INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Performance of Parallel Applications on a Network of Workstations

by

Anatol Piotrowski, B.C.S.

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

15 July, 1997

© 1997, Anatol Piotrowski

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26974-4

Canada

The undersigned hereby recommend to
the faculty of Graduate Studies and Research
acceptance of the thesis,

**“Performance of Parallel Applications on a Network of
Workstations”**

submitted by

Anatol Piotrowski B.C.S.

in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

Abstract

Performance of parallel applications on a network of workstations (NOW) can be different from that on a parallel machine. While such NOW-based parallel systems offer the price advantage over a parallel system, communication overhead is substantially higher. In addition, there are other significant differences between the two types of systems. Therefore, it is important to understand the factors that influence the performance of parallel applications on NOW-based systems. To gain this understanding, we study the performance of a matrix multiplication application as well as generic applications on a NOW-based system that uses PVM software. We have used a network of heterogeneous workstations that uses two networks: Ethernet (10Mbps) and Fast Ethernet (100Mbps). We study several self-scheduling load sharing algorithms that use fixed, variable and adaptive task granularity and do not employ task migration. We report the impact of various system/workload parameters and mapping on the performance of the matrix multiplication as well as generic workload applications.

Acknowledgments

I would like to express my sincere gratitude and appreciation to Prof. Sivarama Dandamudi for his guidance and support during this research.

I gratefully acknowledge the financial support provided by the National Sciences and Engineering Research Council and Carleton University.

I would also like to thank everyone who could have made my life much more difficult but did not.

Table of Contents

ABSTRACT.....	III
ACKNOWLEDGMENTS.....	IV
TABLE OF CONTENTS.....	V
LIST OF FIGURES	X
LIST OF TABLES	XIII
CHAPTER I: INTRODUCTION	1
1.1 WHY NOWs ARE BECOMING ATTRACTIVE SUBSTITUTES FOR MPPS.....	3
1.2 RELATED WORK.....	4
1.3 OBJECTIVES	6
1.4 CONTRIBUTIONS OF THE THESIS	7
1.5 THESIS ORGANIZATION	8
CHAPTER II: EXPERIMENTAL ENVIRONMENT.....	9
2.1 PVM.....	9
2.1.1 What is Provided by PVM	11
2.2 NOW-SYSTEM DESCRIPTION	13

CHAPTER III: FIXED GRANULARITY ALGORITHMS	15
3.1 MATRIX MULTIPLICATION	16
3.2 MATRIX-BASED ALGORITHMS	17
3.2.1 No load sharing (NLS) algorithm	17
3.2.2 Load sharing algorithms	17
3.2.2.1 Broadcast	18
3.2.2.2 Send	19
3.2.2.3 Replicate	20
3.2.2.4 Store	21
3.2.2.5 Hierarchical	23
3.3 SUBMATRIX-BASED ALGORITHMS	25
3.3.1 SubMatrix algorithm	26
3.3.2 CachedSubMatrix algorithm	26
3.4 GENERIC WORKLOAD SIMULATION ALGORITHMS	27
3.4.1 Implementation details of Generic Workload	28
3.4.2 Algorithms implemented	33
3.5 RESULTS AND DISCUSSION	34
3.5.1 Load sharing versus no load sharing	36
3.5.2 Fast master versus slow master	38
3.5.3 Work versus no work	40
3.5.4 Impact of network	41

3.5.5 Task Granularity	42
3.5.6 The effect of sending the second matrix	44
3.5.7 Relative performance of algorithms	45
3.5.7.1 Broadcast versus Send	45
3.5.7.2 Replicate versus Send	47
3.5.7.3 Store versus Send	48
3.5.7.4 SubMatrix versus CachedSubMatrix algorithm.....	50
3.5.7.5 Matrix-based versus Submatrix-based algorithms	51
3.6 RESULTS AND DISCUSSION OF EXPERIMENTS WITH BACKGROUND LOAD	52
3.6.1 Broadcast versus Send under background load.....	53
3.6.2 Replicate versus Send under background load	54
3.6.3 Store versus Send under background load.....	55
3.6.4 Matrix-based versus SubMatrix-based algorithms under background load ...	56
3.7 RESULTS AND DISCUSSION OF GENERIC WORKLOAD	57
3.7.1 Impact of computation to communication ratio	58
3.7.2 Impact of background load	59
3.7.3 Impact of random background load.....	60
3.7.4 Relative performance of algorithms	61
3.7.4.1 Broadcast versus Send	61
3.7.4.2 Broadcast versus Replicate.....	62
CHAPTER IV: VARIABLE GRANULARITY ALGORITHMS	64

4.1	DESCRIPTION OF VARIABLE TASK GRANULARITY ALGORITHMS	65
4.1.1	Guided self-scheduling	65
4.1.2	Factoring	67
4.1.3	Trapezoidal self-scheduling	67
4.1.4	Adaptive task granularity algorithm.....	68
4.2	GENERIC WORKLOAD	69
4.3	RESULTS AND DISCUSSION.....	70
4.3.1	Comparison of fixed and variable granularity algorithms.....	70
4.3.1.1	Variable task granularity algorithms versus Send	70
4.3.1.2	Comparison of Adaptive and Send algorithms	72
4.3.2	Sensitivity to task granularity	74
4.3.2.1	Guided self-scheduling and task granularity.....	74
4.3.2.2	Trapezoidal self-scheduling and task granularity	76
4.3.2.3	Factoring self-scheduling and task granularity	78
4.3.2.4	Adaptive algorithm and task granularity	79
4.3.3	Performance under background load	81
4.3.3.1	Performance of variable task granularity algorithms.....	81
4.3.3.2	Performance of the Adaptive algorithm under a background load	82
4.4	RESULTS FOR A GENERIC WORKLOAD.....	83
4.4.1	Impact of the computation to communication ratio.....	85
4.4.2	Impact of background load	86

4.4.3	Impact of a random background load (versus 1/3 load)	87
4.4.4	Sensitivity to task granularity	88
4.4.5	Relative performance	88
CHAPTER V: CONCLUSIONS		90
5.1	CONCLUSIONS FOR MATRIX MULTIPLICATION APPLICATION	90
5.2	GENERAL CONCLUSIONS	92
5.3	FURTHER RESEARCH	96
APPENDIX 1: COMPLEXITY FOR FIXED GRANULARITY		
ALGORITHMS.....		97
APPENDIX 2: PVM COMMUNICATION PRIMITIVES USED.....		98
REFERENCES.....		101

List of Figures

Figure 1: NOW system used for experiments	13
Figure 2: General structure of the algorithms implemented	15
Figure 3: Hierarchical organization implemented for 7 nodes with branching factor 3	23
Figure 4: Comparison of the performance of matrix multiplication. NLS means no load sharing algorithm, LS means load sharing algorithm	35
Figure 5 : Performance of No Load Sharing (NLS) versus Send	36
Figure 6 : The performance of the Send and CachedSubMatrix algorithms when the master's node works on the multiplication ("Work") and when not ("No work").	38
Figure 7 : Performance sensitivity to task granularity	42
Figure 8: Performance sensitivity of Broadcast and Send algorithms to distributing to all slaves second matrix versus distributing to all slaves first matrix	44
Figure 9: Performance of Broadcast and Send algorithms	46
Figure 10: Performance of Replicate and Send algorithms (Master works)	47
Figure 11: Performance comparison of Store and Send algorithms	49
Figure 12: Performance comparison of SubMatrix and CachedSubMatrix algorithms (divided into 3x3 submatrices)	50
Figure 13: Performance of CachedSubMatrix and Broadcast algorithms	51
Figure 14: Performance of Broadcast and Send algorithms (5 columns granularity)	53

Figure 15: Performance of Replicate and Send algorithms under background load	54
Figure 16: Performance comparison of Store and Send algorithms (task granularity is 5 columns).....	55
Figure 17: Performance of Matrix based algorithms versus SubMatrix-based.....	56
Figure 18: Performance of generic workload algorithm (Full speed - no background load).....	58
Figure 19: Performance of Broadcast algorithm at 50:1 ratio with and without background load.....	59
Figure 20: Performance of Broadcast algorithm (50:1) under background load	60
Figure 21: Performance of Broadcast and Send algorithms	61
Figure 22: Performance of Broadcast and Replicate algorithm at 50:1 ratio	62
Figure 23 :Task sizes for the fixed and variable task granularity algorithms (for a 70 column matrix)	66
Figure 24: Performance of variable task granularity algorithms	70
Figure 25: Performance of Send and Adaptive algorithms.....	72
Figure 26: Performance sensitivity of the Guided self-scheduling algorithm to parameter G	75
Figure 27: Performance sensitivity of Trapezoidal self-scheduling algorithm to parameter T	77
Figure 28: Performance sensitivity of Factoring algorithm to parameter F	78
Figure 29: Performance of Adaptive algorithm with different granularity	80

Figure 30: Performance of variable task granularity algorithms on fast network	81
Figure 31: Performance comparison of Send and Adaptive algorithms	83
Figure 32: Performance of variable task granularity algorithms (no background load)	85
Figure 33: Performance of variable task granularity algorithms at 50:1 ratio	86
Figure 34: Performance of variable task granularity algorithms at 50:1 ratio	87
Figure 35: Comparison of algorithm performance.....	89

List of Tables

Table 1: Comparison of performance of the nodes used. Time that it takes for the node to multiply 720×720 matrices by itself is shown.....	34
Table 2: Comparison of speedups achieved on with different computation to communication ratio	58
Table 3: Corresponding parameters between array multiplication algorithms and generic workload algorithms.....	84
Table 4: Equations for number of data items sent and number of messages send by four fixed granularity algorithms.	97
Table 5: Bandwidths obtained on NOW-system during experiments in Mbytes/sec.	100

Chapter I

Introduction

Exploitation of networked workstations for parallel processing has received substantial interest with the availability of Parallel Virtual Machine (PVM) [12, 23] and Message Passing Interface (MPI) [10, 13, 25] in the public domain. The chief advantage of such distributed parallel systems is the cost-effectiveness.

Parallel computers are very expensive and their availability is not as ubiquitous as the network of workstations. It has been observed that, in practice, up to 80% of the workstations are idle depending on the time of day [15,17]. Thus, network of workstations (NOW) provides an opportunity to emulate parallel machines with large aggregate processing power and memory. An excellent justification for using NOWs is given by Anderson et al. [2]. Several parallel machines are, in fact, based on workstation chips. Some examples of such systems are the Cray T3D (based on DEC Alpha) and the teraflop Intel system (based on PentiumPros). Thus, from the hardware point of view, the main difference between a parallel system (MPP¹) and the one emulated on a NOW is the

¹ MPP stands for Massively Parallel Processors

communication network. Parallel systems use better interconnection networks, better switching techniques (e.g., wormhole routing) whereas the NOW-based systems use high-overhead, low bandwidth local area networks (LANs²). The performance of a NOW-based system improves with the use of high-speed LANs (such as SONET-based ATM networks). There are, however, some other significant differences between the two types of systems. Some of these are listed below:

- *Node heterogeneity*: NOW-based systems typically consist of heterogeneous nodes whereas the real parallel systems typically use homogeneous nodes.
- *Load on nodes*: Since local injection of load is possible by the owner of a workstation, in distributed parallel systems there is a high variability of load on nodes over time. In addition, load imbalances exist among the nodes. These problems are not as serious in parallel systems as the system is under the control of a scheduler.
- *Node failures*: NOW-based systems suffer from node failures as owners have control of their workstations. For example, an owner might reset his/her workstation. Thus, fault-tolerance and fault-recovery is important for these systems.

² LAN stands for Local Area Network

1.1 Why NOWs are becoming attractive substitutes for MPPs

Microprocessors' performance is improving at about 50% per year. Performance and cost-effectiveness of microprocessors are such that many currently produced supercomputers are based on mass produced microprocessors. Some examples of such systems are the Cray T3D (based on DEC Alpha) and the recent Intel system (based on PentiumPros). These microprocessors are in the thousands of workstations.

Hundreds of thousands of workstations produced mean that manufacturers can amortize the massive development cost required to sustain the massive rate of technological innovation over a larger number of units. Other economies of scale further contribute to the improved cost/performance ratio. There is a huge difference between the cost of components for workstations and MPPs not based on workstation parts. For example in 1994 the cost of a megabyte of DRAM was \$40 for a personal computer (today it is closer to \$10 per megabyte) and \$600 for the Cray M90 family [2].

It seems that even MPPs that use standard workstation components have a key weakness – the engineering lag time. MPPs tend to lag one or two years behind the workstations built from comparable parts. At 50% performance improvement per year, a lag of two years translates into more than factor of two in computational performance.

Studies such as [2] and [26] show that NOW-systems with networks based on high-speed switches and efficient protocol implementations which can support high

bandwidth and low latency communication are a practical architecture that is comparable in speed to supercomputers.

