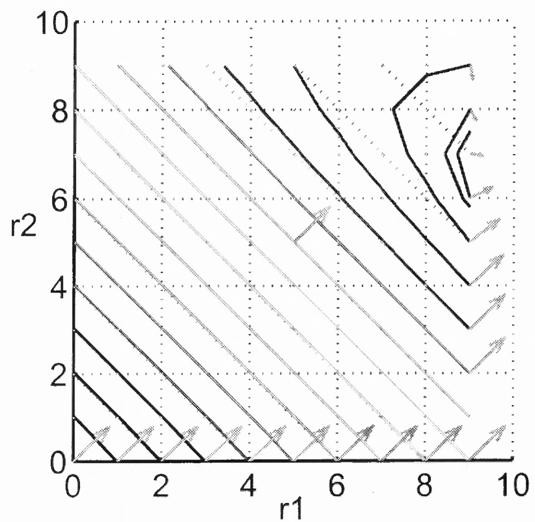
In Fig. 9.1, the pair (r_1, r_2) represents the system configuration and $\log_2(speedup)$ represents the performance metric. The arrows in Fig. 9.1a show dominant scalability increases in the system as a function of (r_1, r_2). In Fig. 9.1b, the solid curves are contour lines of performance (i.e., all system configurations located on the same curve have the same performance). The dashed lines (where $r_1 + r_2$ is constant) indicate configurations settings producing the same system size; the point of contact between a dashed line and a contour curve of performance is the optimal configuration for this system size. The arrows at these points show the gradient of the speedup. Fig. 9.1c shows the direction of system change for each configuration that improves performance most effectively when doubling the system size. The path in red indicates the optimal path to scale up this system while the size of the arrows reflects the amount of scalability. The overall direction of the path is the same as that of the arrows in Fig. 9.1b.

9.4.2 Case 2

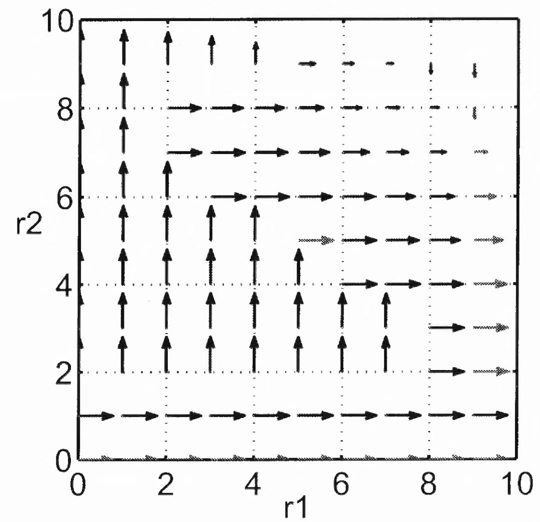
Assume that the overall problem size (i.e., m_1m_2) is still constant when the system is scaled up. The result of the simulation for the scalability of the solution is shown in fig. 9.2. In this simulation, $c_1 = 0.9$, $c_2 = 0.7$, $w_1 = 2^{18}/n_2^{1.5}$ and $w_2 = 8\times n_2^{1.5}$ (i.e., the workload of level-1 SIs increases with the number of PCs in the subcluster). Fig. 9.2 has several differences from Fig. 9.1. The obvious difference is that in Fig. 9.2a the speedup surface turns down for large values of r_1 and r_2. This shows that the solution is not longer scalable at these points. Fig. 9.2b indicates that the optimal direction corresponds to downsizing when r_1 is 9 and r_2 is greater than 7. The reason is that in these locations the number of level-1 SIs is less than the number of level-1 nodes.



(a) Scalability



(b) log(speedup)



(c) the most effective direction to scale up

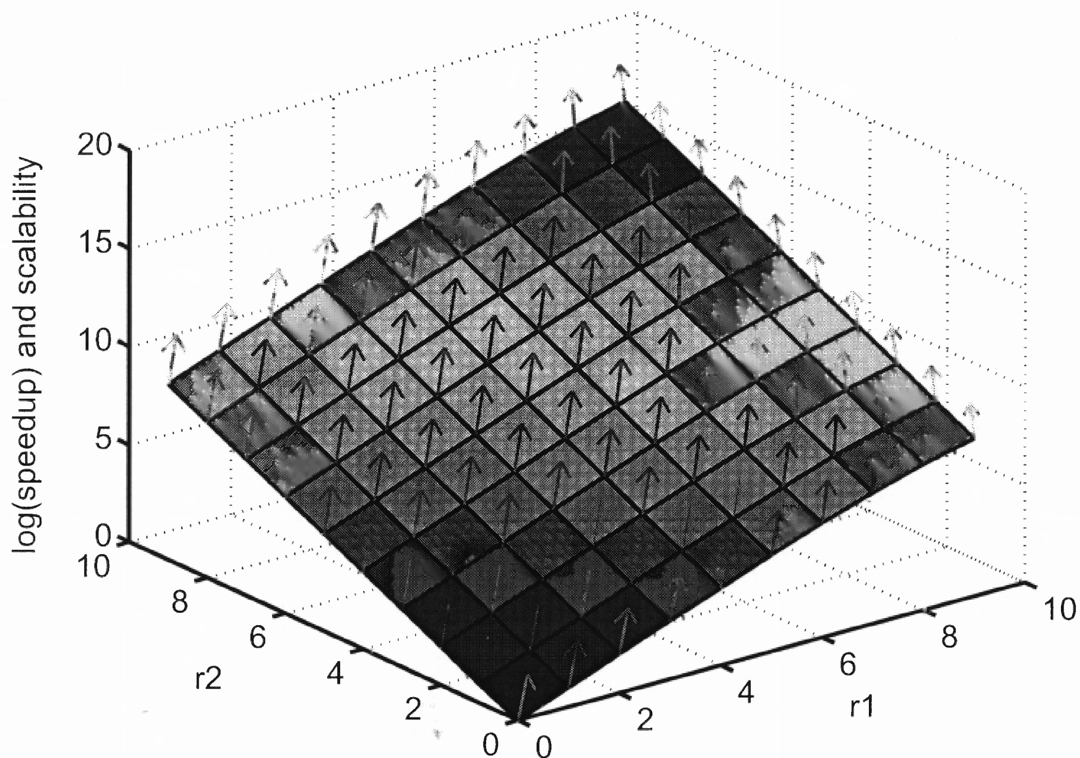
Figure 9.2 Scalability of a two-level cluster for matrix multiplication with fixed overall workload but the workload of level-1 SIs increases with increases in the size of level-1 nodes

9.4.3 Cases 3 and 4

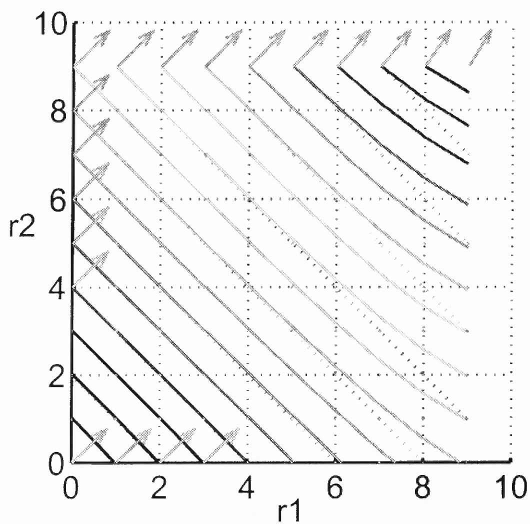
Assume that the workload w1 of the program in number of level-1 SIs is constant when the system is scaled up; also, the workload w_2 of primary (level-1) SIs increases in such a way that the increase in the total space demands for level-1 SDBs matches the size increase of the subcluster as a level-1 node (i.e., the number of PCs in the subcluster). Then, the overall problem size $m_1 \times m_2$ may increase. The result of our simulation for the scalability of the solution is shown in Fig. 9.3. In this simulation, $c_1 = 0.9$, $c_2 = 0.7$, $w_1 = 512$ and $w_2 = 8 \times n_2^{1.5}$.

