A scheduling framework for dynamically resizable parallel applications

Gautam Swaminathan

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications

Dr. Calvin J. Ribbens
Dr. Dennis G. Kafura
Dr. Srinidhi Varadarajan

July 18, 2004
Blacksburg, Virginia

Keywords: Dynamic resizing, parallel applications, MPI-2,
scheduling framework

Copyright 2004, Gautam Swaminathan

A scheduling framework for dynamically resizable parallel applications

by
Gautam Swaminathan
Committee Chairman: Dr. Calvin J. Ribbens
Computer Science and Applications

(ABSTRACT)

Applications in science and engineering require large parallel systems in order to solve computational problems within a reasonable timeframe. These applications can benefit from dynamic resizing during the course of their execution. Dynamic resizing enables fine-grained control over resource allocation to jobs and results in better system throughput and job turn around time. We have implemented a framework that enabled dynamic resizing of MPI applications. Our framework uses the recently released MPI-2 standard that enables dynamic resizing. The work described in this thesis is part of a larger effort to design and implement a system for supporting and leveraging dynamically resizable parallel applications. We provide a scheduling framework, an API for dynamic resizing and libraries to efficiently redistribute data to new processor topologies.

Acknowledgements

I would like to thank the people that helped me through this work in many ways. Without them this research would not have been possible.

First of all, I would like to acknowledge my advisor, Dr. Ribbens, for his support, guidance, and knowledge. His experience in the field was the guiding light behind identifying and formalizing this thesis work out of the vast array of possibilities in this field. His encouragement during difficult phases of the project is greatly appreciated.

I would like to thank Malarvizhi Chinnusamy, my research associate, without whose support and contribution this project would not have been possible. I appreciate her support and dedication, and am truly grateful to her.

Sandeep Prabhakar, Prachi Bora, Satish Tadepalli, and Jeevak Kasarkod; students who had worked with Dr. Ribbens before, who were very easy to approach and gave useful advise and guidance.

Sumithra Bhakthavatsalam, for being the wonderful person that she is.

And last but by no means the least, my parents, who have made great sacrifices throughout their life so that I could have the opportunity to succeed.

TABLE OF CONTENTS

1. **INTRODUCTION** 1

2. **LITERATURE REVIEW** 4

2.1 Communication Software 4
2.1.1 Parallel Virtual Machine (PVM) 4
2.1.2 Message Passing Interface (MPI) 5

2.2 Mathematical Libraries 6
2.2.1 Linear Algebra Package (LAPACK) 6
2.2.2 Scalable LAPACK (SCALAPACK) 6

2.3 Dynamic Resizing Frameworks 7
2.3.1 Piranha 8
2.3.2 Adaptive Multiblock PARTI (AMP) 9
2.3.3 Adaptive MPI (AMPI) 9
2.3.4 Dynamic Resource Management System (DRMS) 10

3. **DESIGN** 13

3.1 Motivation and Approach 13

3.2 System Architecture 14

3.3 Resize Library and API 16
3.3.1 Contacting scheduler at a remap point 16
3.3.2 Expansion 17
3.3.3 Shrinking 17

3.4 Scheduler Components 17
3.4.1 Job Startup 17
3.4.2 Job Monitor 18
3.4.3 Performance Data Gatherer 18
3.4.4 Remap Scheduler 18

4. **EXPERIMENTAL RESULTS** 22

4.1 Experimental Setup 22

4.2 Adaptive Sweet Spot detection 22
4.2.1 Matrix Multiplication 23
4.2.2 LU Factorization 24
4.2.3 Improvement in iteration time 25
4.2.4 Redistribution overhead 26

4.3 Processor allocation decisions for job mixes 27
4.3.1 Job Mix 1 29
29
4.3.2 Job Mix 2 30

4.3.3 System throughput and job turnaround time 31

5. **CONCLUSIONS AND FUTURE WORK** **33**

5.1 Summary **33**

5.2 Future Work **33**

LIST OF FIGURES

Figure 1, section 2.4.1: Tuple Spaces 8
Figure 2, section 2.4.4: DRMS Architecture 11
Figure 3, section 3.2: Architecture diagram 14
Figure 4, section 4.2.1: Sweet spots for matrix multiply on Ethernet 23
Figure 5, section 4.2.2: Sweet spots for LU Factorization on Ethernet 24
Figure 6, section 4.2.2: Sweet spots for LU Factorization on Myrinet 24
Figure 7, section 4.2.3: Matrix Multiplication, Redistribution overhead for N=9600 26
Figure 8, section 4.2.3: LU factorization, Redistribution overhead for N=9600 26
Figure 9, section 4.3.1: Job Mix 1 29
Figure 10, section 4.3.2: Job Mix 2 30

LIST OF TABLES

Table 1, section 4.2.3: Matrix Multiply sample application for N=9600 on Ethernet 25
Table 2, section 4.2.3: LU Factorization sample application for N=9600 on Ethernet 25
Table 3, section 4.3.3: Static processor allocation – job mix 1 31
Table 4, section 4.3.3: Static processor allocation – job mix 2 32

Chapter 1

Introduction

Applications in science and engineering require large parallel systems in order to solve computational problems within a reasonable timeframe. These large parallel systems are common at universities and research institutions and are frequently shared by multiple users. The economic viability and scalability of interconnecting workstations by a high speed network has led to the emergence of clusters of workstations. We expect the use of these clusters to continue to grow in order to support the computational needs of the scientific and engineering communities. However, these systems are still expensive, such that a single user who requires running his engineering simulation on 500 nodes for a period of 10 days, cannot afford such a system. Thus, these systems are generally shared among multiple users, and efficient resource management becomes a key issue.

The problem of efficient resource management in such a system becomes even more difficult when the job arrival rates and workload on the system are varying and unpredictable. Conventional schedulers are static in nature, i.e, once a job is allocated a set of resources, that job continues to use those resources till the end of execution. The system does not have the ability to grant more resources to running jobs when there are idle processors available. When a high priority job arrives in the system, this generally leads to the suspension of lower priority running jobs.

A dynamic resource manager that has the ability to modify resources allocated to jobs during run time would result in more efficient resource management. It has been shown in the literature that dynamic resource management results in better job and system performance [2, 3, 4, 5, 6, 7, 8, 9, 10]. Dynamic resource management enables more fine-grained control over resource usage of jobs, which conventional schedulers do not provide. In a system that employs dynamic resource management, resources allocated to

a job can change due to internal changes in the job's resource requirements or external changes in the systems overall resource availability. Dynamic resource management extends flexibility by enabling applications to expand to a greater set of resources to utilize unused processors. Running applications can also shrink to a smaller subset of resources in order to accommodate higher priority jobs. The system can change the resources allocated to a job in order to meet a QoS deadline. Such a system, which enables resizing of applications, can benefit both the administrators and the users. By efficiently utilizing the resources, jobs can be completed at a faster rate thus increasing system throughput. At the same time, by enabling applications to utilize resources beyond their initial allocation, job turnaround time can be improved. The focus of this research is on reconfiguring parallel applications to use a different number of processes, i.e., on "dynamic resizing" of applications.

Additional infrastructure is required in order to enable resizing. Firstly, a programming model that allows applications to utilize the ability to resize is required. This programming model needs to be simple enough to implement such that existing code can be ported to the new system. Secondly, run time mechanisms to enable resizing are required. This includes the underlying system to provide the facilities required to allow applications to grow and shrink to a varying number of processors and enable the redistribution of the application's global state to this new set of processors. Thirdly, a scheduling framework that exploits the ability to resize to increase system throughput and reduce job turn around time is essential. Such a scheduling framework should support intelligent decisions in making processor allocation and reallocation in order to utilize the system effectively by growing jobs to utilize idle processors, shrinking jobs to enable higher priority jobs to be scheduled, changing resource allocations to meet QoS deadlines, etc.

The work described in this thesis is part of a larger effort to design and implement a system for supporting and leveraging dynamically resizable parallel applications. The overall architecture of this system and our programming model is described in Chapter 3. Algorithms and a library for process and data re-mapping are described in detail in [1].

The focus in this thesis is on the third aspect mentioned above, namely a scheduling framework. The goal is to take advantage of dynamic resizing to improve both system throughput and job turnaround time. Our approach is to allow the application and the scheduler to work together to make resizing decisions. The application supplies good choices for number of processors and processor topologies; the scheduler keeps track of performance data of this application and other applications running on the system. We have extended an existing parallel scheduler [11] to interact with the application to gather performance data, use this data to make decisions about processor allocation, and adjust processor allocations to maximize system utilization.

The remainder of this thesis is organized as follows. Chapter 2 describes background information and a literature review of related work in this field. Chapter 3 describes the architecture of the scheduling framework in detail. Chapter 4 describes experiments that we conducted to evaluate our system. We conclude in Chapter 5 with future work that can be done to improve our system.