Workstations are much more versatile than MPPs and can deliver high throughput for many general purpose tasks. In addition to performance that is comparable to supercomputer's node, the workstations offer large memory and disk capacity. New technologies such as ATM are emerging and are promising very fast communication over LANs. Many organizations already own large number of workstations and since MPPs are prohibitively expensive and inflexible, the idea of using combined resources of many workstations is more attractive than ever.

1.2 Related work

In most cases, a NOW system is available to many simultaneous users and is not dedicated to parallel processing. There might be other local processes that introduce load on some of the workstations, therefore scheduling of parallel jobs on a NOW that makes good use of available resources without (hopefully) interfering too much with other jobs is particularly important. Most existing scheduling methods choose to migrate a parallel process to idle workstation once competing local jobs appear. The studies [15,17] indicate that more than 80% of workstations are idle at any time. Systems that work with large pools of workstations have been implemented such as ones in Condor [5], Utopia [29] and others [18, 22] to name just a few. The idea of migrating a parallel process is based on the

assumption that there is a large pool of workstations of which some are idle. This assumption may not always hold, since quite often a user might not have access to a large pool of workstations. For example, in our labs there are less than 10 workstations available and close to assignment deadlines all machines might be busy. Also, when there are relatively few workstations available, it makes sense to start running a parallel job on all of them. Therefore if local jobs appear on some of workstations, there are no idle workstations available to which parallel jobs might be migrated. Even if the pool of workstations is large, it might be difficult to immediately find an idle workstation where a process may be migrated. Also, there is no guarantee that the process will not have to be migrated again. Overheads of migration and context switching may eliminate any performance gains expected from this technique. There are several implementations of task migration for the PVM software system that we are using - some of them are described in [5, 6].

There are only a few studies that examine non-task migrating scheduling for NOW-based systems. Another approach [4] to scheduling of parallel jobs is to allow them to stay and run (possibly at low priority) when a local job starts executing on that workstation. Thus, the overheads related to process migration are eliminated at the cost of some degradation of performance of the local job. This is the approach that is used to schedule jobs in this study.

Several studies, for example [24], also look at automatic parallelization of

programs for NOWs through the use of special compilers and scheduling.

There have been several studies that show the benefits of variable granularity tasks on MPPs. Several strategies have been proposed to decrease the task sizes [14,21,28]. In this thesis we examine if these strategies are beneficial on a NOW-based system.

1.3 Objectives

Our objective in conducting this study is to gain an understanding of the factors that significantly influence the performance of NOW-based systems. As enumerated, there are significant differences between NOW-based systems and a MPP. In particular, our goal is to provide answers to the following questions:

- How important is the network speed?
- Since the network is heterogeneous, what is the influence of task mapping?
- How beneficial is load sharing?
- What is the effect of frequency of communication? A related question is: What is the effect of the task granularity on the performance?
- Are the variable task granularity schemes proposed for multiprocessors beneficial?

In order to gain this understanding we use the matrix multiplication application running on a network of Pentium and Pentium Pro nodes. The NOW-based system runs

under PVM. In order to look at the intrinsic behavior of the application, experiments were conducted in a controlled matter with no background load. Experiments with controlled background load were also performed. In addition, we have used a generic workload in which we can vary the computation and communication characteristics of applications.

1.4 Contributions of the thesis

There are relatively few studies that present results obtained from running real applications on NOW-based systems that do not use task migration. To the best of our knowledge this thesis is the first study in which:

- the influence of task mapping to slow or fast nodes on NOW-based systems is discussed
- the performance of algorithms with variable task granularity is compared to the performance of algorithms with fixed granularity
- the performance of a NOW connected by 10Mbits/sec Ethernet is compared to the performance of the same NOW connected by 100Mbits/sec fast Ethernet
- all the above experiments were run with and without background load and their results are discussed.

Parts of this study have already been published in [19,20].

1.5 Thesis Organization

The remainder of the thesis is organized as follows. The next chapter gives a description of the experimental environment including the PVM and the NOW system used in conducting the experiments. Chapter III presents the fixed granularity algorithms implemented and discusses the results obtained. The variable granularity algorithms and their performance are discussed in Chapter IV. Conclusions and possible further research topics are given in the Chapter V.

Chapter II

Experimental Environment

In this thesis we examine the performance of parallel applications on a NOW-based system that uses the PVM. This chapter presents background information about the experimental environment that consists of a PVM and a NOW system.

2.1 PVM

PVM is a software package that allows a heterogeneous network of parallel, serial and vector computers to appear as a single concurrent computational resource (distributed memory computer).

The development of PVM started in the 1989 at Oak Ridge National Laboratory and is now an ongoing research project. PVM can be run on over 30 platforms including the LINUX, Silicon Graphics, Cray, Connection Machines, Intel iPSC and Suns.

PVM consists of two parts:

1. A daemon process that any user can install on a machine
2. A user library that contains routines for initializing tasks on other machines, communication and changing configuration of the PVM machine.

Applications using PVM can be written in Fortran77 or C and parallelized by using message-passing constructs common to most distributed-memory computers. PVM supports heterogeneity at the application, machine, and network level. In other words, PVM allows application tasks to exploit the architecture that is best suited to their solution. For example PVM may include parallel machine and graphical workstation. Computation can be done on the parallel machine and the results may be displayed in graphic form on the workstation.

PVM handles all data conversion that may be required if two computers use different integer or floating point representations. If PVM machine is to contain machines with different architectures, programs need to be compiled on each one of the different architectures.

PVM supplies the functions to automatically start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. A task in PVM is defined as a unit of computation analogous to a UNIX process. The PVM application can

then be started from a UNIX prompt on any of the hosts. Multiple users can configure overlapping virtual machines and each user can execute several PVM applications simultaneously. Maximum number of hosts is 4095 with maximum of 262143 tasks per host. In reality, the number of processes allowed on the machine (host) and the number of file descriptors available limit the number of tasks. By default, when the PVM is spawning a task, it tries to spawn in on the least loaded machine. But the least loaded machine might not be the fastest one to complete the task and loads are subject to change. The user may override default spawning behavior by specifying architecture to spawn a task on or even particular host. This provides greater flexibility while retaining the ability to exploit particular strengths of individual machines on the network. If some host fails, PVM will automatically detect it and delete the host from the virtual machine.

PVM supports real broadcast only on certain multiprocessors. In most environments, when a user calls broadcast function, PVM creates a single message buffer and repeatedly sends it to specified tasks. This kind of broadcast should still be faster than a number of Send operations because only one message buffer gets created. In contrast, each Send call creates a new buffer. PVM's detailed description is presented in [23].

2.1.1 What is Provided by PVM

1. *Point-to-point communication*: Communication is usually based on TCP and UDP protocols. Sends are asynchronous and receives can be blocking or non blocking. If data is sent between two machines that use different data representation, data is

packaged before it is sent and later converted to the receiver's format. Multicasting is also supported.

2. *Unix type signals:* They are supported but it is very easy in a parallel environment for interrupts to cause non-deterministic behavior, deadlocks, and even system crashes. For example, if an interrupt is caught while a process is inside a UNIX kernel call, then a graceful recovery may not be possible.
3. *Process groups:* Process group membership is dynamic and group functions are designed to be very general and transparent to the user at some cost in efficiency. A process can belong to multiple groups. Process groups are used for broadcasting messages, barrier synchronization and collective operations.
4. *Environmental management and inquiry:* Utilities are provided for finding task id (similar to process rank), number of existing tasks, number of machines in PVM, error handling, timers and choosing type of communication.
5. *Support for task management and dynamic configuration:* Tasks can be created and killed by other tasks. Tasks can be spawned on specific architecture, even a particular host. PVM machine can be reconfigured from the inside of PVM program by adding and deleting hosts.

2.2 NOW-system description

The network of nodes used in our experiments is based on inexpensive off-the-shelf components, both for processing and communication. The current configuration consists of 6 Pentium and 4 Pentium Pro nodes. All nodes have 32 MBytes of RAM.

Figure 1: NOW system used for experiments

- Nodes 1, 2, 9 are 133 MHz Pentiums;
- Node 4 is a 133 MHz Pentium with a better pipeline burst cache which makes it quite a bit faster than nodes 1, 2 and 9;
- Nodes 5 and 6 are 166 MHz Pentiums;
- Nodes 0, 3, 7 and 8 are 200 MHz Pentium Pros.

These ten nodes are interconnected with a 10Mbps Ethernet “service” network, as well as with a 100Mbps Ethernet high-speed network. The 10Mbps network is used for booting the system, providing the basic communication and remote file service through NFS. Both the 10Mbps network and 100Mbps network can be used for high-level communication under PVM. To an outside user, the system looks like a set of normal workstations. We have conducted experiments with both the slow and fast networks.

Because node 0 has a variety of responsibilities and often has a certain load on it, it was not used in the experiments.

The highest interprocess bandwidth of 7MBytes/sec was achieved between processes on the two fastest nodes (PentiumPro-200). The highest bandwidth achieved between the two slowest nodes (Pentiums-133) was around 5MBytes/sec. Please note that because nodes are communicating over the same Ethernet cable, collisions may frequently occur and will decrease the bandwidth. This is not the case in MPPs.

Our NOW system has significantly a lower bandwidth than commercially available MPPs. MediaCube MPP offers a minimum of 8.8MBytes/sec and can achieve a bandwidth of 352 MBytes/second [16]. Cray Researches’ Cray T3E offers an interprocess bandwidth of 480 Mbytes per second [27].

Chapter III

Fixed Granularity Algorithms

Figure 2: General structure of the algorithms implemented (a) Fork-and-join structure;
(b) Structure of the implemented algorithms

All algorithms that have been implemented are based on the master-slave strategy and follow the fork-and-join structure shown in Figure 2b. In this strategy, a master program acts as the coordinator and spawns as many slave programs as required (typically equal to the number of nodes in the PVM). The slave programs actually perform (part of) the required calculations. The master is responsible for distributing the work to the slaves and also for collecting the partial results returned by the slaves and combining them to get the final result. There is no communication among the slaves. In this model, all

communication is between the master and a slave as shown in Figure 2b. All of the load sharing algorithms are based on the pool of tasks paradigm.

3.1 Matrix multiplication

We have implemented several algorithms to investigate the sensitivity of the performance of the matrix multiplication application to various system and algorithm related issues. Some of the issues investigated include the impact of network, frequency of communication, and the effect of mapping.

We divide the algorithms into two broad classes: algorithms that send one of the two matrices to all slave programs and those that send only a submatrix. We refer to the first class of algorithms as the *Matrix-based algorithms* and the second class as the *Submatrix-based algorithms*. The associated tradeoffs between the two classes are discussed in 3.3.

Matrix-based algorithms require the storage of a complete matrix at each node. The size of the matrix was determined by the amount of memory available on nodes. The matrix size used was 720x720, elements of matrix were floats (4 bytes each) which required 2Mbytes of memory. The matrix size of 720x720 was chosen because 720 is divisible by many numbers (like 2, 3, 4, 5, 6, 8, 9) and also because larger matrices would cause PVM to run out of memory for message buffering. Every matrix multiplication algorithm uses the same function to perform the multiplication itself.

3.2 Matrix-based algorithms

These algorithms send one of the two matrices to all slaves. In the following discussion (and in our implementation), we assume that it is the first matrix that is sent to all slaves. The effect of sending the second matrix instead of the first one is discussed in Section 3.5.6.

3.2.1 No load sharing (NLS) algorithm

In this algorithm, the master program sends the first matrix to all the slaves. The second matrix is distributed equally to all the slaves i.e., the number of columns of the second matrix sent to each slave is C/S , where C is the number of columns in the second matrix and S is the number of slaves. If there is also a slave program running on the master node, S in the formula should be replaced by $S+1$. The algorithm is based on the Send algorithm described in Section 3.2.2.2. That is, the master sends individual packets of data consisting of the whole second matrix plus the slave's share of the first matrix to each slave sequentially.

3.2.2 Load sharing algorithms

All the remaining algorithms allow load sharing and are based on the “pool of tasks” paradigm. Each slave receives a task to work on and when the task is completed it sends results back to the master. The master receives the results and then sends the slave another task to do. There is no communication between the slave programs. The same code is used to multiply matrices or submatrices in every algorithm. The performance of

the algorithm is a function of the task size t (i.e. granularity of the task). For example, if the task size t is fixed at three columns, each time a slave requests work, the next three columns of the second matrix are sent. There is a tradeoff between task granularity and communication overhead. Finer granularity (e.g. one column) introduces more frequent communication but improves load sharing. Coarser granularity decreases the frequency of communication but also decreases the scope of load sharing. In the extreme, we can set $t = C/S$ as in the NLS algorithm with no load sharing. We will discuss the effect of task granularity in Section 3.5.5.

We can further divide the load sharing algorithms into two groups depending on whether the algorithm uses fixed granularity or variable granularity. Variable granularity algorithms are discussed in Chapter IV. We will now describe Broadcast, Send, Replicate and Store fixed granularity algorithms.