Conversely, assume that the workload w_2 of a level-1 SIs is constant, and w1 and the overall workload increase with the number of subcluster nodes in such way that $w_2 = 512$ and $w_1 = 8 \times n_1^{1.5}$. The corresponding result for the scalability of the solution is shown in Fig. 9.4.

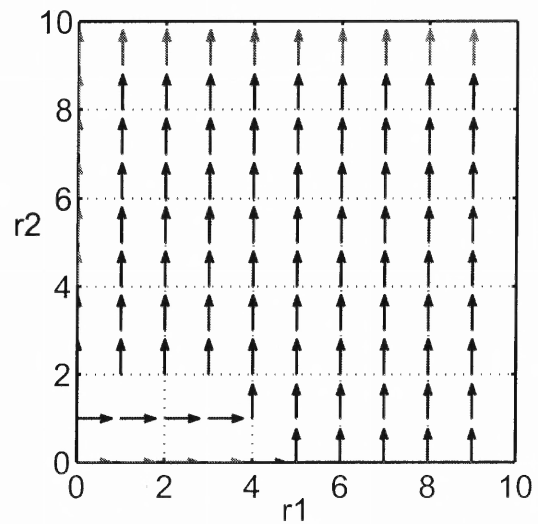
From Fig. 9.3 and Fig. 9.4, we can see that unlike the earlier cases, the optimal directions do not form a path to help us find the global optimal configuration. In fig. 9.3, among all the systems with $r = r_1 + r_2 = 3$, the optimal configuration is at $(r_1, r_2) = (3, 0)$ and the local scalability vector is close to (1, 1); the most effective direction to scale up a single dimension is (1, 0). However, the global optimal configuration for the immediately next size (i.e., $r = r_1 + r_2 = 4$) is at (0, 4). The same jump is also observed in Fig. 9.4. This indicates that even if the starting point is optimal, we cannot find a global optimal scaled up configuration by analyzing local scalability. The task of scaling up multi-dimensional cluster systems for optimal solution is not then simple.



(a) Scalability

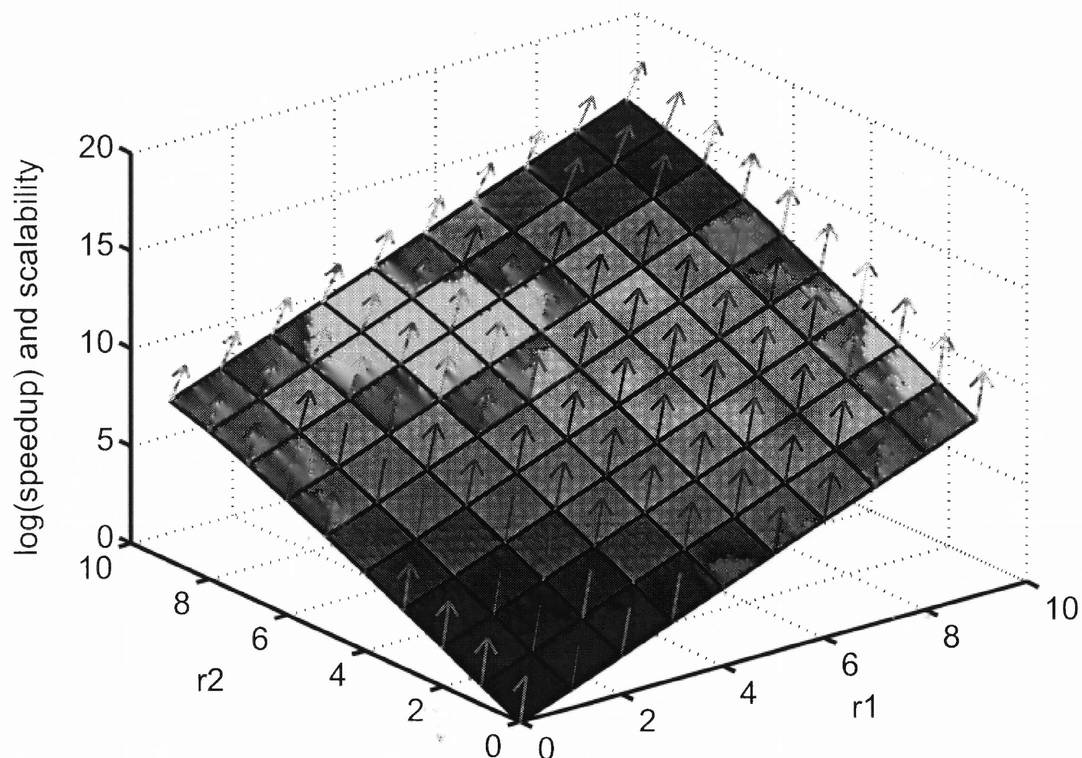


(b) log(speedup)

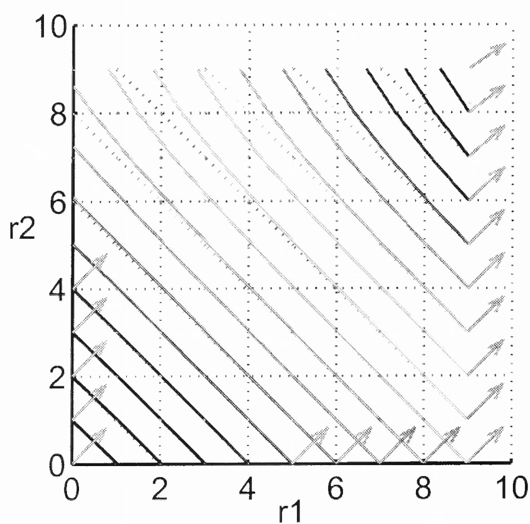


(c) the most effective direction to scale up

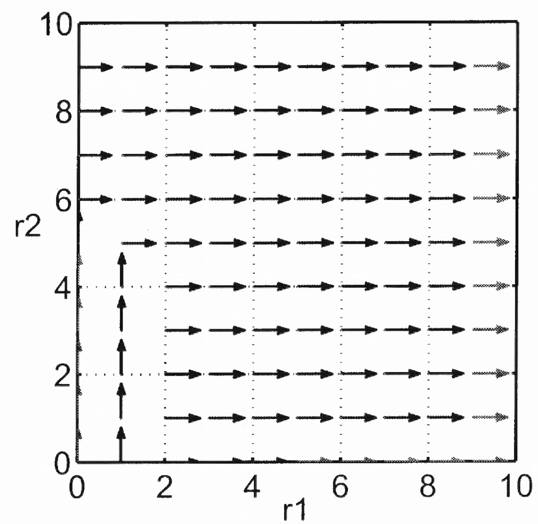
Figure 9.3 Scalability of a two-level cluster for matrix multiplication in which the workload of level-1 SIs increases with the size of level-1 nodes and w_1 is constant