Chapter 2

Literature Review

2.1 Communication Software

A cluster is a collection of machines with their private memory and processor connected by an interconnection network. Nodes communicate by passing messages across this interconnection network. The complexity of message passing is abstracted into a message passing software layer to enable programmers of parallel applications to conveniently employ the underlying system. PVM [12] and MPI [13] are two popular message passing libraries that provide a software layer to enable message passing.

2.1.1 Parallel Virtual Machine (PVM)

Historically, PVM was the first message passing library designed to work on a network of workstations (NOW). However, the popularity of PVM among developers of parallel applications has reduced over the years with the advent of MPI. Although PVM is functionally equivalent to MPI, there are a few philosophical issues that have made MPI more popular.

Control of PVM has always been primarily with the authors. A standards committee, on the other hand, governs MPI, and implementers of MPI are required to strictly follow the MPI standards guidelines. This has resulted in MPI being more portable to various hardware platforms. Moreover, MPI provides a better interface to parallel application developers by employing constructs such as communicators which PVM does not provide. Communicators are a convenient way of partitioning processors involved in a computation into subgroups and to reference these subgroups in order to perform a specific task. However, the PVM legacy continues to exist and PVM implementations are still used around the world.

2.1.2 Message Passing Interface (MPI)

MPI is a message passing standard that is designed by a broad committee of vendors, implementers and users. Many free and vendor supplied implementations of the MPI standard are available today. The most popular free implementations of MPI are MPICH from the Argonne National Laboratory and LAM MPI from Indiana University.

Recently, the MPI-2 standard [13] has been released, which among other features, includes support for dynamic process management. This functionality has been incorporated in the latest release of LAM MPI [33]. MPICH [34], however, has not yet implemented this feature of the MPI-2 standard.

The dynamic process creation and management functionality was added to the MPI-2 standard upon the request from the user community. This feature was desired because of the desire to exploit the dynamic process model discussed in Chapter 1, and was given impetus by the PVM research effort [12] that had demonstrated the benefits of dynamic resource management.

The MPI-2 standard enables process spawning through the API function call MPI_COMM_SPAWN. The MPI-2 standard, however, does not place restrictions or provide guidelines for how the actual spawning takes place, due to the fact that MPI needs to be portable across many different systems. Thus tasks such as process startup, reserving and scheduling resources, and returning information about available resources, are left to third party vendors to build on top of MPI.

In this thesis, we have built a scheduling framework on top of the MPI implementation provided by LAM in order to provide the resource management functionalities required to enable the effective use of the dynamic process model of the MPI-2 standard.

2.2 Mathematical Libraries

A large majority of scientific and engineering applications that run on clusters involve number crunching numerical methods. Numerical libraries have been written that provide a unified API to application programmers and can be optimized for the specific hardware on which these applications are run. Basic Linear Algebra Subprograms (BLAS) are the building blocks of numerical software [14]. Level 1 BLAS provide vector-vector operations, Level 2 BLAS provide matrix-vector operations and Level 3 BLAS provide matrix-matrix operations. Various vendor supplied BLAS exist today that are tuned to run optimally on many hardware configurations. Numerical libraries such as LAPACK [15] and SCALAPACK [16] employ the BLAS to provide methods for commonly used numerical algorithms.

2.2.1 Linear Algebra Package (LAPACK)

LAPACK was designed to run efficiently on modern high-performance processors by taking into consideration the multi-layered memory hierarchies of the machines. Earlier numerical packages such as EISPACK and LINPACK disregarded the memory-hierarchy and thus were prone to inefficiencies by spending too much time in the transfer of data rather than in doing useful computation. LAPACK uses the BLAS to take advantage of optimizations for the underlying hardware. Further, it predominantly uses Level 3 BLAS, i.e., blocked matrix-matrix operations in the inner most loops of the functions to most effectively utilize the memory hierarchy.

2.2.2 Scalable LAPACK (SCALAPACK)

In order to run LAPACK effectively on a distributed memory parallel clusters, a layer was introduced above LAPACK, that employed explicit message passing to incorporate off processor memory into the memory hierarchy. SCALAPACK has been written in Single-Program-Multiple-Data style (SPMD), and is amenable to be used in parallel SPMD programs.

The test applications in this thesis employ SCALAPACK functions to perform matrix-matrix multiplication and LU factorization: two functions that are commonly used and are ideal model applications for a large subset of the scientific code.

2.3 Dynamic Resizing Frameworks

Several mechanisms have been employed to enable dynamic resource management. What follows is a general overview of the work done in this field. The relevant work is dealt with in more detail in Sections 2.4.1, 2.4.2, 2.4.3 and 2.4.4.

In the Master/Worker model [17,18], a global entity distributes the tasks and the data associated with these tasks to worker programs that return the result of their computation to be collected by the global entity. Thread migration is another concept that is used to enable dynamic resource management by enabling threads to migrate from heavily loaded machines to lightly loaded machines [19, 20, 21, 22, 23]. The fork/join model is similar to thread migration and the master/worker model and is used extensively on shared memory multiprocessors [9, 24, 25, 26, 27, 28]. In this model, a few kernel level threads are scheduled for execution on the underlying processors. These kernel threads act as virtual processors for user-level threads that are assigned to them from a shared task queue.

High Performance Fortran [29] is an extension to Fortran that allows data redistribution and virtual processor grid annotation in conventional Fortran code, thus potentially enabling such programs to run on varying processor mappings in a system that supports HPF. HPF provides no constructs for specifying the number of processors that the application can resize to, thus leaving it to the underlying system to provide mechanisms to determine processor allocations. Adaptive Multiblock PARTI [30] is one such library that leverages HPF for adaptive resizing.

The Dynamic Resource Management System (DRMS) [2] enables resizing by extending the SPMD model. In the Single Program Multiple Data (SPMD) model, a single program

executes on multiple machines, and parallelism is achieved by restricting program flow based on processor id's and communicating data and messages between processors by means of an underlying message library. DRMS achieves resizing by linking a library that exposes its API to the SPMD program. The SPMD program uses this API to communicate with the DRMS system at strategic points in the program where resizing is possible. These strategic points are known as schedulable and observable points (SOPs). The SPMD program continues to run at its allocated resource level till it encounters its next SOP. It is at this SOP that decisions on resizing are made based on the internal change in requirements of the application or an external change in resource availability. Section 2.4.4 explains the DRMS architecture in more detail.

2.3.1 Piranha

Piranha is a system that is build upon the Linda [31] worker model. The tuple space in Linda performs the role of the global state pool. This global state pool enables producers and consumers of data to interact in an uncoupled fashion via the tuple space, thus enabling anonymity.

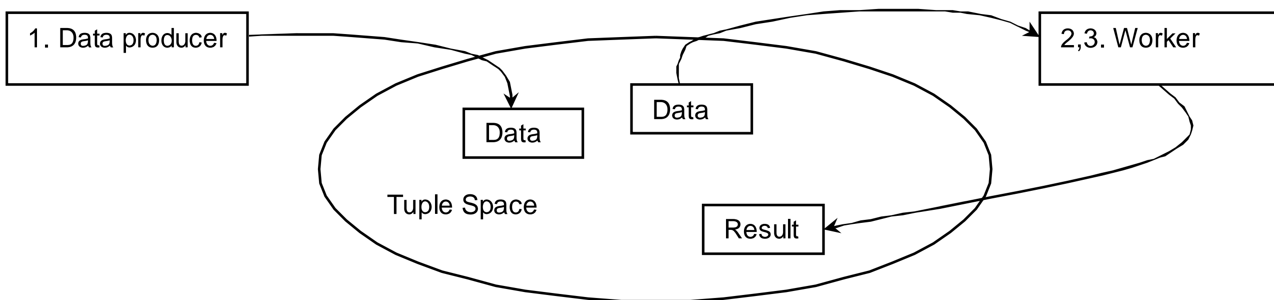


Figure 1: Tuple Spaces

In the above figure, a data producer stores the data on which computation has to be performed in the global tuple space. As processors become available, worker processes on those processors read the data from the global tuple space, perform the computation, and store the result back in the global space.

This model is highly effective for resizable applications as it allows free processors to use the tuple space as the single point of contact for more work. Piranha supports resizable applications by harnessing idle cycles in a workstation environment. Workstations join the computations when they become idle, and withdraw when their owners need them. While Piranha is effective for many applications such as DNA sequencing, it is not very effective for data parallel programs such as LU factorization. As stated in section 2.2.2, our primary focus is the resizing of such data centric applications as they account for a vast subset of current high performance scientific code.

2.3.2 Adaptive Multiblock PARTI (AMP)

The Adaptive Multiblock PARTI (AMP) system enables resizing by spawning a job to the maximum number of processors it can run on. It then enables applications to run on a subset of these machines by redistributing the data to an active partition, and requiring all processes outside of this partition to run as skeleton processes. Resizing is achieved by changing the size of the active partition and redistributing data to this new partition.