3.2.2.1 Broadcast

In this algorithm, the first matrix is broadcast to all slaves. Then the first t columns of the second matrix are sent to the first slave, the second t columns to the second slave and so on. Each slave multiplies the columns that it receives by the first matrix that it already has. When a slave finishes the multiplication, it sends the results back to the master. The master collects the results, writes them into the appropriate place of the result matrix and then sends another t columns of the second matrix to the slave. This continues

until all rows of the second matrix have been sent.

Recall that PVM does not do a real broadcast but a set of sends. Broadcast should still be faster than the number of Send operations because for broadcast PVM creates and sends only one message buffer and not many like for the number of Send operations.

3.2.2.2 Send

A potential problem with the Broadcast algorithm is that it takes some time for the first matrix to be received by all slaves. This is particularly so if broadcast is implemented by simply sending the same data separately to each intended receiver. As the master task is broadcasting the first matrix to all slaves, some slaves may get it earlier than others. Once a slave has received the first matrix, it needs to get t columns of the second matrix to start the multiplication. If the master is still busy broadcasting the first matrix to the other slaves, the slave will be idle until the master finishes the broadcast and sends it t columns of the second matrix.

To avoid this problem, this algorithm sends the first matrix and t columns of the second matrix together to the first slave so it can start the multiplication, then the first matrix and the next t columns to the second slave and so on. We will compare the performance of these algorithms in Section 3.5.7.1.

3.2.2.3 Replicate

A problem with the previous algorithms is that if a task has been sent to a slave that was particularly slow, or had become heavily loaded, the whole computation suffers. This can happen in a heterogeneous environment and/or dynamic environment where load variations are frequent. This problem causes more serious performance problems if the task granularity is coarse. Furthermore, the other algorithms (discussed so far) are not fault-tolerant. If a node fails before returning the results, the whole application suffers. The Replicate algorithm addresses these problems by sending the same task to more than one slave. This algorithm behaves somewhat similarly to the task migration schemes. But instead of moving around the tasks, it replicates them to the other nodes that are not busy.

Our implementation of this algorithm is based on the Send algorithm³. Once all available tasks are distributed to the slaves, the master must wait for all the results to be sent back from the slaves. Next, the Replicate algorithm does something similar to task migration. It duplicates unfinished tasks and sends them to idle nodes. The master keeps track of all tasks that are currently being executed on the slaves. When there are no more tasks to distribute, the master queues all tasks that are being executed on the slaves. It also maintains an idle slave node queue. As long as there are still some tasks whose results are not yet received by the master, it performs the following in a loop:

³ Note that it can also be implemented for other algorithms.

- Priority is given to receiving the results. Therefore, as long as there are some messages with partial results from slaves, these results will be received and processed by the master. If the task result is no longer needed because it has already been received from another slave, it is discarded. If the result is needed, the task whose result was received is removed from the task queue. The slave from whom the results were received is now idle and added to the idle slave queue.
- If the idle slave queue and the task queue are not empty, the task is dequeued and sent to the first idle slave. The slave is dequeued from the idle slave queue. The dequeued task is added to the end of the task queue. The algorithm tries to minimize the number of times a task is replicated but if there are a lot of idle slaves and relatively few tasks left, some tasks may be replicated several times on different slaves. When the master received all task results, it kills all the slaves.

3.2.2.4 Store

This algorithm is similar to the Send algorithm except that the partial results are not sent back to the master immediately as in the Send algorithm. Instead, all partial results are stored locally at each slave. The Store algorithm works as follows: The first matrix and t columns of the second matrix are distributed as in the Send algorithm. Once a slave finishes its computation, it sends a message to the master asking for more columns of the second matrix. The master responds by sending the next t columns. If the master does not have any more columns to send, it sends a message to all slaves asking them to send

the results they have collected. Slaves respond to this request only when they have completed all computations.

There are two main benefits one can expect from the Store over the Send:

- In the Store algorithm, the master is not burdened with handling partial results several times. This improves the response time to the requests from the slaves for more data.
- Generally, it is advantageous to send one large message than several small messages. The Store sends several very short messages (requesting tasks) and several very long messages at the end of computation. On the other hand, the Send sends an equivalent number of messages that contain partial data. The size of these messages in Send depends on the task granularity. While both Send and Store send the same number of messages, we argue that the short messages sent by Store are unavoidable if we want to implement load sharing. The down side of Store, however, is that there is a potential for message congestion as all slaves want to send their results at the end of the computation. If the computation were balanced in time, this would be a problem. If, on the other hand, computation is staggered among the slaves (e.g., due to load variations and task granularity), network congestion may not be a problem.

Section 3.5.7.3 compares the performance of these two algorithms.

3.2.2.5 Hierarchical

All algorithms described use centralized organization. In such algorithms the node maintaining the central task queue (master) could easily become the bottleneck. The hierarchical algorithm tries to reduce the load on the master node caused by the task management. Studies like [3, 7, 9] showed that hierarchical task queue organization could provide significant performance benefits. We want to see if similar benefits can be achieved on the NOW-based system.

Figure 3: Hierarchical organization implemented for 7 nodes with branching factor 3

In the hierarchical organization implemented, new tasks are generated in the master on the root node (Node 1). If slave has finished the task, it returns the results of it to its master (or submaster) and receives from it another task. If the submaster at level L decides that its queue of tasks is empty, it asks its master (or submaster) at level $L-1$ for $s*T$ more tasks where:

- s is a number of slaves below it plus one which is on the same node,
- T is a parameter chosen by the user (at least 1)

The hierarchical algorithm had consistently worse performance than others did for two reasons:

- Master was not the bottleneck in most of the cases.
- This algorithm causes a larger amount of data items to be exchanged between nodes than in other algorithms. Therefore, the network became a bottleneck.

The Hierarchical algorithm was used here to distribute tasks of a single application. It could be beneficial if it was used by the system (for example PVM) to distribute tasks of several applications between nodes.

Because this algorithm performed consistently worse than others did, it will not be discussed any further in this thesis.

3.3 SubMatrix-based algorithms

Algorithms SubMatrix and CachedSubMatrix are based on a different principle than the previous ones. These algorithms are based on the algorithms that are often used for matrix multiplication in multiprocessors (see [1]). We explain the principle of these algorithms by means of an example. Assume that A and B are both $n \times n$ matrices where $n = 2k$. Then A and B can be thought of as conglomerates of four smaller matrices, each of size $k \times k$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Given this partitioning of A and B into blocks, the result of matrix multiplication $C = A \times B$ is defined as follows:

$$C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

A and B matrices can be divided into more than just four pieces. There are tradeoffs between the matrix-based and this class of algorithms. The matrix-based algorithms reduce the frequency of communication but require more memory so that one complete matrix can be stored at each slave. On the other hand, the SubMatrix-based algorithms increase the amount of data being sent but the memory demand is less, as only submatrices are stored at each slave. Only submatrices are being multiplied and this should be quicker

because submatrices fit better into processor cache. In SubMatrix-based algorithms, the master has to do more work, as it has to perform additions.

We have implemented two SubMatrix-based algorithms. Other algorithms (e.g., the replicate algorithm) can augment these algorithms.

3.3.1 SubMatrix algorithm

In this algorithm, every submatrix multiplication is sent to a slave. The master just adds up the results. In the above example, A_{11} and B_{11} would be sent to the first slave, A_{12} and B_{21} to the second and so on. Master collects the results, adds them to the results already received and then sends another piece of matrix A and B to the slave for multiplication. We have conducted experiments to study how varying the number of submatrices (i.e., task granularity) affects the performance (see Section 3.5.5 for a discussion of this issue).

3.3.2 CachedSubMatrix algorithm

This algorithm is very similar to the one above. The major difference is that the master keeps track of which slave has received which submatrix of A. This allows for the caching of the submatrices of A on the slaves. As long as it is possible, the master sends to the slave submatrices of B that need to be multiplied by the submatrix of A that is cached on that slave. If there are no more multiplications to be done with that cached submatrix of A, another submatrix of A is sent to the slave for caching. Care is taken to cache

different submatrices of A on every slave, but once all of them are used it is possible that multiple slaves will be caching the same ones. The effectiveness of caching is discussed in Section 3.5.7.5.

3.4 Generic workload simulation algorithms

Matrix multiplication is just one example of application. One should not make general conclusions based on the behavior and performance of a single application. There is a need to model synthetic applications in which computation and communication can be controlled precisely so that the impact of these factors can be studied. For example, what happens if the application is computation-intensive? What happens if it is communication-intensive?

This information will be useful to programmers in designing their applications. There is also a need to study other applications that exhibit different communication characteristics than the matrix multiplication application considered here. In addition, the beneficial effects of emerging network technologies (e.g., SONET-based ATM) need to be quantified. On such high-speed LANs, PVM communication primitives can be enhanced to improve its performance [7]. These enhancements can significantly affect the application-level performance. Therefore, a number of functions has been implemented to allow simulating the behavior of different algorithms. Basic ideas behind running such simulations are:

- Simulations allow full control over computation to communication ratio. This allows us to study the performance of different task scheduling policies under different conditions.
- Simulations allow full control over the speed of the nodes. It was obvious from matrix multiplication experiments that the performance of the node depends very much on the size of the task given to it. This is mainly due to cache overruns. Generic workload functions make it possible to simulate that particular node has some constant speed independent of the task size given to it.
- Behavior of any class of algorithms can be simulated. Matrix multiplication is just one particular application but generic workload functions allow us to simulate a variety of algorithms. For example with different big “O” like $O(N)$, $O(N \cdot \log(N))$ or $O(N^2)$ complexity times.

3.4.1 Implementation details of Generic Workload

It was determined that the bandwidth of connections over fast network depends greatly on the processing power of the nodes on both ends of the connection. For example bandwidth between two Pentiums Pro was over 7 Mbytes/sec but between two Pentiums 133 was only 5 Mbytes/sec. Such large differences would make the communication to computation ratio vary depending on the computing power of the nodes. Therefore, the slow network was chosen for running generic workload simulations. The bandwidth of the

slow network connections was consistently around 1.07 Mbytes/sec and was independent from the computing power of nodes.

PVM running on LINUX does not provide a real broadcast operation. Some matrix multiplication algorithms use broadcast. Therefore, we wanted to see the difference that a real broadcast would make on the performance. Broadcast was simulated by a generic workload function which sent the full message to one slave and then sent to all other slaves the size of the message. All nodes had to reply that they had received the broadcast before the broadcaster could continue to execute the program.

One of the problems encountered while using PVM was dealing with out-of-memory errors due to buffered messages. Messages are buffered until the receiving task is willing to retrieve them. When there are a lot of messages being buffered but the task is unable or unwilling to receive them, PVM might crash because it has used up all available memory.

In order to eliminate such problems it was necessary to write a special message system based on standard PVM communication functions. The message sending function sends only 200,000 bytes of the message and then it waits for a reply from the message receiving function. Once the reply is received, the next 200,000 bytes of the message are sent. This strategy caused the bandwidth decrease of about 2.2%. This is not a big price to pay to avoid constant crashes.

A program, which simulates the behavior of a certain algorithm, has two logical components - “communication” and “work”. In such a program, data messages are really sent between the nodes but no real computation involving the data is performed. Appropriate generic workload function simulates the time it would take one node to perform the calculations by sleeping a certain number of microseconds calculated according to the following equation:

$$\text{algorithmComplexity} * \frac{\text{currentNumberOfDataItems}}{\text{totalNumberOfDataItems}} * \frac{\text{problemHardness}}{\text{nodeSpeed}}$$

where:

- *algorithmComplexity* is the number of steps required to solve the problem of size N for any given algorithm, for example $O(N)$, $O(N*\log(N))$ or $O(N^2)$
- *currentNumberOfDataItems* - number of data items given node is assigned to work on
- *totalNumberOfDataItems* - N in *algorithmComplexity*, size of the problem to solve
- *problemHardness* is the time it takes to solve one step in an algorithm of given complexity
- *nodeSpeed* allows the user to differentiate between nodes of different speeds, for example a node of speed two is twice as fast as a node of speed one

Please note that in the above equation $\frac{\text{currentNumberOfDataItems}}{\text{totalNumberOfDataItems}}$ is the fraction of

work to be performed by any given node. The total number of microseconds required to perform all of the work is given by $algorithmComplexity * problemHardness$.

Example:

In some algorithms used for matrix multiplication, nodes multiply one matrix by some number of columns of the second matrix. We use matrices of size 720x720 and on the PPro-200 this matrix multiplication takes around 24.5 seconds. Therefore, we want $algorithmComplexity * problemHardness$ to be equal to 24,500,000 microseconds. The basic operation is the simulation of multiplying one column by a matrix. Since we have $N = 720$ columns, we do this basic operation N times and therefore $algorithmComplexity$ here is $O(N)$. Since $algorithmComplexity * problemHardness = 24,500,000$ microseconds, $problemHardness$ is equal to 34000. If we assume the node receives 3 columns to multiply by a matrix and $nodeSpeed$ is the same as PPro-200 then the time is calculated as:

$$720 * \frac{3}{720} * \frac{34000}{1} = 102000 \text{ microseconds.}$$

Hence, instead of doing real calculation, the node will simulate work by sleeping 102000 microseconds.