(a) Scalability



(b) log(speedup)

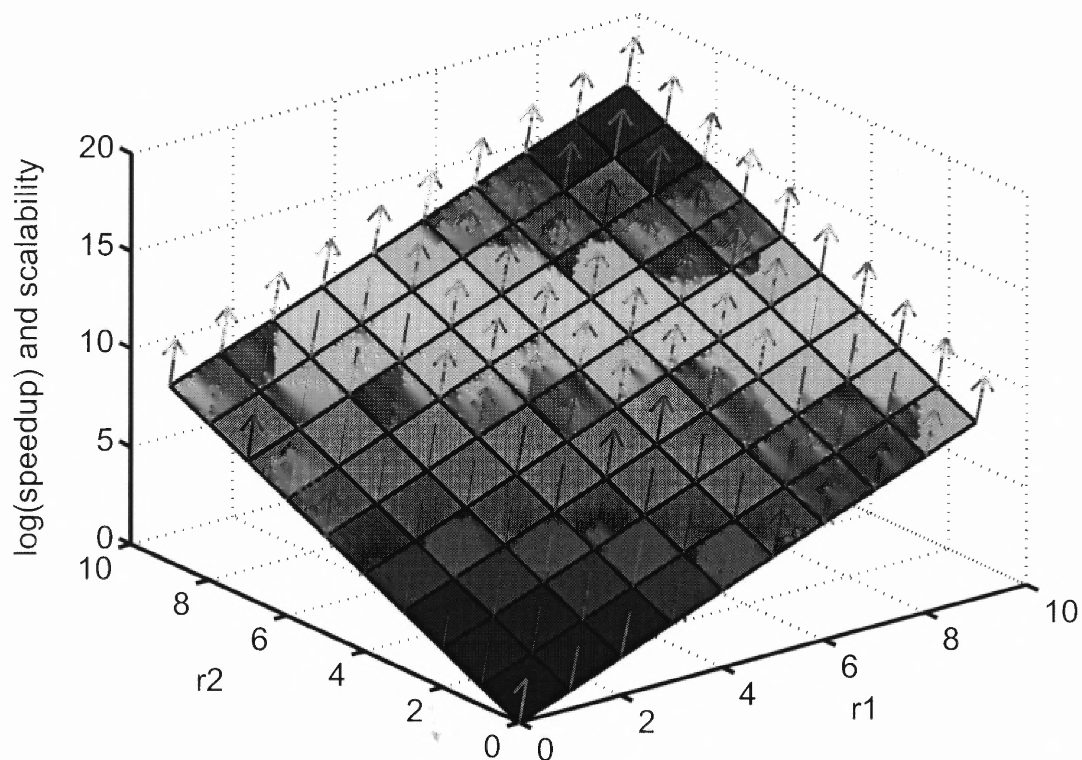


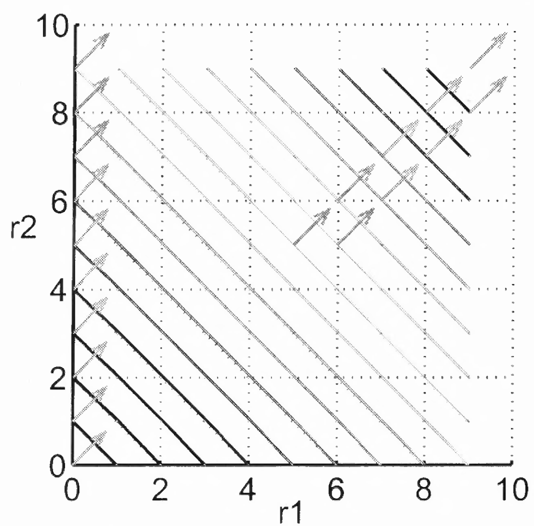
(c) the most effective direction to scale up

Figure 9.4 Scalability of a two-level cluster for matrix multiplication where the w_1 increases with the number of subcluster nodes and w_2 is constant

9.4.4 Case 5

Assume that the workload w_2 of primary (level-1) SIs increases in such a way that increases in the total space demand for level-1 SDBs matches increases in the subcluster size. Also, the workload w_1 of the program in number of level-1 SIs increases similarly (i.e., the overall problem size m1 * m2 may increase) when the system is scaled up. The result of our simulation for the scalability of the solution is shown in Fig. 9.5. In this simulation, $c_1 = 0.9$, $c_2 = 0.7$; $w_1 = 8 \times n_1^{1.5}$ and $w_2 = 8 \times n_2^{1.5}$. Since the workloads at different levels increase independently without any constraint among different dimensions and the workloads are always large enough, the systems always have an optimal configuration. The performance measures are almost identical between different configurations with the same overall system size.





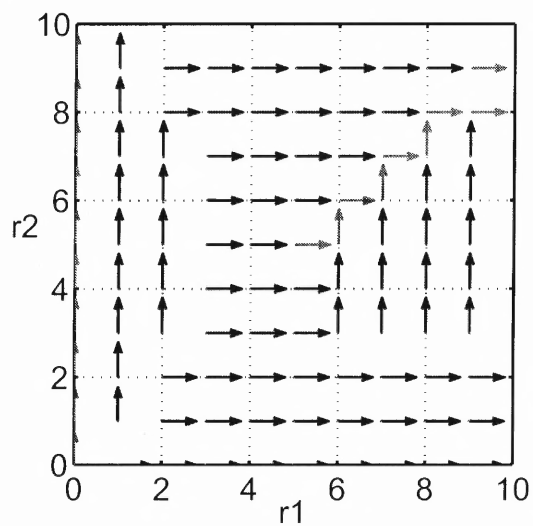


Figure 9.5 Scalability of a two-level cluster for matrix multiplication where w_1 and w_2 increase with the number of level-1 nodes and level-2 nodes, respectively

CHAPTER 10

CONCLUSIONS

In my research, an innovative parallel programming model named SPM was proposed for computations on PC clusters. This model uses standardized coarse-grain tasks named super-instructions (SIs) as the basic components for programming application software, distributing tasks on distributed computer systems and scheduling tasks in parallel at runtime.

The SPM model supports efficient load balancing at run time. The predictable characteristics of SIs make load balancing feasible. Super-programs (SPs) based on this model can efficiently execute on PC clusters. Following this model, a system can efficiently schedule tasks, balance the workload and fully utilize the computing capacity of each computer node. When the super-program has enough parallelism, the member nodes have a little chance to be idle. A couple of application programs, matrix multiplication in the scientific computing domain and mining association rules in the business intelligence computing domain, were developed to illustrate SPM. In all the experimental executions of these super-programs, the workloads were well balanced among the nodes. The scalable behavior of the approach was observed in our experiments.