AMP puts a hard bound on the maximum amount of parallelism that can be employed by an application. Moreover, AMP's skeletons must be time shared with tasks of other applications on the same machine, which is not always supported on clusters that restrict machines to single jobs.

2.3.3 Adaptive MPI (AMPI)

Adaptive MPI is an implementation and extension of MPI that supports processor virtualization. Processor virtualization is achieved by running parallel applications as user level threads and migrating these user level threads from processor to processor. Processor virtualization is enabled by Charm++ [19], which is the underlying framework that supports multithreading and automated load balancing. In order to use AMPI, the user breaks down a parallel application into a number of fine-grained chunks. The number of chunks is independent of the number of available processors and is usually much larger. When the user program is run under the AMPI framework, each of these

chunks are treated as a virtual MPI process (VP) and are implemented as user level threads. AMP and Charm++ support the migration of these user level threads from processor to processor.

The load balancing system of Charm++ automatically instruments the application to gather performance statistics of the VP's. Periodically, or on demand, the system remaps the chunks to processors in order to achieve better load balancing and to reduce the communication overhead. User level thread migration is also used to expand or shrink the number of processors allocated to a job.

Due to the adoption of user level threads, AMPI is very effective for applications that have fine grained parallelism. However, data centric scientific applications employ a far coarser granularity of parallelism, which is not amenable to be split up into a large number of user level threads.

2.3.4 Dynamic Resource Management System (DRMS)

As mentioned earlier, DRMS enables resizing by extending the single-program-multiple-data (SPMD) model. Schedulable and observable points (SOPs) are placed at points in the application code where the applications resource requirements change when it moves from one stage of the computation to another, or when the application can benefit from a resizing operation.

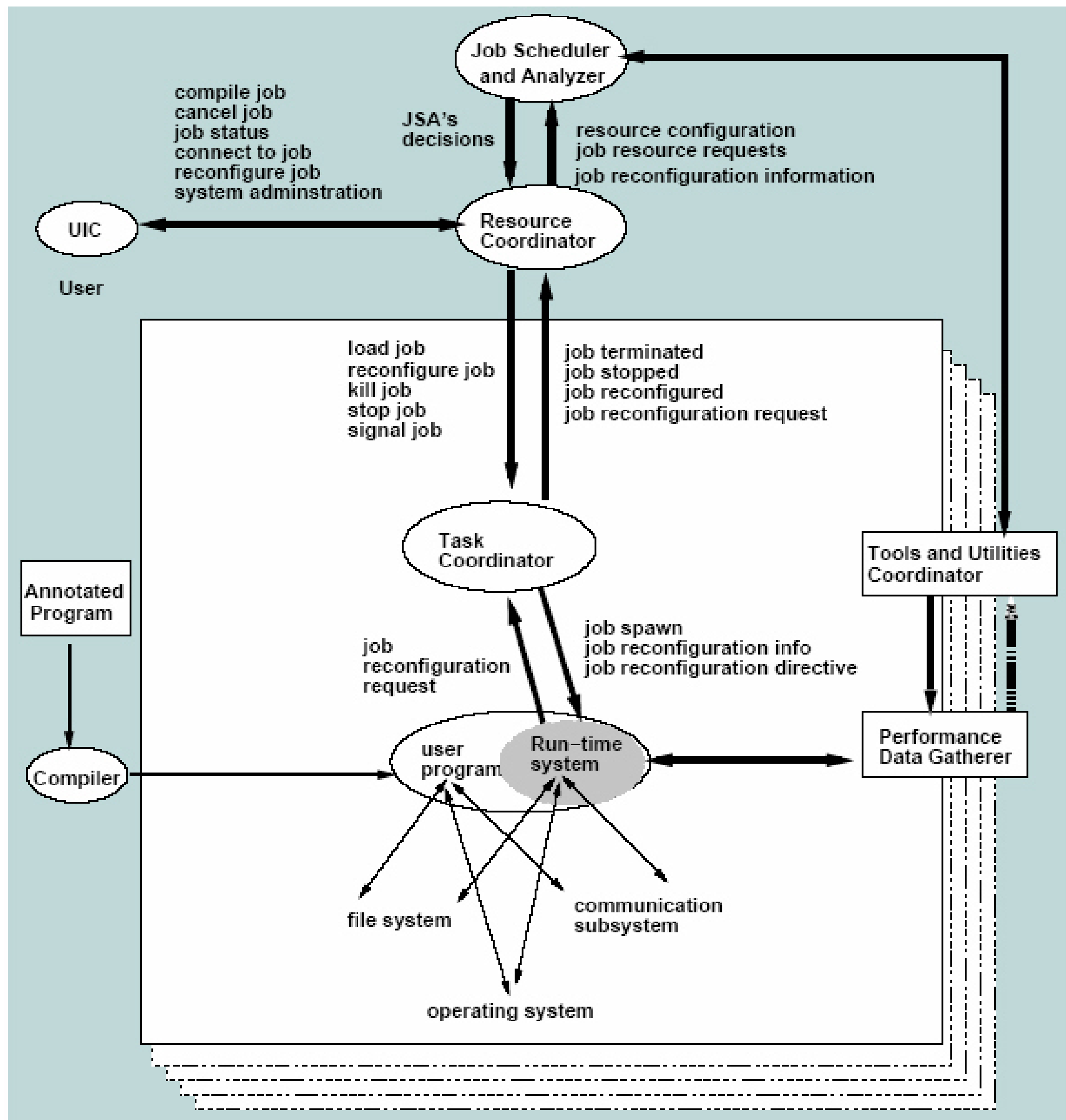


Figure 2: DRMS Architecture

When an application reaches an SOP, it contacts the Task coordinator (TC). It provides the TC a list of possible sizes that the application is willing to grow to. An application developer can provide this list through annotations in the source code. These annotations are converted to DRMS API calls by a precompiler. Every application running under DRMS has a TC associated with it. The applications TC in turn contacts the Resource Coordinator (RC). The RC, in coordination with the Job Scheduler and Analyzer (JSA) decides on a resizing decision and informs the application of the decision. The

application is then terminated and restarted on the new set of processors and its previous state is restored. The application's global state is then redistributed to the new set of processors after which computation can resume.

The current implementation of DRMS is application and user centric. The scheduler makes scheduling decisions based on the applications requirements and processor availability. The scheduler does not make decisions based on the performance of the application. There is a performance data gatherer (PDG), but it does not play a role in the scheduling decision made by the JSA. In our approach the scheduler and the application work together to make resizing decisions. The application provides good choices for the number of processors and processor topologies, and the scheduler takes into account the performance characteristics of the application and other applications in the system while making scheduling decisions.

Under DRMS, remapping requires killing and restarting a job on the new set of processors. This involves a significant overhead in comparison to our system where resizing is achieved without the need to kill and restart the application.

Moreover, DRMS is designed to work on the IBM RS6000 SP. It not been ported to Unix based clusters which are popularly employed today for high performance computing, and does not take advantage of the new MPI-2 dynamic process management functionality.

Chapter 3

Design

3.1 Motivation and Approach

Current scientific codes that run on parallel machines use the MPI message passing layer for communication. The MPI-1 standard assumes a static processor model, i.e. an application cannot dynamically change the number of processors it is running on during the course of its execution. Thus, these applications do not take advantage of a dynamic processor model. The release of the new MPI-2 standard enables the dynamic processor model. The work done in this thesis is part of a wider effort to leverage the ability to dynamically allocate resources to applications during their execution. One of the interesting benefits that our framework provides is the ability to automatically find 'sweet spots' for parallel applications, i.e., to find the number of processors that a particular code and problem instance runs well on.

In order to provide dynamic resizing of applications, work was done in three broad areas. Firstly, a scheduling framework was created. This scheduling framework gathers application performance data and enables the making of scheduling decisions based on this performance data, the current resources allocated to the jobs in the system, and jobs waiting to be scheduled. The work done in this thesis concentrates on this aspect of dynamic resizing.

Secondly, in collaboration with Chinnusamy [1], a programming model for resizable applications was created. This includes the API that applications use to communicate with the scheduler to make resizing decisions and to gather performance data.

Thirdly, mechanisms for mapping processor topologies and redistributing data from the old processor grid to the new processor grid were implemented [1]. In order to enable

ease of portability of existing applications to our framework a library that performs these operations efficiently is included in the framework. This library was implemented on the top of SCALAPACK [16]. BLACS [32], the communication layer of SCALAPACK was modified to support dynamic process management. However, the algorithms are not specific to SCALAPACK and can be easily implemented on the top of other frameworks.

3.2 System Architecture

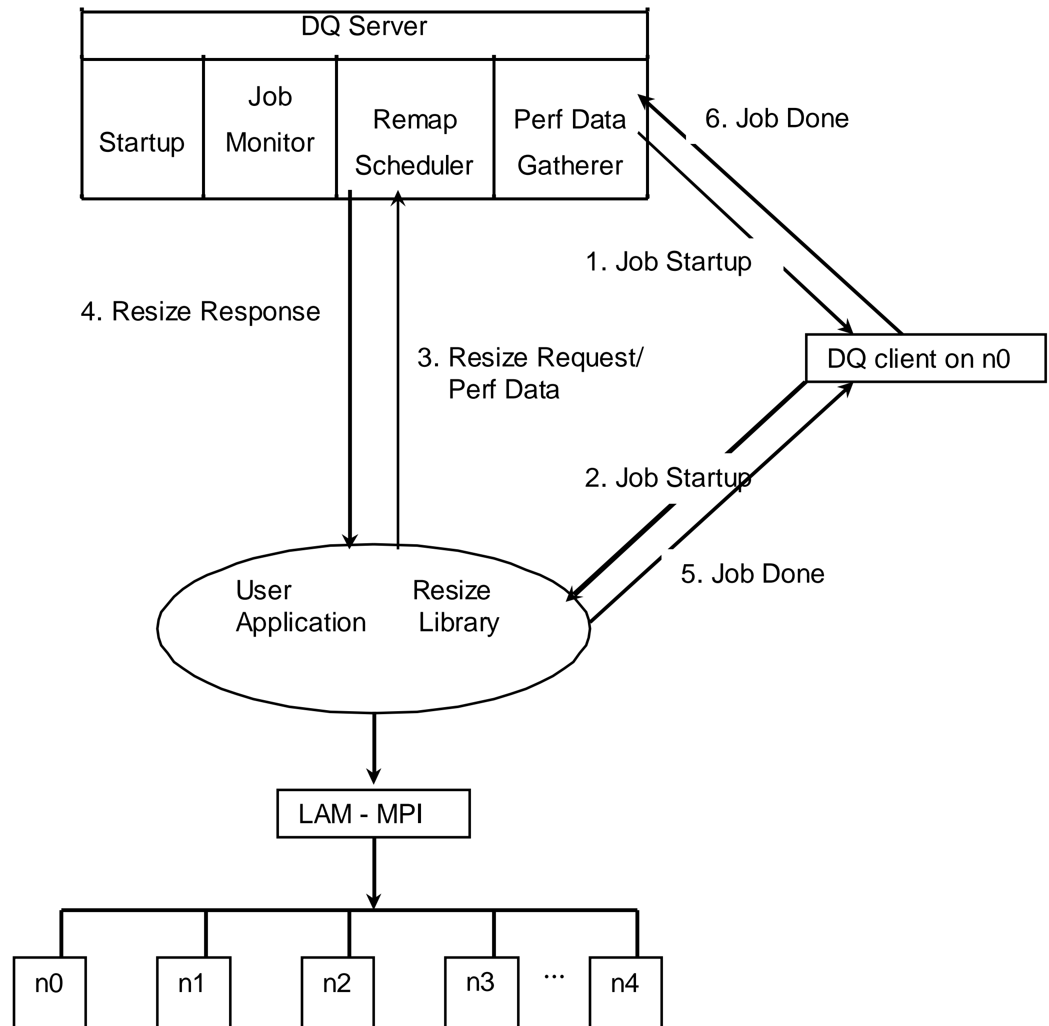


Figure 3: Architecture diagram

The above figure shows the general architecture of our framework. A command line interface is used to schedules an application. The user provides the initial number of processors that the application must start on. Once processors become available for the

application to be started at this initial size, the job startup thread of the scheduler schedules this application on the assigned processors.

When the job is started the Job Monitor is notified. The application also notifies the Job Monitor when the application is terminated, so that the scheduler can reclaim all resources allocated to this application. Notifications to kill the job are sent through the command line, and the Job Monitor is responsible for killing the application on all nodes and reclaiming the processors.

At the end of every computational loop, the application contacts the scheduler. The application provides performance data to the scheduler. Currently the metric used to measure the performance of the application is the time taken to compute per iteration. The Performance Data Gather (PDG), which stores performance information for all applications currently running in the system, gathers the performance data provided by the application. This data is used to make application-resizing decisions.

When the application contacts the scheduler, the Remap Scheduler (RS) makes the decision of whether to allow the application to grow to a greater number of processors, shrink the processors allocated to a job and reclaim the processors to schedule a different application, or permit the application to continue at its current processor allocation. A decision to shrink can be made because the application has grown to a size that has not provided a performance benefit, and hence the RS asks the application to shrink back to its previous size. An application can also be asked to shrink if there are applications waiting in the wait queue to be scheduled. The RS determines the running applications that it needs to shrink such that the overall performance hit to the system is minimized. The heuristic used by the RS to make shrink decisions is described in Section 3.4.4. The RS can also ask the application to expand to a greater number of processors if there are idle processors in the system. The heuristic used by the RS to make expand decisions is described in Section 3.4.4. If the RS is not able to provide more processors for the application to expand, and determines that the application does not need to shrink, it asks the application to continue to run at its current processor allocation.

The resize library, which is linked to the application, is used to perform data redistribution and construction of new processor topologies when the RS asks the application to shrink or expand. After the resize library has performed the resizing of the application, the application can perform its computation on the new set of processors. This process continues for the lifetime of the application

3.3 Resize Library and API

The following pseudocode gives a high level view of a typical scientific application.

1. Do in a Loop
    a. Compute
2. End loop

In our framework we have introduced API calls that communicate with the scheduler and perform expansion and shrinking of the processor set. The following pseudocode gives a high level view of an application that supports resizing under our framework.

1. Do in a Loop
    a. Compute
    b. Contact scheduler at remap point.
        i. if (response is no change)
            1. continue
        ii. if(response is expand)
            1. start up processes on the new processors.
            2. Redistribute data to the new processor set.
        iii. if (response is shrink)
            1. Redistribute the data to the new set of processors.
            2. Terminate processes on the relinquished processors.
            3. Signal the scheduler on completion of the redistribution, so that the scheduler can free the resources.
2. End loop

3.3.1 Contacting scheduler at a remap point

At the end of a computational loop the application reaches its remap point. At this remap point it contacts the scheduler with performance data for the previous computational loop. This performance data includes the time spent computing and the time spent to

remap data if a remap was performed in the previous computational loop. The scheduler makes a decision on whether to allow the application to expand, to ask the application to shrink or to continue at its current processor allocation. The application then uses the underlying remap library to perform expansion or shrinking if required.

3.3.2 Expansion

If the scheduler asks the application to expand, it provides the application with the list of processors that it can use to expand to. The underlying remap library [1] then spawns the applications processes on these new processors and remaps the applications global data to the new processor set.

3.3.3 Shrinking

If the scheduler asks the application to shrink, the application first uses the underlying remap library [1] to redistribute the applications global data to the smaller subset of processors. It then terminates the processes on the processors that it will relinquish and responds to the scheduler asynchronously with the list of processors that have been relinquished.

3.4 Scheduler Components

Our scheduling framework was implemented by extending and modifying the DQ scheduler designed originally by Dr. Srinidhi Varadarajan and implemented by Satish Tadapalli [11].

3.4.1 Job Startup

When an application is scheduled by the user via a command line interface it enters the wait queue of the scheduler. When processors become available to start this application at the initial size requested by the user, the Job Startup thread of the scheduler initiates the applications startup on the set of processors assigned to it. The Job Startup thread sends the job start information to the DQ client on the first node (node 0) of the processors that have been assigned to the job. The DQ client then starts the application on all processors

and is responsible for sending job start, job end signals to the scheduler. The DQ client is also responsible for killing the application if required.

3.4.2 Job Monitor

The Job Monitor thread of the scheduler interacts with the DQ client on node 0 during the lifetime of the application. If an application fails due to internal error the DQ client sends a signal to the job monitor to this effect so that the job can be killed and all resources allocated to the application freed. The DQ client also sends periodic heart beat messages to the Job Monitor thread [11]. If the application fails on one of the processors due to hardware error, then the Job Monitor is able to detect this situation due to missing heart beats and hence is able to kill the application on all processors and free the resources allocated to it.

3.4.3 Performance Data Gatherer

When an application contacts the scheduler at a remap point, it provides application performance data for the just completed computational loop. This data is stored by the PDG to be used by the RS to make future remap decisions.

For every application the PDG stores a sorted list of the various processor sizes the application has run at and the performance of the application at each of these sizes. The PDG also maintains an active stack of the processor sizes that the application can shrink to and the performance penalty associated with shrinking to a smaller size.

3.4.4 Remap Scheduler

The following pseudocode gives a high level view of the remap scheduler.

1. determine available idle nodes in the system.
2. determine the number of nodes required by the next application in the wait queue.
3. If (wait_num_nodes > available nodes)
    a. Should this application shrink?
        i. If Yes ask application to shrink
        ii. Else ask application to continue at its current size.
4. determine if previous expansion was beneficial.

    a. If No, ask application to shrink to previous size. We have determined sweet spot for this application.
5. Determine if this applications can expand to occupy idle nodes in the system.
    a. If Yes, expand.
    b. Else continue at the current size.

When there are applications waiting to be scheduled in the wait queue, the currently running applications are not allowed to expand to occupy idle processors in the system. The RS determines if the current application is one of the applications that needs to shrink in order to make processors available for the applications in the wait queue. This decision is made such that the overall performance penalty to all applications as a result of shrinking is minimized.

However, if the wait queue is empty and there are idle processors in the system, then the RS determines if the current application is one of the applications that can expand to occupy the idle processors. This decision is made such that the overall performance benefit among all the running applications is maximized. An application that has just returned from an expansion and has not benefited from that expansion, i.e. the performance of the application degraded as a result of the expansion is required to shrink to its previous size. This application will not be considered for expansion purposes beyond the point where it did not show a performance gain. This point is known as the sweet spot for that application.

Sweet spot determination

The PDG maintains a sorted list of the various processor sizes that each application has run on and the performance data associated with these sizes. As long as there is performance improvement the scheduler allows an application to grow to a larger processor allocation assuming that idle processors are available in the system. However, if the application reaches its sweet spot, a processor size where the application no longer shows performance gain, the RS will not allow the application to grow beyond this point. The application may shrink to accommodate other applications and then later grow back

to its sweet spot when idle processors become available, but it will not be considered for expansion beyond its sweet spot.

Shrink decisions – calculating performance penalty

An application can shrink under two conditions:

1. The application has determined its sweet spot.
2. The application has to forfeit some of its processors for other applications in the wait queue.

When an application contacts the scheduler at a remap point, the RS determines if this application needs to shrink in order to accommodate the waiting applications. This decision is based on relative performance hits of all applications running in the system.

The application uses the processor size stack information gathered by the PDG to determine the processors each application can relinquish and the performance degradation of the applications for each of the sizes it can shrink to. The RS constructs a shrink points (SP) list. Each entry in the shrink point list consists of the Job ID of the application, the number of processors that are made available if the application shrinks to a smaller size and the performance hit it will suffer as a result of this shrinking. The RS sorts this list in ascending order of performance hit, and picks SP's that will satisfy the processor requirements of the next job in the wait queue.

If the RS is able to find SP's such that applications can shrink to accommodate the waiting application, and the application that has currently contacted the scheduler is one of these applications, then the application is asked to shrink to its SP. As a result of this shrinking more idle nodes become available in the system.

Active applications that are required to shrink will shrink to their corresponding SP's as and when they contact the scheduler until enough processors are available to schedule the job in the wait queue.

Expand decisions – calculating performance gain

The expansion decision is made in a similar fashion to the shrink decision. For all applications that are running at a smaller size than there is performance data available for, a Expand Point (EP) list is created. The EP list consists of the Job Id of the application, the number of processors it will expand by and the performance gain it will achieve as a result of this expansion. This list is sorted and if the current application is one of the applications that can expand to an EP to occupy idle processors it is asked to expand.

In the case where idle processors are available and no running applications have performance data beyond their currently running size, then the applications are allowed to expand to their next greater size. It is in this phase that new performance data is gathered by the PDG.

Chapter 4

Experimental Results

4.1 Experimental Setup

For our experiments, we used a homogeneous cluster comprising of 36, 1.4MHz AMD Opteron processors. Experiments were conducted by using both the Gigabit Ethernet interconnection network and the Myrinet interconnection network.

4.2 Adaptive Sweet Spot detection

The sweet spot detection experiments were performed for SPMD programs whose primary compute intensive tasks were Matrix Multiplication and LU Factorization. Experiments were conducted for various data sizes so as to provide relevant results within the processor limitations of our test cluster.

The input parameters for a given run of an application include the data size (N) and initial processor allocation. After every computational loop the application contacts the scheduler with performance information of the just completed computation. The scheduler gathers this information and allows the application to grow its number of processors as long as there is improvement in the iteration time. Once the application grows to a size that does not provide an improvement in computational time, the application is shrunk back to the previous size where it continues to execute for the rest of the computation. This point is known as the sweet spot. Note that for a large enough value of N the application can grow to occupy all processors and continue to show improvement in iteration time.

4.2.1 Matrix Multiplication

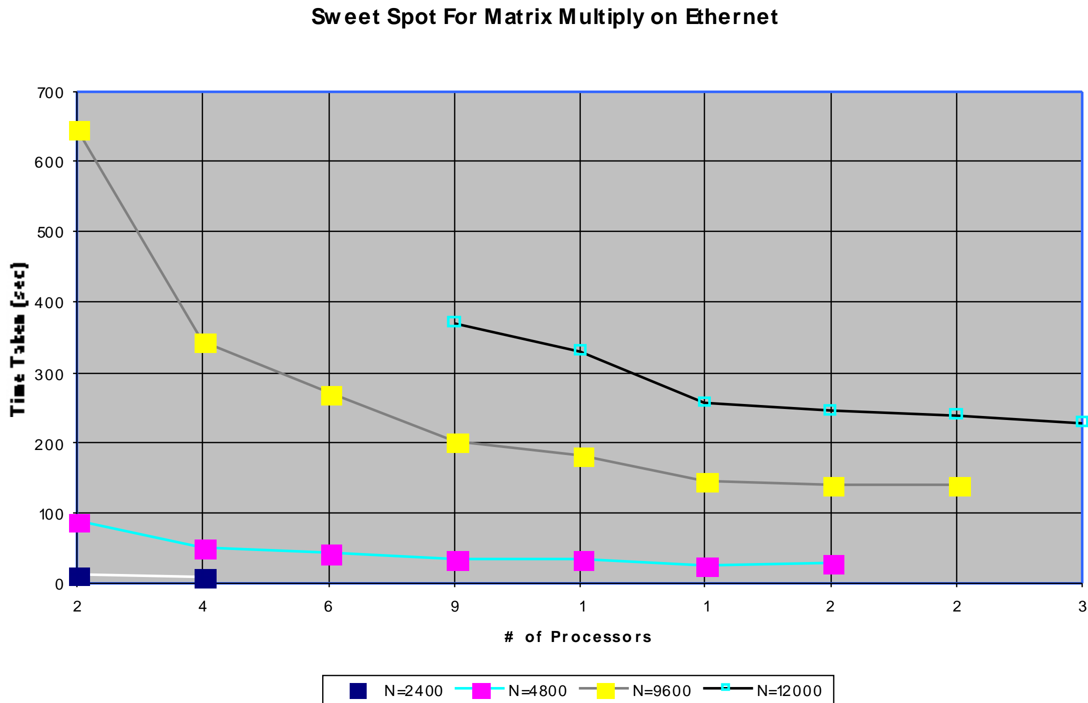
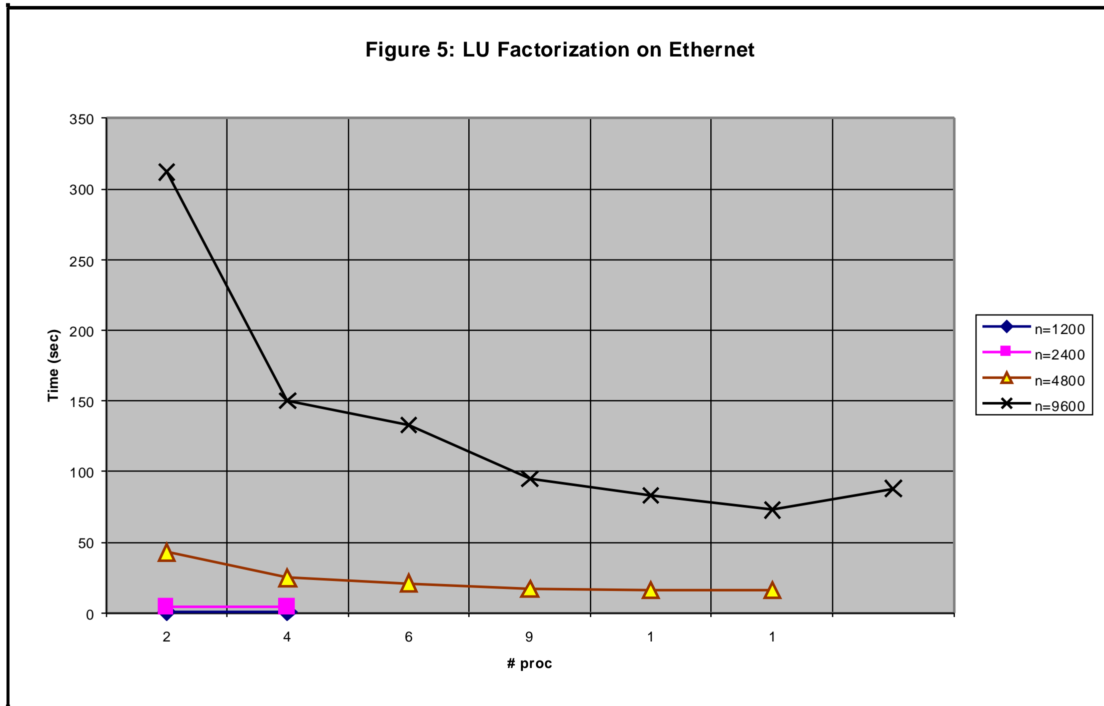


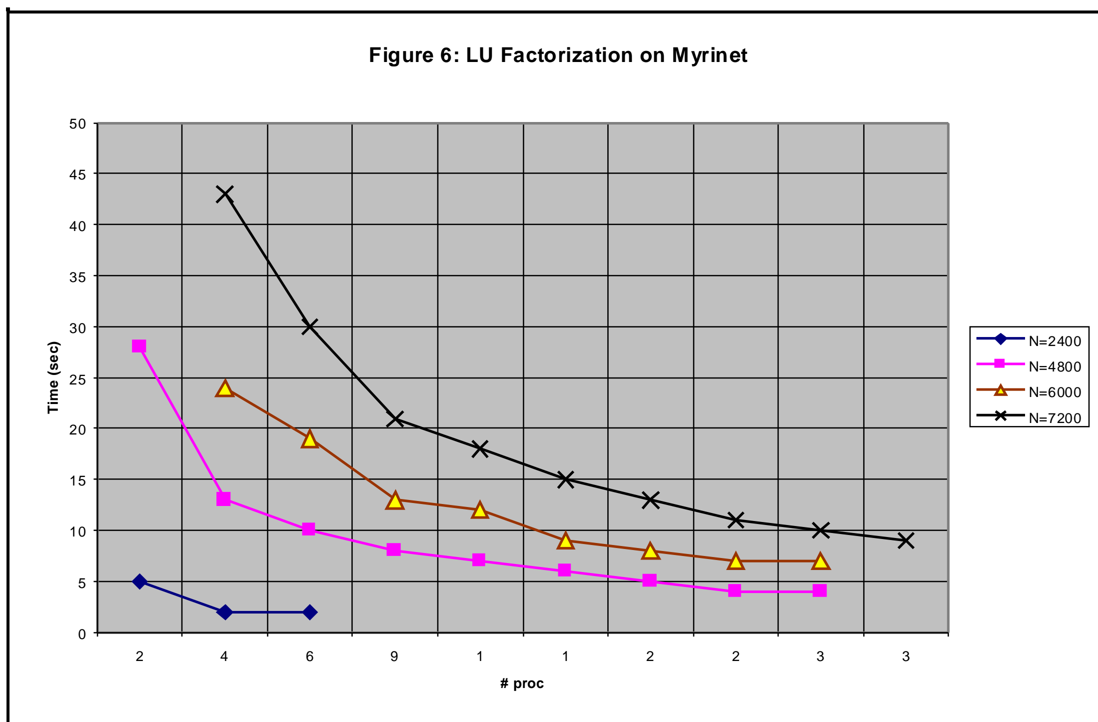
Figure 4: Sweet spot for matrix multiply on Ethernet

The above graph shows the performance data gathered by the scheduler during the matrix multiplication application execution for different sizes of N. For very small values of N, the framework is able to quickly determine the sweet spot within a few iterations. For example, for N=2400, the application grows to 4 processors but it is then realized that the expansion was not useful as it did not result in an improvement in the iteration time. The application then shrinks back to 2 processors and continues at this size for the rest of the computation.

For larger values of N, the framework iterates for a longer period of time before determining the sweet spot. This ramp up time is acceptable under the condition that the application will be iterating many hundreds or even thousands of times during the course of its execution.

4.2.2 LU Factorization





The above two graphs show the performance of the LU factorization model application for different sizes of N as they grow towards their sweet spots on two different

interconnection networks – Gigabit Ethernet and Myrinet. As shown by the above graphs, the performance characteristics are similar to the matrix-multiplication model application and follow the same trend regardless of a change in the underlying interconnection network. A scheduling framework to improve throughput and reduce job turnaround time as shown in Section 4.3 can leverage this nature of data centric applications.

4.2.3 Improvement in iteration time

The following tables illustrate the rate of improvement in iteration time of our sample matrix-multiply and LU factorization applications as they grow towards their sweet spot.

Current processor allocation	New processor allocation	%improvement in iteration time
2	4	46.9
4	6	20.9
6	9	25.8
9	12	9.4
12	16	19.7
16	20	4.1
20	25	0

Table 1: Matrix multiplication sample application for N=9600 on Ethernet

Current processor allocation	New processor allocation	%improvement in iteration time
2	4	51.9
4	6	11.3
6	9	28.5
9	12	12.6
12	16	12.0
16	20	-20.5

Table 2: LU Factorization sample application for N=9600 on Ethernet

As the application grows towards its sweet spot, the improvement in iteration time is not as much as compared to the initial stages of expansion. This property is leveraged, as shown in Section 4.3. Applications can be shrunk to a size that is less than the sweet spot without suffering a large performance penalty. The processors thus freed can be used to schedule other applications in the wait queue.

The sample applications used in our experiments perform computations with two-dimensional matrices of N rows and N columns. The above tables show that the

improvement in performance is more pronounced when the application grows from a non-square processor grid to a square processor grid.

4.2.4 Redistribution overhead

Figure 7: Matrix Multiplication, Redistribution overhead for N=9600

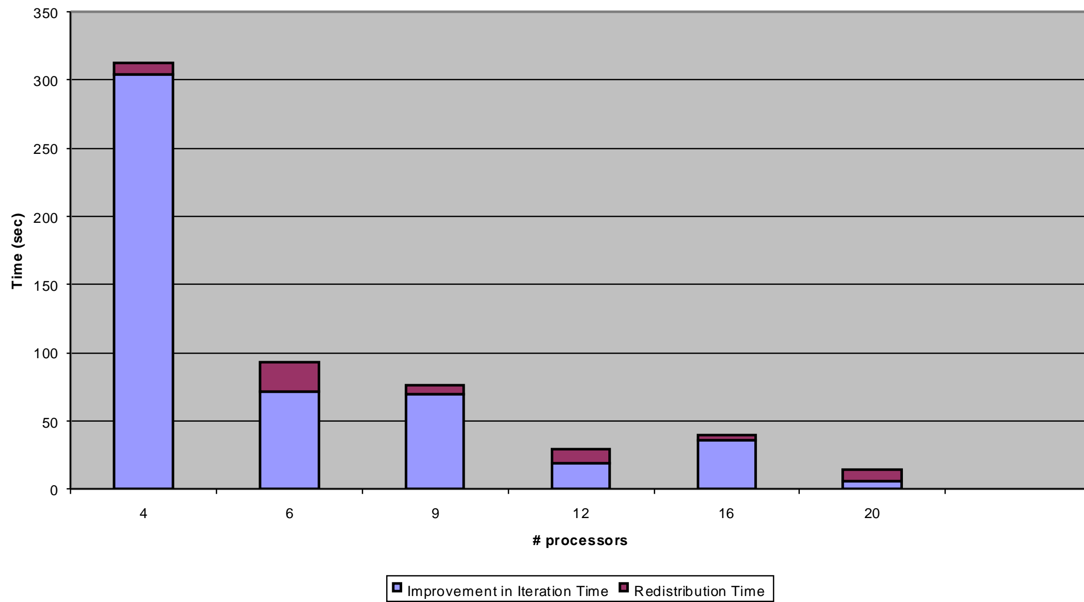
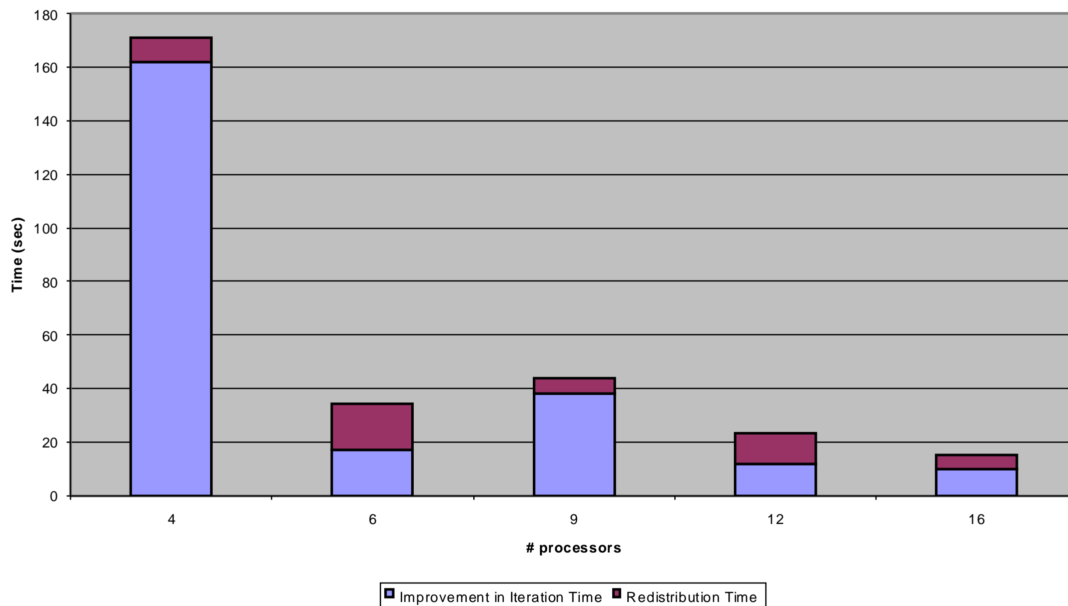


Figure 8: LU Factorization, Redistribution overhead, N=9600



The above tables show the overhead of redistribution in relation to the improvement in iteration time. The processor sizes indicate the new processor set the application is running at. The redistribution overhead is the time taken to redistribute data from the previous processor size to the size indicated in the charts. The application under consideration starts at an initial processor allocation of 2.

The model applications that we have considered are computationally intensive, i.e., the amount of time spent in computation is many magnitudes higher than the amount of time spent communicating data between processors. As a result of this, the time spent in redistributing data from one processor size to another is also far less than the amount of time spent in one loop of the computation for the processor sizes. Thus, even a small improvement in the computational rate can compensate for the redistribution time within a few iterations of the application at the new processor size.

The redistribution time does not result in a significant overhead in our model applications. However, for other scientific applications that employ functions such as the Jacobi method that are not very computationally intensive per iteration, the redistribution cost can be significant in comparison to the improvement in iteration time. Work done by Chinnusamy [1] aims at minimizing the redistribution overhead so that these applications can also benefit from dynamic processor allocation.

4.3 Processor allocation decisions for job mixes

In our framework, the adaptive sweet spot detection was used as a basis to implement scheduling strategies that leverage the fine grained control over processor allocations to jobs concurrently running on our test cluster. Experiments were conducted to demonstrate the potential benefit of dynamic processor allocation and parallel job resizing.

When jobs are concurrently running in a system and are competing for resources, not all jobs can adaptively realize their sweet spot and run at that sweet spot for the entirety of their execution. A job can grow to use the available resources while it continues to achieve a performance benefit. Even though the job might not have achieved its sweet

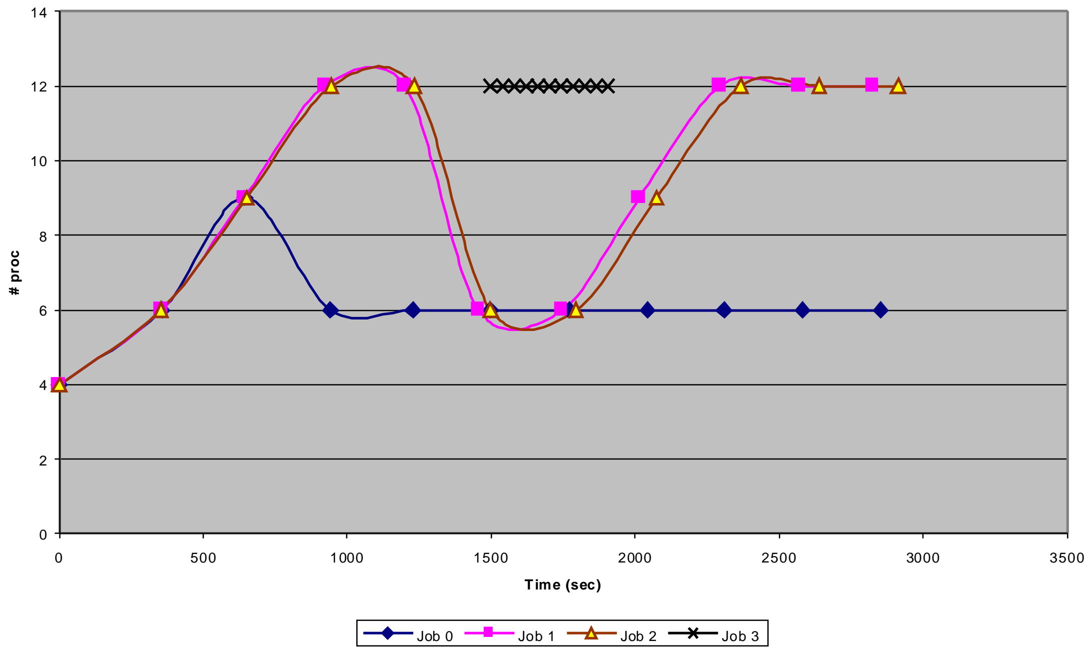
spot, it might have to shrink to accommodate other jobs that have been scheduled based on its relative performance gains while expanding in comparison to other running applications.

Since our system bases shrinking decisions on minimizing the overall performance hit to the system, applications that are closer to their sweet spot, and hence have not shown high performance gains in the later part of their growing phase are more likely to get shrunk than applications who are in the initial stages of their expansion phase and are showing significant performance gains.

Short running jobs benefit greatly in our system. Long running jobs can be shrunk to a smaller size without a significant performance hit, allowing the small job to be scheduled. This would not be possible in a conventional queuing system where the small job would have to wait for the completion of the larger job before being scheduled. The large job can quickly regain its original size after the completion of the smaller job.

Long running jobs also benefit from our system, since they have the ability to utilize the resources beyond their initial allocation, and users are not faced with the decision of scheduling the job on a smaller set of processors than desired based on current resource availability.

4.3.1 Job Mix 1



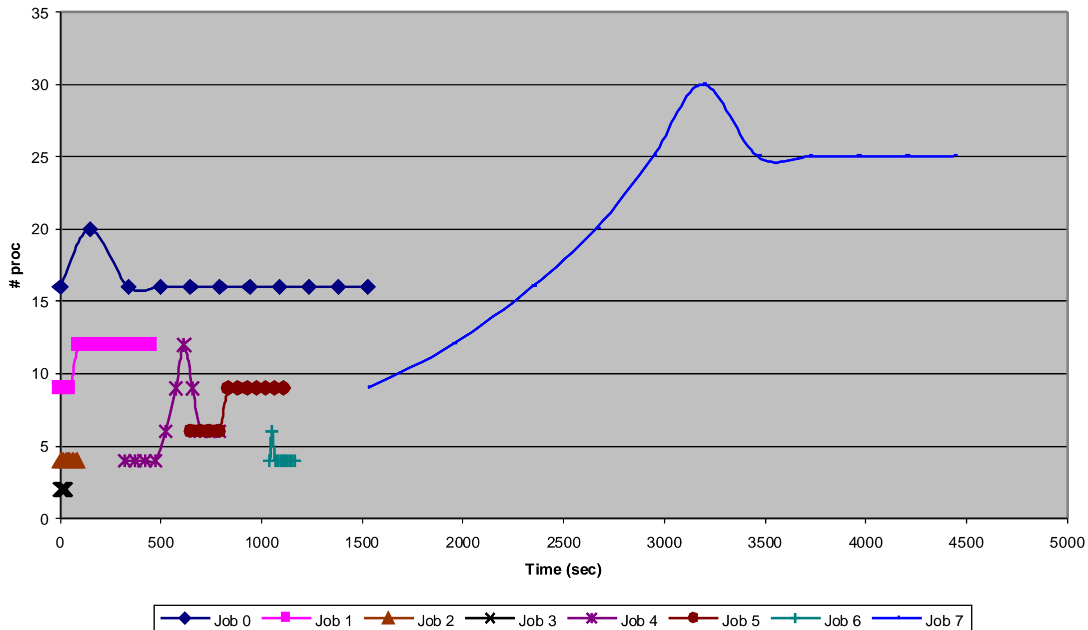
The above graph shows the lifetime of four jobs that were scheduled on our system. The total number of processors available in our test cluster is 32. Applications modify their processor sizes within this constraint. Our sizes for application data were influenced by the limitation of 32 processors. On a larger cluster comprising of hundreds or even thousands of processors, much larger applications can execute: larger in terms of both data size and processor allocations.

Initially, Jobs 0, 1 and 2 were scheduled with processor allocations of 4 processors each. These jobs are long running jobs and expanded to utilize the unused processors in the system. Job 0 was able to determine its sweet spot at 9 processors and continued at this size for the rest of its life time. Jobs 1 and 2 expanded to 12 processors and continued to run at that size till Job 3 was scheduled. As a result of job 3's presence in the wait queue, jobs 1 and 2 shrank to 6 processors in order to accommodate job 3's request of 12

processors. Since job 3 was a short running job, job 1 and 2 were able to expand back to 12 processors after the completion of job 3.

4.3.2 Job Mix 2

Figure 10: Job Mix 2



The above graph shows the life time of 7 jobs that were scheduled on our system. Initially Jobs 0, 1, 2 and 3 were scheduled. The data sizes of these jobs were chosen such that they represented four jobs in the range from very short running job to long running job. Jobs 2 and 3 were short running jobs that remained at their initial processor allocation and terminated, hence releasing these processors to Jobs 0 and 1 running on the system. Job 0 utilized these extra processors in order to realize its sweet spot at 16 processors and continued at this processor allocation for the rest of its life time. Job 1 was able to expand to 12 processors from its initial allocation of 9 processors and continued at that size for the rest of its lifetime. After the completion of Jobs 2 and 3, Job 4 was scheduled. Job 4 was able to expand to a larger size only after the completion of Job 1. It then expanded from its initial allocation of 4 processors to 12 processors. However, Job 4

had to shrink back to 6 processors because of the presence of Job 5 in the wait queue which was waiting for 6 processors. After the competion of Job 4, Job 5 and Job 6 were able to expand beyond their initial allocation. Job 7 was also scheduled during this stage. After the completion of the other jobs, Job 7 was able to expand to occupy all the computational resources of the system and realize its sweet spot.

4.3.3 System throughput and job turnaround time

System throughput is the average time after which a job is completed in the system. It is the time taken by the jobs in the system divided by the number of jobs in the system.

Job turn around time is the total time taken by the applications to execute in the system divided by the number of applications.

The tables below show the comparison of the time spent in the system by the jobs in Job Mix 1 on a static scheduler in comparison to our dynamic processor allocation approach.

Job	static proc allocation	start time	end time	total time	Dynamic start time	End Time	Total Time
0	6	0	3616	3616	0	2853	2853
1	6	0	3652	3652	0	2826	2826
2	6	0	3622	3622	0	2913	2913
3	12	0	408	408	1498	1906	408

Improvement in total time jobs spend in the system = 2298 seconds
Improvement in throughput = 184.8 seconds/job

Table 3: static processor allocation job mix 1

As shown in the above table, if all jobs are scheduled initially in the static scheduler, the system throughput and the job turnaround time are greater than in the case of the dynamic scheduler. This is because jobs 0,1 and 2 are not able to take advantage of the idle processors after the completion of job 3.

In the above case, Jobs 0,1 and 2 executed on 6 processors and could have run more efficiently if they had been scheduled on 9 processors each. This would mean that Job 3 would have to wait for the completion of these jobs before being scheduled.

Job	static proc allocation	start time	end time	Total time	Dynamic start time	End Time	Total Time
0	9	0	2746	2746	0	2853	2853
1	9	0	2887	2887	0	2826	2826
2	9	0	2965	2965	0	2913	2913
3	12	2965	3373	408	1498	1906	408

Improvement in total time jobs spend in the system = 6 seconds
Improvement in throughput = 115 seconds/job

Table 4: static processor allocation job mix 2

If jobs 0,1 and 2 were scheduled with 9 processors instead of 6, as in the previous case, they are able to perform better, hence alleviating the job turnaround time problem. As shown in the above table, the total time spent by jobs in the static system is only 6 seconds greater than the amount of time spent in our dynamic system. However, job 3 is not able to startup until the other jobs in the system have completed. Hence the throughput of the system is still significantly better than in the case of dynamic scheduling.

Chapter 5

Conclusions and Future Work

This chapter summarizes the features provided by our framework and future work that will extend the functionality of our framework.

5.1 Summary

We have provided a framework that enable parallel applications to dynamically resize efficiently during their execution. Our scheduling framework enables applications to iteratively expand to occupy more processors and determine the processor allocation at which the application performs the best. Under the condition of multiple applications being scheduled under our framework, applications are able to shrink to accommodate new applications without undertaking a severe performance penalty and are able to grow back to larger sizes when idle processors are made available. Our framework enables the scheduler and the application to work together to make good resizing decisions. The API provided by us enables conventional SPMD programs to be easily ported to take advantage of dynamic resizing.

We believe that with the introduction of the MPI-2 standard, dynamic processor management will be popularized and application developers will consider this aspect during their application development in order to achieve better throughput and job turn around times.

5.2 Future Work

Exploring different processor topologies and historical data.

Scientific applications can use the processors assigned to them under many different processor topologies. For example, if the application views the processors as a 2-D grid of MxN, then many choices of M and N are possible for any given total processor size. The performance of the application on these different processor topologies will be different based on data distribution and the communication properties of the application. Performance data can be gathered at these different processor topologies to get a better understanding of the application performance. Moreover, recommender systems can be used to use this information to predict performance of applications at topologies for which data is not available. The data collection phase can be significantly reduced if the information from previous runs of the application can be incorporated in the sweet spot determination process.

Other performance metrics

Our framework uses the time spent by the application in one iterative loop of its computation. Performance metrics such as machine load averages, applications parallel efficiency and speed up can be used to explore other definitions of sweet spots. Thresholds need to be determined such that application do not expand if the performance gain is not within a given threshold.

Extend scheduling framework

Our scheduling framework provides the basic functionality required for dymanic processor management. In order to be used as a viable scheduling system constructs such as job priorities, reservations and Quality of Service need to be addressed. With the growing popularity of grid computing through the Globus framework work needs to be done in order to make our work viable across multiple clusters and on a global scale.

REFERENCES

[1] M. Chinnusamy. Data and processor re-mapping strategies for dynamically resizable parallel applications. Master's Thesis. Virginia Tech, 2004.

[2] J. E. Moreira and V. K. Naik. Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications. *IBM Research Report RC 20890,* 1997. *IBM Journal of Research and Development*, Vol. 41, No. 3, pages 303-330, May 1997.

[3] Park, K.-H. and Dowdy, L. W. Dynamic partitioning of multiprocessor systems. *International Journal of Parallel Programming*, 18(2):91–120, 1989.

[4] Tucker, A. and Gupta, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 159–166, December 1989.

[5] Leutenneger, S. T and Vernon, M. K. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 226–236, May 1990.

[6] Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

[7] McCann, C., Vaswami, R., and Zahorjan, J. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, May 1993.

[8] Naik, V. K., Setia, S. K., and Squillante, M. S. Processor allocation in multiprogrammed, distributedmemory parallel computer systems. Technical Report RC 20239, IBM Research Division, October 1995. To appear in Journal of Parallel and Distributed Computing.

[9] McCann, C. and Zahorjan, J. Processor allocation policies for message-passing parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 19–32, May 1994.

[10] Squillante, M. S. *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, chapter On the benefits and limitations of dynamic partitioning in parallel computer systems, pp. 219–238. Springer-Verlag, 1995.

[11] S. Tadepalli. DQ Scheduler. Master's Thesis. Virginia Tech, 2003.

[12] PVM. http://www.csm.ornl.gov/pvm/pvm_home.html

[13] MPI. http://www-unix.mcs.anl.gov/mpi/

[14] J. Dongarra and I. Duff and D. Sorensen and H. van der Vorst. Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia, 1998.

[15] LAPACK Project, http://www.netlib.org/lapack/

[16] ScaLAPACK Project, http://www.netlib.org/scalapack/
[17] D. Gelernter and D. Kaminsky. Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha. *Proceedings of the International Conference on Supercomputing*, ACM, pages 417-427, July 19-23, 1992.

[18] Seti At Home. http://setiathome.ssl.berkeley.edu.

[19] L.V. Kale and Sanjeev Krishnan. CHARM++ : A Portable Concurrent Object Oriented System Based On C++. Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, Sept-Oct 1993.

[20] Sathish S. Vadhiyar and Jack J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems.

[21] Amnon Barak and Oren La'adan. The {MOSIX} multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems.* Volume 13, pages 361-372, 1998.

[22] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. Technical Report CSE-95-002, 1, 1995.

[23] Fred Douglis and John K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Journal of Software - Practice and Experience*, volume 21-8, pages 757-785, 1991.

[24] Robins, K. A. and Robins, S. *The Cray X-MP/Model 24*, volume 374 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[25] Gupta, A., Tucker, A., and Stevens, L. Making effective use of shared-memory multiprocessors: The process control approach. Technical Report CSL-TR-91-475A, Computer Systems Laboratory, Stanford University, 1991.

[26] Moreira, J. E. *On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[27] Polychronopoulos, C. Auto-scheduling: Control flow and data flow come together. Technical Report 1058, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1990.

[28] Edjlali, E., Agrawal, G., Sussman, A., and Saltz, J. Data parallel programming in an adaptive environment. In *Proceedings of 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[29] Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele Jr., G. L., and Zosel, M. E. *The HighPerformance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[30] Edjlali, E., Agrawal, G., Sussman, A., and Saltz, J. Data parallel programming in an adaptive

environment. In *Proceedings of 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[31] Carriero, N. and Gelernter, D., How to Write Parallel Programs: A Guide to the Perplexed, ACM *Computing Surveys,* volume 21, pages 323-358, September 1989.

[32] The BLACS Library. http://www.netlib.org/blacs/

[33] LAM-MPI. http://www.lam-mpi.org/

[34] MPICH. http://www-unix.mcs.anl.gov/mpi/mpich/