We also wanted to simulate the behavior of parallel programs running on heterogeneous nodes or homogenous nodes with variable background load. This was simulated by setting different values of $nodeSpeed$ variable.

In some experiments, we wanted to simulate that the load on each node changes “randomly” every time another task is given to it. It was achieved by modifying the *nodeSpeed*. For this experiment, there were only three node speeds used: 1 (no background load), $2/3$ ($1/3$ background load) and $1/3$ ($2/3$ background load). It is possible that a random number generator might return the same value many times in a row. In order to avoid this situation we randomly arranged all of the three possible node speeds in a three element sequence. Every time a new task was given to a node, the next number from the sequence would be used as *nodeSpeed*. Once the end of the sequence was reached, a new random sequence of *nodeSpeeds* was created.

We wanted simulations to run for a significant amount of time. Some matrix multiplications took under 10 seconds and with such a low response time many factors could cause significant changes in response times. Therefore it was decided to make simulations run for over a minute. It was observed that for $N=20,000,000$ data items (each 1 byte) a Broadcast algorithm takes around 52.6 seconds just to exchange all necessary data without doing (or simulating) any work. To get 1:1 computation to communication ratio, it was necessary for the simulation of calculations to also take 52.6 seconds. Therefore, *problemHardness* was selected to be 2.63 (since $20,000,000 * 2.63 = 52,600,000$).

Accordingly, all algorithm simulations use $N=20,000,000$ data items. Simulations were run at 10:1, 50:1 and 100:1 computation to communication ratio and

problemHardness was set to 26.3, 131.5 and 263 respectively. At computation to communication ratio of 1:1 no performance benefits were achieved as compared to the same simulation running on a single machine. Therefore, no experiments with computation to communication ratio of 1:1 were performed.

3.4.2 Algorithms implemented

We have tried to make algorithms and message exchange mechanisms used in simulation similar to the ones used in the matrix multiplication algorithms. We wanted to gain some understanding of the influence of computation to communication ratio on the algorithms' performance. To keep the communication to computation ratio constant, algorithms with $O(N)$ were used in simulation (unlike matrix multiplication which has other algorithm complexity but as shown in section 3.4.1, can be simulated by $O(N)$).

Three fixed granularity algorithms – Broadcast, Send and Replicate – were simulated using generic load. Please refer to Sections 3.2.2.1, 3.2.2.2 and 3.2.2.3 for description. Because the Send algorithm always proved to perform worse than Broadcast, the computation to communication ratio given in this thesis is the ratio for the Broadcast algorithm. Recall that Send algorithm sends much more data.

The following differences between algorithms used in matrix multiplication and algorithms used in generic workload application:

- In generic workload application there is no first matrix or second matrix so all data items (20,000,000) have to be received by the slave before receiving tasks
- In matrix multiplication the Replicate algorithm is based on the Send algorithm. The Replicate algorithm in generic workload application is based on Broadcast because Broadcast has a computation to communication ratio that we need (all data items are broadcast not sent).

3.5 Results and discussion

This section presents the results of the experiments conducted. Throughout these experiments we have used a 720×720 matrix. Recall that the size of the matrix is determined by the amount of memory available at a node, as the matrix-based algorithms require the storage of a complete matrix at each node. All programs used in our experiments were implemented in the C language.

Node number	Node description	Time in seconds it takes this node to multiply matrices
1	Pentium 133	79.23
2	Pentium 133	79.31
3	PentiumPro 200	24.52
4	Pentium 133 + pipeline cache	70.41
5	Pentium 166	61.01
6	Pentium 166	60.79
7	PentiumPro 200	23.94
8	PentiumPro 200	24.55
9	Pentium 133	78.94

Table 1: Comparison of performance of the nodes used. Time that it takes for the node to multiply 720×720 matrices by itself is shown.

We use the execution time as the main performance metric. The execution time values reported in this section are the averages of 40 runs (with a standard deviation of less than about 1% of the mean value). In order to make fair comparisons among the algorithms, we have used the same function to multiply matrices in all the algorithms. Using results from Table 1, the total combined speed of all of the nodes can be calculated to be about 5 times more than the fastest node (PentiumPro 200).

Figure 4: Comparison of the performance of matrix multiplication. NLS means no load sharing algorithm, LS means load sharing algorithm

From Figure 4, two important observations can be made. This figure clearly shows that there is benefit in using other slower nodes than the three fastest nodes. It also shows that the total speedup⁴ achieved was about 2.8 that is about 56% efficiency⁵.

⁴ The speedup achieved by a parallel algorithm running on n nodes is the ratio between the time taken by that NOW-system executing the fastest serial algorithm (on fastest node, PPro-200 in our case) and the time taken by the NOW-system executing the corresponding parallel algorithm using n nodes.

⁵ The efficiency of a parallel algorithm running on n nodes is the speedup divided by n .

Communication is the bottleneck – it takes about 4.8 seconds to distribute the first array to all slaves on the fast network and about 15 seconds to do the same on the slow net.

3.5.1 Load sharing versus no load sharing

Figure 5 : Performance of No Load Sharing (NLS) versus Send (a) task distribution is "slow first";
(b) task distribution is "fast first"

This section compares the performance with and without load sharing. Recall that the No Load Sharing (NLS) algorithm described in Section 3.2.1 is based on the Send algorithm. Figure 5 shows the relative performance of these two algorithms when the master node also works on the multiplication. Note that both algorithms send the complete first matrix to all slaves. The difference is in sending the second matrix data. Send uses "pool of tasks" whereas NLS sends all required columns in one message. The results for

Send are obtained with a task granularity of 3 columns.

In the NLS algorithm, the first matrix along with the $720/9=80$ columns of the second matrix are sent to slaves serially. In a heterogeneous environment, assuming that we can order the nodes by their speed, task distribution can be done in one of two ways: slowest to fastest (i.e., “slow first”) or fastest to slowest (i.e., “fast first”). To be consistent, we have also done the initial distribution of tasks in Send in both ways.

From the data presented in Figure 5, it can be seen that the task distribution order affects the performance of NLS much more than that of Send. Send exhibits a more robust performance to task distribution order. It is clear that Send does not depend heavily on the initial distribution as the load sharing aspect of this policy will even out the initial task distribution imbalances. The data also shows that the sensitivity of NLS to task distribution order increases when the network is slow.

Performance of Send is better than NLS with one exception. When the master is mapped to the slow node and the network is also slow, NLS performs marginally better than Send. This is because NLS reduces the frequency of communication between the master and a slave substantially (In NLS, communication occurs twice – once to distribute tasks and the other time to communicate the results to the master). This reduced frequency of communication reduces communication overhead as well as the number of interruptions. Both these factors help improve the performance of NLS over Send.

The reason for “Fast First” being slower is simple. Obviously, NLS response time will depend on the response time of the slowest machine. If task distribution is “Fast First”, the slowest node will be the last one to get work. Therefore, performance deteriorates.

While we have done experiments in a controlled environment, in a real system, rank ordering of machines is not possible because the load on machines can vary dynamically. A particularly heavily loaded fastest machine could, in fact, be the slowest from the application point of view. In this type of uncertain environment, the sensitivity of NLS to the task distribution order is a serious drawback and load sharing is preferred.

3.5.2 Fast master versus slow master

Figure 6 : The performance of the Send and CachedSubMatrix algorithms when the master's node works on the multiplication ("Work") and when not ("No work") (a) Send algorithm (b) CachedSubMatrix algorithm.

Figure 6 shows the performance of the Send and CachedSubMatrix algorithms when the master's node works on the multiplication ("Work") and when it does not ("Not work"). Task granularity is 5 columns for the Send algorithm, and 3×3 for the CachedSubMatrix. Note that, in our setup, the fastest machine (200 MHz Pentium Pro) is about 3 times faster in performing the matrix multiplication than the slowest machine (133 MHz Pentium). We note the following from the data presented in this figure:

- *Slow Network:* When the network is slow, it does not make a significant difference whether the master program is mapped to the fastest node or not. Mapping the master program to the fastest node tends to improve performance. But the difference in performance is marginal as shown in Figure 6. This observation is valid whether or not the master performs coordination only or coordination as well as computation.
- *Fast network:* When the fast network is used, the best mapping depends on several factors:
 - *When the master's node performs coordination only:* In this case, mapping the master to the slowest node is better. This is because, if the fastest node is used for pure coordination task, it is taken out of the worker pool of nodes that actually performs the computation. Therefore, the execution time increases. This is true for both algorithms as shown in Figure 6.

- *When the master's node performs coordination as well as computation:* In this case, best mapping mainly depends on the amount of work the master performs in processing replies from slaves. If reply processing takes a negligible amount of time (as in the Send algorithm – see Figure 6 a), mapping the master to the slowest node is better as coordination activities do not overwhelm even the slowest node and faster nodes can be used for computation. If, on the other hand, reply processing time is not negligible (as in CachedSubMatrix algorithm – see Figure 6 b), mapping the master to the fastest node is better as the faster node can respond better to requests from slaves. Mapping the master to the slowest node, for example, could affect the communication performance due to delays caused by the slow master node in processing replies. These mappings become more important with decreasing task granularity as finer task granularity increases the rate of replies from slaves.

In practice, however, it is not possible to determine the fastest node due to dynamic load changes. Note that we are interested in the fastest node from the application point of view (not based on the “raw” speed of nodes).

3.5.3 Work versus no work

The question that we address in this section is whether the master's node should strictly perform coordination (i.e., distribute tasks and process results from slaves) or should it also perform actual computation like a slave. From the data presented in Figure 6

we can observe that if the network is slow, it does not make a significant difference. On the other hand, with the fast node, performance tends to improve, in most cases, when the master's node performs computation in addition to its coordination activities. In practice, therefore, the master should work on computation if it is on a fast node⁶; otherwise, restrict itself to coordination activities. It is, however, not possible to determine if a node is "fast" or "slow" statically due to dynamic load changes. Therefore, the master should be adaptive to the perceived service that a slave is providing. One way of implementing this adaptiveness is to give a low priority to the computation process on the master's node so that the master works on the computation only when it is not busy with coordination activities.

3.5.4 Impact of network

The data presented in Figure 5 and Figure 6 indicate that the network speed has the biggest impact on the performance of the algorithms. Communication is the major bottleneck for the matrix multiplication application. For matrix-based algorithms it takes around 4.8 seconds to distribute the first array to all slaves on the fast network and around 15 seconds on the slow net. This also demonstrates a major drawback of using networks of workstations (NOWs) for parallel processing. Parallel systems have a dedicated interconnection network that is specifically designed to provide a good message-passing performance. This improvement in communication performance is achieved by a

⁶ Fast from the application point of view, not the "raw" speed of the machine

combination of higher bandwidth and better switching techniques (e.g., wormhole routing is used in current parallel systems from Cray, Intel and nCube). This communication bottleneck has been recognized widely as the chief drawback of NOW-based parallel computation (for example, see [2]). The emergence of high-speed, high bandwidth networks will help improve the performance of NOW-based systems.

3.5.5 Task Granularity

Figure 7 : Performance sensitivity to task granularity (a) Send algorithm; (b) CachedSubMatrix algorithm

Task granularity is an important parameter of an algorithm that uses load sharing. Task granularity represents a tradeoff between frequency and the amount of communication and scope of load sharing. Figure 7 shows the impact of task granularity on the performance of the Send and CachedSubMatrix algorithms when the master works

on the computation.

The x-axis in Figure 7a gives the number of columns sent in response to a request from a slave. For the CachedSubMatrix algorithm (see Figure 7b) the x-axis gives the number of sub-matrices that the original matrix is divided into. For example, 3×3 indicates that the matrix is divided into 9 sub-matrices. Thus, in Figure 7a, we move towards finer granularity as we move to the left; in Figure 7b, moving to the right takes us to finer granularity.

We can make two observations based on the data:

- The performance of the matrix-based algorithms is fairly insensitive to task granularity. Also, these algorithms exhibit robust performance to task granularity independent of whether the network is slow or fast. This is because, while the number of messages depend on the task granularity, the amount of data sent is independent of the task granularity used. In our experiments, it is equal to 10×720^4 data items.
- SubMatrix-based algorithms are sensitive to task granularity when the network is slow. However, the use of a fast network makes this algorithm robust to task granularity. This is because, the amount of data sent by SubMatrix-based algorithms is a function of granularity. For example, the amount of data sent is 7×720^2 data items when the task granularity is 3×3 and increases to 19×720^2 data items for 9×9

granularity.

- The best performance point of both algorithms is different on the slow and fast network. It moves from 3 to 5 columns for the Send algorithm and from 3×3 to 4×4 for the CachedSubMatrix algorithm.

3.5.6 The effect of sending the second matrix

Figure 8: Performance sensitivity of Broadcast and Send algorithms to distributing to all slaves second matrix versus distributing to all slaves first matrix

Initially all our algorithms were distributing second matrix to the slaves and then rows of first matrix were used as tasks. Experiments showed that algorithms which send first matrix and columns of second matrix as tasks have better performance. Figure 8 shows the comparison between the two strategies. The granularity of tasks is 3 columns

(or rows) and the master's node is used for work.