In SPM, the parallelization of programs is separated from workload balancing. By asynchronously and dynamically issuing SIs to IEUs, VM can achieve load balancing efficiently. By adopting dynamic scheduling policies for SFs, SPM transfers the overhead of optimizing the execution into a management overhead of VM for scheduling, issuing and committing SIs. I will carry out experiments to show that the latter is kept low for coarse grained SIs. This removes the demand of load

balancing from parallel programs, thus making the development of parallel programs much easier, especially in heterogeneous PC cluster environments.

The SPM model supports also code portability. Since application programs are coded with the SIs for corresponding domains, as long as an efficient implementation exists for each SI on given computing platforms, code portability is guaranteed. The sample SPs were ported successfully on two different PC cluster systems. This process did not involve any modification of these SP codes except for installing and configuring the underlying PC clusters.

Thus, SPM can reuse many achievements in algorithm research and developed codes. All SIs are implemented by sequential procedures. Thus, all libraries can be directly used to develop SIs and SPs. This makes developing parallel programs easier and more efficient. SPM adopts a similar concept with sequential programming models, so it further decreases the effort of creating programs.

In SPM, all data are virtual entities. Each item can be represented with a set of distributed incarnated objects. SPM alleviates various restrictions on accessing data in parallel and eventually increases the effective parallelism in application programs. This reduces the chance of a node being idle while waiting for requested data and eventually improves the overall performance of parallel programs.

Scalability is an attribute of the applied hardware-software solution. It does not solely depend on either the target computer architecture or the algorithm used to solve the problem. It depends on many dimensions that cannot be expressed by a single scalar value. They include the configuration of the computer system and the features of the problem at hand. Currently, definitions of scalability can only characterize partial features of a solution when the system and/or problem are scaled up. A comprehensive scalability expression needs to use a vector as input. Each

value in the vector relates to a single feature change (i.e., a single dimension study). When mentioning the efficiency of scaled up solutions to improve the performance, we must give the direction (i.e., resource) of the change. For an optimally configured multi-level cluster, the scalabilities at all levels should be identical. However, a good solution may not work well with the optimal configuration, especially if such a cluster is built as general purpose. This is because the internal resources of PCs are not exchangeable. We cannot increase the amount of a particular resource by decreasing another type of resource. A type of resource may be over-equipped for a solution but be under-equipped for another solution. Therefore, no PC configuration in these dimensions fits all solutions and the scalabilities in these dimensions may not be the same. Programs developed under SPM match well configurable features of hierarchical PC clusters. The scalabilities of levels can be matched by reconfiguring the hierarchical structure. This approach guarantees that the efficiency obtained by scaling up the resource is higher than a log-linear lower bound.

APPENDIX

A PROOF OF THE OPTIMAL DIMENSION

Theorem: Assume an n-dimensional vector space $\mathfrak{V}$ where for any vector $\mathbf{v} \in \mathfrak{V}$ we have $\mathbf{v} = \sum_{i=1}^{n}(v_i \cdot \mathbf{e}_i)$; for $i = 1, 2, \ldots, n$, v_i is a non-negative real number and at least one of them is non-zero, and $\mathbf{e}_i$ form orthogonal normalized bases for this space. Also, assume a function $\mathbb{E} : \mathfrak{V} \Rightarrow \Re$ that maps a vector v in $\mathfrak{V}$ to a real value in the real numbers domain $\Re$ with $E(\mathbf{v}) = (\mathbf{v} \cdot \mathbf{u})/(\mathbf{v} \cdot \mathbf{w})$; $\mathbf{u}$ and $\mathbf{w}$ are two constant vectors in $\mathfrak{V}$ (i.e., $\mathbf{u} = \sum_{i=1}^{n}(u_i \mathbf{e}_i)$ and $\mathbf{w} = \sum_{i=1}^{n}(w_i \mathbf{e}_i)$). Finally, $x_i = u_i/w_i$.

1) If x_k is the maximum value in the set $\{x_i | i = 1, 2, \ldots, n\}$, then for the vector $\mathbf{y} = x_k \mathbf{e}_k$ the function $\mathbf{E}(\mathbf{y})$ yields the maximum value; i.e., $E(\mathbf{y}) \geqslant E(\mathbf{v})$ for $\forall \mathbf{v} \in \mathfrak{V}$.

2) Assume that there exist multiple values x_i in the set $\{x_i | i = 1, 2, \ldots, n\}$ that are all equal to the maximum value in this set (i.e., $x_{s_1} = x_{s_2} = \ldots = x_{s_k} > x_{s_{(k+1)}} \geqslant \ldots \geqslant x_{s_n}$ and the indices $\{s_i | i = 1, 2, \ldots, n\}$ are a permutation of $\{1, 2, \ldots, n\}$), then, there is a subspace $\mathfrak{V}^k$ that can be constructed with the corresponding base $\{\mathbf{e}_{s_i} | 1 \leqslant i \leqslant k\}$ (i.e., the arbitrary vector $\mathbf{v}^s$ in $\mathfrak{V}^k$ can be expressed as $\mathbf{v}^s = \sum_{i=1}^{k}(v_{s_i}^s \mathbf{e}_{s_i})$) so that in the entire subspace $\mathbf{v}^s$ the function $\mathbf{E}(\mathbf{y})$ is maximized.

Proof: Without loss of generality, assume that $x_1 \geqslant x_2 \geqslant \ldots \geqslant x_i \geqslant \ldots \geqslant x_n$, and also, $\mathbf{v}^k = \sum_{i=1}^{k}(v_i^k \mathbf{e}_i)$, where $v_i^k \geqslant 0$ for $i = 1, 2, \ldots, k$ and $k < n$, and at least one of v_i^k is greater than zero while $v_j^k = 0$ for $k < j \leqslant n$.

1) $E(\mathbf{v}^1) = \frac{(\mathbf{v}^1 \cdot \mathbf{u})}{(\mathbf{v}^1 \cdot \mathbf{w})} = \frac{(v_1^1 \cdot u_1)}{(v_1^1 \cdot w_1)} = x_1.$

For the arbitrary vector $\mathbf{v} = \sum_{i=1}^{n}(v_i\mathbf{e}_i)$:

$$E(\mathbf{v}) = \frac{(\mathbf{v}\cdot\mathbf{u})}{(\mathbf{v}\cdot\mathbf{w})} = \frac{\sum_{i=1}^{n}(v_iu_i)}{\sum_{i=1}^{n}(v_iw_i)} = \frac{\sum_{i=1}^{n}(v_ix_iw_i)}{\sum_{i=1}^{n}(v_iw_i)}$$
$$\leqslant \frac{\sum_{i=1}^{n}(v_ix_1w_i)}{\sum_{i=1}^{n}(v_iw_i)} = \frac{x_1 \times \sum_{i=1}^{n}(v_iw_i)}{\sum_{i=1}^{n}(v_iw_i)} = x_1 = E(\mathbf{v}^1)$$

$E(\mathbf{v}) \leqslant E(\mathbf{v}^1)$, so at $\mathbf{v}^1$ function $E(\mathbf{v})$ reaches its maximum value. The first clause is true.

