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.

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.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

A Resource Monitor for Network-based Parallel Applications

By

Dingyi Jin, B.Eng. (Shanghai), M.Sc. (Michigan)

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Science in Information Systems Science

School of Computer Science
Carleton University
Ottawa, Ontario

February 1999

© Copyright
1999, Dingyi Jin, B.Eng. (Shanghai), M.Sc. (Michigan)

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

395 Wellington Street
Ottawa ON K1A 0N4
Canada

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

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.

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 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.

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.

0-612-43356-0

Canada

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

A Resource Monitor for Network-based Parallel Applications

submitted by

Dingyi Jin, B.Eng. (Shanghai), M.Sc. (Michigan)

Dr. Evangelos Kranakis
(Director, School of Computer Science)

Dr. Jörg-Rüdiger Sack
(Thesis Supervisor)

Dr. John Neilson
(Thesis Supervisor)

Carleton University

February 1999

Abstract

Using a cluster of networked workstations as a single parallel virtual computer has become popular both for scientific and general computation. However, in comparison with dedicated Massively Parallel Processors (MPP), the heterogeneity of the networked parallel systems introduces extra complexity for parallel computation. Effective load balancing has become key to the success of turning such a networked system into a parallel virtual machine.

In general, there are two methods to achieve effective load balancing in the heterogeneous environment of network-based parallel computing. The first method uses algorithmic approach while the second method uses resource information to effectively utilize the available system resources. While the former method has been extensively studied, few studies have been carried out for the latter one.

In this thesis, a resource monitoring tool is developed to study the load balancing method through resource information. The performance of the tool is evaluated through two parallel applications and the performance results are compared with those achieved through algorithmic approaches.

Based on the experiments, this thesis shows that dynamic load balancing using the resource monitoring tool is superior to the simple load sharing algorithms. Although the performance results of using the tool is comparable to those of adaptive load sharing algorithms, the tool provides simpler mechanisms for tuning performance parameters.

Acknowledgments

I would like to especially thank my supervisor, Dr. Jörg-Rüdiger Sack and Dr. John Neilson, for their research guidance and personal encouragement throughout this thesis.

I would also like to thank Dr. Doron Nussbaum for his direction and ideas in this research.

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

I would like to dedicate this thesis to my parents and my wife, LiuHong Wang, for their love and encouragement for all these years.

Contents

Abstract	iii
Acknowledgments	iv
1 Introduction	1
1.1 Research Area	1
1.2 Objective of this thesis	3
1.3 Survey on Related Work	4
1.3.1 Resource Management Tool	5
1.3.2 Message Passing Environment	12
1.3.3 Related Work on Performance Studies	15
1.4 Contribution of the thesis	19
1.5 Thesis Organization	20
2 Design of Resource Monitor Tool	21
2.1 Objectives of the Design	21
2.2 Implementation of the Tool	22
2.3 Collecting Resource Information	23
2.3.1 Percentage of CPU idle time and processor speed	25
2.3.2 Available Memory and Swap Space	26
2.3.3 Disk I/O Data	27
2.3.4 Communication Load	28
2.4 Communication Mechanism of the Tool	29
2.4.1 Spawning of Slave Processes	30

2.4.2	Data Communication between the Master and Slave Processes	32
2.4.3	Takeover by Shadow Master	34
2.5	Fault Tolerance	34
2.5.1	Failure of Slave Process	35
2.5.2	Failure of Master Process	36
2.5.3	Failure of Shadow Master	37
2.6	Dynamics of the Pool and Resource Configuration	38
2.7	Aging of Resource Information	38
2.8	Presentation of Resource Information to User Programs	39
2.9	Graphical User Interface	40
2.9.1	Overview of GUI	41
2.9.2	Configuration of Resource Information	42
2.9.3	Fault Tolerance	43
3	Tool Experiments	45
3.1	Experimental Environment	45
3.2	Testing of the Tool	46
3.2.1	Delay of load information	46
3.2.2	Overhead of the tool	47
3.3	Experimental Applications	47
3.3.1	Parallel Matrix Multiplication	48
3.3.2	Parallel External Memory Sorting Algorithm	53
3.3.3	Background Loads	59
4	Results and Discussions	61
4.1	Parallel Matrix Multiplication	61
4.1.1	General Performance Comparison	61
4.1.2	Automatic Partitioning of Parallel Matrix Multiplication	65
4.1.3	Comparing With Simple Load Sharing Algorithm	65
4.1.4	Comparing With Adaptive Load Sharing Algorithm	70
4.2	Parallel External Memory Sorting	72
4.2.1	General Performance Comparison	72

4.2.2	Comparing With Simple Load Sharing Algorithm	75
4.2.3	Comparison With Adaptive Load Sharing Algorithm	78
4.3	Summary	80
5	Conclusions	82
5.1	Summary of Results	82
5.2	Evaluation of Work	83
5.3	Further Research	83
5.3.1	Resource Monitor Tool	83
5.3.2	Experimental Applications	84
	References	85

List of Tables

3.1	Total Sorting Time With Different Job Granularity	59
4.1	Performance of Parallel Matrix Multiplication	62
4.2	Performance of Parallel Matrix Multiplication With Background Loads	64
4.3	Performance of Automatic Job Partitioning	66
4.4	Comparison with Simple Load Sharing Algorithm	68
4.5	Comparison with Simple Load Sharing Algorithm with Background Loads	68
4.6	Comparison with Adaptive Load Sharing Algorithm with Background Loads	71
4.7	Performance of Parallel External Memory Sorting	72
4.8	A Break-down of Total Sorting Time When Tool is Used	73
4.9	Performance of Parallel External Memory Sorting With Background Loads	74
4.10	Comparison with Simple Load Sharing Algorithm with Background Loads	76
4.11	Comparison with Adaptive Load Sharing Algorithm with Background Loads	79

List of Figures

1.1	Daemons in CODINE	7
2.1	Architecture of the Resource Management Tool	23
2.2	Relationship of master, slave and shell command processes	24
2.3	Remote Command Execution	31
2.4	Graphical User Interface of Resource Monitor Tool	41
3.1	Performance of External Memory Sorting with Different Memory Sizes	55
4.1	Performance of Parallel Matrix Multiplication Without Background Loads	62
4.2	Performance of Parallel Matrix Multiplication With Background Loads	64
4.3	Performance of Automatic Job Partitioning	67
4.4	Comparison with Simple Load Sharing Algorithm with Background Loads	69
4.5	Speed-Up Gained by Using the Resource Monitor Tool	75
4.6	Comparison with Simple Load Sharing Algorithm with Background Loads	77

Chapter 1

Introduction

1.1 Research Area

This thesis falls into area of network-based distributed parallel computing. A tool for collecting resource information on a cluster of interconnected computers is designed and implemented. The tool is then used on a cluster of workstations interconnected by a local area network (LAN). Two experimental parallel applications are studied to evaluate the effectiveness of the tool in improving the performance of these applications. The results are compared with those obtained by using load sharing algorithms without using such a tool.

Using a cluster of networked workstations to serve as a single virtual computer has become popular with the proliferation of powerful workstations and high-speed local area networks. The chief benefit of such a system versus that of a parallel computer is its cost-effectiveness and availability [7]. Massively Parallel Processors (MPP) are

costly and their availability is limited. On the other hand, the combined computational resources of the interconnected general-purpose workstations may exceed the power of a single high-performance computer and these workstations are in idle state 80% - 90% of the time on average [25, 29]. Therefore, utilizing a cluster of computers provides an alternative to parallel machines with large aggregate processing power and memory [31, 3].

Parallel computing on such a heterogeneous environment has been studied intensively over the past years and shows viability both for high-performance scientific computing as well as for more general purpose applications. The main differences of parallel computing between network-based and parallel systems are the performance issues incurred by the heterogeneity of the former [10, 38, 31]:

1. **Communication bottleneck:** Unlike a parallel system where the communication channel is dedicated, the network load on a network-based system is generated both by parallel applications and by more general jobs such as text editing. Furthermore, parallel systems use better hardware for interconnection while the network-based systems use high-overhead and low-bandwidth Local Area Networks (LANs).
2. **Heterogeneity:** The types of heterogeneity of a network-based system includes architecture, data format, machine load, network load and computational speed, whereas a parallel system uses homogeneous processors and the workload is

generated solely by the parallel applications. The heterogeneity of the network-based systems is also observed in the control of the machines. Thus the resource allocation, load balancing, fault-tolerance and recovery and security are more important in network-based systems.

To reduce the difficulties brought by such heterogeneity, resource management becomes important in network-based parallel systems [22, 4]. This thesis attempts to provide a better understanding of the effectiveness in improving the performance of network-based parallel applications with the knowledge of network resource information.

1.2 Objective of this thesis

There are two major objectives of this thesis:

1. **Tool development:** The first objective is to develop a tool for collecting resource information on a network of heterogeneous computers, a set of application program interfaces (API) that can be used by network-based parallel programs for dynamic load balancing and a graphical user interface (GUI) that facilitates the application programmers to gain insights into available resources of their network system.

2. **Performance Study:** The second objective is to compare the performance of experimental network-based parallel applications using the resource information for dynamic load balancing and those using conventional load sharing algorithms.

The experimental applications that are used in this study include parallel matrix multiplication and external-memory sorting programs. The reasons to select these applications include:

1. **Matrix multiplication:** Parallel matrix multiplication is used in [29, 20] and its performance characteristics are understood. Furthermore, the resource requirement of this program is simplified by its CPU-bound nature and the $O(n)$ computational complexity on each local host.
2. **External-memory sorting:** This application places more requirements on resources than the previous one. Its performance is directly influenced by the CPU load, available free memory and disk space of the local host.

1.3 Survey on Related Work

The feasibility and realization of network-based parallel computing is based on the premise that the cluster of independent computer systems, interconnected by networks, can be transformed into a coherent, powerful, and cost-effective concurrent computer by using software tools [38, 22, 10]. These software tools can be roughly

broken down into two categories: message passing environments (MPEs) and resource management (RM) tools. Two traditional paradigms of parallel processing are based on either shared memory or message passing. The current de facto network based parallel computing model is based on message passing [10]. Thus, the experimental parallel applications in this study are limited to MPEs.

Software for cluster resource management is needed to fully utilize the computing resources of a cluster of workstations. Although the MPEs, such as PVM, incorporate some primitives for resource management, RM covers a wider area for resource management and has followed an independent path for its development [33].

This section presents a brief review of the currently available resource management tools, an overview of the message passing systems represented by PVM and related work on performance studies of network-based parallel applications.

1.3.1 Resource Management Tool

The author has found around 30 cluster management tools in commercial and public domains. Twenty-three such tools along with their functionalities are reviewed in [4]. Their development was primarily driven by the need for a good Unix batch and queuing facility in a network environment of Unix machines [4, 22, 15, 9]. This section summarizes their main features.

Design Goals

The design goals of these systems are exemplified by the Network Queuing System (NQS), which is considered to be the progenitor of all the other cluster management tools [22]. The highlights of these goals include [23]:

1. Provide for full support for both batch and device requests.
2. Support resource quotas enforceable by the underlying Unix kernel implementation.
3. Support remote queuing and routing of batch and device request throughout the network of hosts running NQS.
4. Support networked output return, where the standard output can be returned to a remote host.
5. Support status operations across the network so that a user on one machine can obtain information relevant to NQS from another machine.

As indicated by the above goals, the existing resource management tools are mainly for the management of the networked system and batch jobs. Explicit support for parallel applications is neither included nor even defined.

Architecture

The architecture of most of these tools is based on the master/slave model. A representative architecture is exemplified by the Computing in Distributed Networked

Environments (CODINE), a commercial product of Genias Software [16].

CODINE consists of four types of daemons: master daemon, scheduler daemon, communication daemon and execution daemon. The relationships among these daemons are illustrated in Figure 1.1.

Figure 1.1: Daemons in CODINE

The CODINE master daemon runs on the main server and manages the entire cluster. It collects all necessary information for cluster resource management and maintains the CODINE database, which contains information about queues, running and pending jobs and the available resources, etc. The information in the database is periodically updated by the execution daemons. To prevent single point of failure of the master daemon, CODINE provides a shadow master daemon. Should the current

master daemon fails, a new master will resume the work. Communication between all other daemons and the master daemon is via a standard TCP port.

The scheduler daemon is responsible for mapping jobs to the most suitable queues. If a job can not be dispatched owing to a lack of resources, it is put into a waiting queue until the scheduler daemon decides the requirement for resources by the job can be met. The result of mapping is reported back to the master daemon to update the database and for the master daemon to notify the CODINE execution daemon on the corresponding nodes to start the job.

The execution daemon runs on each node in the cluster. It is responsible for starting and stopping the jobs as well as for reporting resource status of its resident workstation periodically to the master daemon.

Functionalities

The tools discussed differ in functionality in the following aspects [4]:

1. **Platform and Operating System Support:** Most of the tools support Unix family operating systems. Some also provide support for multiple processor machines.
2. **Application support:** Nearly all of the tools support batch jobs but some of them do not support parallel jobs. Support for configurable queues for job scheduling also differ in type and size.

3. **Scheduling and Allocation Policy:** Different algorithms and policies are employed for job dispatching, load balancing, job monitoring and job rescheduling. Some tools also support check pointing and process migration.
4. **Configurability:** The tools differ in the way the cluster resources are managed. The items managed include job run-time limits, processes, user job allocation and job statistics.
5. **Dynamics of Resources:** Items in this category include dynamic resource pool, run-time configurability and fault-tolerance.

Scheduling and Resource Allocation

The scheduling and resource allocation of most tools are built around the queue concept. The queue concept in CODINE provides a typical example [16]. Each CODINE queue on a machine represents a job class that can be run on the machine. It can be configured to include attributes such as upper load limits and priority levels. When a job enters the CODINE system, it is first placed in a pending queue by the CODINE master daemon. The scheduler daemon examines the jobs in the pending queue and decides if a job can be sent to a particular machine by matching the resources requested by the job with those available on the machine. The number of queues a machine can have is arbitrary and the queues can be added or modified on the fly. Thus, each queue on a machine is actually representative of a job class that can be run on the machine. Special queues can be defined for parallel or shared

memory computers.

The queue type can be one or several of the following: batch, interactive, checkpointing and parallel. The type determines what kind of jobs can be run in this queue. Other important queue attributes include: 1) the number of jobs per queue; 2) the allocated queue complexes and 3) queue limits.

The queue limits define the resource limits for all jobs in a particular queue and no job is allowed to exceed the limits. The queue limits in CODINE include:

- Time length of job execution
- Wall-Clock Time
- CPU Time
- File, and Data Size
- Stack, Core-file and Resident Set Size
- Queue Complexes.

The queue complexes provide all the relevant information regarding the resource requests a user may place on a CODINE job and how CODINE system interprets these resource request parameters. There are three different types of queue complexes:

1. Default Queue Complex: It contains all the predefined queue attributes such as queue name, host-name, priority and the queue limits.

2. Load Complex: The reported load values by the execution daemon are either the predefined CODINE load values such as the CPU load average, or the load values defined by the CODINE administrator. The host load complex contains all resource information of a particular machine while the global load complex contains cluster wide resources such as the available disk space on a network wide file system or the network traffic. Both the global and host complex can be expanded or modified by the CODINE administrator.
3. User Defined Complex: This is a named collection of attributes and their definitions about how the attributes are to be handled by CODINE.

Parallel Support

Scheduling for parallel applications is very different from scheduling for traditional Unix batch jobs. The interdependence of subtasks of a parallel application makes the synchronization of finishing time among subtasks very important. Ineffective synchronization among the subtasks can deteriorate the overall performance of a parallel application. A quick example is that certain subtasks wait for data from other subtasks which are not effectively scheduled and thus worsen the overall performance.

Although many existing tools claim to support parallel jobs, the exact contents on what and how parallel jobs are supported are not well specified. Typically, a tool performs a selection of an appropriate sub-cluster according to user specified queues.

Based on these information, the tool selects a matching set of queues from the currently available ones using the load level or the fixed ordered selection scheme. It then starts up daemons and other necessary tasks for the parallel execution. We are not aware of any studies of how effective these existing tools are in improving the performance of the network-based parallel applications in comparison with conventional load sharing algorithms.

1.3.2 Message Passing Environment

Beginning from mid 80's, researchers have approached the problem of exploiting the computing power of clusters of computers from the perspective of parallel processing. The development of Message Passing Environments (MPEs) has been the primary result of efforts guided by this approach [33]. Existing message passing environments (MPEs) include PVM (Parallel Virtual Machine) [14], Express [24], Linda[6], p4 [5] and the MPI (Message Passing Interface) standards [26].

The most widely used MPEs include the Parallel Virtual Machine [11, 28] and Message Passing Interface [17, 36]. While MPI provides no resource management functionality, there have been some efforts to provide resource management for the PVM system. This section provides an overview of the PVM software system. A comparison between PVM and MPI can be found in [12].

An Overview of PVM

PVM is a software package that allows a cluster of heterogeneous computers to appear as a single parallel computational resource. The networked computers may be parallel, serial or vector computers. PVM supports heterogeneity at the application, machine and network level [39]. At the application level, PVM allows application programs to explore the architecture that is best suited to their solutions. At the machine level, PVM handles all data conversion between two machines of different data formats. The application programs to be run on the different machines, however, need to be compiled on each of the different architectures. At the network level, PVM supports automatic start-up of tasks on the remote nodes of the virtual machine and allows tasks to communicate and synchronize with each other. The underlying mechanism used by PVM is BSD socket and the network daemons are running on each host node of the virtual machine [28].

PVM consists of two parts: a PVM daemon (pvmd) and a user library. The PVM daemon is a user process that runs on all the hosts within the virtual machine. The daemons are responsible for collecting host and configuration information of the entire virtual machine. The master daemon, which spawns all the other daemons, handles the reconfiguration of the entire system [39]. The user library provides a library of interface functions that user call for message passing, processes creation and task coordination, etc.

The main services provided by PVM include the following [29, 13]:

1. Point-to-point communication: PVM provides the communication mechanism based on BSD socket for application tasks. Data conversion is provided if communicating between two machines of different data format. Multi-cast is provided by repetitive sending of messages to the multi-cast pool.
2. Environment and configuration management: PVM provides for signaling, process control, dynamic process grouping and dynamic configuration. Tasks can be created or killed by other tasks and hosts can be added or deleted from the virtual machine pool.

Resource Management in PVM

Before PVM version 3.3, dynamic load balancing in PVM was done by a method called **Pool of Tasks** paradigm [28]. This is typically implemented in a master/slave model in which the master process decomposes the task and distributes the tasks to the slave processes till the task is completed. Load balancing in this scheme is achieved by keeping all the processors busy as long as subtasks remain in the pool. However, when PVM creates processes, it does so in a round-robin fashion in selecting hosts without regard to the load of the workstations in the virtual machine. Subtasks may be assigned to heavily loaded hosts and thus cause significant load imbalance. PVM did not provide dynamic information on load in the virtual machine.

In PVM version 3.3, a library function `pvm_reg_rm()` is provided for resource management [34, 13]. However, use of this function is nontrivial. Users have to link

it with an outside resource management tool in order to override the round-robin scheduling of PVM [13]. The resource management tool implemented in [34] only detects the PVM processes on the hosts. Since load may come outside from PVM tasks, its function as load balancing is limited.

1.3.3 Related Work on Performance Studies

It has been found that load balancing is the single most factor in influencing the performance in a multi-user environment of networked computers [35]. In our literature study, we find there are generally two approaches to solving the load balancing issue. The first category uses various load sharing algorithms while the other explicitly uses resource information.

Load Sharing Algorithms

Load sharing algorithms in parallel systems are extensively studied; however, their application to network-based parallel applications is quite recent [29, 31, 30]. In general, load sharing algorithms are based on the previously mentioned **Pool of Tasks** paradigm. A master process continues to dispatch subtasks to slave processes till all the task is completed.

Based on the size of the subtasks, the load sharing algorithms include those of fixed, variable and adaptive granularity [19, 32, 40]. In the first case, the size of the subtasks is fixed; while in the latter two, the size of the subtasks either decrease along the task processing as in the variable granularity algorithm or varies according to the

response time of the previous task as in the case of adaptive granularity algorithm. The reason that variable and adaptive granularity can improve performance are [29]:

1. The variable granularity algorithms attempt to minimize the waiting time for the master process to receive the last result. Since the master process has no knowledge about system resources, it can happen that the last task is sent to the slowest machine. If the task size is fixed, then the master process will have relatively long waiting period to receive the last result. When the task size is decreasing, however, the amount of waiting time can be reduced even if the last task is sent to the slowest machine.
2. The adaptive granularity algorithms attempt to vary subtask sizes according to the previous performance of a host. Thus, a slow machine will receive tasks of finer granularity while a fast machine will receive tasks of coarser granularity.
3. It is well known that communication is the bottle-neck for the network-based parallel programs. Thus, performance can be improved by reducing the number of messages with a small number of large messages. Variable and adaptive task granularity algorithms will create less messages than the fixed granularity algorithms in certain situations.

A detailed study on the performance of various load sharing algorithms for network-based parallel matrix multiplication can be found in [29, 31, 30].

Resource Management Approach

The other approach to load balancing is via the use of system resource information. Although extensive research had been done on the development of message passing environment (MPEs) such as PVM and MPI and the resource management (RM) systems, they have been mostly carried out independently of each other [33]. According to our survey, studies on the performance enhancement of network-based parallel applications by explicitly using the resource information are few.

In [20], a simple load balancing mechanism is provided for the PVM software system. The load balancer collects CPU load on each host of the virtual machine pool and allows PVM programs to use the information for dynamic load balancing. The result of the study shows improvement over the situation where load balancing is not provided in some cases.

In [34], a resource manager called Condor is interfaced to PVM to harness the idle cycles of workstations in the cluster. However, given the opportunistic nature of Condor in utilizing idle CPU cycles and its job migration design when the owner starts to use the workstation, it is not clear what the gain would be of using Condor in improving network-based parallel programs.

Discussions

Although using resource management for distributed parallel computing on a cluster of networked heterogeneous computers has been deemed important, there are few

studies comparing the performance of parallel applications explicitly using resource information for load balancing and those using conventional load sharing algorithms.

The study in [20] used parallel matrix multiplication to show the advantages of using resource information. However, the tool developed in that study only reports CPU load and processor speed of each machine in the cluster. Other resources such as free memory and disk space are not reported. Furthermore, the collection interval of the resource data is fixed in 30 seconds in the study. This long interval greatly limits the value of the information available to the parallel applications due to the fact that data may no longer be update-to-date. Indeed, experiments concerning matrix multiplication in the study distribute all the tasks to the machines in the pool in one batch and thus it is impossible to compare the performance results of the load sharing algorithms. In spite of these limitations, the study shows a simple and practical way of making the resource information available to network-based parallel applications.

Other studies such as [33, 41] deal with dynamic load balancing by either migrating whole process or data. Since the number of machines available for parallel programs is normally limited and it is expensive and difficult for job migration, We do not deal with this issue in this study.

One strongest argument against the resource management approach is that the load information on a host is random and therefore can not be used for discretionary load balancing. While this might be true in some situations, cases where programs use up resources for a persistent amount of time widely exist. One such example is the

practice of running long batch jobs in the evening. In addition, some resources such as the available local disk space is relatively static. Thus, studying the implication of using resource information for dynamic load balancing for cluster-based parallel applications is meaningful.

Instead of using an existing resource management tool, we decide to develop a prototype of our own for the following reasons:

1. Most of the existing RM tools are designed for managing Unix batch jobs. Although they have been proved to be effective for serial jobs, they are generally not considered suitable for parallel jobs [33].
2. The existing RM tools generally require super-user privilege for installation while we favor a user-level tool so that it can be easily installed and used.
3. Since there are few studies in this area, we need to have full power in tuning the tool to achieve best results. Developing our own prototype allows maximum flexibility in this regard.

1.4 Contribution of the thesis

According to our literature study, there are relatively few studies on the performance of network-based parallel programs by using the resource information of the system.

To the best of our knowledge, the main contribution of this study includes:

1. Development of a resource monitor tool that is particularly targeted to performance study on network-based parallel applications.
2. Comparative study between the performance of parallel matrix multiplication explicitly using resource information and that of using conventional load sharing algorithm.
3. Study on the performance improvement for parallel external-memory sorting program by explicitly using the resource information.
4. Study on the effectiveness of the tool in automatic partitioning two parallel programs executing in parallel on a cluster of networked computers.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter II gives a description of the design of the resource monitor tool. Chapter III presents the experimental environment and applications. Chapter IV discusses our experimental results. Chapter V gives conclusions and possible future research.

Chapter 2

Design of Resource Monitor Tool

This chapter presents the design and implementation details of the tool prototype that is used to collect resource information of each machine in a cluster of workstations. Section 2.1 presents the design objectives; Section 2.2 and subsequent sections give a detailed description of the implementation of the tool.

2.1 Objectives of the Design

Since the tool is mainly used to provide resource information to parallel applications, resource data collection and timely provision of these data to applications are two chief concerns for the design. To make the tool easily available to ordinary users, we also decided that the tool do not require superuser privilege for its operation and installation. The design objectives are summarized as follows:

1. A stand-alone system that is independent of any existing message passing environments (MPEs).

2. Provides timely resource information to network-based parallel programs without imposing significant additional load on the system.
3. Provides easy-to-use APIs that can be incorporated into network-based parallel programs based on any MPEs such as PVM or MPI.
4. Allow users to set parameters that are important to performance of the network-based parallel programs.
5. Provide user-friendly GUI for the system.

2.2 Implementation of the Tool

The tool follows the master/slave model for data collection. On each host in the cluster pool, a slave process is responsible for gathering resource information on the host and communicating this information to the master process, which is responsible for reporting the resource information of the whole system. Figure 2.1 shows the architecture of the tool.

The details of implementation are presented in the following sections. In particular, the topics include: 1) resource information collection on each host machine; 2) communication mechanism used in the tool; 3) fault tolerance; 4) dynamics of the pool and resource configuration; 5) aging of resource information; 6) presentation of resource information to user applications; and 7) graphical user interfaces (GUIs).

Figure 2.1: Architecture of the Resource Management Tool

2.3 Collecting Resource Information

In Unix operating system, the information necessary for determining machine load is contained in the file `/dev/kmem`, which is updated periodically by the system. However, access to this file requires super-user privilege. Also, the format of this file differs from system to system. Therefore, shell commands provided by Unix systems that access this file have to be used to access system load information.

Most of these shell commands can be launched and instructed to run continuously at specified intervals with a minimum of one second. Each of these shell commands thus constitutes one process. The variation of system resources over that interval is reflected in the output of these commands. Each of the shell commands can be launched within a child process space by a parent process, the slave process in the case of the tool. In case a slave process decides to terminate owing to exceptional conditions such as the detection of the crash of the tool system, it always kills all its

child processes running the shell commands first and then exits. Figure 2.2 shows the relationship of the master, slave and shell command processes. Note that there are also the slave and shell command processes on the machine where the master process resides.

Figure 2.2: Relationship of master, slave and shell command processes

The default types of information collected by the tool are specified below. The experimental applications in this thesis make use of CPU load and Disk I/O data. A user can always add new types of data to be collected and reported by the tool in case of need.

1. CPU load: Percentage of CPU idle time and processor speed.

2. Memory and swap space: Available free memory and swap space.
3. Disk I/O: Percentage of CPU time on disk i/o.
4. Local Disk: Percentage of time the local disk is busy and the available local disk space.
5. Communication load: Number of in- and out-bound TCP/IP packets.

2.3.1 Percentage of CPU idle time and processor speed

There are several ways to obtain CPU load information by commands such as **uptime**, **w**, **iostat** and **vmstat**. However, the CPU load information returned by **uptime** and **w** may not be accurate while the output format of **iostat** varies from system to system. To collect CPU load information, **vmstat** is used as in [20]. The main information contained in the output of **vmstat** command include:

1. Number of processes in run queue, blocked state for resources and runnable state but swapped.
2. Amount of swap space currently available and size of the free list.
3. Information about page faults and paging activity.
4. Information on disk operations.
5. Information on trap/interrupt rates.

6. Breakdown of percentage usage of CPU by user time, system time and idle time.

To extract the data of CPU idle time from a **vmstat** output, a sweep through the headers of **vmstat** output is done to find out the column number of **id** and then the CPU idle time is always obtained from this column in the subsequent **vmstat** output. Other data can be extracted from the output in the similar manner.

The **vmstat** command is launched by **fork** and **exec** calls. The output is redirected via a pipe to the parent slave process. An interval can be specified on the **vmstat** command line and CPU information will be collected continuously at that frequency.

2.3.2 Available Memory and Swap Space

These two pieces of data are also obtained from **vmstat**. As shown in the previous **vmstat** output format, the **swap** field gives the available swap space and **free** field the size of the free list in the memory. It is worth noting that the size of the free list reflects the existing number of free blocks in the memory. The actual size of free memory available to an active process might be larger as some dormant processes may hold onto blocks of memory which may be released and made available to the running process by the swapper.

When a user program tries to allocate a large array, the maximum size of that array is limited by the size of the heap of the process. This limit, along with others, can be obtained from **getrlimit** call declared in the **sys/resource.h** header file. For

different systems, the limits are different.

When a user program allocates an array of size smaller than the limit imposed by the process's heap but larger than the actual free memory available in the system, many page faults may occur and thus slow down the process. Thus, the tool reports both the current free memory space and the maximum of a process's heap size of a particular host.

2.3.3 Disk I/O Data

The **vmstat** command also reports disk statistics, limited up to four disks. The command **iostat**, on the other hand, provides more comprehensive information on each disk. Therefore, **iostat** command is used for disk statistics.

For each disk, **iostat** outputs the following information (in extended display):

1. name of the disk;
2. reads per second;
3. writes per second;
4. kilobytes read per second;
5. kilobytes written per second;
6. average number of transactions waiting for service (queue length);

7. average number of transactions actively being serviced (removed from the queue but not yet completed);
8. average service time;
9. percent of time there are transactions waiting for service (queue non-empty);
and
10. percent of time the disk is busy (transactions in progress).

In the default setup, the tool extracts the percentage of CPU time waiting for I/O and the percentage of busy time of a local disk. To obtain the available space of a particular disk, **statvfs** function call under **sys/statvfs.h** is used. In the default setup of the tool, the space of a local disk is reported.

2.3.4 Communication Load

In many cases, a local area network (LAN) is an Ethernet and contains segments that are connected by transparent bridges. The bridge is intelligent and only forwards non-local traffic from one segment to the other. Thus, communication load on different segments may differ.

To detect exactly the amount of traffic on a segment, network monitoring programs such as **netperf** and **tcpdump** [18, 42] have to be used. However, these programs require the network interface be in promiscuous mode and the processing of each packet on the LAN segment incurs significant CPU load. Since none of our

experimental parallel applications make use of the data on communication load, using of such programs is excluded.

The **netstat** command provides an alternative way to display the number of packets a machine has transmitted and received on each interface. Statistics of interest to the tool will be the TCP/IP interfaces. The **netstat** reports the number of incoming and out-coming packets of a certain host. The intention of reporting these data is to provide an indication on communication load.

Also, to compare the communication delay between two pairs of hosts, the round-trip return time (RRT) can be used. However, to obtain the RRTs between any two hosts, the cost in terms of the number of messages need to be sent will be $O(n^2)$, where n is the number of hosts considered. To reduce the cost for this statistic, an alternative way is to use CPU loads and the network statistics obtained by **netstat** on each machine to collect the inward and outward number of TCP/IP packets. How to use these data is up to the actual parallel applications.

2.4 Communication Mechanism of the Tool

The inter-process communication between the master process and the slave processes is based on BSD socket as it is available on a wide range of operating systems [37]. The tasks that require inter-process communication include the following:

1. Initiation of the tool system;

2. Data collecting by the master process; and
3. Fault-tolerance mechanism in case the master process fails.

The topics covered in the following subsections include: 1) spawning of slave process; 2) data communication between master and slave processes; and 3) takeover process by the shadow master in case the master process fails.

2.4.1 Spawning of Slave Processes

At initiation of the tool system, the master process reads a configuration file that contains the official name of the hosts to be included in the cluster pool and obtains two user-level UDP ports. Then an executable module of the slave daemon is dispatched to each of these hosts.

Under Unix, there are two system calls that provide for remote startup of a process: **rcmd()** and **rexec()**. Since **rcmd()** call requires superuser privilege, **rexec()** is used in the tool. The execution of **rexec()** call is illustrated in Figure 2.3, taken from [37].

There are two potential problems concerning the use of **rexec()** system call:

1. Scalability: As shown in Figure 2.3, creating each remote process requires at least one open file descriptor in the local process. As each Unix process is limited in the number of open file descriptors, the number of remote processes is limited by that number.
2. Security: The **rexec()** call requires the user login name and password on the

Figure 2.3: Remote Command Execution

remote host. Thus, the tool must have access to the array of login names and passwords in order to spawn slave processes on all the host machines in the pool.

To overcome the scalability issue, the master process closes the open file descriptor(s) after each spawning of the slave process. All subsequent data communication between the master and slaves is via a UDP port. For the login names and passwords, the user of the tool can either store the login name(s) and password(s) in the configuration files or input them at the start up of the tool.

The activities for starting up the tool system can be summarized as follows:

1. The master process reads a configuration file to obtain official host names in the cluster and the login name and password to these hosts.

2. The master process obtains two UDP ports, one for slave to send data and the other for user application to query the resource information.
3. The master process spawns a slave process using `rexec()` calls and passing the two user-level UDP ports as arguments to the call.
4. After the slave process is launched, it forks the processes to run shell commands, store the master UDP port for user query in a standard file on the local disk, obtain a user-level TCP port and acknowledges the master of a successful launching. In the acknowledgment packet, the slave informs the master of the user-level TCP port, process identification number, and the processor type and the process limits such as the maximum heap size. The TCP port is used for the master or shadow master to pass control information to the slave processes.
5. After the master process receives all the acknowledgments, it sends the host names along with TCP ports to the shadow master. It then informs all the slave processes to start sending resource data. The issue of failed spawning of slave process is dealt with in later subsections on fault tolerance.

2.4.2 Data Communication between the Master and Slave Processes

The master process collects resource data via a user-level UDP port. UDP is used to prevent the limits on scalability that TCP stream sockets incur [20, 37].

The master process uses **select** system call to monitor the data UDP port. A timeout is set for this call so that the master process won't be blocked forever. On receipt of a datagram from a slave process, the master process sends back an acknowledgment.

The slave processes collect resource data at user-specified intervals. As mentioned previously, this interval can be as low as 1 second. In our preliminary testing, if a slave collects and sends data at this minimum interval, the CPU load imposed by the master process on the resident host approaches 10%.

To reduce this load but to enable the master to be receive information on load variation promptly, a threshold scheme is introduced. In this scheme, a slave will send its load information to the master in fixed period of time or when the load change exceeds a user defined percentage value. The algorithm used by the slave is as follows:

procedure need-send($L[1...n]$, $P[1...n]$, i , v)

{ $L[1...n]$ is an array of accumulated percentage load changes beginning from last time the load was sent to the master; $P[1...n]$ is an array of load data collected in the previous interval; i is the load type and v is the load data of load type i . It returns TRUE if send load data to master is needed; FALSE otherwise.}

$$x = \frac{(v - P[i])}{P[i]}$$

$$L[i] = L[i] + x$$

```
if(absolute(L[i]) > user-defined-value)
    return TRUE;
else
    return FALSE;
```

Using the above scheme reduces the CPU overhead of the master process to less than 1.0% on the resident host. The detailed results on the overhead of the tool will be discussed in the next chapter.

2.4.3 Takeover by Shadow Master

The next section will deal with fault tolerance in case the master process fails. We only discuss the communication mechanism provided for the shadow master to contact the slave processes in the cluster here.

As mentioned above, the master process informs the shadow master process the official name and a user-level TCP port of each host in the cluster. When the shadow master detects the master process failure, it uses these TCP ports to communicate with the slave process.

2.5 Fault Tolerance

There are two main aspects concerning the fault tolerance of the tool system, i.e. the failure of the master process and any of the slave processes. The following subsections deal with these cases and the case of failed shadow master.

2.5.1 Failure of Slave Process

The failure of a slave process may occur either during initiation stage or during normal operation of the tool. A system level POSIX thread is created in the master to deal with slave process failure after the initiation stage. This thread wakes up periodically to check the status of each slave and restarts the slave if necessary.

When the master tries to spawn a slave process to a certain host, it sets a timer. If the timer expires and nothing is heard from the slave, the master process labels the slave process as non-operational. There is no way for the master process to know what is happening to the unsuccessful slave process nor can the master send any signals to that process. Thus, the master will simply ask the thread to keep listening to the unsuccessful slave while going on with normal data collection. If the slave process responds later, the daemon thread will add that process to the normal slave process pool for data collection. Otherwise, the user of the tool has to intervene to kill any processes related to that unsuccessful slave process and explicitly asking the tool to spawn a slave to the host again. The reason the master process does not spawn more than once in this case is to prevent creating too many failed processes on the host.

To detect the failure of a previously operating slave process, the master maintains a counter for each slave. If no datagram is received from a slave process for a regular user-specified data-send interval plus a safety margin, the counter is incremented. After a threshold value is reached, the master process assumes the slave process is dead and labeled it as down. The daemon thread in the master process is responsible

for restarting the dead slaves. The restarter thread first executes a **kill** command to make sure the failed slave process is indeed terminated. If the command is successful, a new slave process is created at the that host by **rexec()** call; otherwise, the restarter thread will keep trying to send **kill** command till it is successful.

2.5.2 Failure of Master Process

The failure of the master process presents a more serious problem as, in that case, the whole tool system will fail. There are two approaches that may be used to resolve this issue. The first approach is to treat the whole pool of hosts as a virtual ring and use a standard ring election algorithm to select a new master. The second approach is to use a shadow master process, which takes over when it decides the master process is dead.

Our tool uses the second approach as the election algorithm incurs additional overhead and we assume the crash of the two hosts at the same time is rare. Since the port used by the master process is at user-level, there must be a way for the shadow master to communicate with the other slave processes. This is realized via the control TCP port each slave processes obtain at initiation. Subsection 2.4 dealt with this issue.

The slave processes, including the shadow master, detect the failure of the master process by maintaining a counter. Whenever a slave process sends a data packet to the master, it increments the counter. Whenever an acknowledgment packet is

received from the master, the counter is reset to zero. If the value of the counter exceeds a user defined threshold value, the slave determines that the master process is dead. If a slave is not the shadow master, it will start a timer and block listening to the control TCP port. If the timer expires and nothing is heard from the TCP port, the slave process will kill all its child processes running the shell commands and then terminate itself.

If a slave process is the shadow master, it forks off a new process to run the shadow master code. The execution sequence by the shadow master is as follows:

1. Execute a remote kill command to make sure the master process is terminated.
2. Obtain two user-level UDP ports.
3. Advertises the UDP ports to all the slaves and wait for acknowledgment.
4. When all the slave replies, select a shadow master and starts to collect data.

2.5.3 Failure of Shadow Master

If the master detects the shadow master fails, it simply appoints a new slave process as the shadow master and passes the necessary information to that new candidate via the control TCP port.

2.6 Dynamics of the Pool and Resource Configuration

The tool provides for dynamic addition and deletion of hosts to the cluster pool. To delete a host, the master process simply sends a remote **kill** command. To add a host, the master asks the previously mentioned restarter thread to do the job.

The tool also provides for dynamic resource configuration. After the tool is initiated, a user can request the tool to report data on a particular field of a shell command output. For instance, a user can add to the report list the data under **us** in **vmstat** command. However, in the current implementation, launching of commands can only specified at tool initiation stage. New commands can also be launched to collect user-specified data.

2.7 Aging of Resource Information

The resource information collected by the master process at regular interval represents the information of the system over the past user-defined interval, which is defaulted at 15 seconds. To provide a historical view of the system, the regular data has to be accumulated and updated. A data window is defined for this purpose.

A data window is an aging scheme that tries to reflect the historical load situation of the cluster covered by the tool. It has a window size, which is the length of history in seconds the window covers. Thus a window of 5 minutes gives the average load

information over the past 5 minutes. An default aging factor for the new data for particular window is derived by dividing the data collection interval by the size of the window. For instance, if data is collected regularly at 15 seconds and the window size is 5 minutes. The aging factor is $15/300$. With each arrival of new data, the new window data is derived by: $(1 - \text{aging-factor}) * \text{current-window-value} + \text{aging-factor} * \text{newly-arrived-data}$. The user of the tool can set the aging factor for each window by himself.

At any particular time, a maximum of ten data windows can be defined. The first one is always the most recently arrived data. The user can change the definition of any of the remaining nine data windows at any time.

2.8 Presentation of Resource Information to User Programs

Since the tool system is stand-alone, a mechanism is needed for the master process of the tool to make the resource information available for user programs. In the initial design, a file is created on network file server (NFS) which is updated periodically by the master process and is accessed by APIs to present data to application programs. However, initial testing shows that NFS may become a bottleneck and access time by an application can take as long as one second. To avoid this bottleneck, the resource information resides in the memory of the machine resident to the master process. The master process advertises a TCP port for querying resource information

to each slave process, which in turn stores the host name of the master and the TCP port in a standard file in */tmp* directory. The user application programs can extract information from this standard file and contact the master process for resource information using the query TCP port. This arrangement allows the application programs to access resource data from any machines in the cluster.

A set of application program interfaces (APIs) are provided to get particular resource information. For instance, the **get_idle_host(char *resource_name, char **host_names, int *data_array, int number_of_hosts, int window_size)** function call allows the user to obtain the requested number of hosts with the lightest load as specified in *resource_name* and the actual values of this resource type on each machine. The detailed function synopsis of these APIs can be found in the user document of this resource monitor tool.

2.9 Graphical User Interface

A graphical user interface (GUI), written in *Tcl/Tk/Expectk*, is created for the resource monitor tool. The main purpose of the GUI is to allow users to query and configure the tool in an easier way. Section 2.9.1 presents an overview of the GUI; section 2.9.2 presents resource parameters that can be configured dynamically through the GUI and section 2.9.3 describes fault tolerance and a command line interface that is complementary to the GUI.

Hosts	Command	CPU	Memory	I/O
gale		186232	25804	98
chopin		61796	612	32
wagner		64828	1904	92
chaos		181457	1576	96
vivado		71712	2656	97

Figure 2.4: Graphical User Interface of Resource Monitor Tool

2.9.1 Overview of GUI

The GUI process communicates with the resource monitor tool through *exp_send* and *expect* commands from *Tcl/Tk/Expecttk*. The GUI consists of two windows. The main window consists of a data displayer and a command menu. The log window records critical system information. Figure 2.4 shows the main window of the GUI.

The machine names that are included in the current cluster appear under the **Hosts** column. The other columns contain the names of resources that are currently being reported and their respective values of each machine. Both machine and resource can be added or deleted dynamically from the system.

2.9.2 Configuration of Resource Information

The GUI provides an easy way for a user to dynamically configure the parameters of the tool through a set of commands. These commands and their descriptions are as follows:

1. **Adding or deleting a machine:** A user can dynamically add or delete a machine to or from existing cluster. When a new machine is added to the cluster, it will show up in the main window and starts to collect the same array of resource data as other machines already in the cluster. When a machine is deleted, its row will also be deleted from the main window. When all machines are deleted but none is added before the timeout of the master process, the master process will view the whole tool system as dead and will eventually terminate.
2. **Adding or deleting a command:** As described previously, the resource monitor tool depends on ever-running Unix performance utility commands to collect resource data. A user can add or delete a command to or from an existing set of commands. When a command is deleted, all the data that are collected under the command will be deleted from the GUI. After a command is added, a user can specify the data to be collected from the command.
3. **Adding or deleting data:** A user can specify data to be collected under a certain command. A new column under the data name will show up in the

GUI's main window to contain values for all existing machines. When a data item is deleted, the column under the data name will also be deleted from the main window.

4. **Adding or deleting data windows:** A user can collect accumulative resource data by specifying data windows. One data window is associated with a time parameter expressed in seconds. Thus, if a data window is 60 seconds, the user will be able to view accumulative resource data values over the last 60 seconds. A maximum of 10 such windows can be specified. A user can also delete a data window in order to make room for new ones.

In addition to the commands described above, the GUI also enables queries on the current configuration of the tool system. The information that can be queried upon include: 1) current set of commands; 2) machine name of the shadow master, if any; and 3) current set of resource data items that are being collected by the tool system.

2.9.3 Fault Tolerance

As described earlier, the single point of failure of the master process of the resource monitor tool is prevented by appointing a shadow master. When the master fails, the shadow master takes over but still attempts to display the GUI on the machine which the old master was operating on. The GUI displayed by the shadow master looks exactly as Figure 2.4 and a user can control the system by interacting with the

GUI.

However, in case the machine on which the old master resided on crashes, the attempt by the shadow master to display the GUI will fail. In this case, a user of the tool can communicate with the new master process via a shell program from any of the machines where the slave processes operate. Basically, the shell program communicates with the master process via the TCP port the master process uses to process data query. As described in Section 2.8, each slave process stores the TCP port and machine name of the master process in a file on local disk. Any process can thus sends query to the master process via this TCP port. The commands that are accepted by the shell program are the same as those accepted by the GUI.

Chapter 3

Tool Experiments

This chapter describes the experimental applications used to demonstrate the effectiveness of the tool in improving the performance of network-based parallel applications. Section 3.1 describes the experimental environment; Section 3.2 gives the testing result of the tool; and Section 3.3 presents the experimental parallel applications, i.e. parallel matrix multiplication and external-memory sorting.

3.1 Experimental Environment

All the experiments described in this chapter are done on a network of 8 Sparc workstations interconnected by a 10MHz Ethernet. Two of these workstations each has a faster processor, 60 MHZ, than the rest (25 MHZ). The operating systems run on these machines are Solaris 2.x. When running the experimental applications, the system is exclusively used by these applications. Background loads are simulated by programs particularly created for this purpose.

Unless otherwise specified, each experiment is repeated ten times and the errors of the experimental data are calculated based on a 90% confidence level. Also, the data collection interval by the tool is set at 1 second unless otherwise specified.

All the parallel programs use PVM as their message passing environment and follow the master/slave model. The master process is responsible for dispatching jobs and aggregating results. The machine on which the master resides also participates in computation. The slave module is dispatched to every machine in the cluster at the initiation of the application program.

3.2 Testing of the Tool

The testing of the tool concentrates on the delay between a data query and the receipt of the data and the overhead of the tool imposed on the system.

3.2.1 Delay of load information

No particular testing programs are devised to get the delay between a data query and receipt of the data by the application. Instead this piece of data is collected through all the experimental applications that use the tool for dynamic load balancing. The delay per query ranges from 0.052 to 0.084 seconds in 100 runs of parallel matrix multiplication. The median value is around 0.060 seconds.

3.2.2 Overhead of the tool

The overhead of the tool is gathered by running the tool for 36 hours. The CPU load imposed on the host running the master process is around 1.7% with data collection interval set at 1 second. The CPU load imposed by each slave process is around 0.8%. When the data collection interval is increased to 3 seconds, these values become 1.0% and 0.2%, respectively.

3.3 Experimental Applications

This section describes the experimental parallel applications used to evaluate the effectiveness of the tool in improving performance for network-based parallel applications. The objectives of the experiments are to investigate the effects of the tool in the following two areas:

1. Automatic parallel job partitioning: In a shared academic environment, there are cases when different people want to run parallel applications on a subset of available machines. The tool is tested for its usefulness in automatically partitioning these jobs on different machines, given that all the machines in the pool qualify for the applications.
2. Automatic selection of idle machines: When a subset of machines are needed for parallel applications, selection of idle hosts can greatly improve performance. The tool is evaluated for this purpose.

3. Comparative study with load sharing algorithms: When all the hosts in a cluster are used for computation, simple load sharing algorithm is considered effective in overcoming the heterogeneity of the system. Parallel matrix multiplication using simple load sharing and using resource information for dynamic load balancing are compared for their performance under situations with and without background loads.

Subsection 3.3.1 gives a description of the parallel matrix multiplication; subsection 3.3.2 describes the external-memory sorting; subsection 3.3.3 describes the artificial background load programs.

3.3.1 Parallel Matrix Multiplication

Two General Algorithms

Two parallel matrix multiplication algorithms are most frequently used [29, 1]. In the first algorithm, the first matrix is sent to all the hosts participating in the multiplication. Then columns of the second matrix are distributed. The participating hosts calculate the columns in the resulting matrix by using the columns received and then send the results back to a master process. The master process simply assembles the received columns in the appropriate places in the final matrix.

In the second algorithm, the two initial matrices are divided into sub-matrices and each host receives two sub-matrices for local calculation. A master process is responsible for assembling all the results. An example taken from [29] is used to

explain this algorithm. The sub-matrices divided out of two initial matrices A and B as well as the result matrix C are shown as follows:

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

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

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

There are tradeoffs between the two algorithms. The first one requires the first matrix be stored on each host and thus larger memory space is needed. The second algorithm requires less memory size on each host but incurs more communication overhead to transfer data. Also, the master process needs to do more processing.

Matrix Multiplication Algorithm Used in This Work

In this work, the first algorithm described in the previous subsection is used. In all the experiments, two 720 * 720 matrices are created in the master process. A slave process is spawned onto every machine in the cluster and the first matrix is sent one by one to each slave. As PVM's broadcast is implemented actually in sequential individual sending, this feature is not used in the program. The slave processes that have received the first matrix block waiting for columns from second matrix to do the multiplication.

After the master finishes sending the first matrix, it starts to send columns of the second matrix to slaves till all the columns are sent. The number of columns in each send is determined by the granularity of the jobs. In a typical experiment, the granularity is 12 columns of the second matrix. When a slave finishes one granularity of job, it sends the result back to the master process and requires one more granularity of job to work on.

To avoid the master being overwhelmed by incoming results sent by slaves at the same time, the following protocol is used for a slave to send results to the master process:

1. When a slave finishes a subtask, it sends a request to the master process for sending the result and blocks waiting for response from the master.
2. When the master receives a request from a slave process, it sends an acknowledgment to the slave to signal it to send the result and blocks waiting for result from the slave.
3. When a slave receives an acknowledgment from the master, it starts to send results to the master and then wait for next task.

In the above protocol, the queuing of slave requests is done by using PVM message buffering scheme.

Determination of Job Granularity

In our experiments using matrix multiplication, job granularity is the number of columns of the second matrix and is the unit of subtask in which the master process sends to slave processes for calculation. Smaller granularity enables more flexible adaptation to the load change of the whole cluster system but gives a lower computation to communication ratio. Larger granularity tends to give a better computation to communication ratio but provides less room for adaptation to system load variations.

The typical job granularity in our experiments is 12 columns of the second matrix. This gives around 15:1 computation to communication ratio for the slower machines and 8:1 for the faster one.

When the resource monitor tool is used, the master process determines the job granularity sent to a slave process according to the CPU load of the slave host. The calculation of CPU load is based on the percentage CPU idle time and the speed of the processor. In our experimental environment, the speed of slower machines is assigned 100 and that of the faster ones 400. The assignment is based on the fact that the faster machines can perform 4 times faster than the slower ones.

A basic granularity, normally 6, is set for a slower machine with 50% idle CPU time. The master process uses the following formula to determine the size of next job that will be sent to a slave host:

$$next_job_size = basic_granularity * \left(\frac{speed_factor * CPU_idle_time}{100 * 50\%} \right)$$

According to this formula, a machine with speed factor 400 and 100% CPU idle time should be assigned 6×8 number of columns.

Simple and Adaptive Load Sharing Algorithms

As discussed in Chapter 1, load sharing algorithms are developed to deal with the heterogeneity of network-based parallel applications. In our experiments, parallel matrix multiplication using the resource monitor for dynamic load balancing is compared with parallel matrix multiplication using simple and adaptive load sharing algorithms.

The simple load sharing algorithm, also called fixed granularity algorithm, dispatches a fixed size of job to a host which just finished its previous job. In this way, faster machines will receive more jobs than the slower ones and thus achieve the goal of dynamic load balancing. One drawback of this algorithm is owing to its fixed job size. If one of the jobs in that last batch of the unfinished jobs is sent to a particular slower machine, that machine will slow down the processing of the whole batch and thus will cause poorer performance.

The adaptive load sharing algorithm attempts to solve the drawback of simple load sharing algorithms by linking the size of a job sent to a host to the response time of the host in finishing the previous jobs. If a host finishes a previous job slower than a predefined time, the host will be sent a smaller job size in the next job dispatching; if it finishes faster than a predefined time, it will receive a larger job size. The algorithm defines a normal job size as well as minimum and maximum job sizes.

Although this algorithm proves to be effective over the simple load sharing algorithm in the multi-processor environment, it requires extensive performance testing in order to determine the different levels of response times.

In our experiments on adaptive load sharing algorithms, we define three levels of job sizes: minimum, normal and maximum. Unless otherwise specified, the minimum and maximum job sizes are one half and double of normal job size, respectively. We will discuss in more detail why three level of job sizes are used in the next chapter.

3.3.2 Parallel External Memory Sorting Algorithm

Description of the Algorithm

Various parallel sorting algorithms are described in [2, 21]. The external-memory algorithm used for our experiment is described as follows [2, 27]:

1. Assuming the maximum number of records that can reside in memory is `MAX_IN_MEMORY` read into memory `MAX_IN_MEMORY` number of records from the input file, sort the records by quick-sort and store the sorted records in a unique file.
2. Repeat Step one until all the data in the input file are sorted. The resulting files are called generation one files.
3. Select a predefined number of files, `MAX_FILE_NUM`, and read into memory the first `MAX_IN_MEMORY/MAX_FILE_NUM` records from each of the file. The block of records from each file forms a bucket and the buckets are organized into a heap using the first record in each bucket.

4. Pop out the first record of the root bucket and delete the record from the heap.

Repeat until a predefined number of records, `MAX_WRITE_OUT`, have been popped out. Write these records out to a unique file. Repeat till a bucket is empty. Then a new batch of records from the same file is read into the bucket. Repeat till all the records in the `MAX_FILE_NUM` files are processed.

5. Repeat step 2 and step 3 till only one single file results.

Resource Requirement of the Algorithm

The performance of the above sequential external memory sorting algorithm depends on the following resources:

1. CPU speed and load,
2. Disk speed and load, and
3. Free memory size.

The dependency on CPU and disk speed and loads is obvious since a faster and lightly loaded CPU makes sorting faster while a fast and lightly loaded disk speeds up the disk access time. The dependency of the algorithm on available memory size may not be obvious and experiments are carried out to gain insight.

Our experiments show that, other things being equal, the factor that mostly influences the performance of this sequential external memory sorting program is the number of files generated at the first generation, assuming the number of files

that can be opened by a process is fixed. This number in turns depends on the MAX_IN_MEMORY, i.e. the maximum array size that can be passed to quick-sort at a time. Of course, The opposite factor in passing larger array to quick-sort is the increased sorting time since quick-sort has an average computational complexity of $O(n \log n)$. Figure 3.1 shows the performance of this program with various memory sizes, with MAX_FILE_NUM at 20 and a total of 2 million randomly generated integers.

Figure 3.1: Performance of External Memory Sorting with Different Memory Sizes

From Figure 3.1, it can be seen that when the memory size is 100000 (in 4 bytes), there is a sharp gain in performance. This is because the number of files created in

the first generation is 20 and thus only one generation of merging is needed. After this point, increasing memory size does not have obvious performance gains. In cases where the size of integers to be sorted is far greater than the memory size, such obvious performance gains should occur at several points. Thus, larger memory size a process can allocate help improving the overall performance of the program.

Actual Resources Used in the Experiments

Based on the analysis in the previous subsection, resource data on CPU, disk and free memory size should be collected by the monitor tool to determine the job size to be dispatched to a particular node. However, it is observed that the *free* memory size obtained from the output of *vmstat* command does not reflect the actual free memory sizes that are currently available as idle processes holding memory pages can be swapped out and thus free up more memory space. Also, the use of virtual memory on Unix systems further complicates the picture. Therefore, all the experiments are done with memory size sets at 1/100th of total number integers to be sorted.

CPU speed and load are in the same as for matrix multiplication described in section 3.3.1. A local disk is used for external memory sorting and is monitored for its activity. The percentage time the disk is busy over the past second is collected and used to determine the load of the disk. This piece of data is collected from the extended output of *iostat* command.

Computation to Communication Ratio

It is found that the computation to communication ratio of the external memory sorting program is low with our experimental environment as the bandwidth of our LAN is only 10 MHz. Typically, the ratio for a slow machine is only 4:1. In order to increase this ratio, we put an empty loop running 200 times in the comparison function and thus this loop is run each time the comparison function is called. As a result, the computation to communication ratio is increased to around 25:1 for a slow machine and 15:1 to a fast machine. Since 100 MHz Ethernet is widely available now, we consider our enhancement of computation to communication ratio as a reasonable simulation of a faster network.

Determination of Granularity

It is found through experiments that the CPU and disk loads slow down a sequential external memory program in about same proportions. One possible reason for this is that the background loads consuming DISK I/O resources also consume CPU cycles. Thus, these background loads slow down the merging process in roughly same proportion as CPU consuming background loads in slowing down sorting process. If a program finishes at 100 second without any background loads, it finishes at about 200 seconds when either CPU or disk is 100% busy. When both CPU and disk are 100% busy, the program takes around 400 seconds to finish. Thus, the equation to determine the granularity of a job to dispatched to a particular node is as follows:

$$next_job_size = basic_granularity * \left(\frac{speed_factor * CPU_idle + speed_factor * Disk_idle}{100 * 50\%} \right)$$

The speed factors for CPU and the disk for slow machine are 100 and those for fast machines are 400. These speed factors are derived from experimental observations that fast machines tend to be 4 times faster under same experimental conditions.

For programs not using the resource monitor tool, the granularity is set at 200000 integers unless specified otherwise. A slow machine takes around 115 seconds to sort 200000 integers while a fast one takes around 30 seconds. The *basic_granularity*, in the above equation, is set at 50000 integers for programs using the resource monitor tool. According to the equation, a slow machine with no background will receive 200000 integers in one job dispatch while a fast one will receive 800000 integers; when heavily loaded, a slow machine will receive 50000 integers at minimum while a fast one will receive 200000 integers.

Table 3.1 gives a comparison of total sorting time when the granularity is set at 50000, 200000 and 500000 integers without using the tool. When job granularity is either too small or too large, the performance tends to deteriorate. An obvious reason for this is that when granularity is too small, the computation to communication ratio tends to be small and thus less parallelism can be achieved. When the granularity is too large, faster machines will be given less data to process and the long run-time for the slow machine(s) in processing the last batch of job can degrade the overall performance. Table 3.1 indicates that our choice of job granularity at 200000 integers

Hosts No.	50K	200K	500K
2	2687.69±17.20	2340.60±61.58	2517.30±139.49
3	1986.76±11.37	1693.20±36.85	1850.10±118.60
4	1486.33±46.19	1368.35±20.23	1481.24±118.35
5	1272.13±5.98	1151.31±14.49	1244.11±41.39
6	1048.97±12.90	1027.05±10.96	1065.86±11.19
7	900.12±10.96	856.57±29.86	914.82±9.87
8	806.18±16.43	780.86±3.45	863.88±2.17

Table 3.1: Total Sorting Time With Different Job Granularity

for our systems is reasonable.

3.3.3 Background Loads

In general, artificial background loads are produced by programs using a tight loop with doing anything significant. The program used to consume CPU cycles is described as follows:

procedurecpu-consumer($T[1...n]$, n)

{This procedure is used to consume CPU cycles to slow down the processing time of a ma

repeat

repeat

$T[i] = i$

until n

until forever

The program used to make the local disk, *tmp.* busy is to continuously write to and read from the disk. The pseudo-code is as follows:

proceduredisk-consumer(T[1...n], n, filename)

{This procedure is used to continuously writing to and reading T[1...n]
from a file called filename. }

repeat

 write T[1...n] to filename

 read T[1...n] from filename

until forever

It is worth noting that when each of the above programs is run alone, it takes about 40% CPU time or makes the local disk busy 70% time. However, to really slow down a processor or disk, the number of jobs waiting in the ready-to-run queue has a more significant effect. Thus, more than one of the above programs need to be launched on a machine in order to really slow down the machine.

Chapter 4

Results and Discussions

This chapter presents the results of the experiments described in the previous chapter. Section 4.1 describes the experimental results obtained for parallel matrix multiplication and Section 4.2 presents the experimental results for parallel external-memory sorting.

4.1 Parallel Matrix Multiplication

4.1.1 General Performance Comparison

Table 4.1 gives a comparison of performance of parallel matrix multiplication programs with or without using resource monitor the tool. The number of hosts used ranges from two to eight, out of a total of eight machines in the cluster and the granularity of subtasks is twelve columns of the second matrix. There are no background loads generated for this run and the error of the data is collected at 90% confidence interval.

Hosts No.	No Tool	Tool
2	515.32±11.10	322.94±15.11
3	362.79±3.61	257.59±3.23
4	286.41±1.86	224.86±2.43
5	241.46±1.19	199.04±4.41
6	209.67±3.01	179.08±2.21
7	190.61±1.31	166.29±1.11
8	156.98±1.08	153.42±5.45

Table 4.1: Performance of Parallel Matrix Multiplication

Figure 4.1: Performance of Parallel Matrix Multiplication Without Background Loads

It is obvious from Table 4.1 that the program using the tool for dynamic load balancing has a better performance than the other program without using the tool. Figure 4.1 shows the difference graphically. When only a subset of all available machines are used, the program using the tool for load balancing always include the fast machines in case they are idle. The program not using the tool adopts the default load balancing scheme of PVM, which is essentially round-robin scheduling. Faster machines are not necessarily included in this case even when they are idle.

However, when all the machines available are used for parallel computation, the difference in performance between programs using and not using the tool becomes very small. The simple load sharing algorithm is as efficient as the program using the tool in utilizing faster machines and thus makes its overall performance comparable to that of the other program. The difference in performance becomes smaller as more hosts are used for computation. Subsection 4.1.3 gives a more detailed study on the case when all the hosts in the pool are used.

When background loads are added to three slow machines with each load consuming around 70% CPU time and the same programs as above were run, the performance of both programs is worse off. However, the program using the tool always performs better than the program without using the tool. Table 4.2 and Figure 4.2 give the results in this case.

Hosts No.	No Tool	Tool
2	602.92±6.19	337.23±9.43
3	417.56±4.19	273.16±10.94
4	331.07±2.58	237.91±5.16
5	273.12±4.20	217.83±9.18
6	239.73±2.37	196.77±5.72
7	204.41±3.78	180.36±4.16
8	177.30±1.14	169.51±1.94

Table 4.2: Performance of Parallel Matrix Multiplication With Background Loads

Figure 4.2: Performance of Parallel Matrix Multiplication With Background Loads

4.1.2 Automatic Partitioning of Parallel Matrix Multiplication

As mentioned in the previous chapter, one of the possible ways to utilize the tool is for automatic job portioning of two parallel programs each requesting a subset of available hosts of the same cluster. In our experiment, two parallel matrix multiplication jobs are launched one after the other from two slow machines.

When less than half of the available machines are requested by each job, the performance gain by using the tool of dynamic load balancing is obvious, as indicated by the results summarized in Table 4.3. Even when five or six hosts out of eight in the pool are requested by each job, the performance gain by using the tool is still substantial. The performance gain by the tool is achieved by maximumly avoiding overlapping the jobs. Since the program not using the tool employs a round-robin scheduling, chances are large to overlap jobs on a subset of machines while other machines remain idle. As expected, however, this performance gain diminishes when each job requests nearly all of the available machines. Figure 4.3 gives a graphical performance comparison of the two situations.

4.1.3 Comparing With Simple Load Sharing Algorithm

The previous results have shown that using the tool for dynamic load balancing for parallel matrix multiplication has an obvious performance gains when a subset of available hosts are used. To compare the performance of using the tool and the simple

Hosts No.	No Tool	Tool
3	453.51±30.67	367.75±15.12
4	415.31±22.45	333.89±15.71
5	351.46±15.67	299.18±16.47
6	329.23±5.55	286.4±10.64
7	293.04±5.18	271.01±6.67
8	259.91±5.42	252.29±5.47

Table 4.3: Performance of Automatic Job Partitioning

load sharing algorithm when all available hosts are used, we carried out experiments with different granularities (in number of columns of the second matrix) and compared the performance of the two programs, in cases with and without artificial background loads.

Without Background Loads

When no background loads are imposed on any of the machines in the pool, the only difference between the slower and the faster machines is the CPU speed. In multiplying 12 columns of integers, a slower machine typically uses 17 seconds while a faster one around 4.5 seconds. In such circumstances, the performance of the program using simple load sharing algorithm and that of the program using the tool are comparable, as indicated in Table 4.4. When job granularity increases, the program using the tool tends to perform better and is relatively insensitive to the increase of granularity. However, the gain in using the tool is limited.

Figure 4.3: Performance of Automatic Job Partitioning

With Background Loads

The small performance difference when the tool is used or not, as presented in the previous subsection, might be owing to the small degree of heterogeneity of the machines in the pool. When the difference of processing speed is much larger, the tool should have the advantage of detecting the particularly slow machines and send smaller jobs to those machines. Thus, we introduced various numbers of CPU consuming background program, as discussed in Chapter 3, to the system to simulate larger heterogeneity in processing speeds.

In each of our experiments, we introduced the same number of background loads,

Granularity	No Tool	Tool
12	156.98 $\pm$ 1.08	153.42 $\pm$ 1.39
18	167.20 $\pm$ 1.56	157.06 $\pm$ 1.50
24	174.31 $\pm$ 1.56	156.67 $\pm$ 4.54
30	179.98 $\pm$ 1.28	162.51 $\pm$ 4.47

Table 4.4: Comparison with Simple Load Sharing Algorithm

No. of Loads	No Tool	Tool
0	161.64 $\pm$ 1.87	162.17 $\pm$ 1.28
1	178.18 $\pm$ 4.24	179.74 $\pm$ 5.70
2	209.20 $\pm$ 2.52	189.13 $\pm$ 4.20
3	234.76 $\pm$ 4.25	203.79 $\pm$ 2.43
4	241.53 $\pm$ 2.79	205.18 $\pm$ 7.56
5	270.00 $\pm$ 2.51	228.00 $\pm$ 2.88

Table 4.5: Comparison with Simple Load Sharing Algorithm with Background Loads

ranging from 1 to 5, on each of the four slower machines. The typical processing time of 12 columns of data is 30, 45, 65, 75 and 95 seconds for 1 to 5 background loads.

The results show that when more than three background loads are planted on the each of the four slow machines, using the tool starts to show obvious gains. When five background loads are planted, the performance gain in using the tool is approaching 20% faster than the case when the tool is not used. Table 4.5 and Figure 4.4 shows the results of the experiment.

There are two possible reasons for this performance improvement:

1. The program using the tool tends to send more jobs to the faster machines.

Figure 4.4: Comparison with Simple Load Sharing Algorithm with Background Loads

2. The program using the tool tends to send smaller jobs to the load ridden machines and thus the delay time caused by these machines in the last batch of jobs is shorter.

After analyzing the data, we find the performance gain is obtained due to the second reason. This gives rise to the question whether performance of adaptive load sharing algorithm should be comparable to that of using the tool. This is the topic of next subsection.

4.1.4 Comparing With Adaptive Load Sharing Algorithm

The previous subsection shows that when background loads are introduced, the program using the tool tends to have a better performance than the program using the simple load sharing algorithm. The reason for the difference is owing to the fixed granularity used in the simple load sharing program. Thus, the last jobs sent to the heavily loaded machines will substantially slow down the whole system. When tool is used, this effect is reduced as smaller size of jobs are sent to heavily loaded machines.

As mentioned in last chapter, adaptive load sharing algorithm is devised to vary next job sizes sent to a machine based on the response time of that machine in carrying out the previous job. Since the gain by using the tool is obtained from smaller job size sent to the heavily loaded machines instead of from sending more jobs to the faster machines, we only used three levels of job sizes in the adaptive load sharing algorithm in our experiment. Also, the granularity of the job sent to a normal slow machine with no background load is 12 columns of the second matrix. If the response time from a machine is slower than the normal response time, the granularity is reduced to 6 columns. This makes the adaptive load sharing algorithm behaves the same as the program using the tool in terms of the granularity of jobs sent to heavily loaded machines. When the response time from a machine is faster than normal, the adaptive load sharing algorithm doubles the granularity to 24 columns.

The response time parameters for the adaptive load sharing algorithm are set as follows: 1) If the response time is longer than 25 seconds, half the granularity to 6.

No. of Loads	No Tool	Tool
0	157.82±1.29	162.17±1.28
1	178.53±3.39	179.74±5.70
2	197.29±2.52	189.13±4.20
3	202.10±2.32	203.79±2.43
4	207.30±2.38	205.18±7.56
5	224.23±2.83	228.00±2.88

Table 4.6: Comparison with Adaptive Load Sharing Algorithm with Background Loads

2) If the response time is shorter than 15 seconds, increase the granularity to 24.

The performance of the adaptive load sharing algorithm turns out to be comparable to the performance of using the tool. The actual data is described in Table 4.6 and the performance data of using the tool is from Table 4.5. The results conform to our previous conclusion that the finishing time of last batch of jobs differentiate the performance of program using the tool from the program using the simple load sharing algorithm. When the size of jobs sent by the load sharing algorithm also varies with the loads, its performance is comparable to the situation when the tool is used. Of course, the tool provides a much easier way to adapt job granularity to load variation while extensive performance tests must be done to determine various response time before adaptive load sharing algorithms can be used.

Hosts No.	No Tool	Tool
2	2340.60±61.58	967.56±19.19
3	1693.20±36.85	885.17±21.45
4	1368.35±20.23	911.37±15.73
5	1151.31±14.49	868.53±34.21
6	1027.05±10.96	817.85±29.98
7	856.57±29.86	802.76±26.27
8	780.86±3.45	770.34±16.85

Table 4.7: Performance of Parallel External Memory Sorting

4.2 Parallel External Memory Sorting

4.2.1 General Performance Comparison

Table 4.7 gives a comparison of performance of parallel external memory sorting programs with or without using the resource monitor tool. The number of hosts used ranges from two to eight, out of a total of eight machines in the cluster. There are no background loads generated for this run and the error of the reported data is collected at 90% confidence interval.

It is obvious from Table 4.7 that the program using the resource monitor tool for dynamic load balancing in general has a much better performance than the program without using the tool. When only a subset of all available machines are used, the program using the tool for load balancing tends to include the fast machines in the cluster in case they are idle; while the program not using the tool adopts the default load balancing scheme of PVM, which is essentially round-robin scheduling. Fast machines are not necessarily included in this case even when they are idle. This

Hosts No.	Total	Merging	Sorting
2	967.56±19.19	189.43±5.61	778.13
3	885.17±21.45	192.67±2.26	692.50
4	911.37±15.73	268.85±5.70	642.52
5	868.53±34.21	273.08±6.75	595.45
6	817.85±29.98	274.67±5.36	543.18
7	802.76±26.27	273.89±5.62	528.87
8	770.34±16.85	279.19±4.11	491.15

Table 4.8: A Break-down of Total Sorting Time When Tool is Used

observation is consistent with that of parallel matrix multiplication as described in Section 4.1.

However, when all the machines available in the cluster are used for parallel external memory sorting, the difference in performance between the programs using and not using the tool becomes very small. The simple load sharing algorithm seems to be as efficient as the program using the tool for dynamic load balancing in utilizing fast machines in the pool. This observation is again consistent with that of parallel matrix multiplication as described in Section 4.1.

The parallel external memory sorting is different from parallel matrix multiplication in that there is a sequential merge stage that has relatively large impact on the resulting performance. In the data reported in Table 4.7, the merge time is around 290 seconds for the program that does not use the tool for dynamic load balancing no matter how many hosts are used. The reason for this is that the granularity of a job is fixed and therefore the number of resulting sorted files for the master process

Hosts No.	No Tool	Tool
2	4127.72 $\pm$ 170.38	1009.42 $\pm$ 17.89
3	2855.10 $\pm$ 86.97	913.07 $\pm$ 5.41
4	2223.99 $\pm$ 82.77	945.26 $\pm$ 16.92
5	1703.54 $\pm$ 116.97	922.80 $\pm$ 33.81
6	1420.96 $\pm$ 15.49	899.36 $\pm$ 28.33
7	1151.06 $\pm$ 19.21	866.36 $\pm$ 37.75
8	919.07 $\pm$ 9.29	895.73 $\pm$ 29.70

Table 4.9: Performance of Parallel External Memory Sorting With Background Loads

to merge is also fixed. For the program that uses the tool for dynamic load balancing, the number of resulting sorted files for the master process to merge varies. A breakdown of the data reported in Table 4.7 for this program is shown in Table 4.8. The merging time when 4 hosts are used increases from 192 seconds when 3 hosts are used to 268 seconds and this increase in merging time more than offset the gains from the parallel sorting part and the overall performance of the program is even worse off. When more than 4 hosts are used, the merging time becomes relatively stable and adding more machines does enhance the overall performance of the program. However, as more and more machines are added, the phenomena of a performance drop owing to rising merge cost is expected to be observed again at certain point.

When one background load is planted on each of four slow machines, the performance of both programs becomes worse. However, the impact of the background loads on the program using the resource monitor tool is much smaller than that on the other program that does not use the tool. Especially, when a subset of available

machines is used, the gain by using the resource monitor tool is substantial. Table 4.9 summarizes the performance data collected in this experiment. Figure 4.5 compares the speed up in total sorting time for experimental data summarized in Table 4.7 and 4.9. The Y-axis in the figure is the ratio of total sorting time when using the tool to the total sorting time when not using the tool.

Figure 4.5: Speed-Up Gained by Using the Resource Monitor Tool

4.2.2 Comparing With Simple Load Sharing Algorithm

The results presented in the previous section shows that using the resource monitor tool for dynamic load balancing for parallel external memory sorting has an obvious performance gains, especially when a subset of available hosts are used. However,

No. of Loads	No Tool	Tool
0	780.86 $\pm$ 3.45	770.34 $\pm$ 16.85
1	919.07 $\pm$ 9.29	895.73 $\pm$ 29.70
2	1114.25 $\pm$ 81.94	955.80 $\pm$ 32.92
3	1223.19 $\pm$ 84.99	968.92 $\pm$ 41.80
4	966.54 $\pm$ 2.79	998.19 $\pm$ 70.20
5	1216.69 $\pm$ 13.28	1070.31 $\pm$ 70.05

Table 4.10: Comparison with Simple Load Sharing Algorithm with Background Loads

when all the available hosts are used, the performance of the two cases is comparable. As shown in the experimental results in parallel matrix multiplication in Section 4.1.3, this comparable performance may be owing to the fact that the pool of machines used is not heterogeneous enough. Thus, more than one artificial background loads are planted in four slow machines out of total eight machines. The result of the experiments is shown in Table 4.10 and Figure 4.6.

Figure 4.6 shows that when some machines are much slower than the rest, as when 2 or more background loads are running on each of four slow machines, the performance gain by using the resource monitor tool is around 20% better than the performance of simple load sharing algorithm. As described in Section 4.1.3, there are two possible reasons for the performance gain: 1) The program using the tool tends to send more jobs to faster machines and thus make better use of resources and 2) The program using the tool tends to send smaller jobs to the load-ridden machines and thus the delay time caused by these machines in the last batch of jobs is shorter.

After examining the trace of experimental data, we find the second reason plays a

Figure 4.6: Comparison with Simple Load Sharing Algorithm with Background Loads

more significant role than the first. This observation is further proven by the fact that when four backgrounds are planted on each of four slow machines, there is a sharp reduction in total sorting time when tool is not used. By examining the experimental data, we found that, in this case, the heavily-loaded machines are so slow in sorting the first batch of job sent to them that their finishing time, around 640 seconds, approaches the finishing time for the rest of the machines to sort the rest of the data, around 630 seconds. When the rest of jobs are done by the other machines, they do not have to wait for the heavily-loaded machines to finish, as in the case of three backgrounds. Thus, the overall performance when four background loads are present

improves. However, it is expected that when more and more background loads are present, the finishing time for the heavily-loaded machines to finish their jobs again determines the overall performance.

Indeed, when one more background load is added to each of the slow machines, the additional total sorting time is now determined by the additional total sorting time of the slowest machine. In our experiment, the slowest machine now takes around 890 seconds to sort the first batch job and the total sorting time thus increased by around 250 seconds.

4.2.3 Comparison With Adaptive Load Sharing Algorithm

The previous subsection shows that parallel external memory sorting using the resource monitor tool for dynamic load balancing tends to perform better when some machines in an available pool are much slower than the other. It also reveals that the performance gain is mainly due to the fact that the tool can guide the application program to variate job granularity in accordance with resources available on a particular host.

As mentioned in Section 4.1.4, adaptive load sharing algorithm is devised to vary next job size sent to a machine based on the response time of that machine in carrying out the previous job. In our experiment with adaptive parallel external memory sorting, we used eight levels of job sizes starting from 50000 to 400000 integers. If a slow machine does not have any background load, the job granularity to this machine

No. of Loads	No Tool	Tool
0	790.87±5.45	770.34±16.85
1	903.40±9.29	895.73±29.70
2	965.59±35.94	955.80±32.92
3	955.78±40.99	968.92±41.80
4	1006.45±35.79	998.19±70.20
5	1056.43±38.28	1070.31±70.05

Table 4.11: Comparison with Adaptive Load Sharing Algorithm with Background Loads

will increase to 200000 integers; for a fast machine without background load, the job granularity is increased to 400000 integers. In between 50000 and 400000, job sizes increase or decrease in 50000 unit, depending on the response time for a machine in processing the previous job. A typical slow machine sorts 50000 integers in less than 30 seconds while a fast machine takes around 8 seconds, when no active background loads are present. The results of this adaptive parallel external memory sorting algorithm are summarized in Table 4.11 along with the performance results of using the tool from Table 4.10.

The performance of the adaptive load sharing algorithm turns out to be comparable to the performance of using the resource monitor tool. The results are consistent with our observation that the finishing time of last batch of jobs differentiate the performance of programs using the tool from the program using the simple load sharing algorithm. They are also consistent with the results observed for matrix multiplication in Section 4.1.4.

4.3 Summary

In this chapter, we presented performance results of two parallel applications when the resource monitor tool and the load sharing algorithms are used. The results follow similar patterns for both parallel matrix multiplication and external memory sorting. The following is a brief summary of these results:

- When a subset of all available machines is used for parallel computing, the parallel programs using the tool for load balancing perform better than the programs using simple load sharing algorithms.
- When all the available machines are used, the performance of the programs using the tool is comparable to that of the program using simple load sharing algorithms when the background loads are absent or light. However, when the background loads are heavy and thus make the cluster of workstations more heterogeneous, the programs using the tool perform better than the programs using the simple load sharing algorithms.
- The major reason that simple load sharing algorithm performs worse than using the tool is that the job size in the former is fixed. Thus, the last jobs that are sent to slower machines slow down the overall finishing time of the entire program.
- When all the available machines are used, the performance of the parallel programs using the tool is comparable to that of the program using adaptive load

sharing algorithms. However, the tool provides a more straight forward way for load balancing while performance metrics have to be collected for using the adaptive load sharing algorithms.

For the parallel matrix multiplication, we also showed that, when two parallel matrix multiplication programs, each using a subset of all available machines, execute concurrently on the same cluster of workstations, the resource monitor tool can be used effectively in partitioning the two programs to reduce overlap of two or more jobs on one machine while leaving other machines idle.

Chapter 5

Conclusions

5.1 Summary of Results

The resource monitor tool, consisting of over 4000 lines of C++ code, developed in this thesis has been shown to be an effective and low-overhead user-level tool in collecting resource information of a network of workstations. Processing of data query by the tool is also effective in that the longest delay observed in our experiments is less than 0.1 second.

The resource monitor tool has also been shown to be effective in improving network-based parallel applications. Our experimental results shows that using the tool for dynamic load balancing is superior to using simple load sharing algorithms when fluctuation of background loads is relatively predictable. Comparing with adaptive load sharing algorithms, our results show that using the tool yields comparable performance results.

Another advantage of using the tool is that it provides a simple mechanism for

tuning performance parameters such as granularity of jobs for network-based parallel applications. Instead of embedding these parameters in the application programs which have to be recompiled at each change, the tool provides simple user application interfaces (APIs) which facilitate these testings.

5.2 Evaluation of Work

This study accomplishes its goal to develop a resource monitor tool that can be used not only to monitor resource information of a network of workstations but also to improve performance of network-based parallel applications. Through experimental results of two network-based parallel applications, this study also provides results to compare dynamic load balancing methods for network-based parallel applications via resource management and algorithmic approaches.

5.3 Further Research

This research can be further extended in the following directions:

5.3.1 Resource Monitor Tool

- The resource monitor tool may collect resource usage information on per process basis and thus avoids the delay in reporting available resources that are to be released by processes that have already terminated. This will provide more prompt and accurate resource information.

- The resource monitor tool takes over parallel job scheduling instead of leaving this to the application programs. This will spare the application programs from scheduling jobs.
- The resource monitor tool can collect traffic load on different segment of a LAN and provide the information to application programs, especially those that rely heavily on inter-process communication over two or more machines.

5.3.2 Experimental Applications

- Find and study parallel applications that can use different algorithms depending on the available resources of a system. Study the usefulness of using the resource monitor tool in adaptively choosing the most appropriate algorithm(s).
- Study a parallel application that is intensive in inter-process communication and the effectiveness of using the tool in choosing an optimal configuration of the processes with respect to communication cost.
- Study the effectiveness of the tool in improving performance for network-based parallel applications in the presence of different patterns of changing background loads.

Bibliography

- [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, Nov. 1994.
- [2] S. G. Akl, Parallel Sorting Algorithm, Academic Press, New York, 1985.
- [3] T. Anderson, D. Culler, D. Patterson and the NOW team, A Case for NOW (Networks of Workstations), *IEEE Micro*, 15(2), Feb. 1995.
- [4] M. Baker, Cluster Computing Review, NPAC Technical Report SCCS-748, Nov. 1995. <http://www.npac.syr.edu/techreports>.
- [5] R. Butler and E. Lusk, User's guide to the p4 programming system, Technical Report ANL - 92/17, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, Oc. 1992.

- [6] N. Carriero and D. Gelernter, Data parallelism and Linda. Technical Report 915, Department of Science. Yale University, Aug. 1992.
<http://www.cs.yale.edu/HTML/YALE/CS/Linda>.
- [7] A. L. Cheung and A. P. Reeves, High Performance Computing on a Cluster of Workstations. *Proceedings of the First International Symposium on High Performance Distributed Computing*, Syracuse, NY, Sept. 1992.
- [8] T. H. Cormen, Charles E. Leiserson and Ronald L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, 1993.
- [9] D. W. Duke, T. P. Green and J. L. Pasko, Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queuing System Version 3.0, Supercomputer Computations Research Institute, Florida State University, Tallahassee, Florida. <http://www.scri.fsu.edu/pasko/dqs>.
- [10] G. A. Geist, Cluster Computing: Wave of the Future? *Springer Verlag Lecture Notes in Computer Science*, 1994. <http://www.epm.ornl.gov/geist>.
- [11] G. A. Geist, J. A. Kohl, R. J. Manchek and P. M. Papadopoulos, New Features of PVM 3.4 and Beyond, http://www.epm.ornl.gov/pvm/pvm_home.html.
- [12] G. A. Geist, J. A. Kohl and R. M. Papadopoulos, PVM and MPI: a Comparison of Features, http://www.epm.ornl.gov/pvm/pvm_home.html.

- [13] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck and V. Sunderam, PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing, MIT, 1994.
- [14] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT press, 1994.
- [15] Generic NQS - Free Batch Processing For Unix, Academic Computing Services, The University of Sheffield, UK. <http://www.shef.ac.uk/uni/projects>.
- [16] Genias Software GmbH. <http://www.genias.de>.
- [17] W. Gropp, E. Lusk and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, Mass., 1994.
- [18] Hewlett-Packard, <http://www.cup.hp.com/netperf/NetperfPage.html>
- [19] S. F. Hummel, E. Schonberg and L. E. Flynn, Factoring: A Method for Scheduling Parallel Loops, *Comm. ACM*, Vol. 35, No. 8, pp. 90 - 101, Aug. 1992.
- [20] C. W. Humphres, A Load Balancing Extension For the PVM Software System, Master Thesis, Department of Electrical Engineering, The University of Alabama, Tuscaloosa, Alabama, USA, 1995.
- [21] J. JaJa, An Introduction to Parallel Algorithms, Addison-Wesley Publishing Company, New York, 1992.

- [22] J. A. Kaplan and M. L. Nelson, A comparison of queueing, cluster and distributed computing systems, NASA Langley Research Center, Technical Report, June, 1994.
- [23] B. A. Kingsbury, The Network Queuing System, Sterling Software, 1121 San Antonio Road, Palo, Alto 94808, USA. <http://shift.cern.ch/pub/NQS>.
- [24] A. Kolawa, The Express programming environment, *Workshop on Heterogeneous Network-based Concurrent Computing*, Tallahassee, FL, Oct. 1991.
- [25] P. Kruegar and R. Chawla, The Stealth distributed scheduler, *Proc 11th International Conf. Distributed Computing Systems*, Arlington, 1991.
- [26] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Application*, 8 (3/4), pp 165-416, 1994.
- [27] D. Nussbaum, Devron Technologies Ltd., 1995.
- [28] PVM 3 User's Guide and Reference Manual, Oak Ridge National Laboratories, <http://www.epm.ornl.gov/pvm>.
- [29] A. Piotrowski, Performance of Parallel Programs on a Network of Workstations, MCS Thesis, School of Computer Science, Carleton University, Ottawa, 1997.
- [30] A. Piotrowski and S. P. Dandamudi, Performance of a Parallel Application on a Network of Workstations, *Int. Symp. High Performance Computing Systems*, Winnipeg, July 1997. (This paper is available from <http://www.scs.carleton.ca>.)

- [31] A. Piotrowski and S. P. Dandamudi, A Comparative Study of Load Sharing on Networks of Workstations, Technical Report TR-97-14, School of Computer Science, Carleton University, Ottawa, 1997.
- [32] C. Polychronopoulos and D. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers, Dec. 1987.
- [33] J. Pruyne and M. Livny, Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generation of Computer System*, Vol. 12, 1996.
- [34] J. Pruyne and M. Livny, Providing Resource Management Services to Parallel Applications, *Proceedings of the Second Workshop on Environment and Tools for Parallel Scientific Computing*, 1994. <http://www.cs.wisc.edu/condor>.
- [35] B. Schmidt, V. Sunderam, Empirical Analysis of Overheads in Cluster Environments, *Concurrency: Practice and Experience* 6 (1), pp1 - 32, Feb. 1994.
- [36] M. Snir, W. Otto, S. Huss-Lederman, D. Walker, and J. J. Dongarra, MPI: The Complete Reference, MIT Press, Cambridge, Mass., 1996.
- [37] W. R. Stevens, Unix Network Programming, Prentice-Hall, 1992.
- [38] V. S. Sunderam, Heterogeneous network computing: The next generation. *Parallel Computing*, 23, pp.121 - 135, 1997.

- [39] V. S. Sunderman, G. A. Geist, J. Dongara, and R. Manchek, The PVM Concurrent Computing System: Evolution, Experiences, and Trends, *Parallel Computing* 20, pp531 - 546, 1994.
- [40] T. H. Tzen and L. M. Ni, Dynamic Loop Scheduling for Shared-Memory Multiprocessors, *IEEE Trans. Parallel Dist. System*, Vol. 4, No. 1, pp. 87 - 98, Jan. 1993.
- [41] S. Zhou, J. Wang, X. Zheng and P. Delisle, Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems, Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, 1992.
- [42] Tcpdump Program, <ftp://ftp.ee.lbl.gov>