The difference in performance between distributing first and second matrix can be explained as follows:

- If second matrix is distributed to the slaves, then the slave has to calculate the cross product of the couple of rows from first matrix with certain columns of second matrix. Every entry in the column is really far away from the next one ($4 \text{ bytes} * 720 = 2880$ bytes). The first element in the column is 2073596 bytes away from the last one. Therefore, quite often during calculation cache misses occur on the node.
- If the first matrix is distributed to the slaves then the slave has to calculate the cross product of certain rows from the first matrix with a couple of columns from the second matrix. Every entry in the column is only as far as four times the number of columns. Every entry in the row is only 4 bytes away. Therefore, relatively few cache misses occur.

3.5.7 Relative performance of algorithms

3.5.7.1 Broadcast versus Send

This section presents the relative performance of Broadcast and Send algorithms. The main difference between the two algorithms is that Broadcast sends the first matrix to all slaves before sending columns of the second matrix. The Send algorithm, on the other

hand, sends the first matrix and the t columns of the second matrix together to slave 1, first matrix and the next t columns of the second matrix to slave 2, and so on, where t is the task granularity. As a result, when there is a significant time lag in receiving the first matrix by the slaves, Send should perform better. This is the case when the network does not support broadcast. Recall that PVM supports broadcast only partially. The relative performance of Broadcast and Send algorithms is shown in Figure 9.

Figure 9: Performance of Broadcast and Send algorithms (a) Master works; (b) Master does not work

The task granularity used in both cases is 3 columns. It is clear from the data in this figure that Send performs better when the network is slow. On the other hand, broadcast gives better performance for the high-speed network. Data from experiments

with generic workload (see Section 3.7.4.1) indicates that if the network supports broadcast, Send is inferior to Broadcast even if the network is slow.

3.5.7.2 Replicate versus Send

Figure 10: Performance of Replicate and Send algorithms (Master works) (a) Task granularity is 3 columns; (b) Task granularity is 30 columns

The Replicate algorithm is proposed for two main purposes: (1) to perform effective load sharing towards the end of computation and (2) to improve fault-tolerance (see Section 3.2.2.3 for details). Figure 10 shows the relative performance of Replicate and Send algorithms. When the master works, the task granularity is 3 columns (Figure 10a) and 30 columns (Figure 10b). The results when the master does not work are very similar and therefore not presented here. Note that the experimental environment used for

these experiments did not involve any background load (and hence no load fluctuations). Under these conditions, the Replicate algorithm shows improvements only in a scenario where the slow machines are delaying the job completion. This, clearly, depends on the task granularity. The larger the task granularity, the more probable is the scenario just described. The data presented in Figure 10 demonstrate that Replicate is not beneficial under no background load variations with small task granularity. Replicate, however, does provide slight performance benefit when large task granularity is used. We have also performed experiments by introducing background load variations. Under such conditions, Replicate can provide reasonable performance improvements (see Section 3.6.2 and 3.7.4.2). It should also be noted that the performance that we have reported is under a no failure case. When there are failures, using Replicate is clearly advantageous.

3.5.7.3 Store versus Send

Relative performance of Send and Store algorithms is shown in Figure 11. The results show that, in most cases, the Store algorithm performs worse than the Send algorithm. Recall that, in the Store algorithm, slaves accumulate the partial results until the computation is done. Then, each slave sends the partial results to the master. On the other hand, in the Send algorithm, slaves send partial result as soon as they compute them. Thus the trade-off between Send and Store is that Send uses several messages distributed over the computation to send the partial results whereas Store uses several messages at the end of computation to send the partial results. Thus, in Send, the master would have to handle

the partial results returned as a part of sending the next task whereas Store postpones such work to the end of the computation. On the other hand, the master could potentially be overloaded at the end of the computation in Store as all slaves send the results of their entire computation at the end.

Figure 11: Performance comparison of Store and Send algorithms (a) Master works; (b) Master does not work

The data suggests that Store performs marginally better than Send only when the master is mapped to slowest node and slow network is used. In this situation, the slow master in the Send algorithm probably spends a significant amount of time on each reply and other nodes have to wait for some time before receiving the next task. The Store algorithm was designed to eliminate this problem.

3.5.7.4 SubMatrix versus CachedSubMatrix algorithm

The performance of SubMatrix-based algorithms can be improved by reducing the amount of communication. In this improved algorithm, called CachedSubMatrix, the message length is reduced by half for a majority of the messages as it sends only one sub-matrix as opposed to two sub-matrices. Figure 12 shows the relative performance of these two algorithms.

Figure 12: Performance comparison of SubMatrix and CachedSubMatrix algorithms (divided into 3×3 submatrices)
 (a) Master works (b) Master does not work

SubMatrix algorithm needs to send 9×720^2 while CachedSubMatrix has to send 7×720^2 data items. The CachedSubMatrix algorithm shows significant improvements when the network is slow. This is a direct consequence of a reduced message length in CachedSubMatrix algorithm.

3.5.7.5 Matrix-based versus Submatrix-based algorithms

In Figure 13, the granularity of the Broadcast algorithm is 3 columns and the matrix is divided into 3×3 submatrices for the CachedSubMatrix algorithm. We have identified two classes of algorithms: Matrix-based and SubMatrix-based.

Figure 13: Performance of CachedSubMatrix and Broadcast algorithms (a) Master works; (b) Master does not work

In Matrix-based algorithms, the first matrix is sent to all slaves and is useful only when there is enough main memory to hold it. In SubMatrix-based algorithms, submatrices are sent to the slaves, which reduces the memory requirement. The relative performance of these two algorithms, which are best in their class of algorithms, is shown in Figure 13. The data in this figure suggests that the relative performance is dependent on the network speed. When the network is slow, Broadcast tends to perform worse than CachedSubMatrix. This is due to the fact that Broadcast transmits the first matrix to

slaves serially (since PVM does not use real broadcast), which causes a larger amount of data to be sent by Broadcast (11×720^2) than by CachedSubMatrix (7×720^2). When the network is fast enough to reduce the adverse effects of the initial transmission of the whole matrix, Broadcast performs better than CachedSubMatrix. Network is no longer the main bottleneck and because the master in the CachedSubMatrix algorithm has to perform large number of matrix additions, this algorithm becomes slower.

3.6 Results and discussion of experiments with background load

Since our objective is to study the performance sensitivity to various parameters and algorithms, we have also conducted these experiments in two controlled environments:

- *With no background load:* In this case, the entire system is dedicated to the matrix multiplication application. These experiments bring out the inherent tradeoffs associated with the algorithms. Further, it also represents the performance on an idle NOW (for example, running after mid-night). Also note that it has been observed that a large proportion of workstations are idle most of the time [15,17].
- *With background load:* In this case, we introduce the background load in a controlled manner so that we can obtain consistent results with multiple runs. The background load induced causes the processor to be shared between the matrix multiplication application and the background computation. Thus, the matrix multiplication

application uses the processor half the time while the background load is on. To introduce variations in the system load, we have turned the background load on for 1 second and off for 1 second cyclically. Note, however, that the background load does not induce network load.

Since there were no real performance benefits obtained on the slow network, we have conducted experiments on the fast network only. We use the execution time as the performance metric. The execution time values reported in this section are the averages of 40 runs (in most cases, with a standard deviation of less than about 1% of the mean value).

3.6.1 Broadcast versus Send under background load

Figure 14: Performance of Broadcast and Send algorithms (5 columns granularity)

(a) no background load; (b) with background load

Even if PVM implements broadcast as a set of sends, the Broadcast algorithm is very often significantly faster than the Send. If the master is fast, then Broadcast is always faster. If the master is slow but there is no load on it, there is no significant difference between Send and Broadcast. Broadcast is only slower when there is a significant load on the slow master (see Figure 14b). In these cases, the master is unable to broadcast the first matrix fast. There is a significant time lag in receiving the first matrix by the slaves and in such situations Send performs better (also see Section 3.5.7.1).

3.6.2 Replicate versus Send under background load

Figure 15: Performance of Replicate and Send algorithms under background load

(a) Task granularity is 3 columns; (b) task granularity is 30 columns

We have seen in Section 3.5.7.2 that the replicate algorithm does not show any significant improvements in a scenario without background load. When background load

was introduced there were some insignificant improvements (see Figure 15a). Replicate, however, can provide some performance benefit when large task granularity is used and background load variations are introduced (see Figure 15b). Larger load variations are needed to show that the Replicate algorithm can be significantly better than the Send. In one of the experiments, the speed of one of the slowest nodes was decreased by a factor of 4 by placing 3 background programs on it. These programs were performing infinite loops. On the fast network with task granularity of 30 columns, the Send algorithm takes on average 23.6 seconds. The Replicate was much faster with average results of 13.9 seconds. In this scenario, the Replicate provided significantly better response time.

3.6.3 Store versus Send under background load

Figure 16: Performance comparison of Store and Send algorithms (task granularity is 5 columns)

(a) No background load; (b) With background load

We can make several observations from the data presented in Figure 16. The results show that, in all cases, the Store algorithm performs worse than the Send algorithm. When fast network is used this algorithms does not seem to be useful. The network is not a bottleneck and messages can be received quickly. Therefore, conditions in which Store should be better do not occur (see also Sections 3.2.2.4 and 3.5.7.3).

3.6.4 Matrix-based versus SubMatrix-based algorithms under background load

Figure 17: Performance of Matrix based algorithms versus SubMatrix-based
(a) no background load (b) with background load

Figure 17 shows the performance of two (Send and Broadcast) Matrix-based algorithms and a SubMatrix-based algorithm (CachedSubMatrix). The main reason why results for Broadcast and Send are shown, is that with background load, Broadcast is

significantly better than Send when master is on fast node, but slower than Send when master is on the slow node (see Section 3.6.1 for reasons why this is so). Showing the performance of just one might suggest that the SubMatrix-based algorithms are better than the Matrix-based under certain conditions. When a slow network was used (see Figure 13a) SubMatrix algorithms offer better performance. Figure 17 shows that on a fast network Matrix-based algorithms are faster or just as fast whether there is a background load or not. With background load:

- *When master is fast:* Broadcast algorithm provides significantly better performance than CachedSubMatrix.
- *When master is slow:* Performance of Send and CachedSubMatrix is similar.

3.7 Results and discussion of generic workload

At computation to communication ratio of 1:1, no speedups were achieved. Therefore, it was decided to run generic workload at 10:1, 50:1 and 100:1 computation to communication ratios. Results of the generic workload with 100:1 ratio did not provide any significant insights beyond those that could have been observed with 10:1 and 50:1 ratios. Therefore, in this thesis, results of 100:1 ratio are not presented.

Experiments were done for a different number of tasks (from 9 to 1080 i.e. workload was divided from 9 to 1080 equally sized pieces respectively). Results obtained for the number of tasks greater than 270 were almost the same as that for 270 and

therefore are not presented here.

Computation to Communication ratio	Calculated execution time on one node (sec)	Best execution time on 9 nodes (sec)	Speedup achieved on 9 nodes
10:1	526	84	6.3
50:1	2630	318	8.3
100:1	5260	610	8.6

Table 2: Comparison of speedups achieved on with different computation to communication ratio

The best speedup of 8.6 was achieved as expected when computation to communication ratio was the largest (100:1).

3.7.1 Impact of computation to communication ratio

Figure 18: Performance of generic workload algorithm (Full speed - no background load) (a) 10:1 computation to communication ratio; (b) 50:1 computation to communication ratio

From Figure 18 it can be observed that the computation to communication ratio has significant impact on the performance. This was to be expected. The higher the computation to communication ratio, the more work can be done in parallel by slaves.

Therefore broadcast's 50:1 results are only around 3.8 times slower than 10:1 results.

3.7.2 Impact of background load

Figure 19: Performance of Broadcast algorithm at 50:1 ratio with and without background load

In some experiments we simulated the background load in a controlled manner so that we can obtain consistent results with multiple runs. The background load induced caused:

- 3 nodes to run as if they had only 1/3 of time available for computation
- 3 nodes to run as if they had only 2/3 of time available for computation and
- 3 nodes were running at full speed.

That created a NOW system that had 2/3 of computing power of an unloaded one. Note, however, that the background load does not induce network load. The results of simulation running with background load were in fact around 33% slower in most cases (see Figure 19).

From Figure 19 we can also see that the performance of simulation running without the background load does not depend on the tasks granularity. But the simulation running with the background load is sensitive to task granularity. When the workload was only divided into 9 tasks, there was only one task per slave. The computation was proceeding only as fast as slowest slave running at 1/3 of full speed allowed it to. Smaller granularity allowed for more load sharing and therefore performance improved. When there were 45 and 63 tasks the results were more than 33% slower because one of the last tasks was given to the slave running at 1/3 of the speed. But the larger the number of tasks, the smaller each task is and the smaller the impact of mapping last tasks to slaves running on slowest nodes.

3.7.3 Impact of random background load

Figure 20: Performance of Broadcast algorithm (50:1) under background load

Because the loads on nodes are random, the results for 9 tasks are not shown here.