2) If $x_1 = x_2 = \ldots = x_k > x_{k+1} \geqslant \ldots \geqslant x_n$, then for the arbitrary vector $\mathbf{v}^k$ in subspace $\mathfrak{V}^k$ which is constructed with $\{\mathbf{e}_i | 1 \leqslant i \leqslant k\}$, i.e., $\mathbf{v}^k = \sum_{i=1}^{k}(v_i^k\mathbf{e}_i)$:

$$E(\mathbf{v}^k) = \frac{(\mathbf{v}^k\cdot\mathbf{u})}{(\mathbf{v}^k\cdot\mathbf{w})} = \frac{\sum_{i=1}^{k}(v_i^ku_i)}{\sum_{i=1}^{n}(v_i^kw_i)} = \frac{\sum_{i=1}^{k}(v_i^kx_iw_i)}{\sum_{i=1}^{n}(v_i^kw_i)}$$
$$= \frac{\sum_{i=1}^{k}({v_i}^kx_1w_i)}{\sum_{i=1}^{n}(v_i^kw_i)} = \frac{x_1 \times \sum_{i=1}^{n}({v_i}^kw_i)}{\sum_{i=1}^{n}(v_i^kw_i)} = x_1 = E(\mathbf{v}^1)$$

Thus, function $E(\mathbf{v})$ maps all vectors $\mathbf{v}^k$ in subspace $\mathfrak{V}^k$ to the maximum value. Therefore, the second clause is true as well. □

BIBLIOGRAPHY

[1] J. Worlton, "Some Patterns of Technological Change in High-Performance Computers," in *Proc. ACM/IEEE Conf. SuperComp.*, Nov. 1988, pp. 312–320.

[2] M. Arlitt, D. Krishnamurthy, and J. Rolia, "Characterizing the Scalability of a Large Web-ased Shopping System," *ACM Trans. on Internet Technology.*, Vol. 1, no. 1, pp. 44–69, Aug. 2001.

[3] P. Bryant, "Levels of Computer Systems," *Comm. of ACM*, Vol. 9, no. 60, pp. 873–876, Dec. 1966.

[4] H. W. Lawson, "Function Distribution in Computer System Architectures," *ACM Computer Architecture News*, Vol. 4, no. 4, pp. 93–97, Jan. 1976.

[5] J. C. King, "Abstract Machines and Software Design," *ACM SIGPLAN Notice*, Vol. 8, no. 9, pp. 86–88, Jan. 1973.

[6] K. J. Gough, "Stacking Them up: a Comparison of Virtual Machines," in *Proc. 6th Australasian Computer Systems Architecture Conf. (Proc. ACM/IEEE Conf. on SC)*, Jan. 2001, pp. 55–61.

[7] B. B. Kristensen and K. Østerbye, "Conceptual Modeling and Programming Languages," *ACM SIGPLAN Notice*, Vol. 29, no. 9, pp. 81–90, Sept. 1994.

[8] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, "Towards Efficiency and Portability: Programming with the BSP Model," in *Proc. ACM Symp. on Paral. Algor. and Arch.*, June 1996, pp. 1–12.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2003.

[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, "SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory," *ACM SIGPLAN Notice*, Vol. 31, no. 9, pp. 210–220, Sept. 1996.

[11] D. A. Patterson, "Latency Lags Bandwith," *Comm. of ACM*, Vol. 47, no. 10, pp. 71–75, Oct. 2004.

[12] I. Sommerville, *Software Engineering*, 6th ed. Harlow, England: Pearson Edu., 2001.

[13] A. B. Bondi, "Characteristics of Scalability and Their Impact on Performance," in *Proc. 2nd Int'l Workshop on Software and Performance*, Sept. 2000, pp. 195–203.

[14] S. D. Galup, "A Training Program for Information Technology Organizations Transitioning to Enterprise-Wide Client/Server Computing," *ACM Computer Personnel*, Vol. 20, no. 2, pp. 27–40, April 1999.

[15] K. Hwang, *Advanced Computer Architecture - Parallelism, Scalability and Programmability.* New York, NY: McGraw-Hill Inc., 1993.

[16] R. W. Sebesta, *Concepts of Programming Languages*, 4th ed. Reading, MA: Addison-Wesley, Longman Inc., 1999.

[17] G. P. B. Gibbons, Y. Matias, and V. Ramachandran, "Can Shared-emory Model Serve as a Bridging Model for Parallel Computation?" in *Proc. ACM Symp. on Paral. Algor. and Arch.*, June 1997, pp. 72–83.

[18] L. G. Valiant, "A Bridging Model for Parallel Computation," *Comm. of ACM*, Vol. 33, no. 8, pp. 103–111, Aug. 1990.

[19] Y. Gurevich, "Sequential Abstract-State Machines Capture Sequential Algorithms," *ACM Trans. on Comput. Logic*, Vol. 1, no. 1, pp. 77–111, July 2000.

[20] A. Blass and Y. Gurevich, "Abstract State Machines Capture Parallel Algorithms," *ACM Trans. on Comput. Logic*, Vol. 4, no. 4, pp. 578–651, Oct. 2003.

[21] F. Ortin, J. M. Cueva, and A. B. Martinez, "Technical Correspondence: The Reflective nitrO Abstract Machine," *ACM SIGPLAN Notice*, Vol. 38, no. 6, pp. 40–49, June 2003.

[22] R. M. Fujimoto and W. B. Campbell, "Direct Execution Models of Processor Behavior and Performance," in *Proc. 19th Conf. Winter Simulation*, Dec. 1987, pp. 751–758.

[23] W. K. Giloi, "Programming Models and Their Interdependence with Parallel Architectures," in *Proc. Programming Models for Massively Parallel Computers*, June 1993, pp. 1–12.

[24] G. Bell and J. Gray, "What's Next in High-Performance Computing?" *Comm. of ACM*, Vol. 45, no. 2, pp. 91–95, Feb. 2002.

[25] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *ACM Computer Architecture News*, Vol. 30, no. 2, pp. 25–34, Oct. 2002.

[26] R. M. Russell, "The CRAY-1 Computer System," *Comm. of ACM*, Vol. 21, no. 1, pp. 63–72, Jan. 1978.

[27] C. L. Seitz, "The Cosmic Cube," *Comm. of ACM*, Vol. 28, no. 1, pp. 22–33, Jan. 1985.

[28] Bell and C. G. Multis, "A New lass of Multiprocessor Computers," *Science*, Vol. 228, pp. 452–457, Apr.25 1985.

[29] S. Microsystem, "Sun ClusterTM Architecture: a White Paper," in *Proc. IEEE Computer Society Int'l Worksh. Cluster Computing.*, Dec. 1999, pp. 331–338.

[30] W. C. Athas and C. L. Seitz, "Multicomputers: Message-assing Concurrent Computers." *Computer*, Vol. 21, no. 8, pp. 9–24, Aug. 1988.