They vary a lot depending on what loads were randomly selected.

From Figure 20 we can see that the simulations with a small number of tasks for random load perform much worse. This is due to situations where one of the last tasks was given to the slave running at 1/3 of the speed. Decreasing task granularity causes lowering the performance difference between the simulations.

3.7.4 Relative performance of algorithms

3.7.4.1 Broadcast versus Send

Figure 21: Performance of Broadcast and Send algorithms (a) 10:1 no background load; (b) 50:1 1/3 background load

It is quite obvious from Figure 21 that the Broadcast algorithm always performs better than the Send algorithm. Recall that the Send algorithm sends a lot more data across the network than the Broadcast. While Broadcast only sends $3 \cdot N^2$ data items, the Send sends $11 \cdot N^2$ data items for $N \times N$ matrices. Computation to communication ratio is actually calculated for the Broadcast algorithm and not for Send. In the Send algorithm

initial distribution of the first data set (see Sections 3.2.2.2 and 3.4) which has to be sent to each slave takes a long time and hurts its performance. The Broadcast algorithm does not have this problem.

With 1:1 computation to communication ratio Send algorithm was also slower than Broadcast. The best result achieved for Send was around 182 seconds – while for Broadcast it was around 50 seconds.

3.7.4.2 Broadcast versus Replicate

Figure 22: Performance of Broadcast and Replicate algorithm at 50:1 ratio (a) all nodes running at full speed

(b) 3 nodes running at 1/4 speed, 3 at 1/2 speed and 3 at full speed

Conclusions that can be drawn from results shown in Figure 22 are similar to those from Sections 3.5.7.2 and 3.6.2. Recall that the Replicate algorithm for generic workload is based on the Broadcast algorithm. When all nodes run at the same speed, the Replicate

will not and should not be faster than Broadcast (see Figure 22a). When nodes run at different speeds, the Replicate can be significantly faster than Broadcast but only for coarse granularity (see Figure 22b). Replicate would be more beneficial if differences between speeds of nodes were even greater.

Chapter IV

Variable Granularity Algorithms

We describe several algorithms that use variable task granularity. These algorithms have been proposed for multiprocessor systems for loop scheduling. All of them are based on the Send algorithm described earlier. The difference between fixed and variable task granularity algorithms is that the task size t decreases with time. For example, the first task may have a size of 10 columns but the last one may have only one column (see Figure 23). This can improve performance in two ways:

- These algorithms attempt to minimize the time the master has to wait to receive the last result. While distributing the tasks, it can happen that the last task is sent to the slowest machine. If the task sizes are static, as in the Send and Store algorithms, then the master will have to wait for a relatively long time to receive the last result. On the other hand, if the task sizes are decreasing, even if the last task is sent to the slowest machine, it should take only a reasonable amount of time to complete the task because the last task size is much smaller than the previous task sizes.

- It is well known that sending many short messages results in worse performance than when a small number of large messages is sent. In case of matrix multiplication, depending on the parameters used, variable task size algorithms can potentially decrease the number of messages sent while increasing their average size. Even though the data in Figure 23 suggests that the variable task granularity algorithms tend to create more tasks than the static, fixed granularity algorithm, we will show in section 4.3 that for the parameters we have used in our experiments, the opposite is true.

Several strategies have been proposed to decrease the task sizes [14, 21, 28]. The following three strategies were implemented: guided self-scheduling [21], factoring [11,14] and trapezoidal self-scheduling [28].

4.1 Description of variable task granularity algorithms

4.1.1 Guided self-scheduling

In guided self-scheduling (GSS) [21], task size is a function of the remaining columns. Typically, task size is set to $1/G$ of the remaining columns. Figure 23 shows the task size distribution using $G = 4$ for a 70 column matrix. For example, we can select G to be equal to the number of slaves (or processors). Choosing G this way can lead to diminished load sharing as large tasks will take much more time on slow machines than on fast ones. Therefore, we have experimented with various G values. Section 4.3.2.1 presents these results.

Figure 23 :Task sizes for the fixed and variable task granularity algorithms (for a 70 column matrix)

4.1.2 Factoring

Factoring [11,14] creates smaller and more uniform task sizes than the guided self-scheduling. Tasks come in groups of S where S is the number of slaves. Each task in a group has the same size. Tasks' size in each group is a constant factor of the size of tasks in the previous group. The optimal factor can be calculated based on the variance in the task computation time [14]. However, in the type of systems that are being considered here, loads can vary dynamically, which makes it unreasonable to assume that variance is known *a priori*. As a good heuristic we set the task size in the subsequent group to be half of the previous group task size. Figure 23 shows the task sizes for a 70 column matrix. In this example, the initial task size F is set to 8 and the number of slaves is assumed to be 4. We have conducted experiments to study the sensitivity to the initial task size F (see Section 4.3.2.3 for details).

4.1.3 Trapezoidal self-scheduling

In trapezoidal self-scheduling [28], task sizes decrease linearly unlike in factoring and guided self-scheduling where they decrease exponentially. The size of first task can be arbitrary. In our implementation, we have used the starting task size as the total number of tasks/(2 * number of slaves). Figure 23 shows the task size distribution for a 70-column matrix. The number of slaves is assumed to be 4. The decrement value T used in this example is 1. We have experimented with various decrement values (see Section 4.3.2.2 for details). A variation of these algorithms specifies the minimum size of a task. This can eliminate the long tail of tasks with size 1 (see Figure 23). We have implemented these

three algorithms (without the modification suggested above) in the Send type algorithm. For this reason, in our results section, we compare the performance of these algorithms with the Send algorithm. Note also that these can also be used to augment the Store algorithm.

4.1.4 Adaptive task granularity algorithm

In this algorithm, the task granularity adapts to the slave node's response time. If a node is slow and/or heavily loaded, it tends to receive finer granularity tasks. The expectation is that the performance can be improved by decreasing the number of messages while increasing their size. For example, if a node is fast, instead of it requesting several times fine granularity tasks, it could be given coarse granularity tasks fewer times. The algorithm works as follows: Initially, C columns are sent to each slave. The master measures the amount of time it takes for each slave to multiply the C columns and return the results. The next time a slave requests more columns, the slave's response time is compared with the average response time of the slaves and the number of columns will be adjusted based on this information. For example, if a slave's multiplication rate is 5 columns/unit time but the average is 10 columns/unit time, it will be given $C/2$ columns as the next task. In order to make sure that the task granularity will be within some reasonable bounds, the algorithm requires lower and upper bounds on the number of columns. As a result, the algorithm takes three parameters: the number of columns C , the minimum number of columns min , and the maximum number of columns max that can be

sent to slaves.

4.2 Generic workload

We have tried to make algorithms and message exchange mechanisms used in simulation similar to the ones in the matrix multiplication algorithms. Three fixed granularity algorithms – Trapezoid self-scheduling, Factoring and Guided self-scheduling – were simulated using generic workload. Please refer to Sections 4.1.1, 4.1.2, 4.1.3 for a description. The same modifications as in fixed granularity algorithms for generic workload were employed here. The following are the differences between algorithms used in matrix multiplication and the generic workload application:

- In generic workload application there is no first matrix or second matrix so all data items (20,000,000) have to be received by the slave before receiving tasks.
- In matrix multiplication Trapezoid self-scheduling, Factoring and Guided self-scheduling algorithms are based on the Send algorithm. Trapezoid self-scheduling, Factoring and Guided self-scheduling algorithms in generic workload application are based on Broadcast because it has the computation to communication ratio that we need (all data items are broadcast not sent).

Variable task granularity algorithms are constantly decreasing the task size. In order to avoid tasks of very small size, minimal task size was set to 20,000 (1/10,000 of 20,000,000).

4.3 Results and discussion

4.3.1 Comparison of fixed and variable granularity algorithms

4.3.1.1 Variable task granularity algorithms versus Send

Figure 24: Performance of variable task granularity algorithms (a) Master works (b) Master does not work

Varying task granularity has been shown to be useful in multiprocessor systems [14]. These studies have shown that fixed task granularity algorithms perform worse than variable task granularity algorithms. Since communication characteristics of multiprocessors and NOW-based systems are quite different, (as discussed in Chapter I), we wanted to investigate the effectiveness of variable task granularity. We have implemented the four variable task granularity algorithms described in Section 4.1. Results obtained for Guided, Trapezoidal and Factoring algorithms are presented in Figure 24. For comparison, we have also included the execution times of Send. Send uses a fixed task

granularity of 3 columns. In Factoring, task granularity starts with 40 columns and the task granularity of each task group is half of the previous group. In GSS algorithm, task granularity is set equal to the $1/14$ of the remaining tasks. In TSS algorithm, task granularity starts at $720/(2*9) = 40$ columns and task granularity decreases linearly by 5 columns. For the parameters in Figure 24 the Send algorithm generates 240 tasks, GSS algorithm generates 74 tasks, Factoring generates 72 tasks, and TSS generates 27 tasks.

- *Fixed versus variable granularity algorithms:* It is clear from this figure that there is no definite advantage to using a variable task granularity algorithm. This result is in contrast to the results reported for multiprocessor systems where the performance gains are significant when the variable task granularity algorithms are used [14,21]. Also note that this is not due to the fine granularity (of 3 columns) used in Send. As shown in Section 3.5.5, the performance of Send is fairly insensitive to the task granularity.
- *Relative performance of variable granularity algorithms:* The data in Figure 24 suggests that, performance of all three algorithms is fairly similar. In general, the GSS algorithm provides the best performance, followed by Factoring and TSS algorithms. This is true even if one compares the best performance provided by these algorithms as shown in Section 4.3.2. Trapezoid self-scheduling in certain cases seems to perform much worse than the other algorithms. This is partially due to this algorithm's sensitivity to the task granularity chosen (see Section 4.3.2.2). Performance on the

slow master as seen on Figure 24 is particularly bad as compared to other algorithms because the task granularity decreases linearly by 5 columns. If task granularity were to decrease by 1 column, then results on the slow master would be very similar to other algorithms, but results on the fast master would be significantly worse than of other algorithms. In practice, however, it is not possible to determine the fastest node or slowest node due to dynamic load changes. Therefore due to the sensitivity of the Trapezoid algorithm to speed of the master node, it is probably the least desired algorithm of the three for practical use.

The performance sensitivity of these algorithms to their parameter values is discussed in Section 4.3.2.

4.3.1.2 Comparison of Adaptive and Send algorithms

Figure 25: Performance of Send and Adaptive algorithms (a) master works (b) master does not work

This section discusses the performance of the Adaptive algorithm described in Section 4.1.4. Recall that this algorithm requires three parameters: a number of columns C that is the basis for the initial task granularity, minimum task granularity min columns, and maximum task granularity max columns. We use the notation (C, min, max) to represent these three parameters. We have implemented the Adaptive algorithm on the basic Send algorithm. Therefore, Figure 25 shows the relative performance of Send and the Adaptive algorithms. The Send algorithm uses 3 column tasks and the Adaptive algorithm uses (3, 3, 9) parameters (remember that the Send algorithm performs best with 5 column tasks on the fast network). The sensitivity of the Adaptive algorithm to its parameters is shown in Figure 29.

It can be seen from these plots that, on the slow network the Adaptive algorithm performs only slightly better than Send. On the fast network, the Adaptive algorithm performs worse than the Send algorithm. While we do not want to generalize this conclusion just based on a single Adaptive algorithm, we do offer some insights into the underlying behavior that causes this relative performance. Adaptive algorithms assume that the past load information can be used to predict the future load and thus can be used to adjust the next task size. However, there may be several factors that may diminish the predictability of response time. These include:

- The response time, as seen by the master node, includes the time required by the master to complete a computation and the delay associated with a round trip on the

communication network. Since the response time depends on the performance of these two system components, the capability of the response time to predict diminishes.

- The predictability of response time information also depends on the task granularity. As the task granularity increases, the system load (both CPU and network loads) should change more slowly. However, network traffic may occur in bursts and may not satisfy this requirement.

For these reasons we are not convinced that Adaptive algorithms would work well for load sharing in this type of system. Further investigation is warranted to establish if Adaptive algorithms would yield substantial performance benefit over simple load sharing algorithms; if so, what type of algorithms are useful and under what system and workload conditions would they perform better?

4.3.2 Sensitivity to task granularity

All four algorithms were sensitive to the task granularity. Trapezoid self-scheduling was the only algorithm whose best performance point changes depending on speed of the master node. The other three algorithms' best points are not significantly influenced by speed of the master node.

4.3.2.1 Guided self-scheduling and task granularity

The GSS algorithm implemented has a single parameter G . Recall that the next task size is equal to $1/G$ of the remaining columns. Thus, a smaller G value causes the task

granularity to increase and vice versa. The performance sensitivity of the GSS algorithm to parameter G is shown in Figure 26. In this plot, we move towards finer granularity tasks as we move from left to right.

Figure 26: Performance sensitivity of the Guided self-scheduling algorithm to parameter G (a) master works; (b) master does not work

On the slow network, the GSS algorithm is fairly insensitive to the value of G . On the other hand, results show that when the fast network is used, performance of the GSS algorithm is quite sensitive to the value of G . Overall, the best value for G seems to be 14. When the slow master is used, the first task is given to the slowest node and therefore results for the slow master are much worse than for the fast master when $G = 6$ or $G = 7$ (if the fast master is used then the first and largest task is given to the fastest node). The

first node will receive a very large task to do and will become the bottleneck on the fast network. All other tasks are smaller and will be completed before the first one.

The sensitivity of this algorithm to the value of G on the fast network increases when the master performs only coordination and is not working on the multiplication.

- *The master is fast:* If it is not used for work then the response time increases at $G = 6$ or $G = 7$ because the fast node has been taken away from the worker pool and the first largest task now goes to a node other than the one on which master runs. This does increase significantly the amount of communication. Now not only the largest task is a bottleneck but also it is on a remote node so once it is finished the master will have to wait a significant amount of time before it receives its results.
- *The master is slow:* If it is not used for work at $G = 12$ or $G = 14$, the response time is lower because only the slowest node was removed from the worker pool and the master on this node can spend all its time managing tasks.

4.3.2.2 Trapezoidal self-scheduling and task granularity

The trapezoidal algorithm uses a single parameter T that is the difference between successive task sizes. Thus, smaller T values result in coarse granularity tasks. The performance sensitivity of the trapezoid algorithm to a decrement value T is shown in Figure 27.

Figure 27: Performance sensitivity of Trapezoidal self-scheduling algorithm to parameter T

(a) Master works; (b) Master does not work

Trapezoid self-scheduling is quite sensitive to the parameter T (see Figure 27). All variable granularity algorithms except this one have usually some initial parameter for which they perform best. This is not the case with Trapezoid self-scheduling. The best performance of this algorithm depends also on the speed of the master node. The reason for the difference in performance can be seen by examining the number of tasks generated by this algorithm for different task decrease parameter T . When T is 1 there are 27 tasks, when T is 3 there are 447 tasks and for $T = 5$, 548 tasks are created.

When work is done on the slow node on which the master resides, the slow master becomes a bottleneck if it has to deal with a large number of messages. Therefore it works

best with a small number of tasks (27).

If the master runs on a slow node that is not used for work and the slow network is utilized then the slow master does not become a bottleneck and is able to manage a large number of tasks. Thus it works best with a large number of tasks (548).

When the master runs on a fast node it is always able to handle a large number of messages without becoming a bottleneck, therefore it works best with a large number of tasks (548).

4.3.2.3 Factoring self-scheduling and task granularity

Figure 28: Performance sensitivity of Factoring algorithm to parameter F (a)

Master works; (b) Master does not work

The factoring algorithm we have implemented uses a single parameter that sets the

initial task size F . Choosing a larger F value moves the system towards coarse granularity. The performance sensitivity of the Factoring algorithm to the initial task size F is shown in Figure 28. Moving to the right implies finer task granularity. Optimum task granularity was obtained with an F value of 40 that generates 72 tasks.

The performance of this algorithm for $F = 70$ and 60, is worse than for lower values of F such as 40. At $F = 70$ and 60 task granularity is very coarse and therefore load sharing is poor. For $F = 70$ there are only 12 tasks and the size of the smallest one is 20 columns. For $F = 60$ there are 15 tasks and the size smallest one is 30 columns.

When $F = 30$ there are 234 tasks, the slow master starts to become the bottleneck and performance suffers.

4.3.2.4 Adaptive algorithm and task granularity

Figure 29 shows the performance of the Adaptive algorithm depending on different (C, min, max) values. Recall that this algorithm requires three parameters: a number of columns C that is the basis for the initial task granularity, the minimum task granularity min columns, and the maximum task granularity max columns. Optimum parameters seem to be (3, 3, 9). Recall that C between 3 and 5 is also an optimum value for Send and Broadcast algorithms. The following observations can be made from Figure 29:

- *If the master works* (Figure 29a): The slow master is much more sensitive to the (C, min, max) values. When $C = 1$, the number of tasks is very large and the slow master

becomes a bottleneck. If $C > 1$ then the slow master on the fast network is as fast or faster than the fast master. The slow network using the fast master produces significantly better performance.

- *If the master does not work* (Figure 29b): Using the slow master always offers better performance than using the fast master. If the master runs on the fast node and it is not used for work, a significant amount of computing power is wasted.

Figure 29: Performance of Adaptive algorithm with different granularity

(a) master works (b) master does not work

4.3.3 Performance under background load

Background loads are introduced in exactly the same fashion as for the static granularity algorithms (see section 3.6 for details).

4.3.3.1 Performance of variable task granularity algorithms

Figure 30: Performance of variable task granularity algorithms on fast network

(a) Without background load; (b) With background load

The performance of the three variable task granularity algorithms running on the fast network with and without a background load is presented in Figure 30. For comparison, we have also included the execution times of Send. Send uses a fixed task granularity of 5 columns unlike 3 in the some of the previous figures. At a task granularity

of 5 columns Send performs better when the fast network is used. In Factoring, task granularity starts with 40 columns and the task granularity of each task group is half of the previous group. In the GSS algorithm, task granularity is set equal to $1/14$ of the remaining tasks. In the TSS algorithm, task granularity starts at $720/(2*9) = 40$ columns and task granularity decreases linearly by 5 columns.

Again similar to the results obtained when there was no background load (see Section 4.3.1.1), when a background load is introduced variable task granularity algorithms did not produce significant performance benefits. With the background load (Figure 30b) the Guided self-scheduling algorithm seems to be always faster than Send but the difference is not significant. The Factoring algorithm seems to be on par with Send, while the Trapezoidal self-guiding is always slower than Send. Again there is no definite advantage in using a variable task granularity algorithm. The reasons for the particularly poor performance of the Trapezoid algorithm are described in Section 4.3.2.2.

4.3.3.2 Performance of the Adaptive algorithm under a background load

In Figure 31 Send granularity is 5 columns and the Adaptive algorithm uses parameters $(C, min, max) = (3, 3, 9)$. From the figure we can clearly see that the Adaptive algorithm does not provide any performance benefits over Send. One second fluctuations in the background load (see section 3.6) can partially explain the poor performance of the Adaptive algorithm because it counts on the predictability of slaves' response times.

Figure 31: Performance comparison of Send and Adaptive algorithms
 (a) without background load (b) with background load

4.4 Results for a generic workload

A background load was implemented in a way which causes the first 3 tasks (and in case of variable granularity algorithms - three largest tasks) to be sent to the nodes that run at 1/3 of the full speed. This is the worst possible case for variable granularity - when the first largest tasks are given to the slowest nodes because the largest tasks can easily become the bottlenecks and they do in several cases shown below. However, due to dynamic load changes, it is not possible to determine the fastest or slowest node, therefore it is possible to send the first largest task to the slowest node.

We conducted experiments with the same initial parameters for Generic Workload

variable algorithms as those used in matrix multiplication variable granularity algorithms. The table below shows which initial parameters in Generic Workload correspond to parameters from array multiplication. Guided self-scheduling parameters (G) are exactly the same as those for matrix multiplication.

Algorithm	Generic workload parameter	Corresponding Array multiplication parameter
Factoring (F)	1,666,666	60
Factoring (F)	1,388,888	50
Factoring (F)	1,250,000	45
Factoring (F)	1,100,000	40
Factoring (F)	900,000	30
Trapezoid (T)	28,000	1
Trapezoid (T)	30,000	None
Trapezoid (T)	32,000	None
Trapezoid (T)	37,000	None
Trapezoid (T)	43,000	5

Table 3: Corresponding parameters between array multiplication algorithms and generic workload algorithms

Parameters (F) for Factoring have been chosen so that they are percentage-wise the same as the parameters in matrix multiplication.

The number of tasks generated for a given T in the Trapezoid self-scheduling algorithm for matrix multiplication application had the biggest influence on performance. It was not desirable to use the same T percentage-wise as in matrix multiplication. For example, for $T=5$ in matrix multiplication, corresponding percentage-wise T in a generic workload would have to be 138,889 which would generate 1508 tasks. Recall that the number of tasks had the biggest influence on performance and here three times as many

tasks as in matrix multiplication would have been generated. Therefore a T was found for which the number of tasks was the same as or very close to the number of tasks in array multiplication. $T=43000$ was used to produce a similar number of tasks (535) as $T=5$ in array multiplication and $T=28000$ was used to produce 27 tasks the as for $T=1$ in array multiplication. Other selected T values used produced a number of tasks between 27 and 535 ($T=37000$ produced 307 tasks, $T=32000$ produced 51 and $T=30000$ produced 29).

4.4.1 Impact of the computation to communication ratio

Figure 32: Performance of variable task granularity algorithms (no background load) (a) 10:1; (b) 50:1

The results obtained are similar to those presented in Section 3.7.1 which examines the impact of the computation to communication ratio on the Broadcast algorithm. From Figure 32 it can be observed that the computation to communication ratio has a significant

impact on the performance. The higher the computation to communication ratio, the more work can be done in parallel by slaves. Therefore algorithms at 50:1 ratio produce results which are only around 3.7 times slower than at a 10:1 ratio.

4.4.2 Impact of background load

Figure 33: Performance of variable task granularity algorithms at 50:1 ratio

(a) no background load; (b) background 1/3 load (non random)

Adding a background load had a severe impact on performance. The resulting NOW should be about 2/3 of the speed of the NOW without a background load. The best results from experiments with a background load were, as expected about 1/3 slower than the best results of experiments without a background load. Recall that a background load was implemented in a way which causes the first three tasks (and in case of variable granularity algorithms - the largest three tasks) to be sent to the nodes that run at 1/3 of

the full speed. This means that the first largest task was given to the slowest node and this node has become the bottleneck.

4.4.3 Impact of a random background load (versus 1/3 load)

Figure 34: Performance of variable task granularity algorithms at 50:1 ratio

(a) background 1/3 load (non random); (b) random background 1/3 load

From Figure 34 it can be seen that with a random background load results are much more random. The overall trend is for results to become more stable and closer to the results of other algorithms as granularity decreases. Because results shown are averages of only three runs, it is easy to make false predictions from them. Because the first largest task might not be sent to the slowest node, results for coarse granularity cases where the first task was a bottleneck improved most of the time. At fine granularity, results with a random background load were significantly worse than the results of a non-

random load. This is again due to the situation where one of the large tasks becomes a bottleneck.

4.4.4 Sensitivity to task granularity

From Figure 32 it seems that the larger the computation to communication ratio the more impact task granularity has on the computation. Result differences between the GSS algorithm at $G=6$ and $G=14$ at 10:1 ratio are 17%, result differences between GSS at $G=6$ and $G=14$ at 50:1 ratio are 43%.

Recall that a background load was implemented in a way that causes the first 3 tasks (and in case of variable granularity algorithms - three largest tasks) to be sent to the nodes that run at $1/3$ of the full speed. The largest tasks are given to the slowest nodes and these nodes became the bottlenecks. This scenario occurs for all 5 results of the GSS algorithm and the 3 first results of the Factoring algorithm in Figure 33b. The results were directly linked to the size of the first task created (3,333,333 for $G=6$ and 1,666,666 for $F=1,666,666$). This scenario also occurs for $G=6$ and $G=7$ on Figure 33a but the impact was not as severe because all nodes run at the same speed.

4.4.5 Relative performance

In Figure 35, the following initial parameters were used: Broadcast algorithm was set to have 270 tasks, for Factorial algorithm $F=900000$, for GSS $G=14$, for TSS $T=43000$. Again similar to results presented in other sections, there does not seem to be any

benefit to using variable task granularity algorithms.

Figure 35: Comparison of algorithm performance. (a) 10:1 computation to communication ratio; (b) ratio is 50:1

Chapter V

Conclusions

We have conducted experiments to study the performance of several load sharing algorithms in a NOW-based parallel system using PVM. A variety of fixed and variable task granularity algorithms were implemented. We have used matrix multiplication and generic workload applications to study the relative performance of these algorithms to various system and workload parameters. We have also used the standard Ethernet (10Mbps) and Fast Ethernet (100Mbps) to investigate the impact of the network speed. The following two sections present some of the conclusions derived from the results reported in this thesis.

5.1 Conclusions for matrix multiplication application

The following conclusions apply to the matrix multiplication problem:

- The speed of the network has a substantial impact on the execution time of matrix multiplication. The computation to communication ratio also has a substantial impact on the performance. The larger the ratio, the higher the efficiency of the NOW-system.

- The SubMatrix-based algorithm seems to be more sensitive to task granularity since decreasing task granularity increases the amount of data that needs to be sent and also increases the amount of computation that the master has to perform.