[31] J. Adams and D. Vos, "Small-College Supercomputing: Building a Beowulf Cluster at a Comprehensive College," *ACM SIGCSE Bulletin*, Vol. 34, no. 1, pp. 411–415, Feb. 2002.

[32] S. Hadjiefthymiades and L. Merakos, "Enhanced Scheme for WWW Computing in Wireless Communication Environments," *ACM Computer Communication Review*, Vol. 29, no. 5, pp. 23–35, Oct. 1999.

[33] M. L. Kherfi, D. Ziou, and A. Bernardi, "Image Retrieval from the World Wide Web: Issues, Techniques, and Systems," *ACM Computing Surveys*, Vol. 36, no. 1, pp. 35–67, March 2004.

[34] A. Snavely, G. Chun, H. Casanova, R. F. V. der Wijngaart, and M. A. Frumkin, "Special Section on Grid Computing: Benchmarks for Grid Computing: a Review of Ongoing Efforts and Future Directions," *ACM Performance Evaluation Review*, Vol. 30, no. 4, pp. 27–32, March 2003.

[35] H. Casanova, "Technical Columns: Distributed Computing Research Issues in Grid Computing," *ACM SIGACT News*, Vol. 33, no. 3, pp. 50–70, Sept. 2002.

[36] F.-D. Kübler and F. Lücking, "Cluster Oriented Architecture for the Mapping of Parallel Processor Networks to High Performance Applications," in *Proc. 2nd int'l Conf. SuperComp.*, June 1988, pp. 197–189.

[37] M. S. Keller, "The Cutting Edge: Moving to SMP," *Linux Journal*, March 2000.

[38] P. M. Chen and E. K. Lee, "Striping in a RAID Level 5 Disk Array," *ACM Performance Evaluation Review*, Vol. 23.

[39] D. Stodolsky, M. Holland, W. V. Courtright, and G. A. Gibson, "Parity Logging Disk rrays," *ACM Trans. on Comput. Systems*, Vol. 12, no. 3, pp. 206–235, Aug. 1994.

[40] T. G. Mattson, "High Performance Computing at Intel: the OSCAR Software Solution Stack for Cluster Computing," in *Proc. IEEE/ACM Int'l Symp. on Cluster Computing and the Grid*, May 2001, pp. 22–25.

[41] J. L. Wolf and P. S. Yu, "On Balancing the Load in a Clustered Web Farm," *ACM Trans. on Internet Technology.*, Vol. 1, no. 2, pp. 231–261, Nov. 2001.

[42] Y.-F. Sit, C.-L. Wang, and F. Lau, "Cyclone: A High-Performance Cluster-Based Web Server with Socket Cloning," *Cluster Computing*, Vol. 7, no. 1, pp. 21–37, Jan. 2004.

[43] M. Andreolini, E. Casalicchio, M. Colajanni, and M. Mambelli, "A Cluster-Based Web ystem Providing Differentiated and Guaranteed Services," *Cluster Computing*, Vol. 7, no. 1, pp. 7–19, Jan. 2004.

[44] V. L. Chung and C. S. McDonald, "The Development of a Distributed Capability System for VLOS," *Aust. Comput. Sci. Comm.*, Vol. 24, no. 3, pp. 57–64, Jan. 2002.

[45] A. S. Z. Belloum, E. C. Kaletas, A. W. V. Halderen, H. Afsarmanesh, L. O. Hertzberger, and A. J. H. Peddemors, "A Scalable Web Server Architecture," *World Wide Web*, Vol. 5, no. 1, pp. 5–23, May 2002.

[46] Y. Saito, B. N. Bershad, and H. M. Levy, "Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-Based Mail Service," *ACM Trans. on Comput. Systems*, Vol. 18, no. 3, pp. 298–332, Aug. 2000.

[47] T. Heath, R. P. Martin, and T. D. Nguyen, "Improving Cluster Availability Using Workstation Validation," *ACM Performance Evaluation Review*, Vol. 30, no. 1, pp. 217–227, June 2002.

[48] V. B. Mendiratta, "Reliability Analysis of Clustered Computing Systems," in *Proc. 9th Int'l Symp. Software Reliability Eng.*, Nov. 1998, pp. 268–272.

[49] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray, "The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability," in *Proc. 28th Int'l Symp. Fault-Tolerant Comput.*, June 1998, pp. 422–431.

[50] Y. Zhou, P. M. Chen, and K. Li, "Fast Cluster Failover Using Virtual Memory-Mapped Communication," in *Proc. 13th Int'l Conf. SuperComput.*, May 1999, pp. 373–382.

[51] D. Nichols, "Using Idle Workstations in a Shared Computing Environment," *ACM Operating Systems Review*, Vol. 21, no. 5, pp. 5–12, Nov. 1987.

[52] P. Enslow, "Multiprocessor Organization-a Survey," *ACM Computing Surveys*, Vol. 9, no. 1, pp. 103–129, Jan. 1977.

[53] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps, "The Starfire SMP Interconnect," in *Proc. ACM/IEEE Conf. SuperComp. (CDROM)*, Nov. 1997, pp. 1–20.

[54] G. F. Pfister, *In Search of Clusters*, 2nd ed. Upper Saddle River,NJ: Prentice Hall Inc., 1998.

[55] K. Wada, S. Yamagiwa, and M. Fukuda, "High Performance Network of PC Cluster Maestro," *Cluster Computing*, Vol. 5, no. 1, pp. 33–42, Jan. 2002.

[56] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, no. 3, pp. 261–322, Sept. 1989.

[57] D. R. Cols, "Business-Structured Client/Server: an Architecture for Distributed Applications," in *Proc. Conf. Centre for Advanced Studies on Collaborative Research: software engineering*, Vol. 1, Oct. 1993, pp. 41–53.

[58] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon, "Requirements for and Evaluation of RMI Protocols for Scientific Computing," in *Proc. ACM/IEEE Conf. SuperComp. (CDROM)*, Nov. 2000, pp. 1–26.

[59] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli, "A Formal Approach for Designing CORBA-Based Applications," *ACM Trans. on Software Engineering and Methodology*, Vol. 12, no. 2, pp. 107–151, April 2003.

[60] Pope and Alan, *The CORBA Reference Guide : Understanding the Common Object Request Broker Architecture.* Reading, Mass.: Addison-Wesley, 1998.

[61] F. E. Redmond, *Dcom: Microsoft Distributed Component Object Model*, 3rd ed. Foster City, CA: IDG Books Worldwide, Inc., Aug. 1997.

[62] C. Munoz and J. Zalewski, "Archictecture and Performance of Java-Based Distributed Object Models: CORBA vs RMI," *Real-Time Systems*, Vol. 21, no. 1-2, pp. 43–75, July 2001.

[63] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielman, C. Jacobs, and R. Hofman, "Efficient Java RMI for Parallel Programming," *ACM Trans. on Programming Languages and Systems*, Vol. 23, no. 6, pp. 747–775, Nov. 2001.

[64] Grosso and William, *Java RMI.* Sebastopol, CA: O'Reilly, 2002.

[65] B. H. H. Juurlink and H. A. G. Wijshoff, "A Quantitative Comparison of Parallel Computation Models," *ACM Trans. on Comput. Systems*, Vol. 16, no. 3, pp. 271–318, Aug. 1998.

[66] D. Skillicorn and S. Pelagatti, "Building Programs in the Network of Tasks Model," in *Proc. ACM Symp. on Applied Computin.* ACM, March 2000, pp. 248–254.

[67] V. Blanco, J. A. González, C. León, C. Rodríguez, G. Rodríguez, and M. Printista, "Predicting the Performance of Parallel Programs," *Parallel Computing*, Vol. 30, no. 3, pp. 337–356, March 2004.

[68] M. Kühnemann, T. Rauber, and G. Rünger, "Performance Modelling for Task-Parallel Programs," in *Performance Analysis and Grid Computing.* Kluwer Academic Publishers, Jan. 2004, pp. 77–91.

[69] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* Redwood City, CA: Benjamin/Cummings Inc., 1994.

[70] D. B. Skillicorn and D. Talia, "Models and Languages for Parallel Computation," *ACM Computing Surveys*, Vol. 30, no. 2, pp. 123–169, June 1998.

[71] W. W. Gropp and E. Lusk, "TA Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters," in *Programming Models for Massively Parallel Computers*, Oct. 1995, pp. 2–7.

[72] D. W. Walker, "Portable Programming within a Message-Passing Model: the FFT as an Example," Vol. 2, pp. 1438–1450, 1989.

[73] S. A. Ward and R. H. Halstead, "A Syntactic Theory of Message Passing," *Journal of the ACM*, Vol. 27, no. 2, pp. 365–383, April 1980.

[74] E. D. Brooks and K. H. Warren, "A Study of Performance on SMP and Distributed Memory Architectures Using a Shared Memory Programming Model," in *Proc. ACM/IEEE Conf. SuperComp. (CDROM)*, Nov. 1997, pp. 1–16.

[75] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *ACM SIGPLAN Notice*, Vol. 25, no. 3, pp. 168–176, Feb. 1990.

[76] G. Fritsch, W. Henning, H. Hesenuer, R. Klar, C. U. Linster, C. w. Oehlrich, P. Schlenk, and J. Vokert, "Distributed Shared Memory Multiprocessor Architecture MEMSY for High Performance Paralel Computations," *ACM Computer Architecture News*, Vol. 17, no. 6, pp. 22–35, Dec. 1989.

[77] B. Manchek. PVM List of Frequently Asked Questions. [Online]. Available: http://www.netlib.org/pvm3/faq_html/node5.html (Accessed on Sept. 2004)

[78] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Journal of the ACM*, Vol. 42, no. 1, pp. 124–142, Jan. 1995.

[79] K. H. Lee and C. H. Lam, "Massage-Passing Controllerr for a Shared-Memory Multiprocessor," *ACM Computer Architecture News*, Vol. 17, no. 6, pp. 142–149, Dec. 1989.

[80] A. C. Klaiber and H. M. Levy, "A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs," *ACM Computer Architecture News*, Vol. 22, no. 2, pp. 94–105, April 1994.

[81] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol. 2, no. 4, pp. 315–339, Dec. 1990.

[82] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker, "A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment," ORNL, Tech. Rep. TM-12231, Feb. 1993.

[83] W. Gropp, E. Lusk, and A. Skkjellum, *Using MPI: Portable Parallel Processing with the Message-Passing Interface.* Cambridge, MA: MIT Press, 1994.

[84] MPICH Home Page. [Online]. Available: http://www-unix.mcs.anl.gov/mpi/mpich/ (Accessed on Sept. 2004)

[85] LAM Home Page. [Online]. Available: http://www.lam-mpi.org/ (Accessed on Sept. 2004)

[86] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, "Questions and Answers about BSP," *J. of Sci. Programming*, Vol. 6, no. 3, pp. 249–274, Fall 1997.

[87] C. W. Keßler, "NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model," *The Journal of Supercomputing*, Vol. 17, no. 3, pp. 245–262, Nov. 2000.

[88] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Abstractions for Dynamic Data Distribution," in *Proc. IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[89] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder, "The High-Level Parallel Language ZPL Improves Productivity and Performance," in *Proc. IEEE Int'l Workshop on Productivity and Performance in High-End Computing*, 2004.

[90] R. W. Numrich and J. Reid, "Co-array Fortran for Parallel Programming," *ACM SIGPLAN Fortran Forum*, Vol. 17, no. 2, pp. 1–31, Aug. 1998.

[91] W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," George Washington University, Center for Computing Sciences, Tech. Rep. Technique Reports CCS-TR-99-157, May 1999.

[92] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Language Specification. [Online]. Available: http://www.gwu.edu/~upc/docs/upc_spec_1.1.1.pdf (Accessed on Sept. 2004)

[93] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, P. C. D. Gay, and A. Aiken, "Titanium: A High-Performance Java Dialect," in *Proc. ACM Workshop on Java for High-Performance Network Computing*, Feb. 1998, pp. 1–12.

[94] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP.* Morgan Kaufmann, 2000.

[95] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *ACM SIGPLAN Notice*, Vol. 33, no. 5, pp. 212–223, May 1998.

[96] Supercomputing Technologies Group. Cilk-5.3 Reference Manual. [Online]. Available: http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf (Accessed on Sept. 2004)

[97] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an Efficient Multithreaded Runtime System," *ACM SIGPLAN Notice*, Vol. 30, no. 8, pp. 207–216, Aug. 1995.

[98] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM*, Vol. 46, no. 5, pp. 720–748, Sept. 1999.

[99] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, "Algorithms: On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs," in *Proc. ACM Symp. on Paral. Algor. and Arch.*, June 2004, pp. 133–144.

[100] G. E. Kaiser and B. Hailpern, "An Object-Based Programming Model for Shared Data," *ACM Trans. on Programming Languages and Systems*, Vol. 14, no. 2, pp. 201–264, April 1992.

[101] P. Wegner, "Dimensions of Object-Based Language Design," *ACM SIGPLAN Notice*, Vol. 22, no. 12, pp. 168–182, Dec. 1987.

[102] L. V. Kale and S. Krishnan, "CHARM++: a Portable Concurrent Object Oriented System Based on C++," *ACM SIGPLAN Notice*, Vol. 28, no. 10, pp. 91–108, Oct. 1993.

[103] Charm++ Programming Language Manual. Parallel Programming Laboratory, Department of Computer Science, University Illinois at Urbana-Champaign. [Online]. Available: http://charm.cs.uiuc.edu/manuals/pdf/charm++.pdf (Accessed on Sept. 2004)

[104] M. O. Neary and P. Cappello, "Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing," in *Proc. ACM-ISCOPE Conf. on Java Grande*, Nov. 2002, pp. 56–65.

[105] I. Vakalis and E. de Doncker, "Parallel Global Adaptive Integration and Dynamic Load Balancing on Loosely Coupled Systems," in *Proc. ACM Symp. on Applied Computin: States of the Art and Practice*, Feb. 1993, pp. 554–561.

[106] D. Jin and S. G. Ziavras, "A Super-Programming Technique for Large Sparse Matrix Multiplication on PC Clusters," *IEICE Trans. Inform. Systems*, Vol. E87-D, no. 7, pp. 1774–1781, July 2004.

[107] R. Agrawal and R. Srikant, "Mining Association Rules between Sets of Items in Large Databases," in *Proc. ACM Special Interest Group on Management of Data*, May 1993, pp. 207–216.

[108] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," in *Proc. ACM Special Interest Group on Management of Data*, May 2000, pp. 1–12.

[109] T. Shintani and M. Kitsuregawa, "Hash Based Parallel Mining Algorithms for Mining Association Rules," in *Proc. IEEE Conf. on Paral. and Distr. Inform. Systems*, Dec. 1996, pp. 19–30.

[110] D. W. Cheung, K. Hu, and S. Xia, "Asynchronous Parallel Algorithm for Mining Association Rules on a Shared-Memory Multi-Processors," in *Proc. ACM Symp. on Paral. Algor. and Arch.*, July 1998, pp. 279–288.

[111] E. Han, G. Karypis, and V. Kumar, "Scalable Parallel Data Mining for Association Rules," pp. 277–288, May 1997.

[112] R. Agrawal and J. Shafer, "IEEE Trans. Knownledge and Data Eng." *Comm. of ACM*, Vol. 8, no. 6, pp. 962–969, Dec. 1996.

[113] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li, "Parallel Data Mining for Association Rules on Shared-Memory Multi-Processors," in *Proc. ACM/IEEE Conf. SuperComp. (CDROM)*, July 1996, pp. 1–18.

[114] D. Jin and S. G. Ziavras, "Load Balancing on PC Clusters with the Super-Programming Model," in *Workshop on Compile/Runtime Techniques for Parallel Computing (in conjunction with the International Conference on Parallel Processing-ICPP03)*, Kaohsiung, Taiwan, Oct. 6-9 2003, pp. 63–70.

[115] D. Lin and Z. M. Kedem, "Pincer_Serch: An Efficient Algorithm for Discovering the Maximum Frequent Set," *IEEE Trans. Knownledge and Data Eng.*, Vol. 14, no. 3, pp. 553–566, Dec. 2002.

[116] R. Agrawal, T. Imielinski, and A. Swami, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proc. 20th Very Large Database Conf.*, Sept. 1994.

[117] I. S. Duff, M. A. Heroux, and R. Pozo, "An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum," *ACM Trans. on Mathematical Software*, Vol. 28, no. 2, pp. 239–267, June 2002.

[118] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli, "Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User-Level Interface," *ACM Trans. on Mathematical Software*, Vol. 23, no. 3, pp. 379–401, Sept. 1997.

[119] D. Jin and S. G. Ziavras, "A Super-Programming Approach for Mining Association Rules in Parallel on PC Clusters," *IEEE Trans. Paral. Distr. Syst.*, Vol. 15, no. 9, pp. 783–794, Sept. 2004.

[120] D. Jin and S. G. Ziavras, "A Super-Programming Technique for Large Sparse Matrix Multiplication on PC Clusters," in *2nd Workshop on Hardware/Software*

Support for High performance Scientific and Engineering Computing (SHPSEC-03) [in conjunction with the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT-03)], New Orleans, LA, Sep. 27–Oct. 1 2003, pp. 1–12.

[121] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix Multiplication on Heterogeneous Platforms," *IEEE Trans. Paral. Distr. Syst.*, Vol. 12, no. 10, pp. 1033–1051, Oct. 2001.

[122] D. W. Cheung, S. D. Lee, and Y. Xiao, "Effect of Data Skewness and Workload Balance in Parallel Data Mining," *IEEE Trans. Knownledge and Data Eng.*, Vol. 14, no. 3, pp. 498–514, Dec. 2002.

[123] Synthetic Database Generator. [Online]. Available: http://www.almaden.ibm.com/cs/quest/syndata.html (Accessed on Sept. 2004)

[124] E. W. Felten and D. McNamee, "Improving the Performance of Message-Passing Applications by Multithreading," in *Proc. Scalable High Performance Comput. Conf.*, April 1992, pp. 84–89.

[125] D. Jin and S. G. Ziavras, "Modeling Distributed Data Representation and its Effect on Parallel Data Accesses," *J. Paral. Distr. Comput.,Special Issue on Design and Performance of Networks for Super-, Cluster-, and Grid-Computing*, (accepted for publication).

[126] D. Jin and S. G. Ziavras, "Scalability: A Multidimensional Entity," *IEEE Trans. Paral. Distr. Syst.*, (submited on April 2005).

[127] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran, "An Approach to Scalability Study of Shared Memory Parallel Systems," *ACM Performance Evaluation Review*, Vol. 22, no. 1, pp. 171–180, May 1994.

[128] A. H. Karp and H. P. Flatt, "Measuring Parallel Processor Performance," *Comm. of ACM*, Vol. 33, no. 5, pp. 539–543, May 1990.

[129] X.-H. Sun and J. Gustafson, "Towards a Better Parallel Performance Metric," *Parallel Computing*, Vol. 17, pp. 1093–1109, 1991.

[130] D. R. Law, "Scalable Means More than More: a Unifying Definition of Simulation Scalability," in *Proc. 30th Conf. on Winter Simulation*, Washington, D.C., Dec. 1998, pp. 781–788.

[131] M. D. Hill, "What Is Scalability?" *ACM Computer Architecture News*, Vol. 18, no. 4, pp. 18–21, Dec. 1990.

[132] D. Nussbaum and A. Agarwal, "Scalability of Parallel Machines," *AC*, Vol. 34, no. 3, pp. 57–61, Mar. 1991.

[133] R. Deters, "Scalability and Information Agents," *ACM Applied Computing Review*, Vol. 9, no. 3, pp. 13–20, Sept. 2001.

[134] G. Brataas and P. Hughes, "Exploring Architectural Scalability," *ACM Software Engineering Notes*, Vol. 29, no. 1, pp. 125–129, Jan. 2004.

[135] J. L. Gustafson, "Reevaluating Amdahl's Law," *Comm. of ACM*, Vol. 31, no. 5, pp. 18–21, May 1988.

[136] S. G. Ziavras, "On the Problem of Expanding Hypercuber Based System," *J. Paral. Distr. Comput.*, Vol. 16, no. 1, pp. 41–53, Spet. 1992.

[137] S. G. Ziavras, "RH: A Versatile Family of Reduced Hypercube Interconnection Networks," *IEEE Trans. Paral. Distr. Syst.*, Vol. 5, no. 11, pp. 1210–1220, Nov. 1994.

[138] J. S. Vetter and M. O. McCracken, "McCracken, Statistical Scalability Analysis of Communication Operations in Distributed Applications," *ACM SIGPLAN Notice*, Vol. 36, no. 7, pp. 123–132, June 2001.

[139] M. D. Greenberg, *Advanced Engineering Mathematics.* New Jersey: Prentice-Hall, 1988.