Relative performance of algorithms:

- *Broadcast vs. Send* – Send algorithm is faster for matrix multiplication on the slow network. This is due to PVM not implementing a real broadcast operation. Data from experiments with a generic workload indicate that if the network supports broadcast, Send is inferior to Broadcast even if the network is slow.
- *Send vs. Store* – In most cases, the Send algorithm performs better than the Store algorithm. However, these results should be viewed in the context of the matrix multiplication application used. The matrix multiplication does not involve significant computation by the master in processing the partial results received from the slaves. In other applications, this may not be true.
- *SubMatrix vs. CachedSubMatrix* – The CachedSubMatrix algorithm is always better because it decreases the amount of communication.
- *Matrix-based vs. SubMatrix-based* – It seems that the only reason for the better performance of SubMatrix-based algorithms on the slow network is that they require

less data to be sent. Therefore, if a real broadcast is supported by the software then Broadcast (Matrix based algorithm) should always perform better because it involves much less communication than any of the SubMatrix-based algorithms. If broadcast is not supported then the SubMatrix-based algorithm might prove to be better if the network becomes the main bottleneck.

5.2 General conclusions

The following conclusions should apply to most applications no matter whether or not there is a background load on the nodes:

- The speed of the network has a substantial impact on the execution time of application. The computation to communication ratio also has a substantial impact on the performance. The larger the ratio, the higher the efficiency of the NOW-system.
- Load sharing algorithms provide better performance than the no-load sharing ones even when the network is slow. However, no load sharing algorithms perform as well as the load sharing ones if the task distribution order is proper (i.e., from the fastest machine to the slowest one). Determination of this order, however, is not possible in practice due to dynamic load changes.
- Performance benefits of mapping the master program to a faster or slower node

depends on several factors including the speed of the network and whether the master node is strictly responsible for coordination or whether it performs computation in addition to coordination. In practice, however, it is not possible to determine the fastest machine due to dynamic load changes.

- The master should work on computation if it is on a fast machine. Since it is not possible to determine if a machine is “fast” or “slow” due to dynamic load changes, the slave process should be given much lower priority than the master.
- Task granularity is important - very coarse granularity might severely decrease load sharing among the nodes because the slowest node might become a bottleneck. After a certain point, a further decrease in task granularity does not improve the performance; it can even hurt the response time because a large number of tasks means a large number of messages and a high load on the master.
- If the network has a low bandwidth it might be beneficial to use small task sizes (assuming the task size is proportional to the amount of data sent). If a large task, which contains large amount of data, is sent to the first slave, then a significant time lag is introduced in receiving the tasks by the other slaves. This would cause less work to be done in parallel by the nodes as can be seen in section 3.5.7.1.

- The Replicate algorithm will provide substantial benefits if the task granularity is coarse or if there are large variations in the speed of the nodes. Using Replicate is clearly advantageous when there are node failures.
- The Hierarchical algorithm has worse performance than the other algorithms. This algorithm tries to stop the master from becoming a bottleneck. However, the master was not the bottleneck in our experiments, the network was. This algorithm causes a larger amount of data items to be exchanged between the nodes than in the other algorithms.

Variable granularity algorithms:

- Variable task size algorithms, proposed and found to be beneficial for multiprocessors, are not found to be useful in our experiments. Their performance is worse or the same as fixed granularity algorithms. In addition, because the first tasks generated by these algorithms are large, there is a chance that these large tasks will become a bottlenecks if they are assigned to slow nodes. A possible research topic might be to investigate whether variable task size algorithms are beneficial if the master has to do a lot of work for each task because these algorithms decrease the number of tasks.
- A problem with the variable task granularity algorithms is that it is important to set appropriate parameters to get a good performance. Since NOW-based parallel

systems have highly volatile system loads, it is probably difficult to predict appropriate parameters. Further investigation is required to study this problem.

- Performance of the GSS and Factoring algorithms seem to be similar. The performance of the TSS algorithm appears to be worse than that of these two algorithms.
- The Adaptive algorithm we have implemented did not provide performance improvements over the Send. In fact, Send performed better in almost all cases. We have identified some possible reasons for this. Further investigation is warranted to establish if Adaptive algorithms would yield a substantial performance benefit over simple load sharing algorithms; if so, what type of algorithms are useful and under what system and workload conditions would they perform better?

5.3 Further research

In addition to the suggestions for further research made in the last section, this study can be extended in several other directions:

- What are the potential benefits of using both networks?
- What are the advantages (if any) of using a combination of algorithms? For example a hybrid algorithm that combines a variable task granularity algorithm with a Replicate or a CachedSubMatrix with a Replicate.
- How does the network load influence the performance of the algorithms?
- Whether or not the Hierarchical algorithm will work well when the network is not a bottleneck or network topology is such that communication between the supermasters and submasters does not interfere with communication between submasters and slaves?

Further study is also needed to understand the impact of I/O intensive applications.

Appendix 1: Complexity for Fixed Granularity Algorithms

This appendix gives equations for the number of data items sent and the number of messages sent by four algorithms. Because all other algorithms are based on Broadcast and Send, the same equations apply to the algorithms they are based on.

	Broadcast	Send	SubMatrix	CachedSubMatrix
Data items sent	$3N^2$	$(2 + nnodes)N^2$	$3dN^2$	$(1 + 2d)N^2$
# of messages	$1 + \frac{N}{C}$	$nnodes + \frac{N}{C}$	d^3	d^3

Table 4: Equations for number of data items sent and number of messages sent by four fixed granularity algorithms.

In above tables:

- N - size of array (720 for matrix multiplication); number of data items (20,000,000) for generic workload
- $nnodes$ - number of nodes (9 for our NOW)
- C - Task size
- d - Matrix is divided into $d \times d$ submatrices

Appendix 2: PVM Communication Primitives Used

This appendix provides information on the kind of performance that can be expected from the PVM communication primitives. There is a variety of PVM message routing and message packing options. Each combination of these produces different message latency and bandwidth.

There are two routing options in PVM, `PvmDontRoute` and `PvmRouteDirect`. Option `PvmRouteDirect` allows direct routing where one task sends messages to another through a TCP link, avoiding the overhead of copying them through the pvmds. In `PvmDontRoute` any message sent to another task is routed to the pvmd, which forwards it to the destination. The advantage of direct links is the observed factor of two boost in communication performance. Option `PvmRouteDirect` does not scale well because some UNIX systems allow only a small number of direct links.

Since `PvmRouteDirect` offers double the bandwidth and in our case the number of tasks is small, only the `PvmRouteDirect` message routing option was used in the experiments.

There are three encoding (packing) options available in PVM:

- `PvmDataDefault` - XDR encoding is used by default because PVM does not know if the user is going to add a heterogeneous machine before this message is sent. Because

all nodes in our NOW-system use the same encoding, this option is not used in experiments to save on encoding costs.

- `PvmDataRaw` - no encoding is done. Messages are copied to the buffer and sent in their original format.
- `PvmDataInPlace` - data left in place. Buffer contains only sizes and pointers to the items to send. When `pvm_send()` is called, the items are copied directly out of the user's memory. This option decreases the number of times the message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent.

PVM also supplies `pvm_psend()` routine, which combines many steps required for sending messages with above packing options into a single call. But `pvm_psend()` only packs and sends a contiguous array of a single data type.

`PvmDataInPlace` was used whenever possible because it results in the largest bandwidth. It's not always used because it copies data directly from user space and data may be modified before it is sent.

The second highest bandwidth was obtained by using `pvm_psend`. The drawback is that `pvm_psend` can only send a single continuous chunk of data.

It was discovered that when a message is composed of many disjoint chunks of

memory, PvmDataInPlace performs much worse than the PvmDataRaw option. Therefore, in such cases, PvmDataRaw was used.

Table 5 below shows the bandwidths that were obtained during the experiments. In the table “Fast network” means 100 Mbit/sec Ethernet, “Slow network” means 10 Mbit/sec Ethernet, “Slow nodes” indicate that bandwidth was measured between Pentiums-133 and “Fast nodes” indicate that the bandwidth was measured between PentiumPros-200.

Communication Method	Fast network Slow Nodes	Fast Network Fast Nodes	Slow network Slow Nodes	Slow Network Fast Nodes
PvmDataDefault	1.71	3.24	0.80	0.88
PvmDataRaw	2.74	5.31	0.96	0.99
PvmDataInPlace	5.04	6.84	1.04	1.04
pvm_psend	2.39	6.82	1.02	1.04

Table 5: Bandwidths obtained on NOW-system during experiments in Mbytes/sec.

References

- [1] R.C. Agarwal, F.G. Gustavson, M. Zubair, "A high performance matrix-multiplication algorithm on a distributed memory parallel computer, using overlapped communication", *IBM Journal of research and development* vol. 38 NO. 6 November 1994
- [2] T. Anderson, David Culler, David Patterson, and the NOW team, "A Case for NOW (Networks of Workstations)", *IEEE Micro*, 15(2), February 1995, pp.54-64.
- [3] S. Ayachi and S. P. Dandamudi, "A Hierarchical Processor Scheduling Policy for Multiprocessor Systems", *IEEE Int. Symp. Parallel and Distributed Processing*, New Orleans, October 1996.
- [4] D. Babbar and P. Krueger, *Stealth: a liberal approach to distributed scheduling for networks of workstations*, Technical Report, OSU-CISRCI/93-TR6. Ohio State University, 1993
- [5] A. Bricker, M. Litzkow and M. Livny, *Condor technical summary* Computer Sciences Department University of Wisconsin - Madison, 1991
- [6] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, J. Walpole, "MPVM: A Migration Transparent Version of PVM", Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology, February 1995

- [7] S. L. Chang, D. H. C. Du, J. Hsieh, and R. P. Tsang, "Enhanced PVM Communications over a High-Speed LAN", *IEEE Parallel and Distributed Technology*, Fall 1995, pp. 20-32.
- [8] S. P. Cheng, and S. P. Dandamudi, "Scheduling in Parallel Systems with a Hierarchical Organization of Tasks", *Proceedings of the 12th International Conference on Distributed Computer Systems*, June 1992.
- [9] S. P. Dandamudi and S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems", *IEEE Trans. Parallel and Dist. Syst.*, Vol. 6, Jan. 1995, pp. 1-16.
- [10] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "A Message Passing Standard for MPP and Workstations", *Comm. ACM*, Vol. 39, No. 7, July 1996, pp. 84-90.
- [11] L. E. Flynn and S. F. Hummel, "Scheduling Variable-Length Parallel Subtasks", *IBM Research Report RC 15492*, 1990.
- [12] G. A. Geist and V. S. Sunderam, "Network-Based Concurrent Computing on the PVM System", *Concurrency: Practice and Experience*, Vol. 4, No. 4, June 1992, pp. 293-311.
- [13] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass., 1994.
- [14] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops", *Comm. ACM*, Vol. 35, No. 8, August 1992, pp. 90-101.
- [15] P. Kruger and R. Chawla, "The Stealth Distributed Scheduler", *IEEE Int. Conf.*

Dist. Computing Systems, 1991, pp. 336-343.

- [16] "MediaCUBE Performance Brief" the web page of nCube
<http://www.ncube.com/products/1kstream.html>
- [17] M. Mutka and M. Livny, "The available capacity of a Privately Owned Workstation Environment", *Performance Evaluation*, Vol. 12, No. 4, July 1991, pp. 269-284.
- [18] B. C. Neuman, S. Rao, "Resource Management for Distributed Parallel Systems", *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, Spokane, July 1993.
- [19] A. Piotrowski, S. P. Dandamudi, "Performance of a Parallel Application on a Network of Workstations", *Proceeding of the 11th Annual International Symposium on High Performance Computing Systems*, July 1997, pp. 429-440
- [20] A. Piotrowski, S. P. Dandamudi, "A Comparative Study of Load Sharing on Networks of Workstations", *International Conference on Parallel and Distributed Computing Systems*, New Orleans, October 1997
- [21] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers", *IEEE Trans. Computers*, Vol. C-36, No. 12, December 1987, pp. 1425-1439.
- [22] R. Prouty, S. Otto and J. Walpole, "Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations" *Technical report number: CSE-94-012*, Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology March 21, 1994

- [23] PVM 3 User's Guide and Reference Manual, Oak Ridge National Laboratories, 1993. (Manual and source code available via anonymous ftp from *netlib2.cs.utk.edu* in directory */pvm3*).
- [24] B. S. Siegel, P. Steenkiste "Controlling application grain size on a network of workstations", Department of Electrical and Computing Engineering, Carnegie Mellon University, 1995
- [25] M. Snir, S. W. Otto, S. Huss-Lederman, D. Walker, and J. J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, Mass., 1996.
- [26] P. Steenkiste, "Network-Based Multicomputers: A Practical Supercomputer Architecture", *IEEE Transactions on Parallel and Distributed Systems* Vol. 7, No. 8, August 1996
- [27] "The CRAY T3E Series", the web page of Cray Research
<http://www.cray.com/products/systems/crayt3e/overview.html>
- [28] T. H. Tzen and L. M. Ni, "Dynamic Loop Scheduling for Shared-Memory Multiprocessors", *Proc. Int. Conf. Parallel Processing*, Vol. II, 1991, pp. 247-250.
- [29] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems" *Technical Report CSRI-257* April 1992. Computer Systems Research Institute University of Toronto

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved