12-15-2007

Modeling of an Adaptive Parallel System with Malleable Applications in a Distributed Computing Environment

Sheikh Khaled Ghafoor

MODELING OF AN ADAPTIVE PARALLEL SYSTEM WITH MALLEABLE APPLICATIONS IN A DISTRIBUTED COMPUTING ENVIRONMENT

By

Sheikh K. Ghafoor

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2007

MODELING OF AN ADAPTIVE PARALLEL SYSTEM WITH MALLEABLE APPLICATIONS IN A DISTRIBUTED COMPUTING ENVIRONMENT

By

Sheikh K. Ghafoor

Approved:

Dr. Ioana Banicescu Professor of Computer Science and Engineering (Major Professor and Joint Dissertation Director)	Dr. Tomasz Haupt Research Associate Professor Center for Advanced Vehicular Systems (Committee Member and Joint Dissertation Director)
Dr. Susan Bridges Professor of Computer Science and Engineering (Committee Member)	Dr. Thomas Phillip Professor of Computer Science and Engineering (Committee Member)
Dr. Edward B. Allen Associate Professor of Computer Science Engineering (Graduate Coordinator)	Dr. Anthony Skjellum Professor of Computer and Information and Sciences University of Alabama at Birmingham (Committee Member)

Dr. Roger King
Associate Dean, Research and Graduate Studies
Bagley College of Engineering

Name: Sheikh K. Ghafoor
Date of Degree: December 2007
Institution: Mississippi State University
Major Field: Computer Science
Major Professor: Dr. Ioana Banicescu
Title of Study: MODELING OF AN ADAPTIVE PARALLEL SYSTEM WITH MALLEABLE APPLICATIONS IN A DISTRIBUTED COMPUTING ENVIRONMENT
Pages in Study: 173

Candidate for Doctor of Philosophy

Adaptive parallel applications that can change resources during execution, promise increased application performance and better system utilization. Furthermore, they open the opportunity for developing a new class of parallel applications driven by unpredictable data and events. The research issues in an adaptive parallel system are complex and interrelated. The nature and complexities of the relationships among these issues are not well researched and understood. Before developing adaptive applications or an infrastructure support for adaptive applications, these issues need to be investigated and studied in detail. One way of understanding and investigating these issues is by modeling and simulation. A model for adaptive parallel systems has been developed to enable the investigation of the impact of malleable workloads on its performance. The model can be used to determine how different model parameters impact the performance of the system and to determine the relationships among them.

Subsequently, a discrete event simulator has been developed to numerically simulate the model. Using the simulator, the impact of the variation in the number of malleable jobs in the workload, the flexibility, the negotiation cost, and the adaptation

cost on system performance have been studied. The results and conclusions of these simulation experiments are presented and discussed, suggesting interesting insight for further investigation.

In general, the simulation results reveal that the performance improves with an increase in the number of malleable jobs in a workload, and that the performance saturates at a certain percentage of rigid to malleable jobs mix. A high percentage of malleable jobs is not necessary to achieve significant improvement in performance. The performance in general improves as the flexibility increases up to a certain point; then, it saturates. The negotiation cost impacts the performance, but not significantly. The number of negotiations for a given workload increases as number of malleable jobs increases up to a certain point, and then it decreases as number of malleable jobs increases further. The performance degrades as the application adaptation cost increases. The impact of the application adaptation cost on performance is much more significant compared to that of the negotiation cost.

DEDICATION

To my friend and life partner Kakon

ACKNOWLEDGMENTS

I would like to express my sincerest appreciation for the people who helped me in many ways during the course of this dissertation. My earnest gratitude goes to my major professor and joint dissertation director, Dr. Ioana Banicescu, who guided me throughout my academic term here at Mississippi State University (MSU) with valuable advice, constant encouragement . I would also like thank her for the patience to accommodate my mistakes and to make me learn from them.

I am extremely grateful to Dr. Tomasz Haupt, joint dissertation director, for his invaluable advice and gracious help throughout my dissertation work. I am grateful for the generous financial support and research opportunities he has provided throughout my appointment as Research Associate in the Center for Advanced Vehicular Systems (CAVS) at MSU.

I am grateful to Dr. Susan M. Bridges, member of my graduate committee, for her invaluable advice and gracious help throughout my graduate studies. I would also like to gratefully acknowledge Dr. Thomas Philip and Dr. Anthony Skjellum for serving as members in my graduate committee, and providing me with helpful suggestions and comments.

I wish to thank all the faculty members at the Department of Computer Science and Engineering whom I have come across and learned my way into the world of

teaching and research. I would like to take this opportunity to acknowledge the support for infrastructure and resources needed in this work as provided by CAVS at MSU. A special appreciation goes to my colleagues and fellow Vortal group students at CAVS specially to Greg, Anand, Igor, Bhargavi, Mahbubur, Satya and Nisreen. I would like to specially acknowledge the friendly staff at the Department of Computer Science and Engineering (Brenda Collins, Jo Coleson, and Brandi Velcek), who was always willing to help me with anything and everything that I needed during my time at MSU

On a personal note, I wish to express my heartfelt gratitude to my spouse and friend, Ambareen without whose love, sacrifice & support, I would not come this far.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

LIST OF TABLES

TABLE Page

LIST OF NOMENCLATURE

Adaptive RMS	An RMS capable of managing rigid as well as malleable application.
Adaptive Parallel Application	A parallel application that can change number of processor during execution.
Adaptive Parallel System	A system that consists of a adaptive RMS managing a cluster of nodes, and a set of parallel applications containing rigid, as well as malleable applications.
Job	A parallel or sequential application submitted by a user for execution to a computer system.
Workload	A set of independent jobs submitted to s system for execution. Each job has an arrival time, resource requirement, and execution time.
Rigid Workload	A workload consists of rigid jobs only
Malleable Workload	A workload consists of a mixture of rigid and malleable jobs or malleable jobs only.
Arrival Time	$\boldsymbol{T_a}$ - The time when an application is submitted for execution.
Start Time	$\boldsymbol{T_c}$ - The time when a job start executing (running) on computing node(s).
Completion Time	$\boldsymbol{T_c}$ - The time when a job finish executing (running) on computing node(s)
Execution Time	$\boldsymbol{T_e}$ - The difference between start time and completion time of a job.
Rigid Execution Time	Execution time of a job if it executes as a rigid job.
Turn Around Time	**TAT –** The difference between arrival time and completion time of a job.
Average Turn Around Time	**TAT –** The average of TAT of all jobs in a workload.
Schedule Span	Time between the arrival of first job and completion of last job in workload.
Utilization	$\boldsymbol{U}$ – Fraction of total CPU cycles jobs in workload are executing during the schedule span.

CHAPTER I

INTRODUCTION

This dissertation addresses the problem of modeling and simulation of an adaptive parallel system with malleable applications in a distributed environment. In order to demonstrate the utility of simulation environment, we present results, analysis, and interpretations of simulation experiments with malleable applications in an adaptive parallel system. The purpose of this chapter is to outline the research work conducted, provide necessary background, present the motivation for the research, postulate the research hypothesis, discuss contributions, and clarify the terminology used throughout the document.

1.1 Adaptive Parallel Applications

Feitelson and Rudolph [1] classified parallel applications into four groups based on who decides the number of processors to be used by a parallel application and on when the decision is made. The classification is shown in Table 1.1.

Table 1.1 Classification of parallel applications

Who decides	When decided	
	At submission	During execution
Application	Rigid	Evolving
System	Moldable	Malleable

A *rigid* application requires a certain number of processors as specified by the user at application submission time, and it cannot execute with fewer processors or make use of any additional processors. In the case of a *moldable* application, the number of processors assigned is determined by the system scheduler within certain constraints and the application may use only that particular number of processors throughout its execution. An *evolving* application may initiate a change in the number of processors during execution. In the case of a *malleable* application, the number of processors assigned to an application may change during execution as a result of the system offering it additional processors or requiring that the application releases some. Throughout this dissertation we define *adaptive applications* as applications that are evolving or malleable according to Feitelson's classification. The change in resource requirements in an evolving application is triggered by the application itself due to the nature of the problem and employed algorithms, while in malleable applications, change is triggered by events external to the application such as changes in hardware availability, applications of higher priority requiring more resources, and others reasons.

Characteristics of Adaptive Parallel Applications: There are many scientific and engineering applications which work with large input data and are computationally intensive. Some of these applications are highly parallel (including embarrassingly parallel) in nature. Among these there are highly scalable applications operating over a long range of resources. The total amount of computation for these applications does not vary from execution to execution for the same input data. Currently, these applications are implemented as rigid or moldable parallel applications. These types of applications

are good candidates for conversion to malleable applications where the resource management system (RMS) can provide them with idle resources whenever available and take away resources whenever high priority applications or other evolving applications need them.

Examples of evolving applications with predictable computation are parallel applications where computational workload varies during the execution due to nature of the problem, the employed algorithm, and an unpredictable non-uniform distribution of input data. The total computational workload of these applications does not vary from execution to execution for the same input data. The parallel Fast Multipole algorithm for *N-body* simulation is an example of such applications [2][3]. The N-body simulations consider *N* particles, their positions and velocities, and the problem is to compute the forces they exert on each other, and then calculate their new positions. This problem has been widely used in a broad class of application areas of science such as astrophysics, molecular dynamics, biophysics, molecular chemistry, and others. Even though the amount of total computation does not vary, due to the nature of algorithm, the computational requirements and consequently, the resource requirements of these types of applications change during a single run.

There are evolving applications in the fields of science and engineering where computational workload varies from execution to execution for the same input data. An example of such a field is weather prediction. Traditional weather forecasting systems are static in nature. They run simulations on a pre-scheduled space and forecasting time. Such systems cannot respond to mesoscale weather changes that can appear suddenly and

evolve rapidly. The researchers are working on the next generation of forecasting systems which will adapt dynamically both in time and space. The simulations will generate successive forecasts more frequently, and they will run on higher resolution grids for more accurate predictions. The resource requirements (processors, disk storage and network bandwidth) of such applications change dynamically during execution. One of the major obstacles of realizing such adaptive weather prediction systems is the absence of "adaptive cyber infrastructure" capable of supporting adaptive applications [4][5]. The amount of computation of these types of applications varies from execution to execution, even for same input data due to unpredictable conditions that can occur during execution. The resource requirements of such an application may also vary during its executions and cannot be determined precisely before the execution starts. Such applications are an example of evolving applications.

In general adaptive applications may have the following characteristics:

1. In a parallel application, the amount of computation for the same input data may remain the same from one execution to another or it may vary due to unpredictable events or data during execution. One may expect that the amount of computation for malleable applications will be constant, while for evolving applications the amount of computation may change during execution. That would require consumption of additional resources or release of idle resources.
2. Applications may have certain resource utilization characteristics. They may not be able to utilize any arbitrary amount of resources between a maximum and a minimum amount. For example, many applications require that the number of

processors be a power of two. If such an application is running on eight processors, it would not be able to utilize four additional processors or release two processors. However it may be able to utilize eight additional processors if allocated or release four processors if required.

3. Adaptive applications consist of phases. The amount of resource consumed within a phase remains unchanged. Applications can change resource consumption at phase boundaries. Parallel applications have natural breakpoints where processes of an application can synchronize, exchange data, or redistribute data among themselves. As a result, in such applications, adaptation (consumption of additional resources or release of idle resources) can occur at these breakpoints, not arbitrarily at any point in time during execution. If a malleable application is allocated additional resources or asked to release some resources, it consumes or releases resources at the next breakpoint where it can reconfigure itself to consume or release resources. Consequently, there may be a time gap between the adaptation decision by the Resource Management System (RMS) (asking an application to consume or release resources), and the actual adaptation (consumption or release of resources). The interval between two consecutive breakpoints constitutes a phase. Rigid and moldable applications have only one phase while evolving and malleable applications have multiple phases.
4. Both malleable and evolving applications require communications and negotiate resources with the RMS. The negotiations can be initiated either by the RMS in

case of malleable applications or by the applications in case of evolving applications.

1.2 A Resource Management System for Adaptive Parallel Applications

An adaptive parallel system is defined as a system that consists of a Resource Management System (RMS) managing a cluster of nodes and a set of parallel applications containing rigid as well as malleable applications. The goal of an RMS is to provide support for efficient utilization of computational resources and for resolving conflicts of interests between the end users. The schedule determines the order in which the applications are executed on computing nodes. The schedule is generated according to system policies. The RMS functionality also includes actual resource allocation (submitting the applications) or de-allocation (terminating the applications), and reporting the applications' status (such as pending, running, or completed) to the system administrator and to the end users. To perform its functions, the RMS monitors its resources and accepts messages about resource status changes such as a processor failure, or application completion and a resulting release of resources. Typically, an RMS has two main components: the server and the node controllers.

The Server: The server is the central coordinator of the RMS and is responsible for gathering information about the available resources, accepting applications from the users, organizing those applications in queues, and initiating a schedule cycle. Once a schedule is contrived, the server contacts the individual node controllers which place applications into execution. The server contains a scheduler that computes a schedule

when the server initiates a scheduling cycle. The scheduler implements a scheduling algorithm to satisfy the system policies taking into account the current status of the system (i.e. information about the state of the resources and queued applications). The generation of a schedule is triggered by scheduling events. Traditional RMS has two types of scheduling events: i) submission of an application by user, ii) termination of a running application.

The Node Controller: There is one node controller for each computing node and each acts as an agent of the server. The node controller starts and controls applications on the nodes and reports node and application status to the server.

If a workload contains malleable and evolving applications in addition to rigid applications, a traditional RMS will require additional functionalities to manage adaptive applications. In an adaptive parallel system, while computing a schedule, if enough resources are not available, the RMS has the option of preempting resources from running malleable applications. When running evolving applications require additional resources, they must ask the RMS for resources and the RMS must consider these requests while computing a schedule. Multiple evolving applications may request additional resources at the same time. The RMS must be able to handle multiple simultaneous resource requests from evolving applications efficiently. If idle resources are available and there are no pending applications, they may be allocated to running malleable applications. If computing a schedule involves preemption of resources from multiple malleable applications and/or allocation of resources to multiple evolving applications, the RMS must communicate and negotiate with all these running

applications within a single scheduling cycle. The RMS must also efficiently manage these negotiations with multiple applications in a single scheduling cycle.

1.3 Motivation

Current resource management systems for clusters primarily support rigid applications. A few systems support moldable applications, where there is some flexibility in the amount of resources that can be assigned before the applications start. However, the resources are fixed once the applications start execution. A new paradigm for cluster resource management is required as result of the dynamic resource requirements of adaptive applications, as well as of the unpredictability of resource utilization in a cluster. On one hand, an evolving application may require additional resources during execution to accomplish its objectives or relinquish resources not needed for its level of current workload. On the other hand, changes in the computing environment may require that a currently executing application be malleable. In this case, the application should be capable of relinquishing currently assigned resources for use by other applications, which are critical, or be capable of using additional resources that are otherwise idle, for earlier completion.

In addition to the possibility of underutilizing resources, fixing the number of processors assigned to an application may also prevent another application with a higher priority or revenue from being started due to an insufficient number of available processors in the system. To retain the capability of the system to immediately start higher priority applications without preempting running applications and to avoid the

penalty for lost opportunities, it makes sense for the resource manager to be able to request processors from currently executing applications that have lower priorities or revenues.

Enabling the execution of malleable and evolving applications on clusters leads to the potential of birth of a new paradigm for cluster computing. This paradigm will provide a framework for improving the utilization of existing clusters by allowing idle resources to be assigned to currently executing applications. It will also provide a framework for supporting mission-critical applications on general-purpose clusters by allowing a reassignment of already committed resources and excluding the need for expensive dedicated resources. The paradigm has the potential to improve the capability, responsiveness, and efficiency of existing clusters, as well to create new ways of utilizing current technologies for emerging applications characterized by dynamic resource requirements. It will also open up the opportunity of implementing a new class of parallel applications driven by unpredictable data and events. The paradigm will further propel the research for fundamental understanding of evolving and malleable applications, and their interactions with resource management systems and scheduling techniques, leading to the design and building of supportive computing system software.

The research issues in an adaptive parallel system are complex and interrelated. The nature, complexities and relationship of these issues are not well researched and understood. Before developing adaptive applications or an infrastructure support for adaptive applications these issues need to be investigated and studied in detail. One way of understanding and investigating these issues is by modeling and simulation. Modeling

and simulation of an adaptive parallel system will help us to understand, analyze and predict the behavior of adaptive applications and of the new RMS.

1.4 Research Issues in an Adaptive Parallel System

From the description presented in sections 1.1 and 1.2, it is evident that the management of adaptive applications in a cluster environment is a complex and multi-faceted problem. Some of the most important research issues are described below.

1.4.1 The Communication and Negotiation with Running Applications

In the current job-scheduling paradigm, once a job starts executing, no communication takes place between the application and the RMS. In an adaptive parallel system, communication and negotiation between the RMS and adaptive application is necessary. One of the research problems in an adaptive parallel system is concerned with what would be the communication mechanism and negotiation protocol between the adaptive applications and the RMS and with how the negotiation would be carried out.

1.4.2 The Negotiation Management

In an adaptive parallel system the need for negotiations arises from the characteristics of adaptive applications. These negotiations need to be managed efficiently, and that is a complex problem. For example when there are multiple requests from several running evolving applications, the RMS can initiate a scheduling cycle for every request, or it can group a set of requests and handle them together. Also, during

computation of a schedule, the RMS has to communicate and negotiate with multiple running applications. These negotiations may be carried out sequentially one after another. Alternatively, they can be carried out in parallel. Different negotiation management strategies would require different design and implementations of the RMS and they may impact the performance of the system and applications in different ways.

1.4.3 The Scheduling

Scheduling algorithms for adaptive applications are more complex than those for rigid applications. The scheduler has to respond in a timely manner to the demands of both running and queued applications. If enough resources are not available, it has to choose resource preemption candidates among running malleable applications. When idle resources are available, the algorithm must decide how to allocate the idle resources among the running malleable applications. If multiple evolving applications are requesting additional resources and if enough resources are not available even after preemption, the scheduling algorithm has to decide how to address those requests. Essentially, scheduling in an adaptive parallel system is a multi-step procedure that involves computing a schedule, negotiation with running applications, and re-computing the schedule. The re-computation of a schedule may result in requiring further negotiations.

1.4.4 Programming Model for Adaptive Applications

The structure of adaptive applications is different from that of rigid applications. They need to communicate and negotiate with the RMS using a protocol that the RMS

understands. Adaptive applications require reconfiguring themselves when additional resources are consumed or idle resources are released. The reconfiguration may require data redistribution, creation of new processes or deletion of existing processes etc. Currently, to the best of our knowledge, there is no programming model which captures all the aspects of an adaptive parallel application.

1.4.5 Overall Model of an Adaptive Parallel System

The research issues mentioned above are not independent but interrelated. For example, computing a schedule may involve negotiations with running adaptive applications. Consequently, the outcome and cost of scheduling depends on how negotiation is carried out and managed. The negotiation involves the RMS and the running applications. As a result, the characteristics and the model of adaptive applications influence the negations. The interdependencies and relationships of scheduling, negotiation, and application behaviors impact the overall architecture of an adaptive RMS. The relationships among these research issues and their overall impact on the model of an adaptive parallel system are not yet very well understood.

1.5 Ongoing Work in Adaptive Parallel Systems

This section presents a brief summary of related research in adaptive parallel systems, while a detail review of related research is presented in Chapter II. The adaptive parallel system is not a well-researched area in computer science. Some research has been conducted on scheduling of moldable and malleable applications [6][7][8][9]. Some work has been accomplished on developing infrastructure support for adaptive applications

[10][11][12][13][14][15]. None of these efforts have explicitly targeted the communication and negotiations between the applications and the RMS or dynamic resource reallocation that are imperative to support adaptive applications.

We have been working on developing infrastructure support for adaptive applications in cluster environments. The outcome of our efforts so far has been reported in [16][17][18]. A protocol for resource negotiation between adaptive applications and the RMS has been developed. Experiments with prototype implementations show that the protocol works. It covers a wide range of interaction scenarios between applications and the RMS, and the overhead of the protocol is very low [16]. We have also proposed an architecture for an RMS capable of managing adaptive applications. An early prototype implementation indicates that developing a robust RMS capable of managing adaptive applications in cluster environment is possible, and that adaptive applications show promise for improving system performance [7][18]. However, during our research work, we have realized that the area of adaptive parallel applications and system support for executing such applications is not a well-researched area in computer science. The properties of adaptive applications, their impact on the RMS and on the middleware are not well understood and require further investigations. This leads to our current research effort on modeling an adaptive parallel system as a whole to understand and analyze properties of adaptive applications, the RMS, and their effects on each other.

1.6 Dissertation Objectives

The context of this dissertation is to develop a model for an adaptive parallel system with malleable applications and to investigate it using simulation. In general, the

presence of malleable applications imposes new requirements on all the components of an adaptive parallel system. The dissertation objective is to investigate the impact of these requirements by modeling and simulation. In particular, the objectives of this dissertation are to investigate the impact of the characteristics of the workload and those of the RMS on system performance. More specifically, the effect of the number of malleable applications in the workload, their flexibility, cost of negotiation, and cost of adaptation on system performance will be investigated.

1.7 Hypothesis

A model for adaptive parallel systems can be developed to enable the investigation of the impact of malleable workloads on its performance. The model can be used to determine how different model parameters impact the performance of the system and, to develop a better understanding of the relationships among them. This model can be used to predict the performance variation in the presence of malleable applications as opposed to the same applications being rigid.

1.8 Contributions

The primary contribution of this dissertation is the design and implementation of a model for an adaptive parallel system, and the demonstration of how this model can be used to gain new knowledge about the impact of the model parameters on performance of the adaptive parallel system. The main contributions of the dissertation are summarized below.

The design and implementation of:

a. A conceptual model and subsequently a semi-formal mathematical model for an adaptive parallel system.

b. A model for generating malleable workload.

c. A discrete event simulator to numerically simulate models of adaptive parallel systems. In particular, the simulator can be used to determine the impact of RMS, application and workload parameters on system performance.

d. A prototype RMS system capable of managing malleable as well as rigid applications.

- The demonstration of model utility through discovery of the following knowledge about an adaptive parallel system with malleable applications:

e. The performance in general improves with an increase in number of malleable jobs in a workload and the performance saturates at a certain job mix. At this point, a higher percentage of malleable jobs do not result in significant improvement in performance.

f. Presence of malleable jobs in a workload decreases the average turn around time and average wait time. However, the presence of malleable applications increases the average execution time.

g. The performance in general improves as the flexibility increases up to certain point.

h. The negotiation cost does not significantly impact the performance.

i. The number of negotiations for a given workload increases as number of malleable jobs increases up to a certain point. As the number of malleable jobs increases further the number of negotiations decreases and it reaches a minimum with 100% malleable jobs.

j. The performance degrades as the application adaptation cost increases. The impact of application adaptation cost is much more profound compared to negotiation cost.

1.9 Organization

This remainder of the dissertation is organized as follows. Chapter II presents a review of the background literature related to the current work. Chapter III discusses a conceptual and mathematical model for an adaptive parallel system. A discrete event simulator and a prototype implementation of an RMS for an adaptive parallel system are presented in chapter IV. Chapter V describes simulation experiments to validate the simulator. In Chapter VI results and analysis of experiments to investigate the impact of model parameters on performance are discussed. Finally, Chapter VII presents a summary and describes future extensions and possible applications of this research.

CHAPTER II

BACKGROUND AND RELATED WORK

The area of adaptive parallel system is a promising research area that has only recently begun to attract attention. It is not a well researched topic in computer science and therefore today, there is only a handful of literatures available in this area. The related works in this area can be grouped in three main areas:

(a) Resource Management System (RMS) and language support for adaptive parallel applications

(b) Scheduling of adaptive parallel applications

(c) Programming models for adaptive parallel applications

The following sections review selected research work that is deemed relevant to the dissertation.

2.1 RMS And Language Support for Adaptive Parallel Applications

Much work has been conducted on Resource Management System (RMS), both in academia [9][20][21][22] and industry [23][24][25][26][27][28], addressing many vital aspects of efficient resource management. These systems represent the current state-of-the-art for problems that can be solved using static and open-loop scheduling approaches. They deal with both simple workloads (independent sequential and parallel jobs) and dependant workloads (workflows) [28]. Some implementations support moldable jobs

[28]. Other implementations address resource co-allocation through advance resource reservation.

The need for RMS support for adaptive applications on distributed memory systems has been recognized and addressed by many researchers. In the simplest form, the adaptation can be achieved by checkpointing a running application and restarting it with a different resource allocation. Vadhiyar and Dongarra [29] developed a framework for malleable jobs called SRS (Stop and Restart System) based on check pointing and migrations. Using this framework users can checkpoint and stop a parallel application and then restart the application on a different number of processors to continue from the checkpointed state.

To expand or shrink an application, users have to checkpoint and stop a job and then restart the application on the different number of processor, which incurs large overhead. Data redistribution is done by the SRS library and it supports very simple parallel applications. SRS does not support applications with file I/O, structures and pointers. Users need to modify their applications to use this framework. SRS does not address the resource allocation or scheduling problem directly. It is more of a checkpoint-restart system than a resource management system for malleable jobs.

One of the early research efforts in developing RMS was a malleable job system for time shared parallel machine developed by Kale et. al. [14][15]. This work concentrates on developing a framework for malleable applications for timeshared parallel machines. The framework consists of three components, a scheduler, the Charm [30][31] runtime system based on converse [32], and a set of malleable jobs developed using AMPI [33] or

Charm++[31]. AMPI is an adaptive version of MPI implemented as user level threads. Malleable programs based on AMPI consist of a large number of virtual processes implemented as user level threads. Typically the number of virtual processes is larger than the number of actual processors. Charm++ is an object-based language for parallel programming. A Charm++ parallel program is mapped to a large number of parallel objects that communicate with each other by message passing. The Charm runtime system has a load balancer that balances the load between processors by redistributing threads or objects. Parallel program developed on the Charm system has the ability to accept a processor map (a bit vector) from external programs. A set bit in the processor map indicates that the processor is allocated to the program. In Charm a new parallel job is started on all processors in the system but load is only allocated to the processors enabled in the processor map. The scheduler adapts (shrinks or expands) a malleable job by sending a new processor allocation (bit vector) to the job; the run time system then balances load by migrating processes from old allocation to new allocation. A skeleton process is left behind on each vacated processor to forward messages meant for processes that were previously running on that processor.

As there is no support for accepting a request from a running job, the charm framework does not support evolving jobs. There is only one-way communication from the scheduler to the running job. Therefore, there is no support for negotiation between a running job and the scheduler. Applications developed in this framework are not truly malleable applications because there are fixed number of processes throughout the life time of an application with shrinkage and expansion achieved by folding or unfolding

processes onto physical processor. In this system a malleable application has to be developed in AMPI or charm++. A bit vector is too simplistic and cannot be used for scheduling other resources (memory, disk space, bandwidth etc.).

The Distributed Resource Management System (DRMS) [13][34][35] is an integrated environment for the development, execution and resource scheduling of adaptive applications. DRMS supports adaptive applications based on SOP (Schedulable and Observable Point) model. In the SOP model, the execution of a parallel program consists of a sequence of stages. Each stage is like a conventional SPMD program, and the number of tasks and the association between data spaces and tasks is fixed during a stage. The boundary between stages is called SOP. The stage of a program can only be examined at an SOP and the number of tasks and association between tasks and data space can be altered. Each stage in a SOP program consists of four sections: resource, data, control and computation. The resource section specifies the number of tasks in the form of a range of valid number of tasks. Once a specific number of tasks are selected for execution of a stage, the data section specifies an association between data space and tasks. The control section specifies the values for control variables pertinent to the stage. Finally, the computation section specifies the computations and communications that each task performs during the stage.

The main components of the DRMS framework are: the DRMS compiler, job scheduler and analyzer (JSA), resource coordinator (RC), and a run time system (RTS). The DRMS compiler translates a program written in the DRMS language, linking them with RTS to create a reconfigurable (malleable according to our definition) SOP

program. DRMS language is an extension to FORTRAN that includes directives for creating a SOP program. The resource allocation and scheduling decisions are made by the JSA on the basis of implemented scheduling policies. The RC on behalf of JSA communicates with running applications to convey reconfiguration decisions. An application compiled under the DRMS has an associated task coordinator (TC) consisting of multiple agents. One of the agents acts as master coordinator and communicates with RC. The TC delivers reconfiguration messages to RTS. At the next SOP the RTS in conjunction with TC, redistributes application data so that application can run with a new set of tasks on a new set of processors.

The DRMS is a tightly coupled integrated environment. Applications have to be developed in the DRMS environment and DRMS supports adaptive applications developed in FORTAN based on SOP model only. There is no support for resource negotiation in DRMS. Therefore, the JSA needs to know the valid resource configuration of applications in advance to be able to reconfigure an application. The data redistribution after reconfiguration is done by the system, not by the application as a result the DRMS supports a limited class of parallel application.

Hungershofer et. al. [10][11] developed a resource management system called Application Parallelism Manager(APM) for scheduling malleable jobs on shared memory machines. APM determines processor assignment to running jobs based on estimated current speedup to maximize system utilization. APM is implemented as a single server consisting of a database and two threads which access the database. One thread listens for information from the running application and stores it in the database. The other thread

reads information from the database, computes processor assignments and sends the assignments to running applications. Entries in the database are created when a running application connects to the server for the first time and the entries are deleted when an application disconnects after its completion. The APM and malleable applications communicate with each other using TCP sockets. Applications send runtimes of parallel and sequential phases to the APM. The information sent out by the application has to be provided by the application itself. Applications send their information periodically and also send information when major changes occur. For communication with the APM, applications need to be compiled with a library provided by APM. Applications also need to incorporate codes to monitor their sequential and parallel runtime. The information sent by the APM to applications contains only one single value specifying the number of assigned processors.

The APM implemented two scheduling policies. One is equipartitioning where processors are distributed equally among running jobs. The other policy is based on application speed up. The system was tested on a 16 processor SMP system with 4 instance of the same application (a multi-threaded finite element simulation). The results indicate that both scheduling policies achieve a shorter schedule span than optimal offline schedule with moldable jobs. The authors reported that scheduling many malleable applications on a large system leads to complex behavior. It is not clear from their publications how APM will perform with large number of applications on a large system. APM works on shared memory machines only and it does not support evolving applications

The work by Jha et al. [36][37] addresses the adaptive resource allocation problem for a pool of dependent applications (subtasks) cooperating in real time towards a common goal. The applications are event driven and data dependent and their computational needs and resource requirements vary due to runtime changes in event rates and input data content. Jha et al. have adopted a four step operational model for dynamic resource allocation to meet the deadline of the entire application.

Monitor application performance using real time instrumentation.

Detect deviation in performance from desired performance level.

Compute a new resource allocation that would likely improve performance significantly.

Effect the new resource allocation in a manner that minimizes the perturbation to the application due to the transition.

The system does not accept any new job arrival and minimizes the total execution time of the complete task. The applications are instrumented to monitor their performance and to report the current rate of data processing to the system. The system analyzes the reports from all concurrently running applications and reallocates resources in order to meet the deadline for the entire task. This solution is application domain-specific as the system is responsible for converting the rate of data processing of the subtasks into an estimate of the completion time, leaving no room for the resource negotiations.

Little research has been reported in the literature about language and runtime support for developing adaptive applications. Edjlai et al. developed a runtime library called Adaptive Multiblock PARTI (AMP)[38][39] that enable users to create adaptive applications. AMP can be used by compilers for data parallel applications such as HPF or

it can be used by a programmer for developing adaptive parallel applications by hand. According to the authors there are two major issues in executing applications in an adaptive environment:

> Redistributing data when the number of available processors changes during the execution of the program.
>
> Handling work distribution and communication, insertion and optimization when the number of processors on which a given parallel loop will be executed is not known at compile time.

AMP addresses both of these issues and supports parallel programs using the single program multiple data (SPMD) model of execution. It is targeted towards an environment in which a parallel program must adapt according to the system load. AMP assumes that

> The adaptive program does not remap immediately when the system load changes.
>
> When the program remaps from a larger number of processors to a smaller number of processors, it may continue to use a small number of cycles on the processors it no longer uses for computation.

An application based on AMP is marked with remap points, and adaptation can occurs only at remap points. Remap points can be specified by the programmer if the program is parallelized by hand, or may be inserted by the compiler if the program is compiled by a parallel compiler such as HPF. An AMP program is spawned on the maximum number of processors on which it can run. At remap points AMP determines if there is a need for adaptation. If the system load requires adaptation, data redistribution is used to move the active data to a different subset of processes called active processes. Only the processes that belong to this subset perform computation. The processes from

which all data has been removed are called skeleton processes. They still execute the code for the application but, since they have no data associated, they do not perform intensive computations. AMP imposes a hard limit on the maximum number of active processes, namely the number of processes that were originally spawned. The skeleton processes can interfere with other applications that have active processes on the same physical processors.

2.2 Scheduling of Adaptive Applications

Scheduling algorithms that have been developed so far vary in their objectives and approach to solving the problem at hand. Most of these algorithms have their roots in queuing theory, graph theory, dynamic and linear programming, and most recently, genetic and annealing algorithms. This section presents a review of related research works on scheduling algorithms for adaptive application.

Most of the current work on scheduling adaptive applications has been theoretical. Turek, Wolf, and Yu [40] used approximation algorithms and were the first to propose a two-phase approach to schedule malleable jobs. Their goal was to find a non-preemptive schedule that minimizes the make span (total of maximum execution time for all jobs). The basic idea is to select an allotment (number of processors allocated to each task) using a packing algorithm, and then solve the scheduling problem of these tasks using non-malleable scheduling algorithms. This effort resulted in a polynomial time allotment selection algorithm. The results were further improved by Dutot, Mounie and Trystram[41]. Dutot et al. [6][7][8][9] proposed several approximation algorithms for scheduling moldable and malleable applications. Two scheduling criteria were considered

for evaluating the algorithms: minimization of the makespan and minimization of the average completion time. For the problem of scheduling malleable applications, their approach was based on batch scheduling; applications may arrive at anytime, but are scheduled in successive batches. Mounie, Rapine, and Trystram [10] adopt a two-phase approach similar to the method of Turek et al. [40] and focuse on the first phase, the allotment selection, to reduce its complexity. The allotment selection was done by either using the canonical list scheduling algorithm or a knapsack algorithm, resulting in a linear time complexity. They defined a moldable job using Feiteson's [1] definition as malleable job. Their research does not consider true malleable jobs (i.e. job that can change number of processors during execution) in their scheduling algorithms.

Most current RMS use simple scheduling schemes such as First-Come-First-Served and priority with some variation of backfilling or gang scheduling. Only a few of these RMS are aware of adaptive applications, and they use a heuristic approach for scheduling adaptive applications.

The dynamic resource management system DRMS [16] uses a reconfigurable scheduling (RS) policy. Each malleable application has to provide a set of acceptable numbers of processors it is capable of executing on. Under the RS policy, whenever processors are available to schedule jobs, the scheduler tries to schedule jobs in the order in which they arrived. However, instead of scheduling the earliest job on the maximum possible number of processors, it tries to schedule as many of the currently waiting jobs in the pending queue as possible. When not enough free processors are available and there are jobs waiting to run, it tries to free up processors from jobs that are

currently running on more than their minimum number of processors. Similarly, when there are no jobs waiting to be run and free processors are available RS tries to expand one or more of the running jobs to run on a larger set of processors. The authors compared the performance of reconfigurable scheduling with maximum-to-fit non-adaptive scheduling. A non-adaptive scheduling workload consists of moldable jobs only. Simulation experiments showed that, compared to non-adaptive policy, a reconfigurable policy always performs better.

Kale et. Al. [14][15] adopted a similar approach to schedule malleable jobs. Each arriving jobs specifies the minimum and maximum number of processors it can use. When a new job arrives, the scheduler recalculates the number of processors allocated to each running jobs. All jobs, including the new ones, are allocated their minimum number of processors. Leftover processors are shared equally, subject to each job's maximum processor usage. If a new job cannot be scheduled, it remains in the pending queue. When a running job finishes, the scheduler applies the same algorithm to allocate the free processors. Experiments with actual applications and simulation showed that both system utilization and mean response time improves with adaptive scheduling.

Hungershofer et al. [10][11]] implemented adaptive scheduling algorithms based on two different policies: equipartitioning and accumulated speed up. In both policies malleable jobs are started with a minimum number of processors. In the equipartitioning policy, free processors are distributed evenly among running malleable jobs. The other policy is based on application speed up. APM estimates the speed up of all running applications on an additional processor. The application with the highest differential

speed up (difference between current speed up and speedup on one additional processor) is allocated an additional processor. Experimental results show that the equipartitioning policy leads to better response times, while the accumulated speed up policy increases the throughput of the system.

2.3 Programming Model for Adaptive Applications

The major issues in developing adaptive applications are programming abstraction and efficient support for runtime adaptation. Programming abstractions should be easy to use so that application developers are motivated to develop adaptive applications. The programming abstraction should be easy such that they can be added to non-adaptive applications without much effort to convert them into adaptive applications. The mechanisms for supporting adaptive applications that has been reported in the literature for various programming models are described below. The programming models can be grouped into five categories: master-worker model, fork-join model, fixed task model and SOP model.

Master-worker model: Applications in this model consist of a master and several workers and computation for worker tasks is dynamically carved out. The master is a global entity that defines the tasks that must be executed and the data on which they operate. The master can be active (a process) or passive (a global state pool). The workers are given or fetch tasks from the master, execute them and return the result to the master. The Piranha[42] system is an example of a mechanism to support adaptive applications developed based on the Linda workers model [43][44]. The Linda model is a general model for parallel programming based on distributed data structures. In Linda the

total work to be done by the program is broken into a number of discrete tasks, which are stored in a global data space. One process known as the master is responsible for generating the tasks and gathering and processing the results. Actual program execution involves a number of component processes known as workers. Each worker removes a task, completes it, and then grabs another until some condition is met. Workers may encounter a special type of task known as a poison pill telling it to terminate [44]. Linda supports dynamic creation of processes. Piranha was mainly developed to utilize idle cycles in a network of workstations. It moves Linda processes from heavily loaded workstations to idle workstations.

Fork-Join Model: In this model a number of kernel level threads are scheduled for execution on physical processors; these kernel threads are then used as virtual processors for the execution of user level threads. The user level threads are created to execute tasks from a shared task queue. Examples of systems that use the fork-join model to support adaptation are Cray Multitasking [45], Process Control [46], Minos [47] and Autoscheduling [48][49]. The work on fork-join models mentioned above is all in the context of shared memory multiprocessors machines.

Fixed Task Model: This model supports adaptation of SPMD programs with in a fixed number of executing tasks. An application in the fixed task model is started with the maximum number of tasks (either thread or process) on which it can run. Examples of the fixed task model are programs based on AMPI [14][15] and AMP [38][39]. This model supports malleable applications only. Programs based on AMPI work in shared memory machines and consist of a large number of virtual processes implemented as user threads.

Typically the number of virtual processes is larger than the number of actual processors. Adaptation (shrinkage and expansion) is achieved by folding or unfolding threads onto allocated physical processors. In the case of AMP, adaptation is achieved by redistributing the active data to a subset of processes leaving some skeleton processes.

SOP Model: In the SOP programming model, the execution of a parallel program consists of a sequence of stages called schedulable and observable quanta (SOQ). The number of tasks is fixed during an entire stage, and the association between data spaces and tasks is fixed one-to-one. Therefore, each stage behaves like a conventional SPMD program. Boundaries between stages are defined as schedulable observable points (SOP). Reconfiguration of a parallel program can only occur at a SOP. During reconfiguration the associations between tasks and data spaces are altered. The stage following a reconfiguration point executes on a new configuration of tasks and data until it reaches a new SOP. A reconfiguration from one stage to the next may involve a change in the number of tasks, a change in the association of data with tasks, or both.

Each stage of a SOP program consists of four sections: resource, data, control and computation. The resource section specifies the number of tasks needed for the execution of the stage. This specification is usually in the form of a range of valid number of tasks. Once a specific number of tasks is selected for execution of the stage, the data section specifies an association between the data space and tasks. The control section specifies values for control variables pertinent to the stage. Control variables are used to control the flow of the execution inside a stage, which may vary depending on the number of tasks and data association. Finally, the computation section specifies the computations

and communications that each tasks performs for the execution of the stage. The computations and communications are usually steered by the control variables specified in the control section.

2.4 Limitation of Existing Research

RMS support for adaptive applications has been restricted to malleable applications only. Most existing systems assume that the malleable applications can operate on any number of processors between a minimum and maximum. This is a strict restriction on malleable applications. In practice, from logs of supercomputing centers it has been observed that many parallel applications require that the number of processors be a power of two. Current resource management systems for adaptive applications do not support evolving applications. None of the RMS mentioned in this chapter explicitly target the communication and negotiations between the applications and the RMS or dynamic resource reallocation, both imperative to support adaptive applications. Another limitation of the existing systems is that the RMS and applications are tightly coupled. There is no clear separation or interface between them. Data redistribution is performed by the RMS. Consequently, applications have to be developed in the framework provided by the RMS.

The scheduling approaches discussed in this chapter use heuristic techniques. So far, these scheduling algorithms focused on malleable applications only. Most of the algorithms adopted an equipartitioning policy for processor preemption or free processor allocations. No effort has been made to compare different preemption or allocation policies (for example, preempt resources from longest running job vs. shortest running

job). None of the scheduling algorithms have considered evolving applications in the workload. The scheduling algorithms assume that the malleable applications will be able to use any number of processors between the maximum and minimum number of processors. They have not considered negotiations as part of scheduling, which would change the scheduling strategy.

Though the existing programming models can be adopted for developing adaptive applications, none of these models address all aspects of adaptive applications. For example none of the existing models support the notion of negotiation. Some of the existing programming models do not support dynamic task creation and task destroying.

The main goal of existing research in adaptive parallel systems so far has been limited to accommodate a restricted model of malleable application to determine the impact of the presence of a malleable workload on system performance. None of the research addresses negotiation between the RMS and applications, and none of them address evolving applications. No research has been targeted towards determining the requirements imposed on the RMS by the presence of adaptive applications in a workload. No attempt has been made so far to study and understand the overall model of an adaptive parallel system.

CHAPTER III

THE ADAPTIVE PARALLEL SYSEM

The goal of this dissertation is to model and simulate an adaptive parallel system with malleable applications in a distributed computing environment. This goal is achieved through the following steps: 1) develop a conceptual model of the system, 2) design a mathematical model from the conceptual model, 3) develop a simulator, 4) validate the model using the simulator, and 5) gain new knowledge about the adaptive parallel system through simulation experiments. Figure 3.1 shows a dependency graph of the research methodology, and the main steps are briefly described below.

1. *Conceptual Model:* In this stage, an adaptive parallel system has been studied carefully to develop an understanding of the system. The important components of the system and interactions between them have been identified.

2. *Mathematical Model:* From the conceptual model a mathematical model has been developed. The components and their interactions are described using variables and equations. An adaptive parallel system is highly complex, so that developing an accurate and valid mathematical model is extremely difficult, if not impossible. Therefore, we have developed a semi-formal mathematical model and studied the model by means of numerical simulation.

3. *Simulator:* A discrete event simulator has been developed from scratch to simulate the model developed in the previous step.

4. *Validate the Simulator:* In this stage, the simulation experiments have been conducted using real world data. The goal of the simulations was to validate how accurately the model approximates the real system.

5. *Gain New Knowledge:* Simulation experiments with synthetic data have been conducted in this step. The goal of these experiments was to gain insight about adaptive applications, the RMS, and their interrelationship.

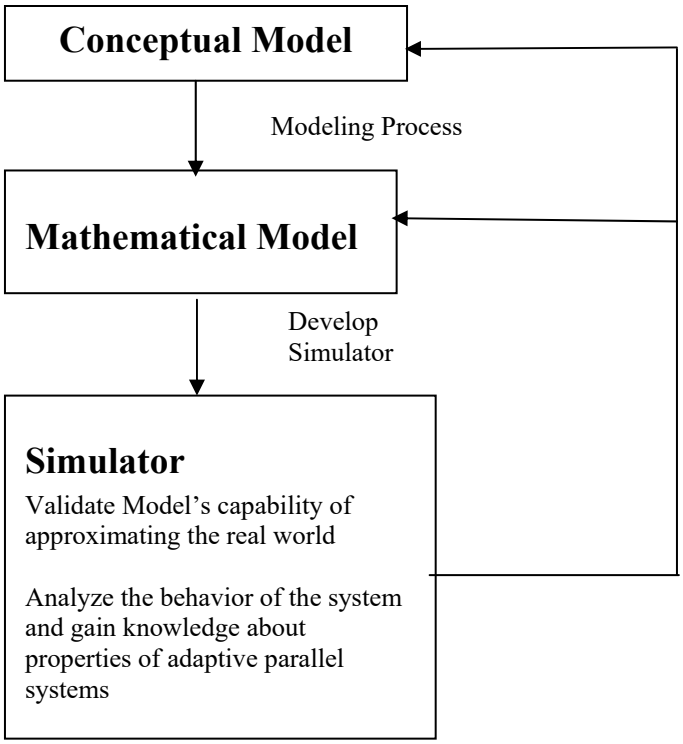


Figure 3.1 Research methodology

3.1 Description of an Adaptive Parallel System

The goal of a Resource Management System (RMS) is to provide support for an efficient utilization of computational resources and to resolve conflicts of interests between various end users. Typically, this goal is achieved by organizing the workload, composed of sequential and/or parallel applications, into queues and by creating a schedule. The schedule determines the order in which the applications are executed on computing nodes. The schedule is generated according to some system policies. The RMS functionality also includes the actual resource allocation (starting the applications) or de-allocation (terminating the applications), and reporting the applications' status (pending, running or completed) to the system administrator and to the end users. If the workload contains malleable and evolving applications in addition to rigid and moldable applications, a traditional RMS will require additional functionalities to manage adaptive applications. In an adaptive RMS, if enough resources are not available while computing a schedule, the RMS has the option of preempting resources from running malleable applications. When running evolving applications that require additional resources ask the RMS for resources, the RMS has to consider these requests while computing a schedule. Alternatively, if idle resources are available and there are no pending applications or pending resource requests from running evolving applications, the idle resources may be allocated to running malleable applications. In order to manage adaptive applications, interactions between applications and the RMS are required. To determine the functionalities that an adaptive RMS must have, we need to analyze the

requirements of an adaptive RMS. One way to analyze its behavior and requirements is to develop use cases for an adaptive parallel system.

3.1.1 Use Cases for an Adaptive Parallel System

From a high level perspective an adaptive parallel system can be viewed as being composed of three actors: users, RMS, and running applications that include malleable, evolving, and non-adaptive applications. One can think of four high-level use cases in an adaptive parallel system.

Submission of a new application

Completion of a running application

Idle resources released by a running evolving application

Additional resources requested by a running evolving application

The high-level use cases are described below.

3.1.1.1 Use Case: Submission of a New Application

Actors: users, RMS, running malleable applications

Overview: This use case starts when a user submits an application to the RMS for execution. The RMS starts the application if resources requested by the application are available. If resources are not available, depending on the policy, the RMS either tries to preempt resources from already running malleable applications and starts the application or puts the application in a pending queue. Running malleable applications may agree to release less or more resources than the amount requested by the RMS. For example, an application submitted by a user may require eight processors, and there are three idle

processors available, while a malleable application is running on eight processors. The requirement of the malleable application is that the number of processors be a power of two, and the minimum and maximum number of processor requirements are two and sixteen, respectively. The RMS may ask the malleable application to release five processors. Since the malleable application can operate on only a power of two number of processors, it may agree to release only four or six processors. In reply, the RMS may order the application to release six processors, and starts the new application on eight processors (three idle, and five from the six released processors). This use case ends when the submitted application starts execution or it is queued in the pending job queue. The use case is depicted in Figure 3.2.

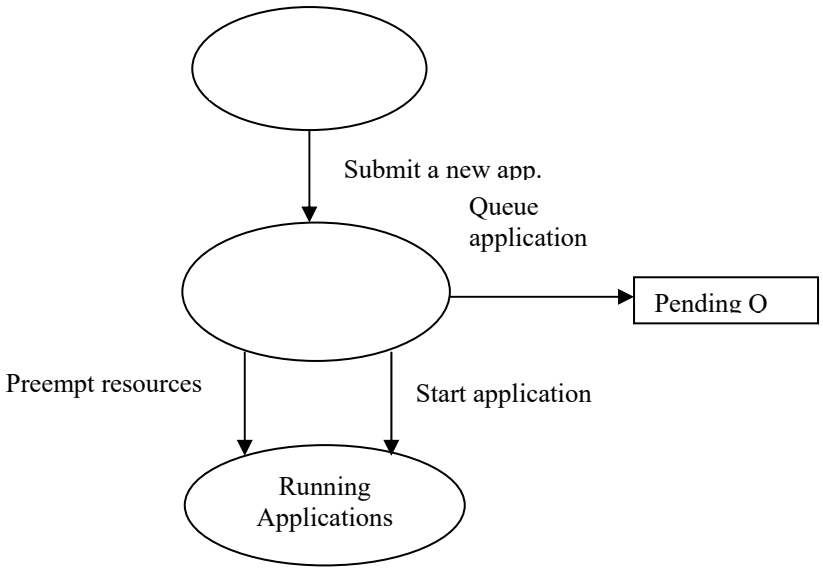


Figure 3.2 Diagram of interactions for the use case “submission of a new application”

3.1.1.2 Use Case: Completion of a Running Application

Actors: RMS, running application

Overview: This use case starts when a running application (non-adaptive or adaptive) completes its execution. The RMS updates the system state, and depending on the policy, it starts one or more pending applications from the queue (if there are any) and/or distributes the idle resources among running malleable applications. This use case ends when the RMS completes allocating the idle resources released by the completed application according to the system policy. Figure 3.3 shows this use case.

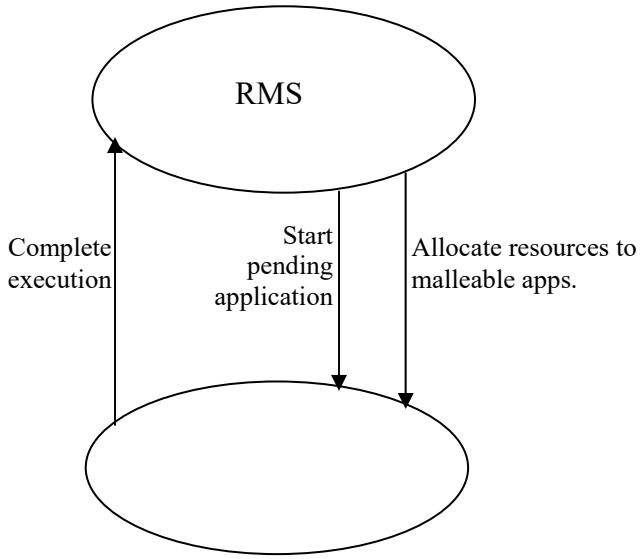


Figure 3.3 Diagram of interactions for the use case "completion of a running application"

3.1.1.3 Use Case: Resource Released By a Running Evolving Application

Actors: RMS, running evolving applications

Overview: This use case starts when a running evolving application releases some unutilized resources and informs the RMS. The RMS updates the system state and

depending on the policy, starts one or more pending applications from the queue (if there are any) and/or distributes idle resources among running malleable applications. This use case ends when the RMS completes allocating the idle resources released by the running evolving application according to the system policy. This use case is presented in Figure 3.4.

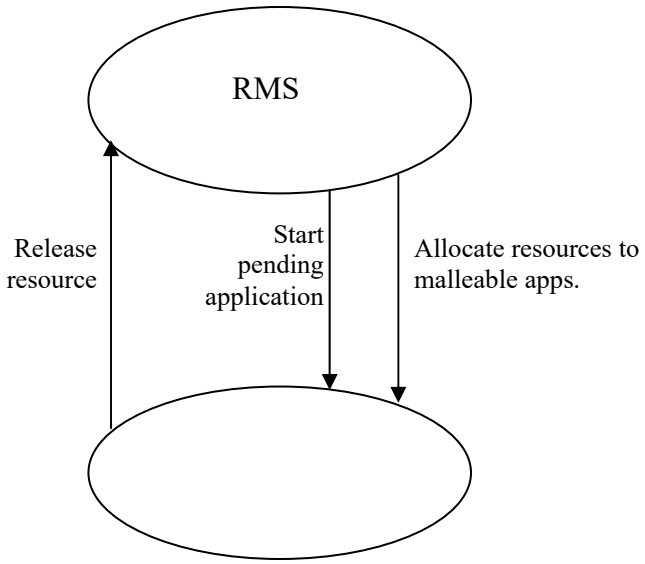


Figure 3.4 Diagram of interactions for the use case "resource released by a running evolving application"

3.1.1.4 Use case: Evolving Application Requesting Additional Resources

Actors: RMS, running evolving application, running malleable applications.

Overview: This use case starts when a running evolving application asks the RMS for additional resources. If resources are available, the RMS allocates the requested resources. If enough resources are not available, the RMS tries to preempt resources from one or more running malleable applications and allocates the requested resources to the running evolving application. If some, but not all resources are available (even after

preemption), the RMS may offer the available resources to the requesting application, instead of rejecting its request. The evolving application may or may not accept the offered resources. For example, consider an evolving application which requires that the number of processors be a power of two and is executing on eight processors. In mid execution, the application needs additional resources and asks the RMS for twenty four additional processors. The RMS may have only five idle processors available, and it can preempt ten more processors from three running malleable applications (one from the first, two from the second and seven from the third). Instead of rejecting the request, it may offer the evolving application fifteen processors. Since the application can use only eight additional processors out of fifteen offered, it may ask the RMS to allocate eight additional processors. The RMS may then ask malleable applications one and two to release one and two processors respectively, and allocate 8 processors (3 preempted and 5 idle) to the evolving application. The use case ends when the RMS completes resource negotiation with the requesting applications and allocates resources agreed during the negotiations, or rejects the request. This use case is presented in Figure 3.5.

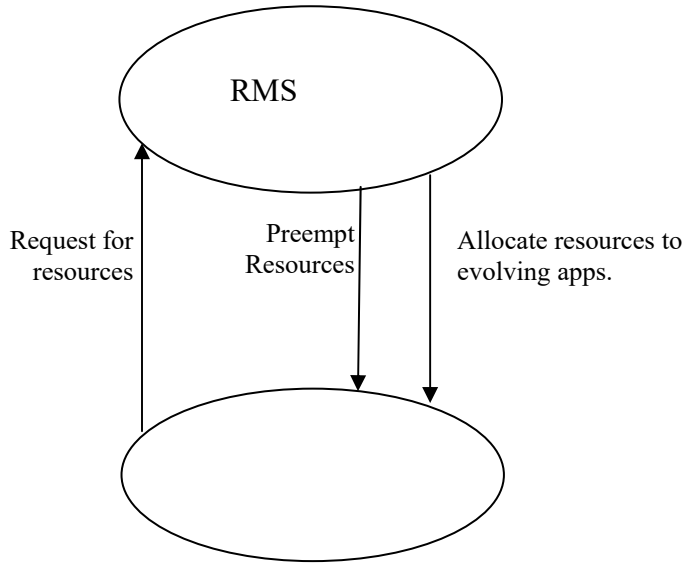


Figure 3.5 Diagram of interactions for the use case "evolving application requesting additional resources"

3.1.2 Requirements of an Adaptive Parallel System

From the use cases described above, it is evident that a complex multi-round, negotiation mechanism between applications and the RMS is required to support a wide variety of parallel adaptive applications. In an adaptive parallel system, multiple evolving applications may request additional resources at the same time, or one request is followed by another in a very short period of time. If enough resources are not available to fulfill the resource requirement of a new application submitted by a user or a request for additional resources by an evolving application, the RMS may have to communicate with one or more malleable applications to preempt resources. The RMS must manage these communications and negotiations efficiently. Clearly, managing negotiations with running adaptive applications is one of the critical requirements of an adaptive RMS. For

the negotiation management, an adaptive RMS is required to perform the following additional functionalities compared to their counterpart in the traditional RMS:

Receive requests from running evolving applications.

Handle multiple simultaneous requests form running evolving applications.

Carry out negotiations with running adaptive applications.

Allocate additional resources to running adaptive applications.

Claim resources from running adaptive applications.

3.2 Conceptual Model of an Adaptive Parallel System

From the use cases described in the previous section, a conceptual model of an adaptive parallel system has been developed. The conceptual model has been developed using a bottom up approach. First a conceptual model of an adaptive application was developed and then gradually the model for other components of the system was developed, and finally a conceptual model of the system was developed. An adaptive parallel system has three main components: applications, RMS and users.

3.2.1 Adaptive Applications

Adaptive applications are capable of expanding by dynamically creating new processes and redistributing data among its processes while executing. They are also capable of shrinking by self reconfiguration, and dynamically destroying processes. In practice, it is possible to create such a parallel application using PVM and MPI, both of which support dynamic process creation and destruction. For expansion or shrinkage, an adaptive application requires negotiation with the RMS to acquire additional resources for expansion or to inform the RMS about the resources it will release.

For expansion or shrinkage, an adaptive application must negotiate resources with RMS. Consequently, adaptive applications must be able to send negotiation requests to the RMS and be able to accept negotiation requests from the RMS. They must also have the intelligence to carry out and conclude a negotiation. The conclusion of negotiation results in an agreement, which the application must execute by consuming or releasing, agreed upon resources. Two parties (the RMS and the application) are involved in a negotiation. In the case of malleable applications, the RMS initiates the negotiation by sending an offer to the application. In the case of evolving applications, the negotiation is initiated by the applications. The negotiation continues by exchanging the offer and counter offer between the applications and RMS. For successful negotiations, both the application and the RMS must communicate with each other according to some agreed upon negotiation protocol.

Once an agreement is reached between an application and the RMS, the application should be able to execute the agreement. The execution of an agreement may involve receiving of a list of allocated resources from the RMS, and reconfiguring itself to make use of the additional resources. Alternatively, the execution of the agreement may involve the application reconfiguring itself, freeing up agreed upon resources, and releasing the free resources to the RMS.

Our model of an adaptive application is based on a master-worker hierarchy employed in many data parallel scientific and engineering application. It consists of a coordinating process which is the master and a set of computing processes which are the workers. The computing processes perform the application specific computation. The coordinating

process distributes the initial workloads among workers, monitors worker loads, and balances loads among workers. In an adaptive application, the coordinating process is also in charge of the resource negotiation with the RMS and execution of the agreement. The coordinating process executes the agreement by reconfiguring the running application to execute on a changed number of resources. Consequently, execution of an adaptive application consists of a number of phases. During each phase the amount of resources the application uses remains unchanged. As a result of the negotiation, the application may reconfigure itself and enter a new phase where it will be running on a different number of resources.

In general, a parallel application may not be able to reconfigure itself at any arbitrary point in time. It may have to wait for a synchronization point where it can reconfigure itself by creating/destroying processes, and/or redistributing data. Consequently there may be a time gap between reaching an agreement, and execution of the agreement by the application. This has a critical implication for the RMS in the case of agreements involving resource release, because the RMS requires allocating the released resources to other running or pending applications. The application should be able to determine when it can release agreed upon resources, and inform the RMS during negotiation, so that the RMS can make informed decisions regarding allocating the released resources.

3.2.2 Resource Management System

In an adaptive parallel system the goal of the Resource Management System (RMS) is to provide support for efficient utilization of computational resources and for

resolving conflicts between interests of various end users. Typically, this goal is achieved by organizing the workload into queues, and by creating a schedule. In order to create a schedule, the RMS may need to negotiate with running adaptive applications to allocate them additional resources or preempt resources from running applications and allocate the released resources to waiting applications. The schedule determines the order in which the applications are executed on computing nodes. In an adaptive RMS if enough resources are not available while computing a schedule, the RMS has the option of preempting resources from running malleable applications. When running evolving applications require additional resources, they ask the RMS for resources and the RMS has to consider these requests while computing a schedule. Alternatively, if idle resources are available and there are no pending applications or pending resource requests from running evolving applications, the idle resources may be allocated to running malleable applications. The RMS functionality includes actual resource allocation (starting the applications), de-allocation (terminating the applications), reallocation (allocating and preempting resources to/from running application), and reporting the applications' status (pending, running or completed) to the system administrator and to the end users. An adaptive RMS has the several components to achieve the above mentioned functionalities. Figure 3.6 shows the component of an adaptive RMS.

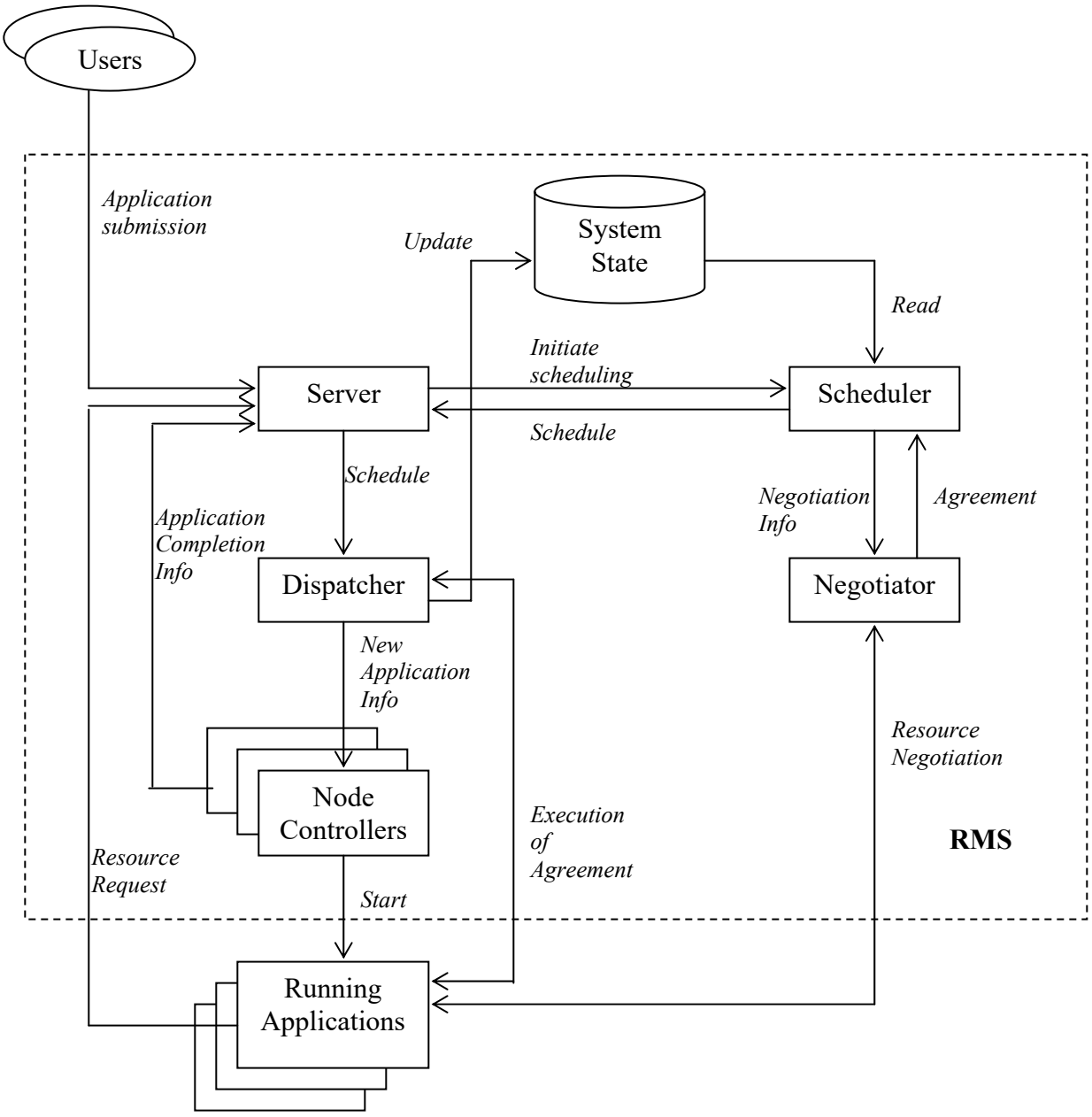


Figure 3.6 Component of an adaptive RMS

3.2.2.1 Server

The server acts as the central coordinator of the RMS. The server is capable of accepting events from other components of the RMS and also events from external

entities such as users, or running applications. In response to an event (internal or external) the server performs some functionality. For performing its functionality the server maintains a system status. The system status consists of a list of pending applications, a list of running applications, resource information, and a list of resource request from running applications.

Users submit applications for execution to the server. The server updates the system state by storing the application's information in the pending list. It receives application completion information from the node controllers, and updates the system status by updating the resource information and the running application list. When a running evolving application requires additional resources, it sends a request to the server. Also when an evolving application has idle resources, it releases idle resources and informs the server. In these events the server updates the system status.

One of the main functionalities of the server is to initiate the scheduling cycle and the server decides when to initiate the scheduler. The server initiates scheduling in response to a scheduling event (application submission, application completion, request for additional resources by running evolving applications, and voluntary resource release by evolving applications). When the scheduler is computing a schedule, if the server receives another scheduling event, the server may queue the event and initiate scheduling later. When the server receives a schedule, it sends the schedule to the dispatcher for execution

3.2.2.2 Node Controller

A node controller manages a computing node and manages applications running on that node. The node controllers act as server agents and start new applications on behalf of the server. The node controller receives application information and resource assignment from the dispatcher, and then launches the application on the assigned resources. When an application completes its execution and terminates, the corresponding node controller sends an application completion notification to the server.

3.2.2.3 Scheduler

The scheduler computes a schedule according to the system policy, the objective function, and the current system state. In an adaptive parallel system the scheduler has to consider the request for additional resources from running evolving applications in addition to the demands of pending applications. If enough resources are not available to meet the demands of pending and running evolving applications, the scheduler has the option of preempting resources (claiming resources without terminating the application) from running malleable applications. Alternatively, if idle resources are available, the scheduler can allocate them to running malleable applications. Consequently, computing a schedule involves deciding which pending applications to execute, selecting evolving and malleable applications to allocate additional resources, selecting preemption candidates among malleable applications, and assigning resources to the selected applications. If a schedule involves allocation or preemption of resources from running adaptive applications, negotiations with those applications are necessary.

The computing of a schedule in an adaptive parallel system may be a multistage process. The scheduler computes an initial schedule according to the system policy and objective function. It negotiates with running applications through the negotiator. Depending on the negotiation outcome, the scheduler may need to re-compute the schedule, which may require further negotiations. This re-computation of schedule and negotiations may go on multiple times until a final schedule is computed. The final schedule contains a list of pending applications to be started and a list of agreements with running malleable and evolving applications.

Some component of the RMS must decide when to compute a schedule and direct the scheduler to compute a schedule, and send information necessary to compute the schedule. In our proposed model, the server decides when to compute a schedule and invokes the scheduler.

3.2.2.4 Negotiator

It is the RMS agent that carries out the negotiation with adaptive applications. The negotiator must know the initial offers that it needs to make to the applications. In addition, it must also know the negotiation policy. The negotiation policy dictates how the negotiation will proceed and converge, when to accept or reject a counter offer by an application, when to terminate a negotiation, and what to do if an application doesn't respond to an offer or counter offer. The negotiator may have to negotiate with multiple applications; it may carry out these negotiations with multiple applications simultaneously, or sequentially one after another. At the end of negotiations, the negotiator sends the outcome of negotiation (list of agreements) back to the scheduler.

3.2.2.5 Dispatcher

Once a schedule has been computed, the server sends the schedule to the dispatcher for execution. The schedule consists of three lists: i) a list of pending applications to be started, each with a list allocated resources; ii) a list of running applications, each with amount of resources to be released; iii) a list of running applications, each with a list of additional resources to be allocated to them. The dispatcher contacts the running applications which are required to release resources and get the list of released resources from applications. The dispatcher communicates with running applications which are required to expand and sends them the list of additional resources allocated to them. It sends information of pending applications to be started to the assigned node controllers, which in turn start application on allocated resources.

3.2.3 Users

Users submit their application for execution on the cluster. The user provides all the information required by the RMS to execute their application. The information includes the executable name, resources required for the application, and how long the resources are required. Users may also provide additional application constraints such as a deadline by which the application must complete execution, the minimum wall clock time after which the application must start, or the minimum and maximum number of processors that the application can utilize etc. From the RMS points of view, the applications submitted by the users are independent of each other. A set of applications submitted by different users over a certain period of time is called *workload*.

3.2.4 Relationship between the Components

The conceptual model depicting the components of an adaptive parallel system and their association is shown in Figure 3.7. The association between the components in the conceptual model can be *one to one* (i.e. at a particular point in time one instance of a component interacts with exactly one instance of another component), *one to many* (one instance of a component interacts with one or more instances of another component), or *many to one* (one or more instances of a component interacts with exactly one instance of another component).

Multiple users may submit applications for execution to the servers at a discrete point in time. As a result, the association between users and server is many to one. There are multiple node controllers in the system, each managing a single node, and the association between the server and the node controller is one to many. One or more running evolving applications may send requests for additional resources to the server at a particular moment in time. The association between running applications and the server is many to one.

Figure 3.8 shows the collaboration diagram of the conceptual model for an adaptive parallel system. Each object in the model performs some functions and interacts with other objects by exchanging messages. Objects have one or more inputs from other objects and have one or more outputs to other objects. To perform its functions, an object may be required to maintain some internal information apart from the inputs from other objects. Objects perform well-defined functions in response to inputs from other objects.

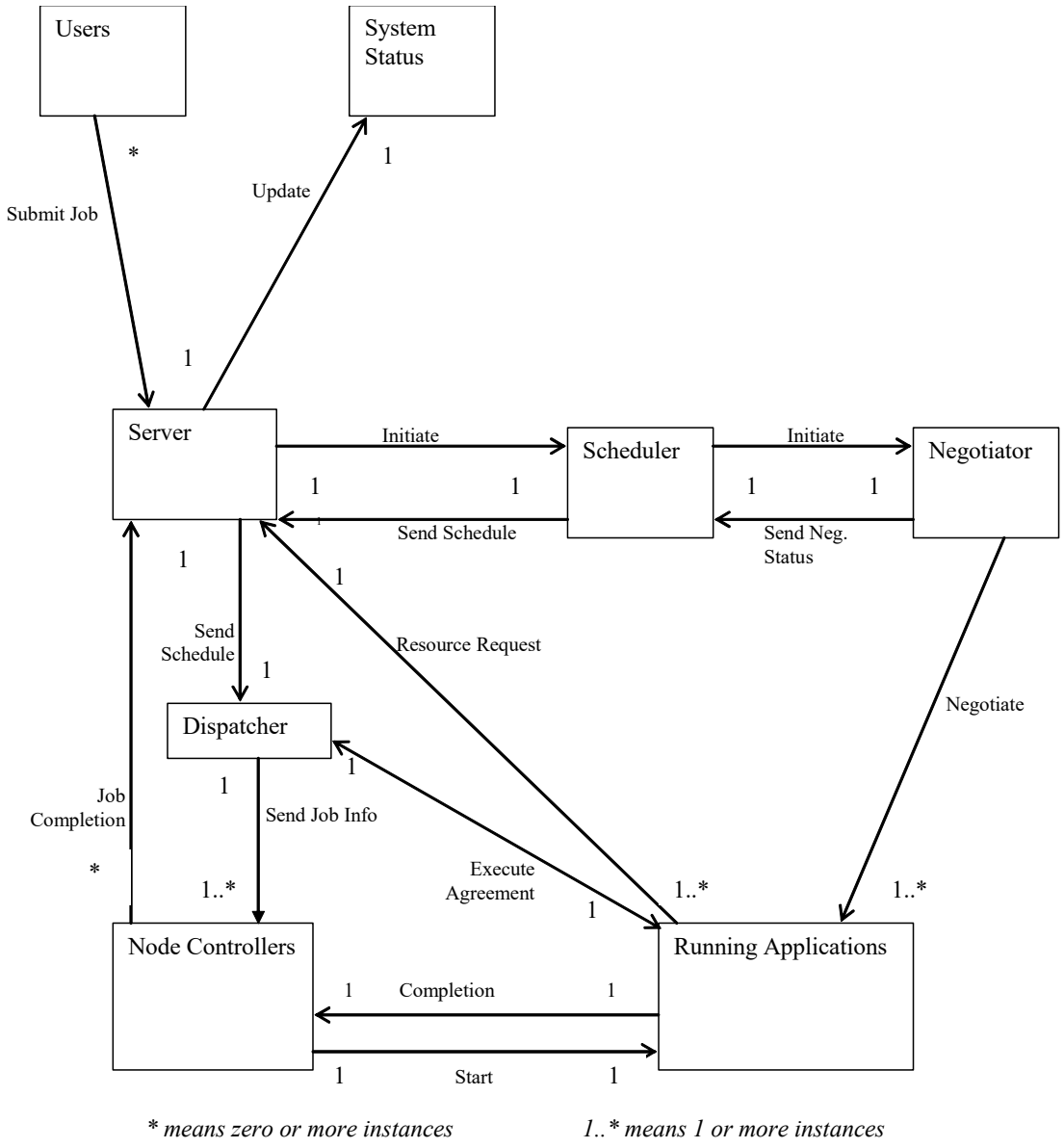


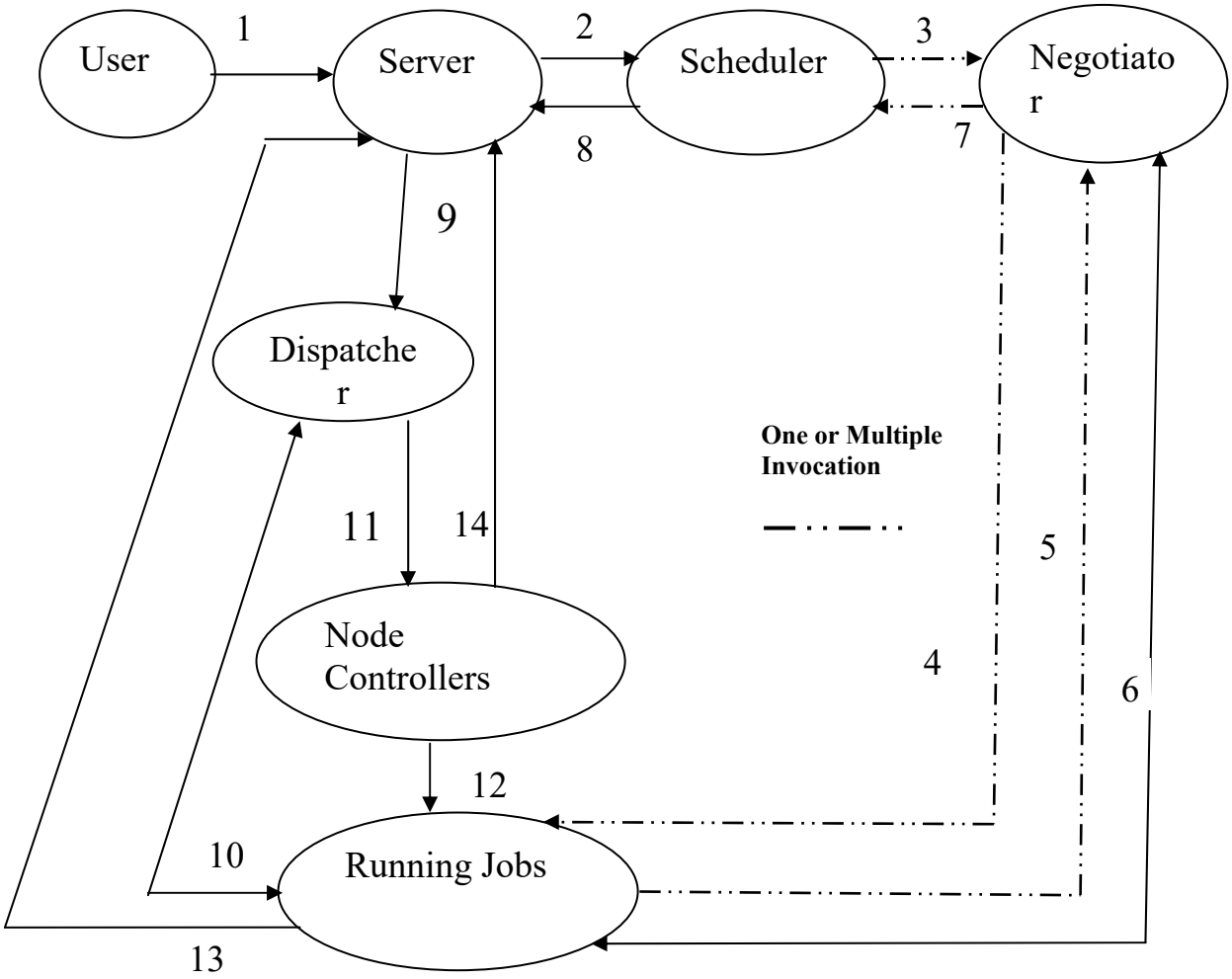
Figure 3.7 Conceptual model of an adaptive parallel system

The model can be considered as an event driven model. An input is an event, and an object performs a certain function in response to an event and may generate one or more events for other objects. Some events such as submission of a job, completion of job, request for negotiation, and release of resource by an evolving job, can occur at any

point in time. Some events occur in response to other events or execution of some functionality. The server, the scheduler, the negotiator, the dispatcher, and the node controllers represent the RMS. The objects user and running jobs are external to the RMS.

Table 3.1 shows the object relationship and information flow among objects. The first row of the table lists the functionalities that an object performs. The second row shows the information required to perform the functionalities. Each entry in the cells after the second row shows the information exchange between the objects in the column heading and the row heading of the cell. The cell entry is the output of the row object to the column object. An empty cell indicates the information exchange between the objects of the column and those of the row.

After the second row, the contents of a row show the output of an object in the row heading, and contents of the columns show the inputs to the object in the column heading.



Interactions between objects

1. Submit Job (Job info)
2. Schedule (System State)
3. Negotiate (proposal)
4. Offer/Counter offer
5. Counter offer
6. Accept/Reject Notification
7. Negotiation outcome (Agreements)
8. Execute (schedule)
9. Execute schedule
10. Execute (Agreement)
11. Start New Job (Job execution info)
12. Launch Job
13. Request for negotiation
14. Complete job (Job completion info)

Figure 3.8 Collaboration diagram of the conceptual model for an adaptive parallel system

Table 3.1 Object relationship and information flow

	User	**Server**	**Scheduler**	**Negotiator**	**Node Controller**	**Running Malleable**	**Running Evolving**
Functional-ity	Submit job	Initiate Scheduling Update System status Execute Schedule	Compute Schedule	Negotiate with malleable and evolving jobs	Start new job	Negotiate Execute agreement.	Negotiate Execute agreement. Request for resource Release idle resource
Info. Required		System State ***New job info*** ***Job completion info*** ***Resource request*** ***Resource release*** ***Schedule***	System state Agreement. List Policy Objective function	Neg. Info Neg. Policy Offer	Job info Allocation info.	Offer Agreement.	Offer Agreement.
User		New Job info.					
Server			System State		Job execution Info	Agreement.	Agreement.
Scheduler		Schedule		Neg. Info			
Negotiator			Agreement List			Offer	Offer
Node Controller		Job Completion Info					
Running Malleable		Released resource		Offer			
Running Evolving		Neg. Request Released resource		Offer			

3.3 Mathematical Model

The mathematical model describes the conceptual model in terms of variables and equations. The goal of the mathematical model is to provide qualitative and quantitative information on system and application performance for a given adaptive parallel system

(a given workload and a RMS). While developing the mathematical model, we need to represent the objects, procedures, and the information flow described in the conceptual model presented in section 3.5 in terms of variables and equations. As described in the previous sections, an adaptive parallel system is a very complex system. In this dissertation we have attempted to model and simulate an adaptive system with rigid and malleable applications only. Once a model for an adaptive system with malleable applications is developed and validated, it can be enhanced to incorporate evolving applications in the future. Before developing the mathematical model, we will define the assumptions that have been made about the system in section 3.3.1.

3.3.1 Assumptions

For developing the mathematical model, and subsequently to develop the simulator, we made the following assumptions about an adaptive parallel system.

1. All applications in the workload are parallel applications.
2. Workload consists of rigid and malleable applications only.
3. Sequential computation is negligible compared to parallel computations.
4. The computation is linearly distributed over time.
5. Only processors are considered as resources.
6. The negotiator performs negotiations sequentially one after another.
7. Malleable applications perform negotiation in a non blocking manner. That is while the coordinating process is engaged in negotiation, the computing processes can continue with computation
8. Adaptation cost varies from application to application.

9. Adaptation cost is proportional to the change in the number of processors.
10. Adaptation cost does not vary from shrinkage to expansion.

3.3.2 Workload

There are two trends in generating the workloads among the researchers working in the resource management and scheduling communities. One trend is to use actual workload data derived from job traces from supercomputer centers [50][57]. The other trend is to generate workloads from workload models [51][52]. As discussed in Chapter II, using a workload model has advantages over an actual workload derived from job traces.

For the simulation experiments presented in chapter VI, we have used the workload data generated from a workload model. There are several validated workload models available for rigid applications [52][58][59][60][61]. Currently none of the models available are able to generate a workload containing rigid as well as malleable applications. These models provide arrival time, execution time, and number of processors required for a rigid workload. Instead of developing a model from scratch we have decided to adopt one of the validated models for rigid application and extend it to accommodate malleable applications.

For simulation experiments in this research we need realistic workload data. The reasons for the need of realistic workload data are as follows. i) To validate the simulator and consequently the model. If the model is validated against realistic data we have more confidence about the models capability of approximating a real adaptive parallel system. ii) If realistic data is used to investigate the impact of model parameter of system

performance, then we will have more confidence on the experimental results and subsequent knowledge gained from the experiments.

In this research we have selected the workload model by Allen B. Downey [52] for modification to generate a malleable workload. There are three primary reasons for selecting Downy's model. First, Downey's model has been validated against workload logs from San Diego Super computer center (SDSC) and Cornell Theory Centers (CTC), and it can generate realistic data. Second, the model has been used by several researchers to generate workloads for their experiments. Third, an open source implementation of Downey's model is available which makes it easier to extend and modify.

Downy's model takes the maximum number of processors that an application can have, the minimum and maximum runtime of applications in the workload, and the number of jobs to be generated as input. The model provides the arrival time, the number of processors required, and the run time for each application in the workload. A model for malleable application must able to generate a workload with rigid as well as malleable applications. To accommodate malleable applications the existing Downy's model required three modifications: decide how many of the applications should be malleable, what should be the distribution of malleable applications in the workload, and what will be the flexibility range (minimum and maximum processors) of each malleable application. The modified model has four additional parameters to convert the rigid applications generated by Downey's model into malleable applications. The parameters are the number of malleable applications as percentage of total number of applications in

the workload, the distribution of malleable applications in the workload, and the minimum and maximum number of processors that a malleable application can have.

3.3.3 Application

The execution of a malleable application consists of phases. Within a phase, the number of processors used by the application remains unchanged. The coordinating process of the application carries out negotiation with the RMS and enters into an agreement which involves either releasing some processors or receiving some additional processors. The application executes the agreement by reconfiguring itself. The reconfiguration involves redistribution of data, and the consumption or the release of processors. As a result of reconfiguration, the application enters into a new phase of execution. There is an additional cost of reconfiguration at each phase change which is called *adaptation cost*. The adaptation costs bring overhead to running malleable applications.

Let the total computation of a malleable application be W and it takes t_d time to complete the computation on p_d processors. Then:

$$W = p_d \times t_d \quad \text{.....(3.5)}$$

Let the application consist of n phases and uses $p_1, p_2, \ldots p_n$ processors in phases $1, 2, \ldots n$ respectively.

Let the execution times of phases $1, 2, .. n$ be $t_1, t_2, \ldots t_n$, respectively

Let the adaptation costs for the phases be $c_{a1}, c_{a2}, \ldots c_{an}$, respectively.

Then, the remaining computation after phase 1 is:

$$Wr_1 = W - W_1 \text{ .....(3.6)}$$

Where $\boldsymbol{W_1}$ is the computation completed in phase 1

$$Wr_1 = p_d \times t_d - p_1 \times t_1 \textbf{ .....(3.7)}$$

If there is no phase change after phase 1, then the time required to complete the remaining computation $\boldsymbol{Wr_1}$ after phase 1 is:

$$tr_1 = \frac{Wr_1}{p_2} \text{ .....(3.8)}$$

or $$tr_1 = \frac{(p_d \times t_d - t_1 \times p_1)}{p_2}$$

If the application consists of three phases only then the remaining computation after phase 2 is

$$Wr_2 = p_d \times t_d - p_1 \times t_1 - p_2 \times t_2$$

or $Wr_2 = Wr_1 - p_2 \times t_2$ **.....(3.9)**

And the time required to complete the remaining computation $\boldsymbol{Wr_1}$ after phase 2 is

$$tr_2 = \frac{Wr_2}{p_3} \text{ .....(3.10)}$$

or $$tr_2 = \frac{(p_d \times t_d - t_1 \times p_1 - t_2 \times p_2)}{p_3}$$

In general, after the ith phase remaining computation

$$Wr_i = p_d \times t_d - \sum_{i=1}^{m} p_i \times t_i \text{ .....(3.11)}$$

or $Wr_i = Wr_{i-1} - p_i \times t_i$ **.....(3.12)**

And the time required to complete the remaining computation $\boldsymbol{Wr_i}$ after phase $\boldsymbol{i}$ is

$$tr_i = \frac{Wr_i}{p_{i+1}} \text{ ……..(3.13)}$$

$$tr_i = \frac{Wr_{i-1} - p_i \times t_i}{p_{i+1}} \text{ ……(3.14)}$$

If the application consists of $\boldsymbol{n}$ phase the execution time of the application is

$$t = (t_1 + C_{a1}) + (t_2 + C_{a2}) + \cdots\cdots + (t_{n-1} + C_{an-1}) + \frac{Wr_{n-1}}{p_n} \textbf{ …. (3.15)}$$

3.3.4 Negotiation

If the workload consists of malleable and rigid applications only, all negotiations are initiated by the RMS. The negotiation cost has two components: the communication time required to send and offer or counteroffer from the RMS to the application and vice versa, and the time required by the application or RMS to make a decision. In a cluster environment where load is controlled, it is reasonable to assume that the communication cost between the RMS and the application do not vary from one negotiation to another. Since the RMS knows the global system state and it follows a specific system policy, we assumed that the decision making time for the RMS does not vary from negotiation to negotiation or from application to application. However, since each application is different, the time required to respond to an offer from the RMS may vary from application to application. For most applications, the response time is low and for few applications it may be high. In addition, the number of rounds in each negotiation may vary from application to application and negotiation to negotiation.

From the above discussion it is apparent that the negotiation cost depends on the communication time, the response time and the number of rounds required to conclude a negotiation. We have modeled the negotiation cost by aggregating these factors into a single parameter C_n. We have chosen two models for the variation of the negotiation cost. In the simpler model the variation is zero. That is, for a given workload, the negotiation cost is constant. In the other model, negotiation cost varies from application to application and negotiation to negotiation, and it is low for most applications and it maybe high for a few applications. In this model the negotiation costs follow a random ramp distribution. The probability decreases linearly as the negotiation cost increases.

3.3.5 Adaptation Cost

An application executes an agreement by shrinking or expanding. Shrinking involves redistributing data, destroying processes and releasing idle processors. Expanding involves spawning processes on the additional allocated processors and redistributing data. The cost of data redistribution depends on application's business logic and data structures. As a result, it varies from application to application. In addition, for the same application the redistribution cost depends on the variation of the number of processors that the application is using before and after adaptation. We have assumed that for a given application, the adaptation cost is linearly proportional to the change in number of processor.

We have modeled the adaptation cost of an application for change in one processor by single parameter ***C***. After execution of an agreement if an application changes the number of processors from $\boldsymbol{p_1}$ to $\boldsymbol{p_2}$ the adaptation cost is

$$C_a = C \times |p_1 - p_2| \quad \text{.................................. (3.16)}$$

Like the negotiation cost, the adaptation cost is chosen in two ways to model the variation of adaptation cost per processor from application to application. In the simpler model the variation is zero. Therefore, for a given workload the cost is constant. In the other model, the adaptation cost varies from application to application, and it is low for most applications and high for a few applications. Parallel applications which have low adaptation cost are most amenable to conversion into malleable application; consequently, it is reasonable to assume that for most applications the adaptation cost will be low. For the adaptation cost we have adopted the same distribution as negotiation cost.

3.3.6 Performance Metrics

The average turn around time has been selected as a metric for measuring application performance. Utilization has been selected as a metric for measuring system performance. The turn around time (TAT) for an application is defined as the time taken from submission of an application to the system until its completion of execution. The average turn around time is defined as the arithmetic mean of the turn around time of all applications in the workload. The schedule span is defined as the time between the arrival of the first application in the system and the completion of the last application for a given

workload. Utilization is defined as the fraction of total available CPU cycles during the schedule span that have been used by all the applications in the workload.

Let us assume that for a given workload ***W*** the

Total number of applications is: m
Arrival time of application ***i*** is t_a^i
Start time of application ***i*** is: t_s^i
Completion time of application ***i*** is: t_c^i
Execution time of application ***i*** is $t_{ex}^i = t_c^i - t_s^i$
Arrival time of first application is: t_a^1
Completion time of last application is: t_c^m
Total number of processors in the system is: p

Then,

The waiting time for application ***i*** is: $t_w = (t_s^i - t_a^i)$

The turn around time for application ***i*** is: $TAT = (t_c^i - t_a^i)$ ……..(3.1)

The average turn around time for workload ***W*** is

$$\text{<Avg. } TAT\text{>} = 1/m \sum_{i=1}^{m} (t_c^i - t_a^i) \quad \text{……(3.2)}$$

The schedule span is:

$$ss = (t_c^m - t_a^1) \quad \text{……(3.3)}$$

The cost of an application is defined as total the CPU time consumed by an application during its execution. For example, if a rigid application runs on 4 processors for 100 seconds the cost of the application is 4 x 100 = 400 seconds, or if a malleable application runs on 4 processors for 100 seconds, and then on 8 processors for 100 seconds, then the cost of the application is 4 x 100 + 8 x 100 = 1200 seconds.

Let the cost of an application i be C_{app}^i

The total processor cycle available during the schedule span is ***(p*ss)***.

Therefore the utilization is: $$U = \sum_{i=1}^{m} C^{i}_{app} / (p^{*}ss) \quad \ldots\ldots\ldots\ldots\ldots\ldots (3.4)$$

To compute performance metric the arrival time, the start time, and the completion time is needed for each application in a workload. Numerical simulation of the mathematical model described in the previous sections will provide these values. The workload model provides the arrival time. Scheduling algorithm and negotiation model provide the start time. Application model and adaptation model provide the completion time. From the numerical simulation the performance metric can be computed for a workload.

3.4 Summary

We have described an adaptive parallel system and a conceptual model for an adaptive parallel system. We have developed a semi-formal mathematical model of an adaptive parallel system with malleable applications which is numerically simulated with a discrete event simulator presented in chapter IV. Utilization and average turn around time has been selected as measure of performance for RMS and application respectively.

A malleable application consists of phases. During a phase, a negotiation occurs between a running malleable application and the RMS. If there is an agreement between the application and the RMS, then at the phase boundary the application adapts and change the number of processors utilized. To calculate the performance metric for a given workload, we need to know the arrival time, the starting time and the completion time of each application in the workload. The workload generated from the workload model described in section 3.4.3 provides the arrival time, processor requirements, and total

computation of each application. The numerical simulation of a workload provides the start time for each application. During a simulation the remaining computation time after a phase change can be predicted using equation 3.14. During a simulation the scheduler determines whether a running application will go through a phase change.

Chapter IV describes a discrete event simulator developed for numerical simulation of an adaptive parallel system with malleable application. A prototype implementation of an adaptive RMS capable of handing malleable application has also been presented in chapter IV.

CHAPTER IV

SIMULATOR FOR AN ADAPTIVE PARALLEL SYSTEM

In Chapter III a model for an adaptive parallel system has been presented. In this chapter, we present the design and implementation of a discrete event simulator of the model discussed in chapter III. This chapter also presents the implementation of a prototype system for an adaptive parallel system. The experimental results with the prototype system are discussed in this chapter.

Simulation is one of the most widely used scientific techniques for studying a system [56]. A system is defined to be a collection of entities, e.g. people, machine etc. that act and interact together towards the accomplishment of a goal [64]. The entities of the system are: the users, the resource management system, and the running applications. An adaptive parallel system can be considered as a discrete event system because events such as the submission of an application or the completion of an application occur at discrete points in time. As a result, the state of the system changes at discrete points in time.

4.1 Discrete Event Simulator

Although discrete event simulators are used to simulate different type of real world system, they all share a number of common components and logical

organization[56]. The following components are present in most discrete event simulators.

System State: A collection of variables called state variables which describe the state of a system at a particular time.

Simulation Clock: Because of the dynamic nature of the simulation, a mechanism keeping track of simulation time is required, and a mechanism is required to advance the time as the simulation progresses. A variable that contains the current value of simulation time is used to keep track of time. There are two main mechanisms of advancing the simulation clock: the fixed-increment time advance and the next-event time advance. In fixed-increment time advance, the simulation clock is advanced by a fixed value Δt. After each increment, a check is made whether any events have occurred during the previous interval Δt. If any events have occurred, they are processed and the system state is updated accordingly. In next-event time advance approach the simulation clock is advanced to the time of occurrence of the next event, and the event is processed.

Event List: A list containing the events and time when they will occur. The list is usually sorted in increasing order of event time.

Statistical Counter: Contains variables used for storing information about system performance.

Initialization Routines: Subprograms to initialize the simulation model at time 0.

Timing Routine: A subprogram that determines the next event time and advances the simulation clock to the next event time.

Event Routines: Subprograms that process events and update the system state when an event occurs. In general there is at least one subprogram for each type of events.

Report Generator: Subprograms that computes estimates of the desired performance measures at the end of simulation.

Main Program: The subprogram that coordinates the simulation. It is also known as the simulation executive. The main program invokes the timing routine to determine the next event time, and it invokes the corresponding event routine to update the system state.

The flow of control and relationship among the components are shown in figure 4.1. At time t = 0 the main program invokes the initialization routine, which sets the simulation clock to zero, initializes the state variables and creates the event list. The initialization of state variables and the event list may involve reading simulation inputs from a file. After the control is returned to the main program, it invokes the timing routine to determine the next event type, and then it invokes the appropriate event routine to process that event. Event routines generally perform three types of activities, i) updating the state variables, ii) updating the statistical counter, and iii) generating future events and inserting them into the event list. The invoking of the timing routine and of the event routine is repeated until all the events in the event list is processed. The main program then calls the report generator to compute the performance indicator and generate a report.

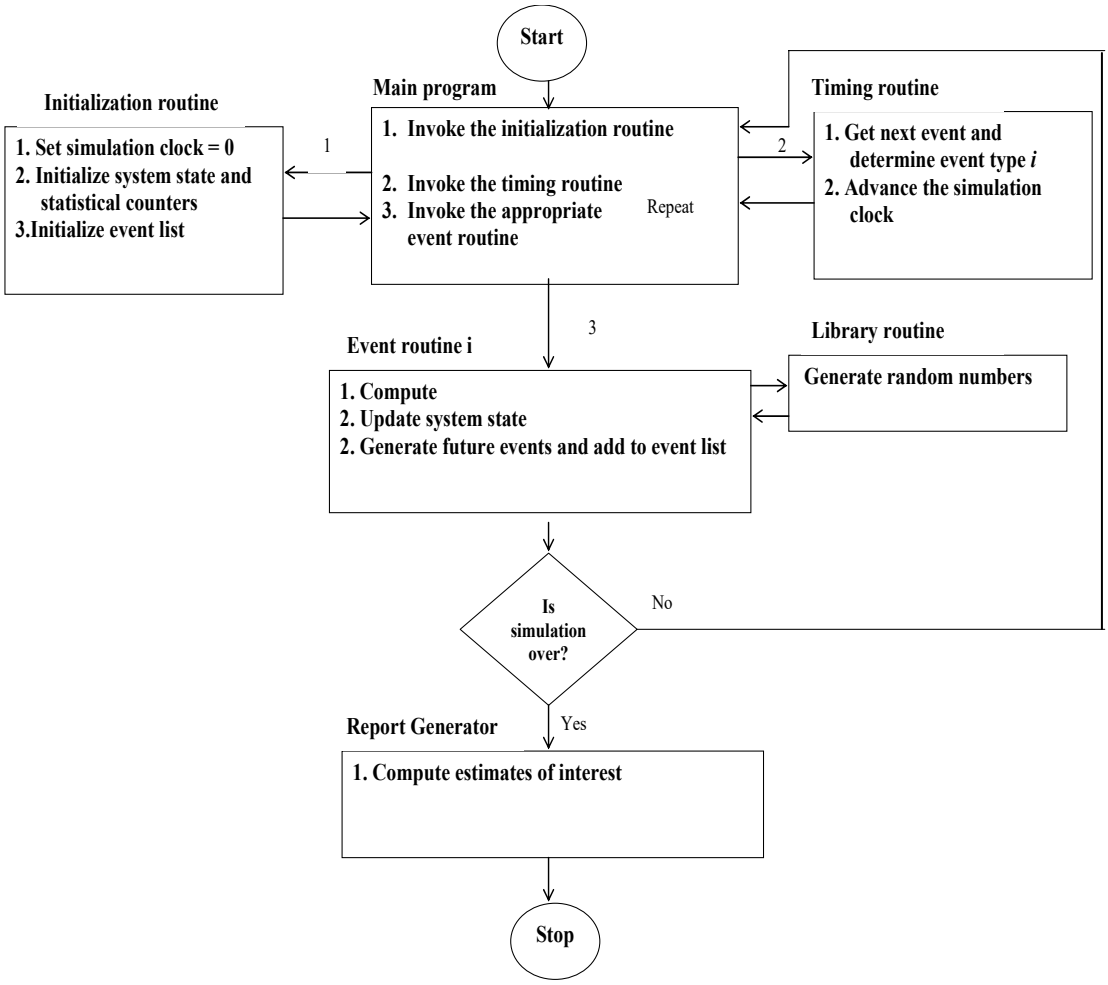


Figure 4.1 Flow control of discrete event simulator

4.2 Simulator for Adaptive Parallel System

A simulator for the numerical simulation of an adaptive parallel system such as the one described in chapter III has been developed from scratch. We have examined a few open source off the shelf simulators. None of them were found to be suitable for our simulation purpose. Modifying them to adapt for our purposes seemed to be more work than developing a new one from scratch. Figure 4.2 shows the logical organization of the simulator. The simulator is composed of the following major modules: the executive, the initialization routine, the scheduler, the negotiator, and the dispatcher. In addition, the

simulator uses several data structures which constitute the system state. The following subsections briefly describe the modules of the simulator.

4.2.1 System State

The system state contains the several data structures representing the state of the adaptive parallel system during a simulation. The system state contains the following data structures.

Event List: This list contains the events that will occur during the simulation. Each entry in the list represents an event. Each event has a type, time of occurrence, and the *job id* with which the event is associated. Our simulator handles only two types of events: a job submission event and a job completion event. The event list is created by the initialization routine. The event list is updated by the executive and the dispatcher. The event list is sorted in increasing order of event time.

Pending Job List: This list contains information about all the applications in a given workload. The list is sorted in order of job arrival time. The initialization routine creates the pending job list. Each entry of the list contains the following information: the job id, the job type (rigid/malleable), the arrival time, the default processor requirement, the execution time on the default number of processors, the minimum and the maximum processor that the job can utilize.

Running Job List: This list contains information about all the jobs that are currently running. In addition to the information of pending job, each entry in the list contains the following information: the start time, the number of processor currently allocated, the amount of work remaining after the last phase change, the time when last phase change has occurred. The list is created and updated by the dispatcher.

Job Completion List: This list contains information about all the jobs that have completed their execution. In addition to the information present in an entry in the running job list, each entry in the list contains the completion time. The list is created by process completion module. The performance metrics are calculated from this list.

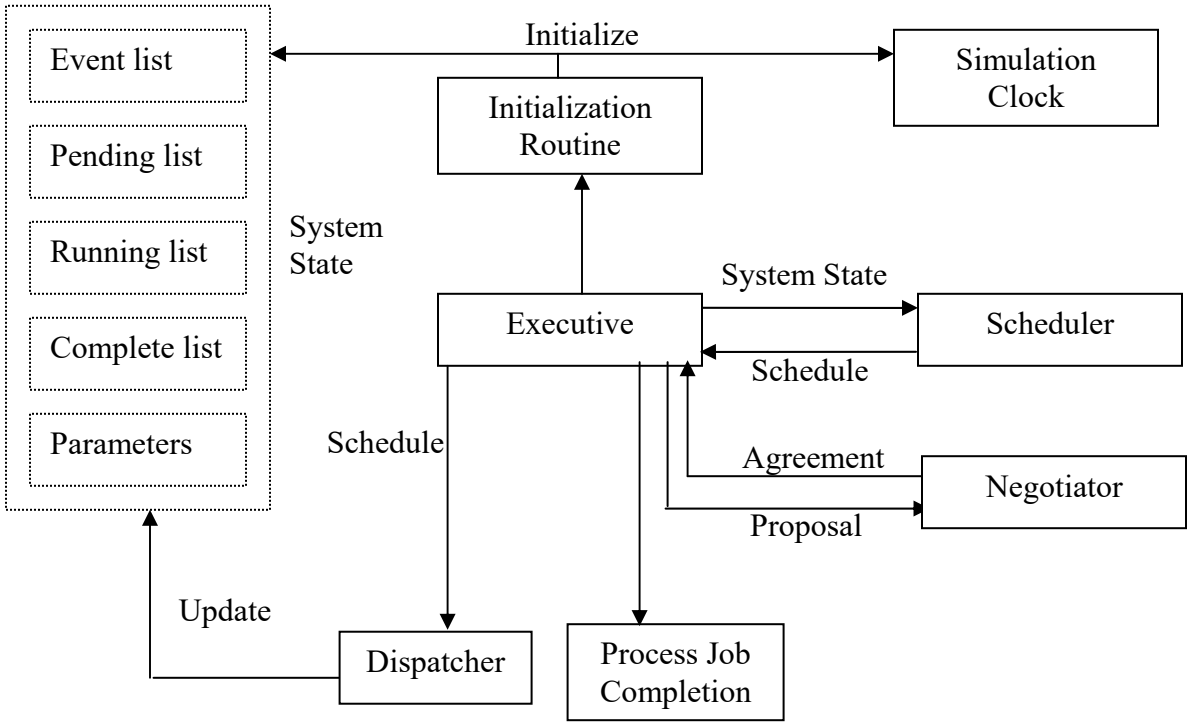


Figure 4.2 Organization of the simulator for an adaptive parallel system

System Parameters: The parameters are: the minimum negotiation time, the maximum negotiation time, the percentage of negotiation success rate, the minimum adaptation cost, the maximum adaptation cost, the total number of processors in the cluster, the name of the workload file, and the name of the statistic file. These values are initialized by initialization routine.

4.2.2 Executive

The executive is the coordinator of the simulator. It coordinates the other modules and drives the simulation. The algorithm and flow diagram for the executive are show in Figure 4.3 and Figure 4.4. The coordinator first calls the initialization routine which initialized the system state. It then enters into a loop. Inside the loop it removes the next event from the event list and processes the event by invoking other modules. While

processing an event, the executive updates the simulation clock and the system state. This process continues until the event list is empty and the simulation ends.

```
Input:    None
Output:  A file containing the statistics of the simulation run
1. Initialize
2. while event list not empty
3.        Remove next event from event list
4.        if event = job completion
5.                move completed job from running list to completion list
6.                update system state
7.        if event time <= simulation clock
8.                go to step 3.
9.        simulation clock ← event time
10.      schedule ← scheduler(System State)
11.      agreement ← negotiator(proposed agreement)
12.      update schedule
13.      simulation clock = simulation clock + negotiation overhead
14.      dispatcher(schedule)
15. end while
16. calculate statistics
17. write statistics to output file
```

Figure 4.3 Algorithm for the simulation executive

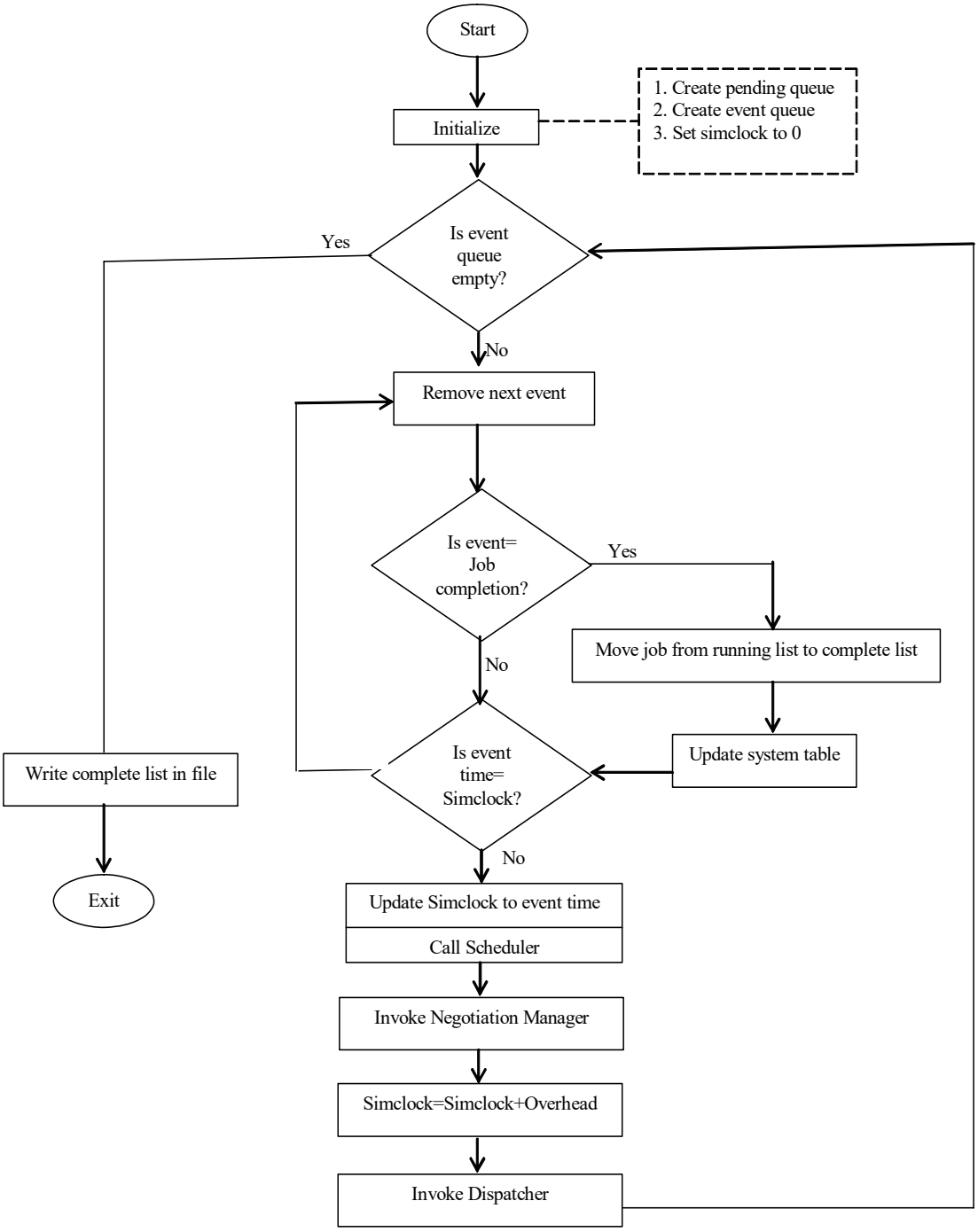


Figure 4.4 Flow diagram for the simulation executive

4.2.3 Initialization Routine

The initialization routine is invoked by the executive. It reads the workload and parameter files and initializes the system state. It also sets the simulation clock value to zero. Figure 4.5 show the algorithm for initialization routine.

```
Input:    workload file, system parameter file
Output:  Updated system state.
1.  Read workload file
2.  Create pending job list
3.  for each job in the pending list
4.        create an event
5.        event type ← job submission
6.        event time ← arrival time
7.        id ← job id
8.        insert event in event list
9. end if
10. sort event list on event time
11. sort pending list on arrival time
12. read system parameter from parameter file
13.  simulation clock ← 0
14. end
```

Figure 4.5 Algorithm for the initialization routine

4.2.4 Scheduler

The scheduler module is invoked by the executive. The scheduler creates a schedule and returns the schedule to the executive. A schedule consists of two data structures: a jobs to start list and a negotiation proposal list. Each entry in the jobs to start list contains the job id of a pending job and the number of processors allocated to it. Each entry in the proposal list contains the id of a running malleable job, an indicator whether the job needs to shrink or expand and the number of processors the job needs to release or receive. The algorithm and flow diagram for scheduling are shown in Figure 4.6 and Figure 4.7. The scheduler creates the schedule according to the following policy.

First Come First Serve: Pending jobs are scheduled according to the arrival time.

Maximum fit: Schedule as many job as possible in a scheduling cycle. This is done by allocating the minimum number of processor for pending malleable jobs.

Pending Job Priority: Pending jobs are given priority over malleable jobs. If idle processors are available the scheduler tries to schedule pending jobs first. If enough idle processors are not available to schedule pending jobs the scheduler tries to preempt the required number of processors from running malleable jobs. If idle processors are available after scheduling pending jobs, or there are no pending jobs then only the idle processors are allocated among running malleable jobs.

Shrinkage and Expansion: Preempt processors starting with the running malleable job which started earliest, then from the next one and so on. Preempt the maximum possible number of processors from jobs starting with the first. This policy reduces the number of negotiations required. For example, assume eight additional processors are required to schedule a pending job and there are four running malleable jobs each of which can release five processors. The scheduler will decide to preempt five processors from the first job and three from the second job, which will require negotiation with two applications, as opposed to preempting two processors from each job, which will require negotiation with four jobs. Allocation of idle processors among running jobs adopts the same policy.

The jobs to start list is created in two steps. In the first step, an initial list is computed according to the maximum fit policy. If after the first step idle processors are available, they are allocated to the jobs in the initial list on a FCFS basis in the second step. For example, assume that there are ten idle processors and three pending malleable jobs. Each of the malleable jobs can run on any number of processors between 4 and 12. In the first step, job one and jobs two will be allocated 4 processors each following the maximum fit policy. Job three cannot be included in the list because it requires a minimum of four processors and there only two processors available. Now in the second stage, the two remaining processors will be allocated to job one. The final jobs to start list will contain job one with six processors and job two with 4 processors.

If idle processors are available after refining the initial list, and there are pending jobs and running malleable jobs, then the scheduler tries to schedule pending jobs by preempting processors from running malleable jobs. The scheduler first computes the maximum number of processors that can be preempted from running malleable jobs. The scheduler then tries to allocate processors to pending jobs according to the maximum fit policy, assuming that the preemptable processors are available. If idle processors are available after refining the initial list, and pending jobs cannot be scheduled by preempting processors from running jobs, then the idle processors are allocated to running malleable jobs on a first start first get basis. The scheduler does not actually preempt processors from running jobs. It just makes the decision on which pending jobs to start, which running jobs to shrink or expand. In other words it proposes a schedule but does not execute a schedule.

```
Input:    System State
Output: Schedule
1.  create initial schedule
2.  Refine schedule
3.  if idle processor > 0 and running malleable > 0
4.       if  no. of pending job > 0
5.                calculate preemptable processors
5.                schedule pending job
6.                create proposal for processor preemption
7.       end if
8.       create proposal for processor allocation to running job
9.  end if
10. return schedule
```

Figure 4.6 Algorithm for the scheduler

4.2.5 Negotiator

The module negotiator simulates the negotiation between the RMS and the applications. The negotiator is invoked by the executive. The input to the negotiator is a proposal list and the output is an agreement list and the negotiation overhead. Each entry in the proposal list contains a job id, a shrink or expansion indicator, the number of processors to be released or received by the running malleable jobs, and a negotiation status field. The algorithm and flow diagram for the negotiator are shown in Figure 4.8 and Figure 4.9. The negotiation is carried out sequentially one after another. At the beginning of the negotiation cycle, the negotiation overhead is set to zero. The negotiation overhead is the sum of costs of all negotiations carried out in a negotiation cycle. To determine the negotiation cost, the negotiator invokes a routine with maximum and minimum negotiation costs as parameters. The routine stochastically selects the

negotiation cost from a random ramp distribution. Whether the negotiation succeeded or failed is determined stochastically. In case of shrinkage the agreed number of processor is determined by randomly selecting an integer between 0 and the maximum number of processors that the job can release. In case of expansion the agreed number of processor is determined by randomly selecting an integer between 0 and the maximum number of additional processors that the job can consume.

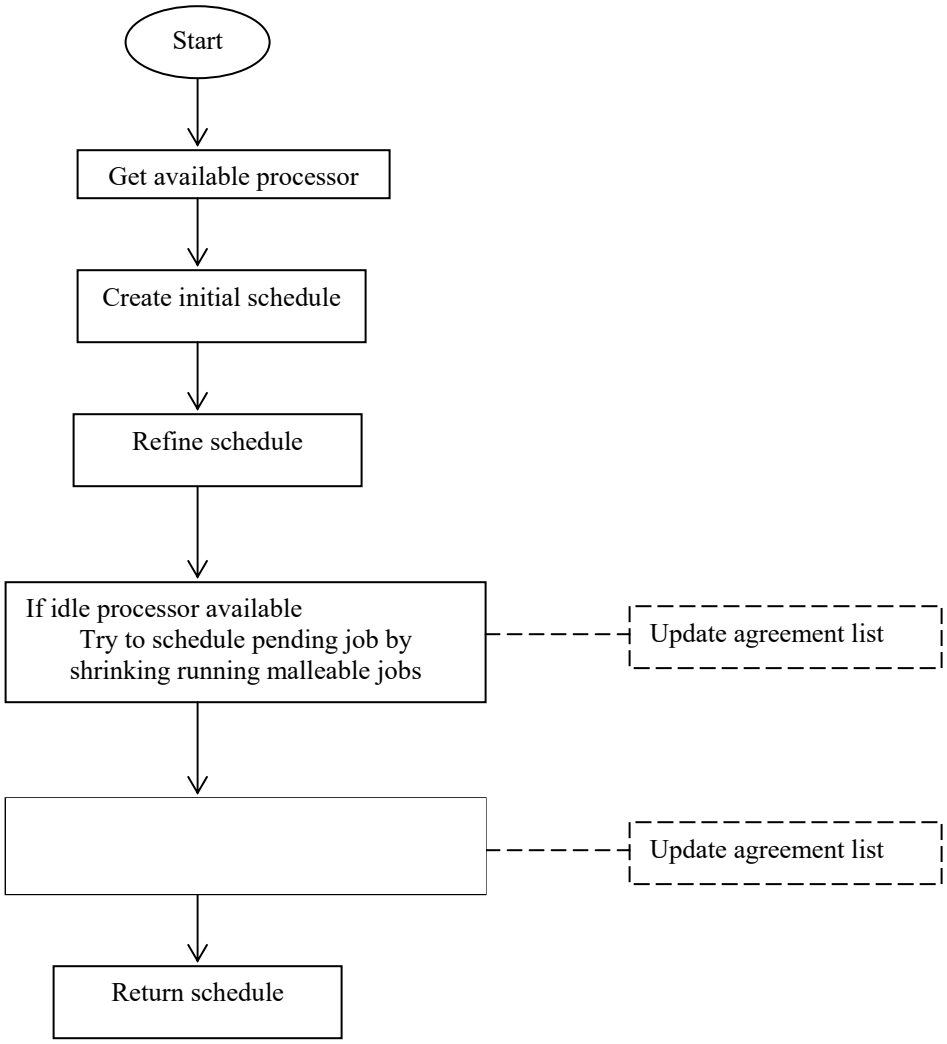


Figure 4.7 Flow diagram for the Scheduler

```
Input:    Proposal list
Output: Agreement
1.  overhead ← 0
2.  while proposal list not empty
3.        get next proposal
4.        overhead ← overhead + get negotiation cost(min. neg. cost, max.
neg. cost);
5.        negotiation status ← get negotiation status(success rate)
6.        processor ← get negotiated processor no()
7.        update agreement list
8.  end while
6.  return negotiation overhead and agreement list
```

Figure 4.8 Algorithm for the negotiator

4.2.6 Dispatcher

The dispatcher executes a schedule. The algorithm and flow diagram for the dispatcher are shown in Figure 4.10 and Figure 4.11. The schedule consists of two lists: the jobs to start list and the agreement list. The executive invokes the dispatcher by passing the schedule. The dispatcher first executes the agreements in the agreement list. It removes the next agreement from the agreement list. It computes the remaining computation Wr_i of the application after adaptation according to equation 3.12. It computes the time required to finish the remaining computation tr_i according equation 3.13. The dispatcher then stochastically determines the cost of adaptation by invoking the get adaptation routine. It then computes the new completion time of the running malleable jobs. It updates the event list by changing the corresponding event time to new

completion time. The dispatcher also updates the current allocated processors in the running job list and the number of idle processors available.

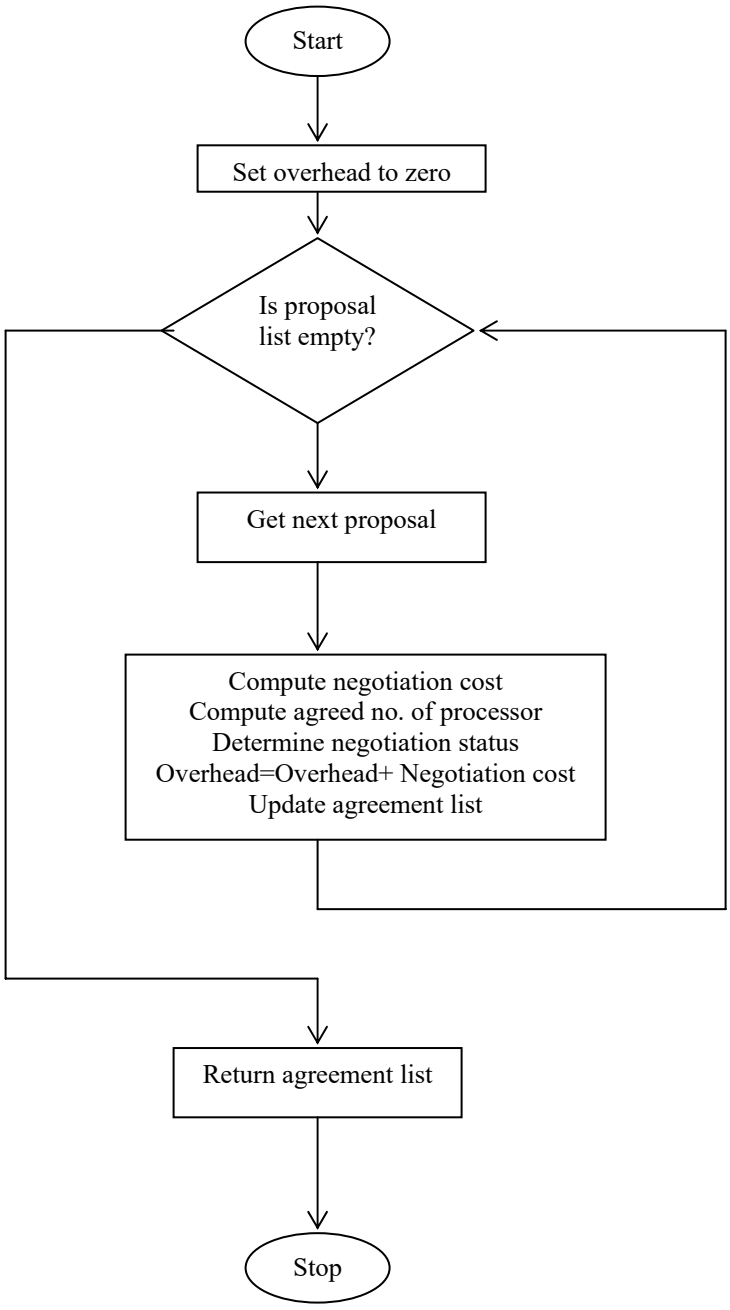


Figure 4.9 Flow diagram of the negotiator

After executing all the agreements, the dispatcher executes the jobs to start list. For each job in the list it computes the completion time of the job, creates a job completion event and inserts the job completion event in the event list. It also moves the job from the pending list to running job list and sets the start time and current allocated processor. The dispatcher then sorts the event list in order of event time.

Input: schedule (jobs to start list and agreement list)
Output: updated system status
1. while agreement list is not empty
2. remove next agreement
3. $Wr_i \leftarrow Wr_{i-1} - p_i * t_i$
4. $tr_i \leftarrow Wr_i / p_{i+1}$
5. adaptation cost ← get adaptation cost(min. adapt. cost, max. adapt. cost)
6. $t_c \leftarrow$ simulation clock + adaptation cost + tr_i
7. update job completion event in the event list
8. update job info in running job list
9. update available processor
10. end while
11. while jobs to start list not empty
12. remove next job
13. $Wr \leftarrow (pd * td)$
14. $t_c \leftarrow$ simulation clock + W_r/p_1
15. $t_s \leftarrow$ simulation clock
16. insert job completion event in the event list
17. move job from pending list to running job list
18. update available processor
19. end while
20. sort event list

Figure 4.10 Algorithm for the dispatcher

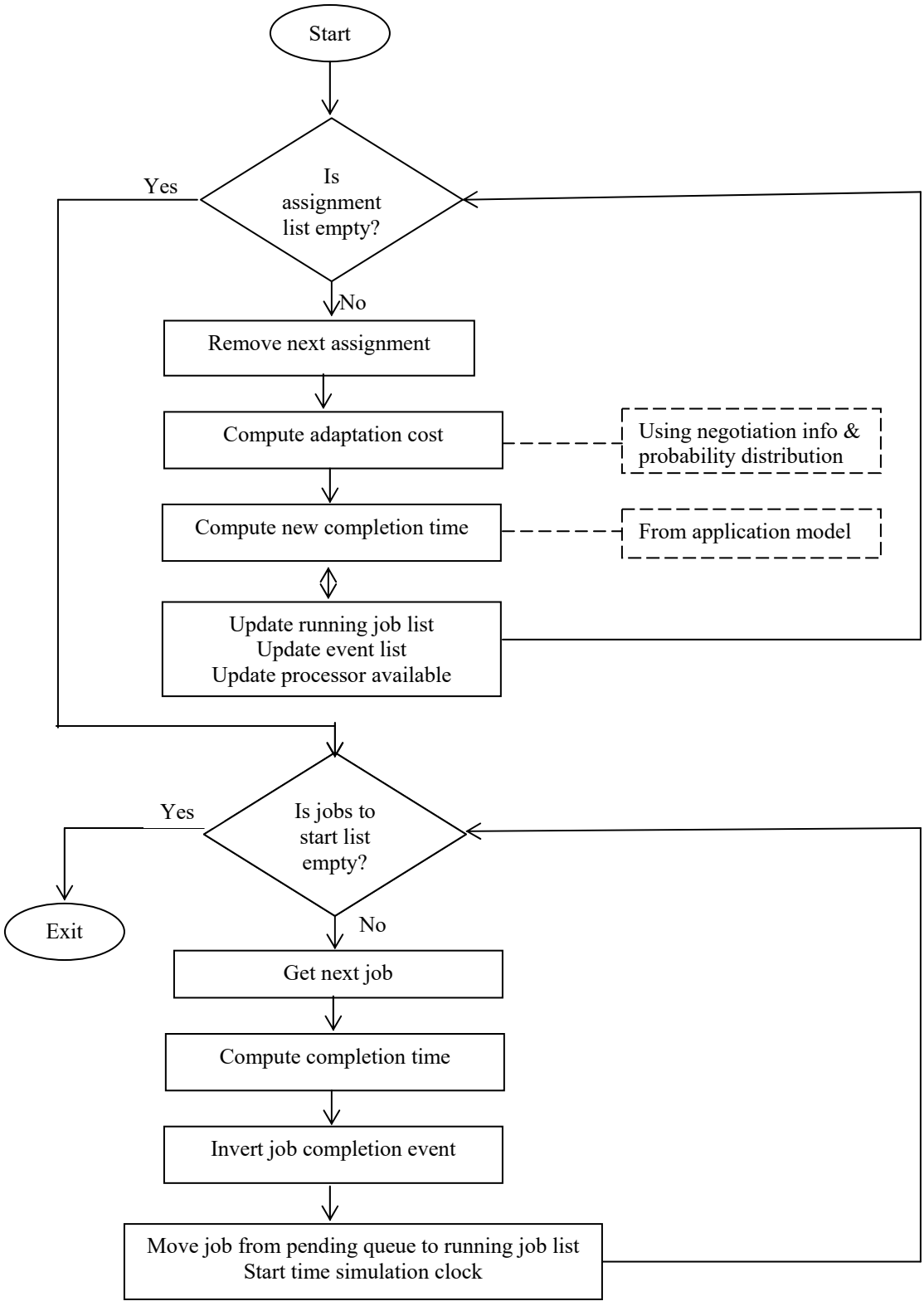


Figure 4.11 Flow diagram for the dispatcher

4.2.7 Implementation

The simulator has been implemented as a single program with one thread of execution. All the modules were implemented as separate subroutines. The simulator was implemented in C on a Linux platform. During the development of the simulator, the incremental development model was followed. As each increment was built the components underwent thorough unit testing. Each developed increment was continuously integrated with the previous one and an integration test was performed.

4.3 Prototype Resource Management System

We have developed a prototype resource management system capable of handling malleable jobs. There were three objectives for the development of the prototype RMS. The first objective was to use it as a tool to study an adaptive parallel system. The second objective was to get an idea about the realistic values of system parameters such as negotiation costs, adaptation costs etc. The third goal was to generate some real world data which could be used to validate the simulator described in the previous section. Developing a full blown RMS and malleable applications with different characteristics is labor intensive, time consuming and a costly task. We have implemented the prototype RMS with minimum functionality and implemented one malleable application. We took the path of least resistance to get to a working system, so that we can use it to study an adaptive parallel system and generate some real data. The implementation of the prototype system and the results of experimentation with it are discussed in this section.

In order to manage malleable applications, interactions between the RMS and the applications are required. Because of the resource utilization pattern of malleable

applications, a simple accept/reject type of communication with the RMS is not enough [16]. Managing negotiations with running malleable applications is one of the critical requirements of an adaptive RMS. For such negotiation management, an adaptive RMS must perform the following additional functionalities compared to a traditional RMS: i) carry out negotiations with the running malleable applications; ii). allocate/claim resources to/from the running malleable applications; iii). make decisions to allocate idle resources among the running malleable applications; and iv) choose resource preemption candidates among the running malleable applications.

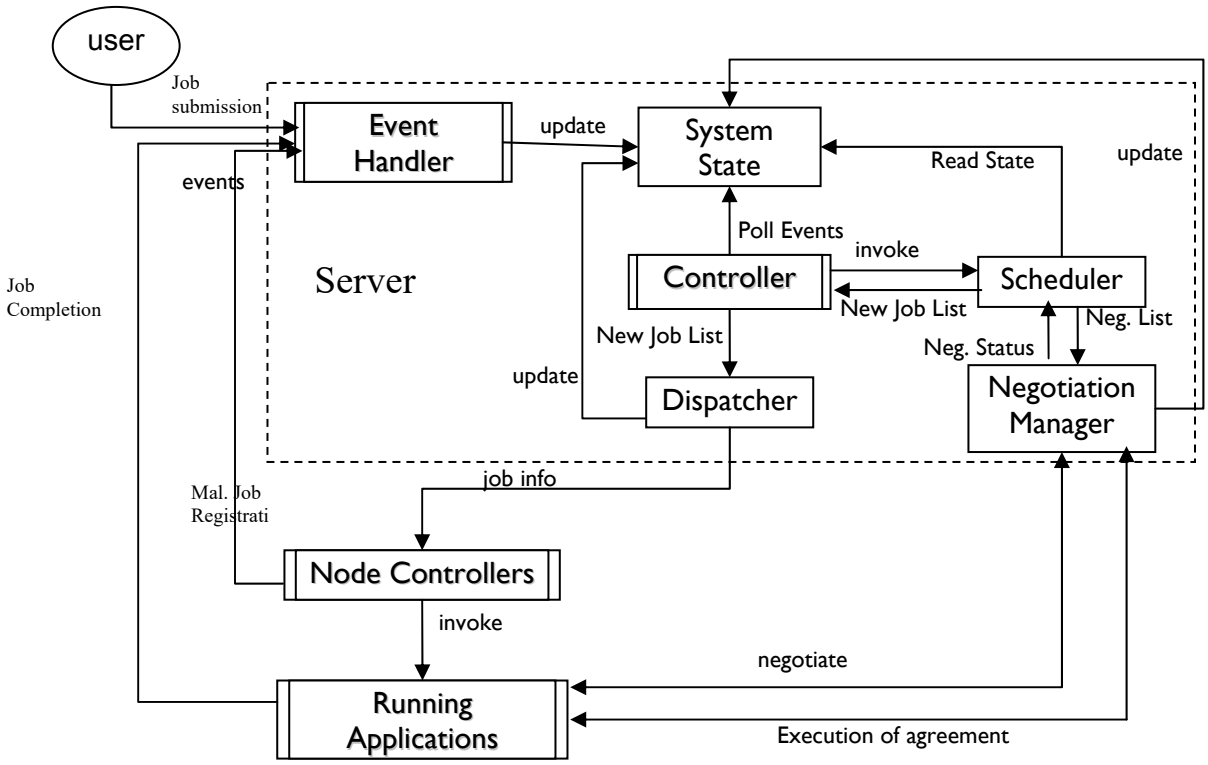


Figure 4.12 Architecture of the prototype RMS.

4.3.1 Architecture and Implementation

Figure 4.12 shows the architecture of the RMS for malleable applications, based on the requirements mentioned above. The RMS has two parts: a server and node controllers. The server is responsible for gathering information about the available resources, accepting jobs from the users, organizing those jobs in queues, and initiating a schedule cycle. Once a schedule is contrived, the server contacts the individual node controllers, which place applications into execution. There is one node controller per computational node. Each controller acts as an agent of the server and starts and controls applications on the nodes.

The server consists of the following components: the event handler, the controller, the scheduler, the negotiation manager and the dispatcher. The event handler and controller are two separate threads of execution that are running concurrently. The server also manages information about the system through a data structure called system state. The system state maintains information about the current state of the RMS. It consists of following information: i) pending job queue - list of jobs submitted by users organized in a FIFO Queue, ii) list of running rigid jobs, iii) list of running malleable jobs, iv) resource information, and v) pending event queue. The event handler accepts events external to server and updates the system state. Currently the event handler responds to three types of events: job submissions by users, job completion notifications by the node controllers and registrations by running malleable jobs. Out of these events, job submission and job completion are scheduling events, meaning that the server initiates a scheduling cycle in response to these events.

A user submits a job for execution by submitting a script containing information about the application to executed. Jobs are submitted through a client program. In the submission script the users have to mention the job type (whether a job is rigid or malleable), and the minimum and the maximum processors requirement of the job (in case of malleable job). The event handler receives the job information from the client program and inserts the job information in the pending job queue; it also pushes the job submission event in the event queue. When a running job completes its execution, the node controller that had started the job sends a job completion notice to the event handler. The event handler updates the resource information, running job list, and pushes a job completion event in the event queue.

When a malleable job starts execution, it first opens a socket and sends the host and port information to the event handler. This process is called job registration. When the event handler receives this information it stores this information in the running malleable job list. The purpose of registration is to enable communication between the server and running malleable application, whenever expansion or shrinkage of running malleable jobs is required.

Whenever there is a scheduling event in the event queue, the controller removes the events and initiates a scheduling cycle by calling the scheduler. There can be two events (ex. a job submission and running job completion at the same time) arriving at the same time in the event queue. In such a case, the controller component of the server removes both the event from the queue and initiates a single scheduling cycle. The scheduler computes a schedule and sends a list of pending jobs to be started with logical

processor assignment to the controller. The controller sends the list of jobs to be started to the dispatcher.

The scheduler computes a schedule based on system state and scheduling policy. A schedule consist of three lists: a list of pending jobs to be started with logical processor assignment, a list of running malleable jobs with number of processor to be releases by shrinking, and a list running malleable jobs with number of additional processors allocated to them for expansion. After a scheduling cycle, all or any of these lists can be empty. The scheduling policy adopted in present RMS is FCFS (First Come First Serve), and pending new jobs are given priorities over running malleable jobs.

During a scheduling cycle, first a jobs to start list is created by allocating a minimum number of processors to the pending jobs as long as resources are available. If enough processors are not available to schedule pending jobs, the scheduler makes decision to preempt processors from running malleable jobs (if there are any) in order to schedule pending jobs. The scheduler computes the maximum number of processors that can be preempted from all running malleable jobs and tries to schedule as many pending jobs as possible. Once scheduling decisions of pending jobs are made, the scheduler selects the preemption candidates among the running malleable jobs. The required number of processors is preempted starting with the maximum possible number of processors from the running malleable job that has started the earliest, continuing with next malleable job and so on. If idle processors are available and no jobs are pending, or idle processors are not enough to fulfill the minimum processor requirement of the next pending job in the queue, the idle processors are allocated to running malleable jobs.

During allocation of idle resources, the malleable job which has started the earliest is given its maximum number of processors, provided enough processors are available. Otherwise the job is allocated the available number of processors. This continues as long as processors or running malleable jobs are available.

The scheduler does not start a pending job or preempt processors from running jobs. It just makes the decision which pending jobs to start, which running jobs to shrink or expand. Once a schedule is computed, the scheduler sends the list of jobs to shrink and the list of jobs to expand to the negotiation manager. When the scheduler receives the negotiation status from the negotiation manger, it sends the list of pending jobs to be started to the controller. Ideally the scheduler may need to re-compute the schedule, based on the negotiation status, which in turn may require further negotiations; i.e. the scheduler may work in multiple stages until a final schedule acceptable to all parties involved is computed. The scheduler in the prototype implementation is a single stage scheduler, it does not re-compute based on the result of the negotiation.

The negotiation manager negotiates with running malleable jobs, executes the expansion or the shrinkage of the running jobs, and also updates the system state. The expansion and shrinkage of malleable applications is a two-step process. In the first step, the negotiation is carried out; once an agreement is reached, the agreement is executed in the second step. In the case of expansion, the execution involves sending the list of physical processors to the application, and updating the system state. In the case of shrinkage, the execution involves receiving the list of processors released by the application, and updating the system state. In the present implementation, both the

negotiation and the implementation of an agreement are carried out by the negotiation manger. The negotiation is carried out sequentially starting with the first job in the negotiation list. Once negotiations and execution of agreements are complete, the negotiation manger sends the negotiation status back to the scheduler. For the prototype implementation, the negotiation cost between the scheduler and the malleable applications has been measured and was found to be very low (1.5 milliseconds on average for one round of negotiation) compared to the typical execution time of parallel applications.

The controller invokes the dispatcher and sends the list of pending jobs to be started. The dispatcher reads the system state and assigns physical processors to the pending jobs to be started. It then creates a configuration file containing the list of physical processors allocated to the job, much like PBS node file for MPI jobs. This file is used by the application to spawn processes on the allocated processors. The dispatcher then sends the job information to the designated node controller to start the job. After the job is started, the dispatcher updates the system state. Like the negotiation manager, the dispatcher also works sequentially.

4.3.2 Resource Negotiation Protocol

In order to manage adaptive applications, interactions between applications and the RMS is required. The communication scenarios that may occur between adaptive applications and an RMS may widely vary: Two examples are briefly described below:

1. A malleable application, which requires that the number of processors be a power of 2. The minimum and maximum processor requirement is 8 and 32, respectively. Currently, the application is executing on 8 processors. In mid execution, the RMS may offer 15 processors to the application. Since the application can use only 8 additional processors out of 15 offered, instead of rejecting the offer, the application may ask the RMS to allocate 8 additional processors.
2. In an environment where applications pay for resources, when some idle resources are available, the RMS may offer the resources to a malleable application for a price. The application may be willing to accept the additional resources at a lower price, and therefore makes a counter offer. Depending on the policy, the RMS may accept or reject the offer or even make another counter offer.

From the above scenarios, it is evident that a simple accept/reject type of communication is not enough. A complex multi-round negotiation between applications and the RMS is required to support a wide variety of parallel adaptive applications. For negotiation of resources between adaptive applications and the RMS, a negotiation protocol has been developed and implemented.

Figure 4.13 shows the finite state machine representation of the negotiation protocol. The initiator (either applications or the RMS) specifies resource requirements and associated terms and conditions. The other party examines the resource request and responds. The response can be accept, reject, or a counter offer with modified

requirements. At any time during the negotiation, any party can send a final offer indicating that no further negotiation can be done. The other party can either accept or reject the offer. Also, during the negotiation, any party can accept or reject an offer and thus terminate the negotiation. The negotiation results in either accepted or rejected status. In the case of an accept, the agreed resources are allocated. All the information regarding a negotiation (resources requested, terms and condition etc.) is encapsulated in an object, which we call the Negotiation Template (NT). The negotiation takes places by exchanging this template. It is composed of three sections. The first section contains general information related to the two parties. The second section contains the status of the negotiation, and the third section contains a list of resource objects that are being negotiated. Each resource object has information about the resource being negotiated, the quantity of the resource being requested, and the status of each resource request. An NT can have more than one resource request inside it. Each resource request has its own terms for the negotiation and its own status. The overall status of the negotiation depends on the combined status of all the requested resources. A detailed description of the resource negotiation protocol is described in [16][17]. A set of APIs that can be used by adaptive applications and the RMS for resource negotiations has been developed. The negotiation manager uses these APIs to communicate and negotiate with malleable applications.

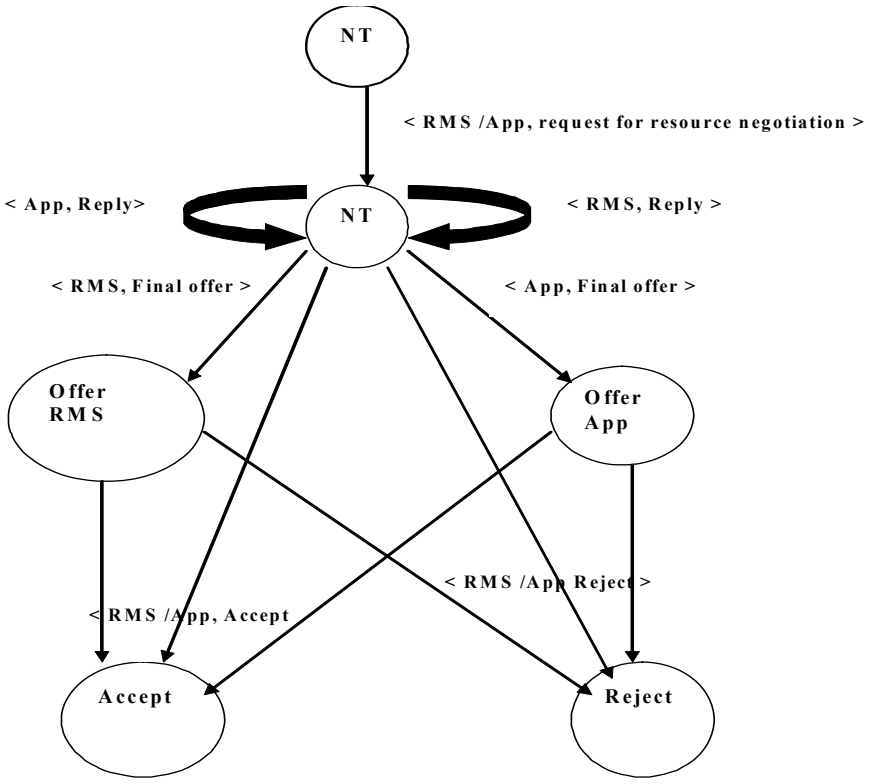


Figure 4.13 Finite state representation of the negotiation protocol

4.3.3 Workload

The workload creation for experiments in this research faces a unique problem. The problem is to generate workloads with actual applications, which will run on the test bed in a reasonable amount of time (since simulation is not being used) and that will be similar in characteristics to workloads from a validated model or to traces from some supercomputer center.

The workload model by Allen B. Downey [52] has been used to generate workloads for experiments. Using Downey's model, a rigid workload containing 120 jobs has been generated. The number of jobs has been limited to 120, so that the jobs can run

on a sixteen processors cluster (our test bed) in a reasonable amount of time. Actual applications have been created according to the workload generated from the model. From this original workload, several other rigid workloads have been created by decreasing the inter-arrival time. Since the processor requirement and the execution time remains unchanged, this results in a higher system load for the RMS. The factor by which the inter-arrival time is decreased is called the shrinking factor. The shrinking factor of the original workload is 1. When the load is low, the system utilization is low. As the load increases, the utilization increases and reaches a saturation point. The saturation for the generated workloads occurs at shrinking factor of 0.4.

From the rigid workload with shrinking factor of 0.4, several workloads containing malleable jobs have been created. In order to study the impact of malleable applications, three parameters of the workload have been varied: the number of malleable jobs as percentage of total jobs, the processor flexibility range of malleable jobs, and the distribution of malleable jobs in the workload. The percentage of malleable jobs was varied between 0% and 100% to investigate the impact of number of malleable jobs on performance. Workloads with 10%, 25%, 33%, 50%, 75 % and 100% malleable jobs have been created. Both the flexibility range and flexibility of malleable jobs determines how efficiently the RMS can adapt malleable jobs to utilize all the processors in the cluster. The flexibility range of the malleable jobs was varied between 2 to 5, 2 to 6, 2 to 8, and 4 to 8 processors. To utilize all the processors in the system there have to be some malleable jobs running at all times throughout the execution of a workload. A smart scheduler is needed to make sure that some malleable jobs are always running. Since the

RMS is using a FCFS scheduler, to avoid fragmentation, there has to be at least one malleable job either running or waiting at the front of the pending queue at the time of scheduling. The malleable jobs are distributed uniformly through out the workload.

4.3.4 Application

The Open System for Earthquake Engineering simulation (OpenSees) [62] software for simulating seismic response of structural and geotechnical systems has been selected as the test application. The application works in a master-worker fashion. It consists of one coordinating process and one or more computing processes to perform the actual simulation of structures' response. The coordinating process takes a list of allocated processors and a list of structures as input. Implemented using PVM [63], it starts computing processes on each of the allocated processors and distributes the structures for simulation until all the structures are simulated. The execution time of the application is proportional to the number of structures to be simulated.

The model of the malleable version of this application have four properties: i) the application can dynamically create and destroy processes; ii) it can accept a negotiation request from the RMS and carry out negotiation; iii) the application can run on any number of processors between a minimum and a maximum number of processors; iv) it can release agreed upon processors without delay. The coordinating process has the capability of changing the number of computing processes during execution by dynamically creating new computing processes or destroying existing computing processes. The coordinator process is also capable of accepting requests from the RMS and engaging in resource negotiation. The negotiation results in either accepting or

rejecting the offer from the RMS. The malleable application has two parameters: the minimum and the maximum number of processors. As long as the negotiation results in the total number of processors of the application to be between the minimum and maximum, the offer from the RMS is accepted. Otherwise the offer is rejected. In case of rejection, the RMS doesn't terminate the application. The application continues to run on currently allocated processors.

4.3.5 Results and Analysis

To evaluate the impact of malleable jobs on system and application performance, the generated workloads have been executed on a dedicated cluster of 16 processors. The results of the experiments, namely, the system utilization, the schedule span, and the turn around time as a function of the composition of the workload (percentage of malleable jobs) for different flexibility ranges are shown in table 4.1 and figures 4.14 to 4.18 respectively.

From figure 4.14 it can be seen that the utilization increases as the number of malleable jobs in the workload increases. The utilization saturates at a job mix of about 33%, and it increases very little with the increase of number of malleable jobs after saturation. For a flexibility range of 2-8, the utilization increases from 87% for an all rigid workload to 90% for 10% of malleable jobs, and reaches a maximum of 100% for all malleable jobs. The utilization is 98% for a job mix of 33%.The results also show that for the same job mix the utilization is little lower for lower flexibility range. Almost full (i.e., 99%) utilization is achieved with a workload containing all malleable jobs for all

flexibility range. This indicates that the overhead of managing malleable applications is very low.

Figure 4.15 shows the variation of utilization with the flexibility range for all job mixes. Experiments have been carried out for three flexibilities: 3 processors (range 2 to 5), 4 processors (range 2 to 6), and 6 processors (range 2 to 8). In can be seen from the figure that the utilization increases with the increase of flexibility of malleable jobs for all job mixes. This is because with higher flexibility the scheduler has a higher probability of filling up the cluster. The result also shows that the number of malleable jobs in the workload has a more prominent impact on utilization than the impact of flexibility on utilization.

For the same flexibility, the utilization also depends on the minimum processors requirements. Figure 4.16 shows the variation of utilization for two flexibility ranges (2 to 6 and 4 to 8) with same flexibility (4 processors). It can be seen from the figure that the utilization for the range 4-8 is lower than the utilization for the range 2-6 at lower job mix, and the and the gap in utilization decreases as the number of malleable jobs increases. This is because with lower minimum processor requirement, the scheduler has a higher probability of scheduling a pending malleable job. For example consider the scenario that there are 2 idle processors, all the running jobs are either rigid and/or malleable running on minimum processors, and the job at the head of the pending queue is malleable. In such a case, if the flexibility range of the pending malleable job is 2-6, it could be started immediately, but if the flexibility range is 4-8, the job cannot be started. It will have to wait until a running job exits and releases at least 2 more processors

making the number of idle processors 4 or more. As the number of malleable jobs increases, the probability of a malleable job running on more than its minimum processors increases, which in turn increases the probability of releasing 2 processors by shrinking running malleable to start the pending malleable jobs. As a result, the gap in utilization decreases as the number of malleable job increases.

Figure 4.17 shows the plots for the schedule span as function of job mix. Similar trends as those of utilization can be seen for the schedule span. The schedule span decreases 668 seconds from 5058 for a rigid workload to 4390 for all malleable jobs for a flexibility range of 2-8. For 10% malleable jobs the schedule span decreased to 4861 seconds. Moving from an all rigid workload to a 10% malleable job mix saves 10688 cpu seconds.

The average turn around time (TAT) of figure 4.18 shows similar trends as schedule span. The average turn around time decreases from 1968 seconds for a rigid workload to 1849 seconds for 10% malleable jobs to a minimum of up to 1657 seconds for 100% malleable jobs. For 33% malleable jobs the turn around time decreases to 1675 seconds. It means that for a job mix of 33% on average, an user has to wait 273 seconds less to get his/her result after the submission of a job.

The improvement in performance in the presence of malleable jobs in the workload comes at the expense of the execution time of the malleable jobs. The experimental results show that in malleable mode, the execution time of a job increases in general. Since the scheduler has to shrink a malleable job to accommodate the pending jobs, on average, a job runs on a lower number of processors in malleable mode than it

does in rigid mode. Table 4.2 presents the average execution time, average turn around time, and average wait time for different job mixes for the flexibility range 2-8. The graph of figure 4.19 shows the average execution time, the average turn around time, and the average wait time as function of number of malleable jobs in the workload. From the figure it can be seen that as the number of malleable jobs increases, the job execution time increases on average. However, as the number of malleable job increases, the average turn around time decreases. This is because with higher number of malleable jobs, a job has to wait less in the pending queue.

Table 4.1 Utilization, schedule span and average TAT for different job mix in workload

%Job Mix	Utilization				Schedule Span(Sec)				Average TAT (Sec)			
	2-5	2-6	2-8	4-8	2-5	2-6	2-8	4-8	2-5	2-6	2-8	4-8
0%	.87	.87	.87	.87	5058	5058	5058	5058	1969	1969	1969	1969
10%	.90	.91	.91	.89	4884	4872	4861	4949	1866	1858	1849	1886
25%	.95	.96	.96	.95	4653	4613	4581	4661	1737	1711	1691	1735
33%	.96	.97	.98	.96	4620	4560	4490	4569	1712	1696	1675	1711
50%	.98	.98	.98	.97	4495	4487	4486	4539	1650	1647	1641	1665
75%	.99	.98	.99	.98	4472	4483	4456	4486	1668	1658	1658	1637
100%	.99	.99	1.0	.99	4470	4445	4390	4455	1672	1669	1657	1633

Table 4.2 Average execution time, turn around time and wait time for different job mix in workload

% of Job Mix	Avg. Exc. Time	Avg. Turn Around Time	Avg. Wait Time
0%	154	1969	1815
10%	156	1849	1694
25%	161	1691	1530
33%	160	1675	1514
50%	192	1641	1449
75%	239	1658	1418
100%	284	1657	1373

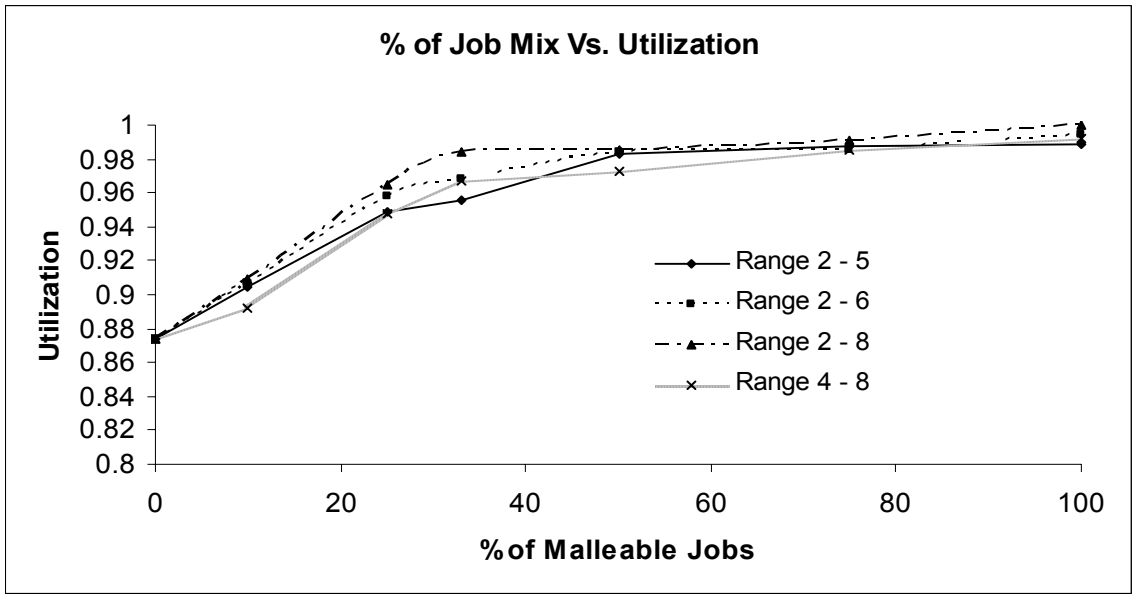


Figure 4.14 Utilization as function of job-mix

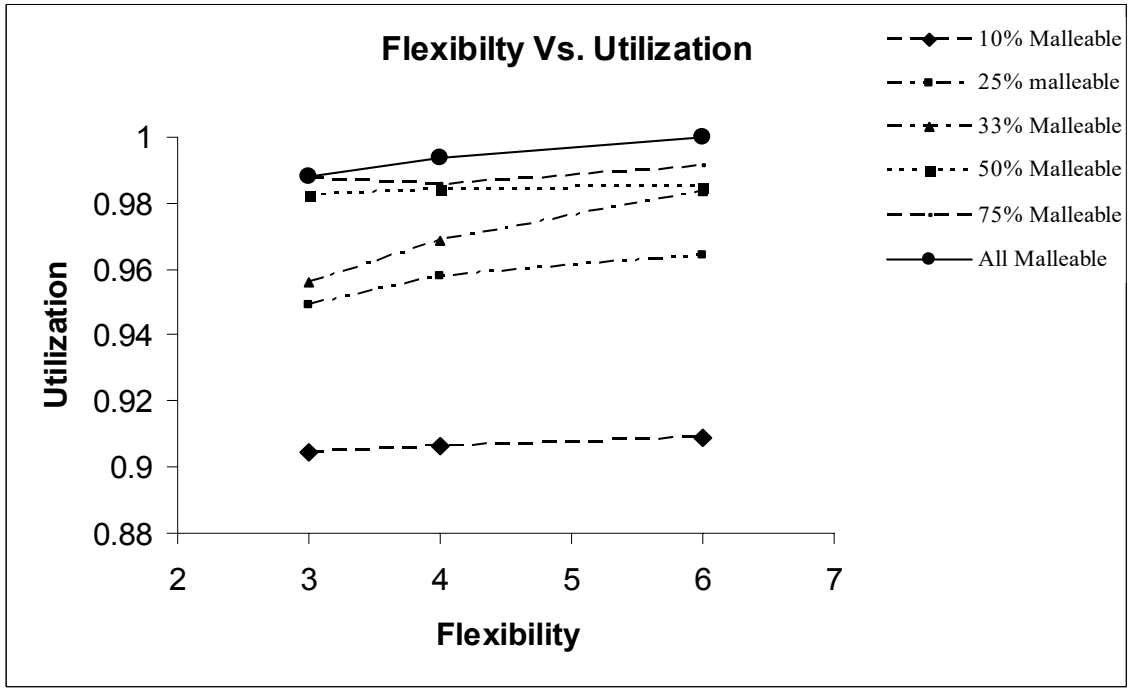


Figure 4.15 Utilization as function of flexibility

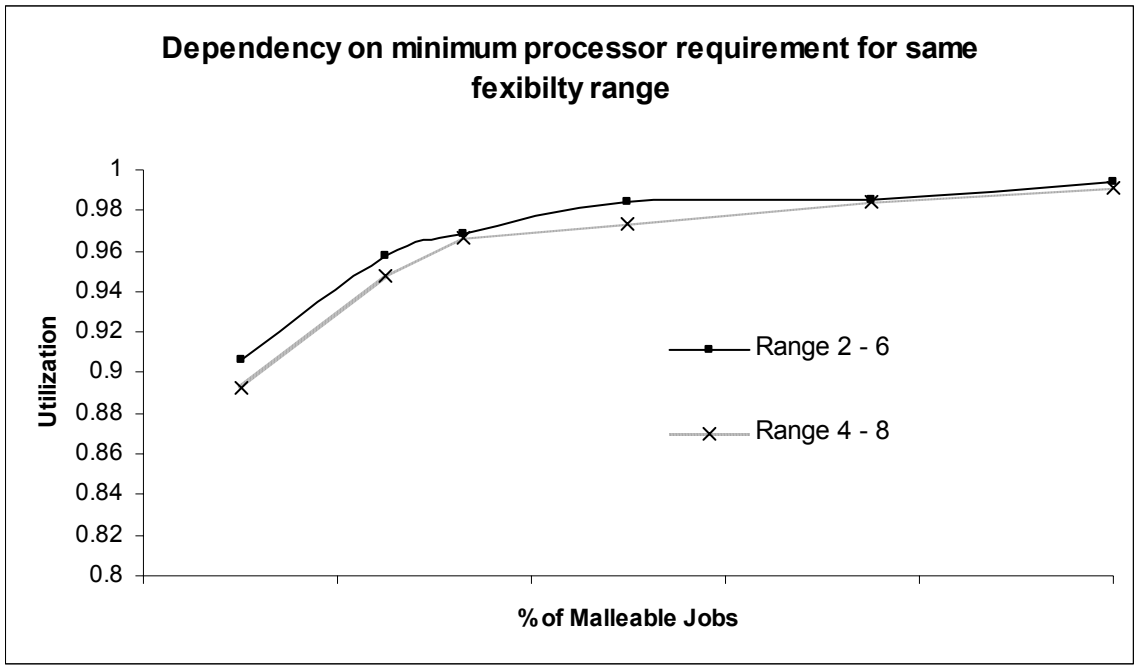


Figure 4.16 Utilization as function of percentage of malleable jobs

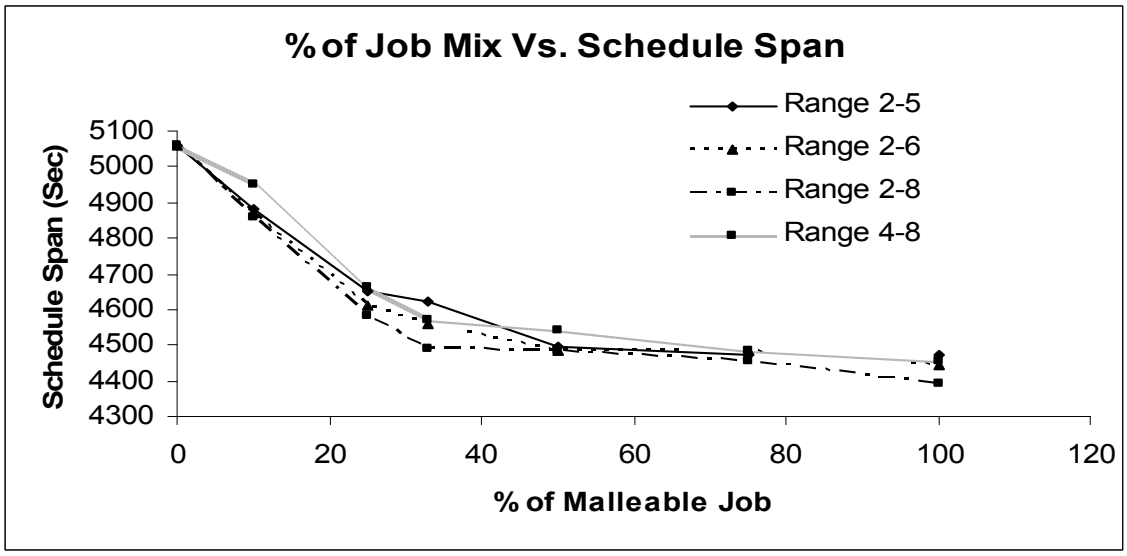


Figure 4.17 Schedule span as function of job-mix

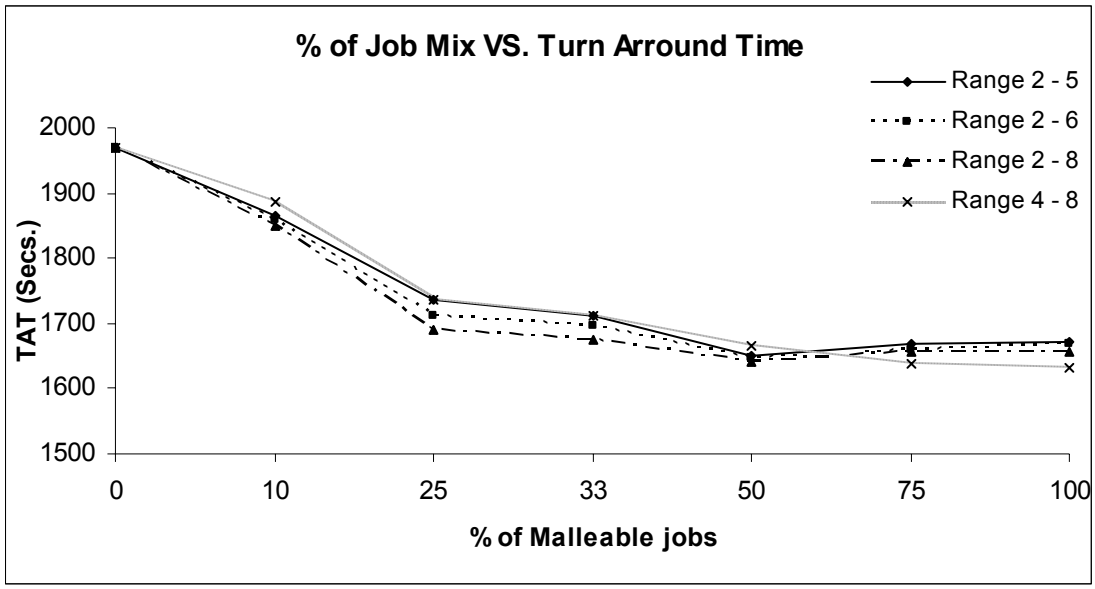


Figure 4.18 Turn around time as function of job-mix

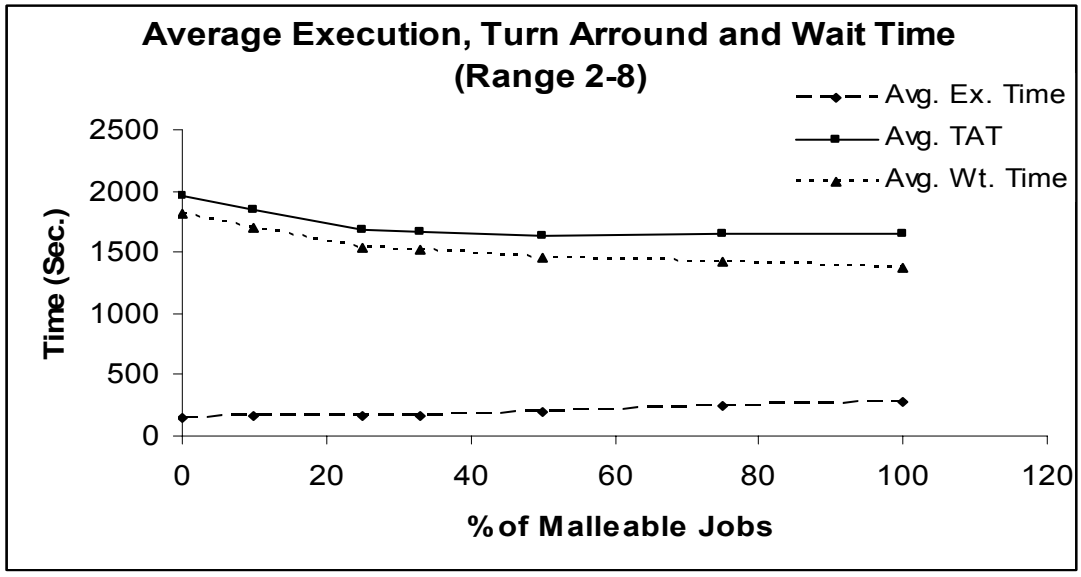


Figure 4.19 Average execution, turn around, and wait time as function of job-mix

CHAPTER V

VALIDATION

In chapter III we have described the model of an adaptive parallel system and in chapter IV we have described a discreet event simulator to simulate the model. Chapter IV also presents a prototype implementation of the model. In this chapter and next chapter we present the experimental results with the simulator. Figure 5.1 shows the experimental procedure with the simulator.

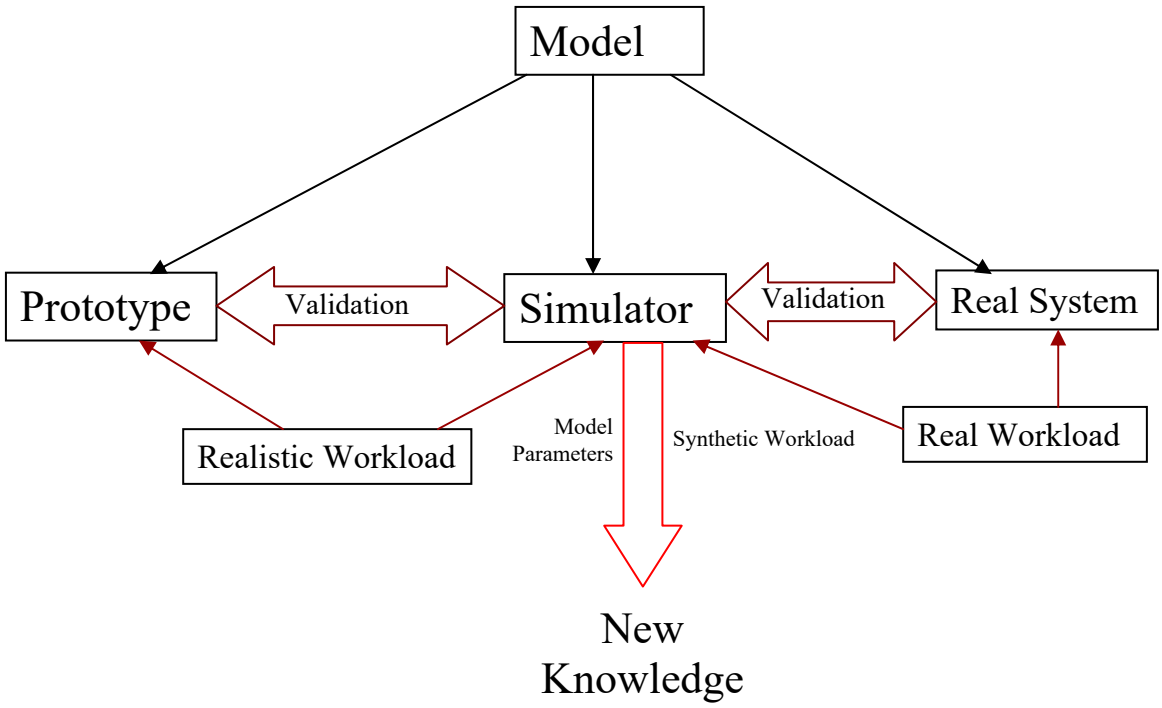


Figure 5.1 Experimental procedure with the simulator

One can think that the model we have developed has three representations: the prototype, the simulator and the real rigid parallel systems. The model was developed based on real rigid parallel system as well as a hypothetical adaptive parallel system. Consequently, one can think that the read rigid system is a representation of the model and vice versa. To validate the model we need to validate the simulator against real systems. We have validated the simulator against a rigid parallel system at San Diego supercomputer center as well as against the prototype. For validation against rigid system we conducted simulation experiments with real data obtained from workload logs at SDSC [65] and compared the outputs of the simulation with the outputs of SDSC system. For validation against adaptive system we conducted experiments with prototype system with realistic workload. Simulation experiments were conducted with the same realistic workload as input, and the output of the simulator and the prototype was compared The realistic workload is generated using modifies Downy's workload model. Once the simulator is validated against the rigid and adaptive parallel system we conducted simulation experiments with synthetic workloads and with different model parameters to gain new knowledge about adaptive parallel systems.

In this chapter we present a set of experiments to validate the simulator while Chapter VI presents simulation experiments to gain new knowledge about adaptive parallel system. Two sets of experiments were performed for the validation of the simulator. The first set of experiments is directed towards validating the model for rigid parallel systems. The second set of experiments validates the model for an adaptive

parallel system with malleable applications. Workload data from the prototype system described in chapter IV has been used for the validation of the simulator.

The workload data for simulation experiments consists of information about a set of applications. For each job, the workload data provides application type (rigid/malleable), arrival time, number of processors required, and the execution time on the required processors. In addition, for malleable applications the minimum and the maximum number of processors that an application is capable utilizing are also provided.

5.1 Evaluation Method

In general a simulator is valid if it can accurately approximate the real system it is simulating. The simulator is evaluated using two approaches: individual application data and group data.

5.1.1 Individual Application Data

In this approach we run the simulator using data set from real systems. The individual application data from the simulator output is compared with the output of the real system. The start time and the completion time of each individual application from simulator and real system is compared. For each data set the Euclidean Distance $\boldsymbol{d_e}$ of each application's start time, and completion time from simulator and real system output is computed.

The Euclidean distance d_e between two points is $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$, in Euclidean *n*-space, is defined as:

$$d_e(p,q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Euclidean distance between two one dimensional points $P = (p_x)$ and $Q = (q_x)$, the distance is computed as:

$$\sqrt{(p_x - q_x)^2} = |p_x - q_x|$$

Since time is a one dimensional data one dimensional Euclidean distance is used as a metric to compare the start time $\boldsymbol{T_s}$ and completion time $\boldsymbol{T_c}$ between real system and simulator output. The distance $\boldsymbol{d_e}$ should very small and should not vary from job to job. A plot of $\boldsymbol{d_e}$ against job number should be a straight line.

5.1.2 Group Data

In this approach the performance metric of a workload (data set) is computed from simulator and real system output. The performance metric for a workload are system utilization $\boldsymbol{U}$ and average turn around time $\boldsymbol{ATAT}$. The Euclidean distance between utilization of real system $\boldsymbol{U_r}$ and simulator $\boldsymbol{U_s}$ for all data sets are computed. Similarly the Euclidean distance between average turn around time of real system $\boldsymbol{ATAT_r}$ and simulator $\boldsymbol{ATAT_s}$ for all data sets are computed. The distance $\boldsymbol{d_e}$ should be very small and should not vary from data set to data set. A plot of $\boldsymbol{d_e}$ against data set should be a straight line.

In addition, for malleable workloads the trends in variation of $\boldsymbol{U}$ and $\boldsymbol{ATAT}$ between simulator outputs are compared to real system outputs to see whether the trends

are similar. To evaluate whether the behavior of the simulator scales properly, simulation experiments are conducted with several synthetically generated malleable workloads. The properties of the synthetic workloads are kept the same as those of the real malleable workloads, except that the number of jobs in the workloads and the maximum number of processors is increased proportionally. For synthetic workloads the simulation parameters are set equal to the values measured from the real system except for the cluster size which is increased proportional to the increase in number of maximum processor. For the synthetic malleable workloads the trend in variation of utilization is compared to variation in utilization in real system with malleable workload.

5.2 Experimental Data

For experiments with rigid data, workload log from San Diego supercomputer center (SDSC) and data from the prototype system has been used. The SDSC data set consist of 59725 jobs from April 1998 to April 2000. The jobs were executed on an IBM SP2 computer comprised of 128 processors**.** The prototype data set consists of 120 jobs. The jobs were executed on a cluster of 16 processors.

For experiments with malleable applications six data sets from the prototype system have been used. Each data set consists of 120 parallel jobs with a processors requirement vary between 2 to 8 processors. The minimum and maximum processor requirement of malleable jobs were 2 and 8 respectively. The rigid run time of the job varies between 100 and 800 seconds. The differences between the data sets are in number of malleable jobs and in number of minimum and maximum processors for each malleable job. Table 5.1 summarizes the characteristics of malleable datasets. To evaluate

the scalability of the simulator, synthetic malleable workloads were generated using the workload model described in chapter III Table 5.2 shows the characteristics of synthetic malleable workloads.

Table 5.1 Malleable workload from prototype system

Data Sets	No. of Jobs	Min. # of Proc.	Max. # of Proc	Min. Run Time	Max Run Time	Pmin for Malleable Jobs	Pmax for Malleable Job	% of Malleable Job
1	120	2	8	100	800	2	8	10
2	120	2	8	100	800	2	8	33
3	120	2	8	100	800	2	8	50
4	120	2	8	100	800	2	8	75
5	120	2	8	100	800	2	8	100

Table 5.2 Malleable workload generated synthetically

Data Sets	No. of Jobs	Min. # of Proc.	Max. # of Proc	Min. Run Time	Max Run Time	Pmin for Malleable Jobs	Pmax for Malleable Job	% of Malleable Job
1	1200	2	32	100	800	2	32	10
2	1200	2	32	100	800	2	32	33
3	1200	2	32	100	800	2	32	50
4	1200	2	32	100	800	2	32	75
5	1200	2	32	100	800	2	32	100

5.3 Experimental Results

In order to validate the simulator we run two set of simulation experiments. The First set of simulation experiments was conducted with rigid data sets. The second set of f experiments was conducted with dataset containing malleable jobs.

5.3.1 Simulation with Rigid Data

The normalized Euclidean distance between the real system output and simulator output for both application start time and application completion time is used as metric to

evaluate the how accurately the simulator approximate real system. The normalized value is computed according to the following equation.

Normalized value = |(real system output – simulator output)| / real system output

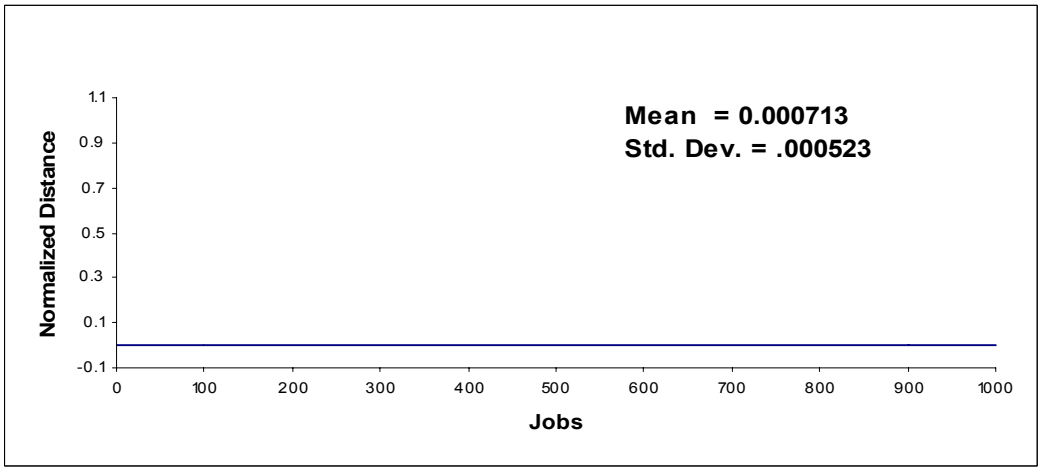


Figure 5.2 Comparison of start time of simulator and SDSC output

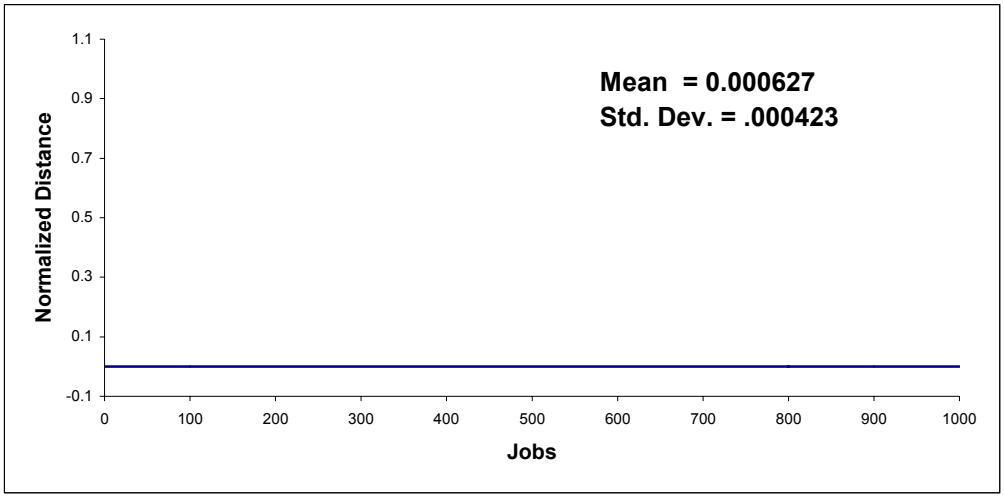


Figure 5.3 Comparison of completion time of simulator and SDSC output

Figure 5.1 through 5.2 shows the graph of normalized Euclidean distance d_e for application start time T_s and application completion time T_c for rigid data from San Diego supercomputer center and prototype system.

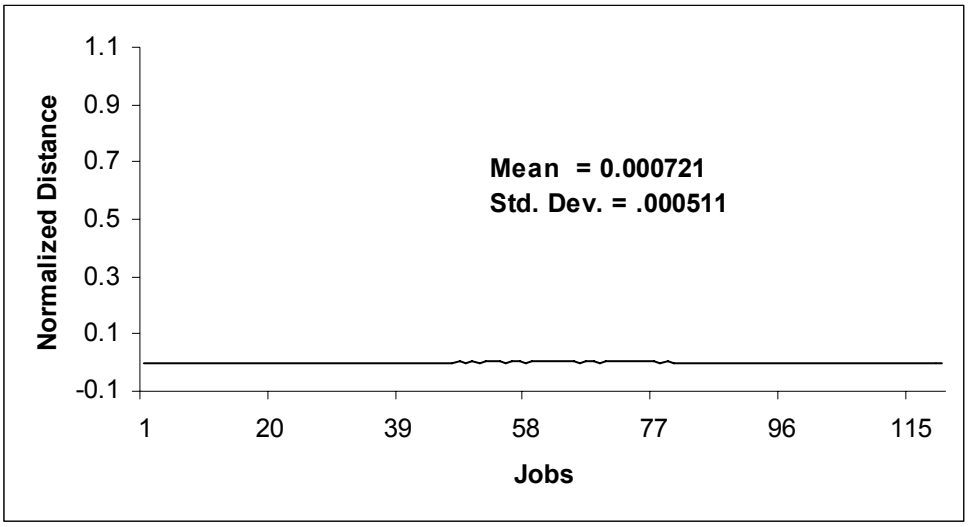


Figure 5.4 Comparison of start time of simulator and prototype output

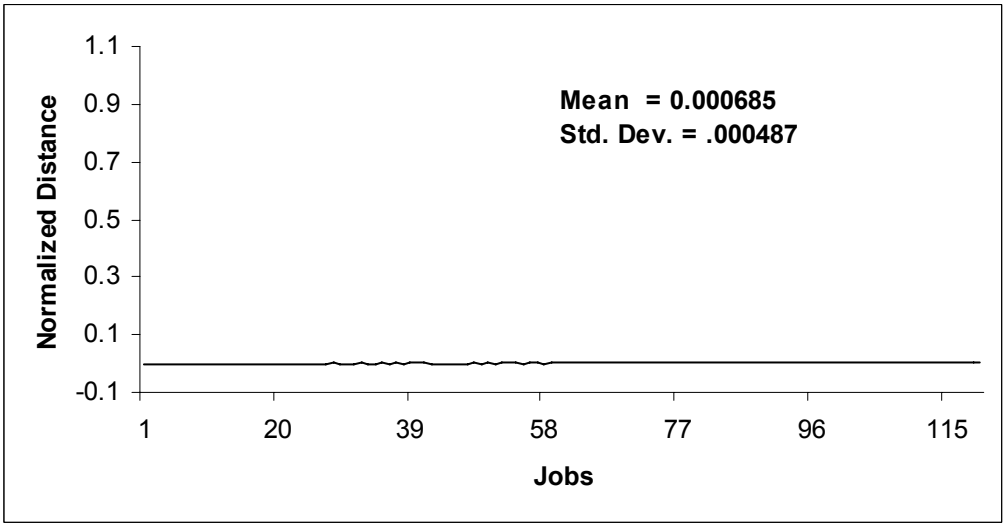


Figure 5.5 Comparison of completion time of simulator and prototype

From Figures 5.2 - 5.5, it can be seen that the simulator outputs very closely approximate the outputs of the real system. For all applications the difference between real data and simulator output is less than 0.1 percent for both application start time and completion time. The mean and standard deviation of distances for start time and completion time is very low. Table 5.3 shows the comparison of system utilization and average turn around time between real system and simulator output. From Table 5.3 it can be seen that the system utilization and average turn around time of simulation output very closely approximate those from real system. From experimental results presented above it can be concluded that the simulator is a faithful representation of real system with rigid applications.

Table 5.3 Comparison of utilization and average turn around time

	Utilization		Normalized Distance	Avg. TAT		Normalized Distance
	Real	Simulator				
Prototype	0.8733	0.8736	0.00037	1968	1997	0.0140
SDSC	0.7432	0.7434	0.00027	3157	3142	0.0047

5.3.2 Simulation with Malleable Data

For simulations with malleable applications, the negotiation cost and adaptation cost of the actual prototype system for malleable application was measured. The simulator parameters were set to actual measured values. Table 5.4 shows the simulator parameter for experiments with malleable workloads presented in Table 5.2.

Table 5.4 Parameters for simulation with malleable applications

Parameter	**Value**
Minimum Negotiation Cost	0.0015 sec
Maximum Negotiation Cost	0.0015 sec
Minimum Adaptation Cost	0.002sec
Maximum Adaptation Cost	0.002 sec
Percentage of Successful Negotiation	100
Size of cluster	16 processors

Figure 5.6 to Figure 5.15 shows the comparison of start time and completion time of individual jobs of malleable workloads for data set 1 to data set 5. From the figures it can be seen that for all workloads that the difference between the start times $d_e(T_s)$ from simulator of real system is very low. Similarly for completion time the difference $d_e(T_s)$ is very low. However, for some jobs the difference is larger than expected. For example $d_e(T_s)$ for job 19 of data set 3, $d_e(T_c)$ for jobs 15, 19, 23, 26, 33, 36 of data set 2 jobs 13, 16 and 49 of data set 4 and job 36 of dataset 6 is more than 0.1. This means for these cases the simulator output varies more than 10 percent compared to the real data.

There are two explanations for this variation. First, the model and subsequently the simulator don't take into account the unpredictable system variance in the real system. For example in real system a job or the RMS may have to share a computing node with other system processes, and, as a result, a job may start or finish its computation later that expected. The average negotiation and adaptation cost from one workload (data set 1) the real system has been used. as parameters for simulation experiments, In reality these costs varies a little within a workload and from workload to workload.

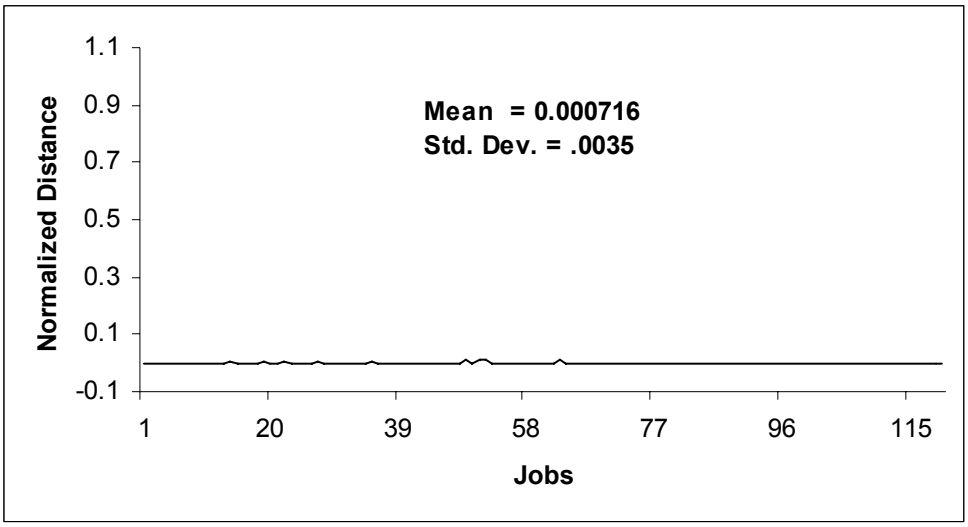


Figure 5.6 Comparison of start time of for malleable data set 1 (10% job-mix)

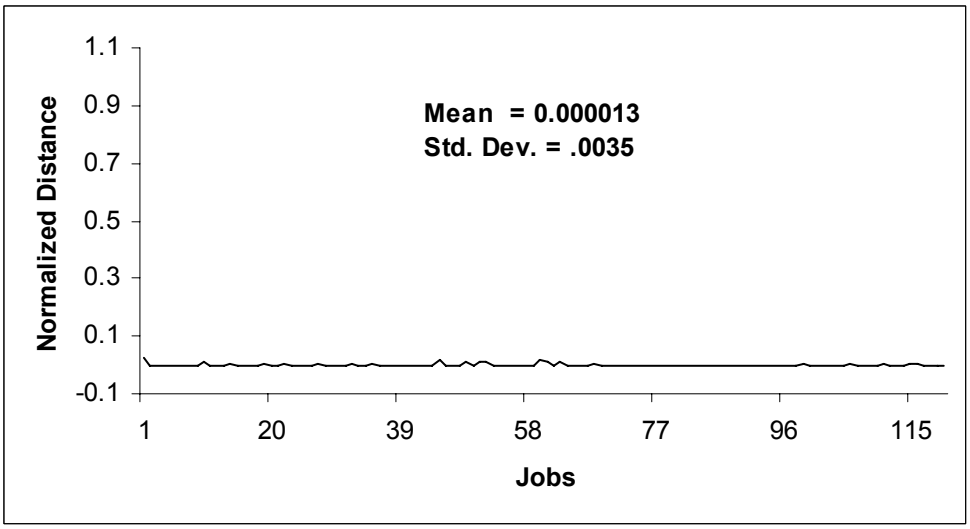


Figure 5.7 Comparison of completion for malleable data set 1 (10% job-mix)

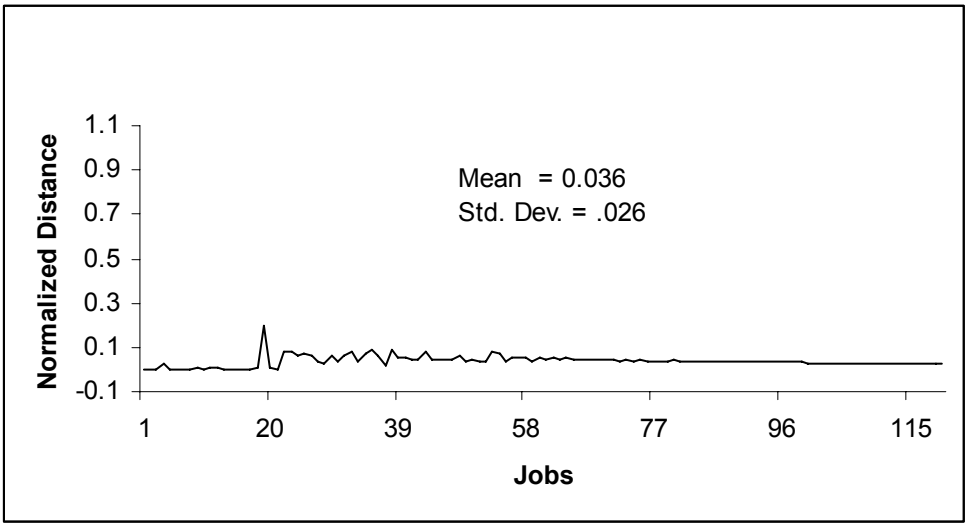


Figure 5.8 Comparison of start time of for malleable data set 2 (33% job-mix)

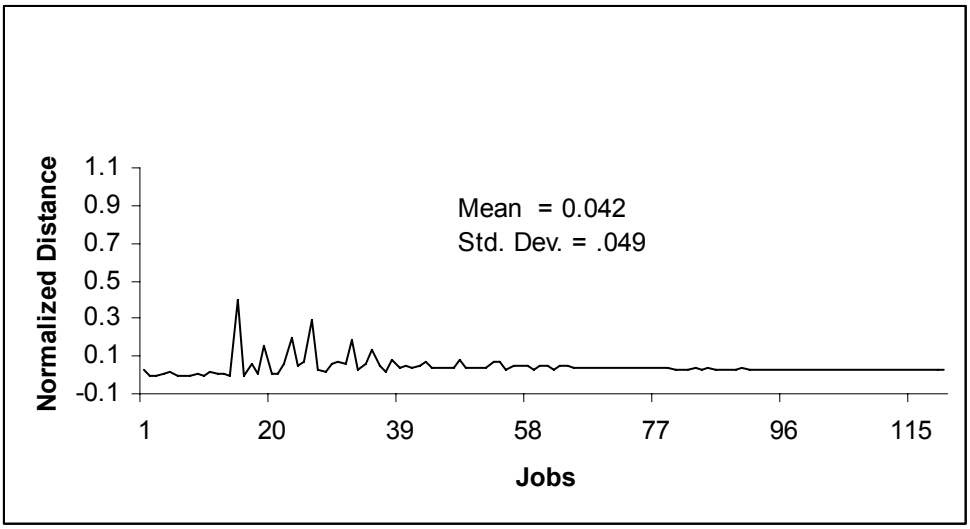


Figure 5.9 Comparison of completion for malleable data set 2 (33% job-mix)

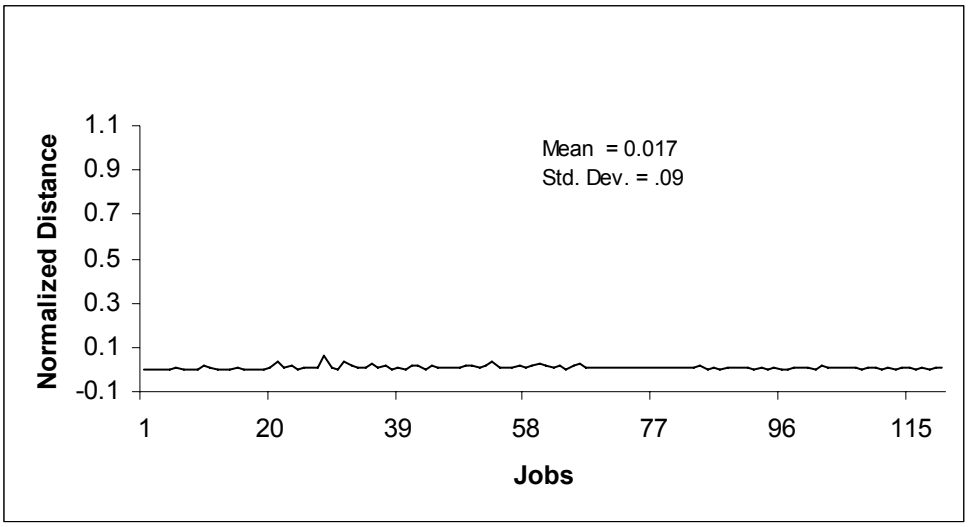


Figure 5.10 Comparison of start time of for malleable data set 3 (50% job-mix)

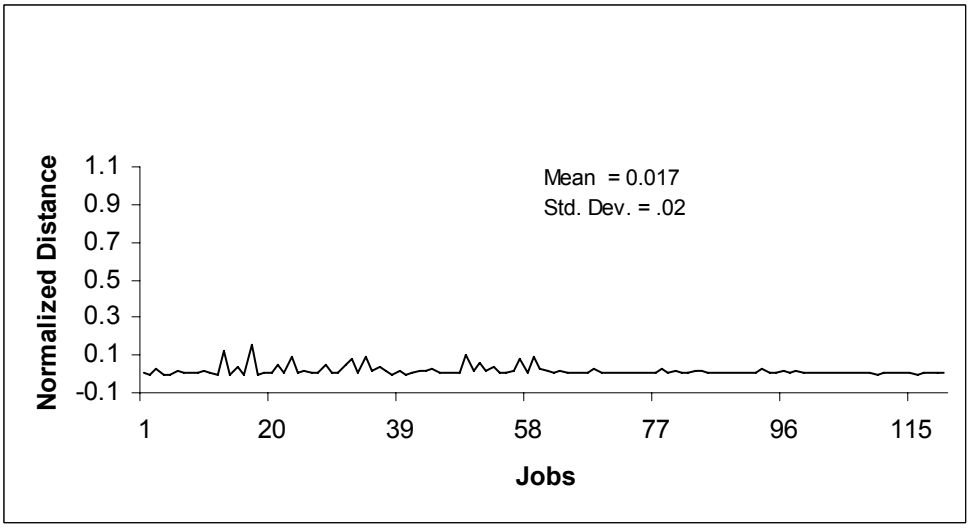


Figure 5.11 Comparison of completion for malleable data set 3 (50% job-mix)

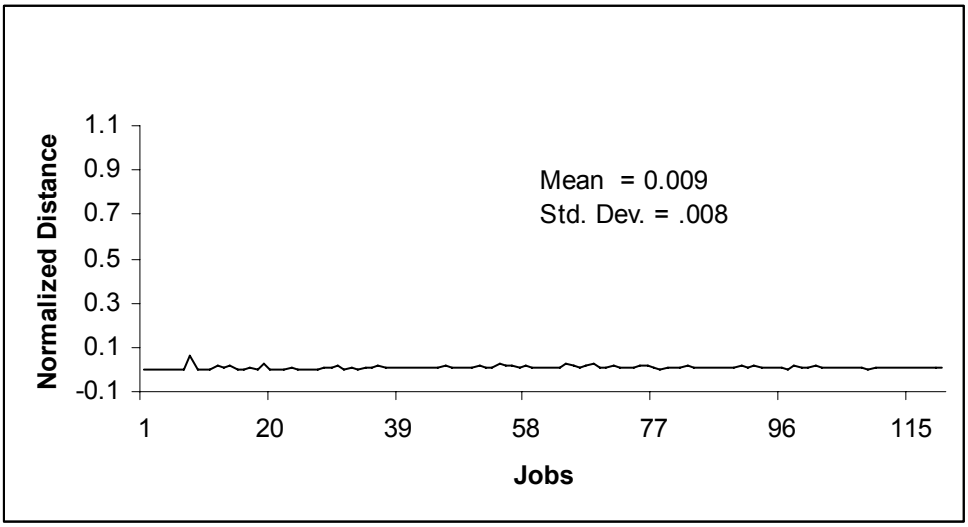


Figure 5.12 Comparison of start time of for malleable data set 4 (75% job-mix)

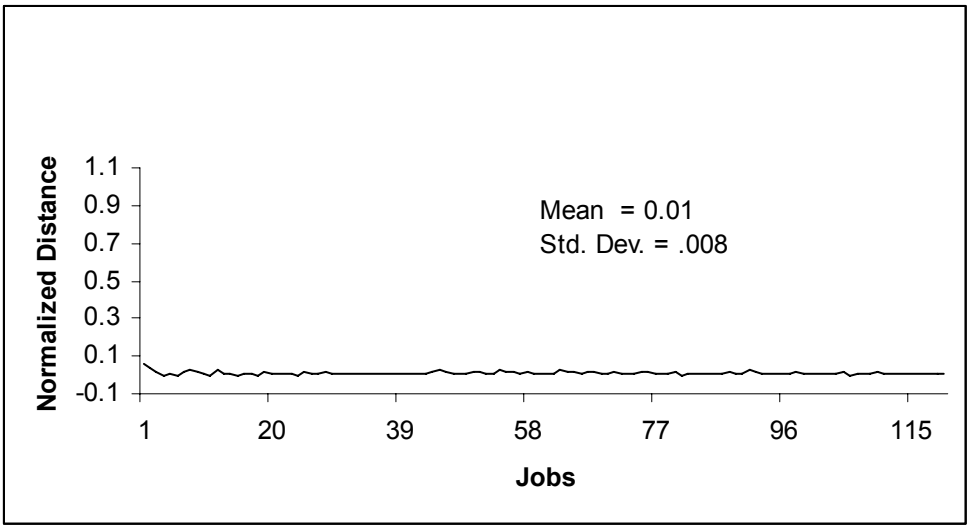


Figure 5.13 Comparison of completion for malleable data set 4 (75% job-mix)

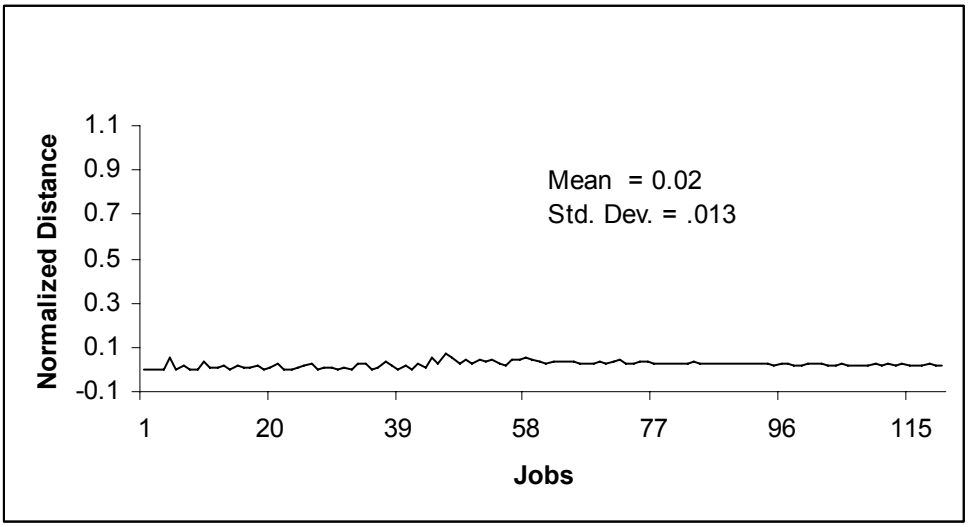


Figure 5.14 Comparison of start time of for malleable data set 5 (100% job-mix)

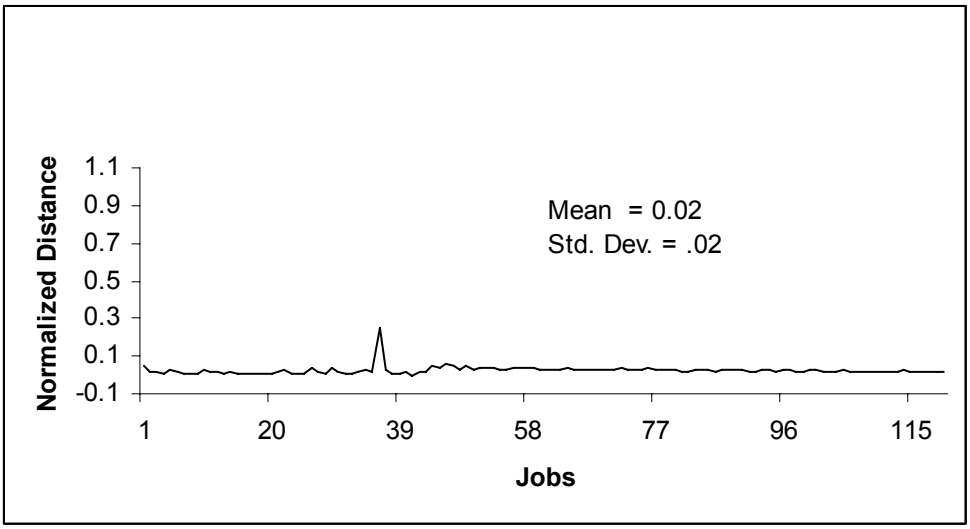


Figure 5.15 Comparison of completion for malleable data set 5 (100% job-mix)

Table 5.5 shows the mean and standard deviation of $d_e(T_s)$ and $d_e(T_c)$ for all workloads. It can be seen from the table, that the mean and standard deviation is less than 0.02 for all data sets, except for data set 2. Table 5.6, Figures 5.16 and 5.17 show the comparison of utilization and average turn around time for rigid data set and malleable

data sets. It can be seen from the table that the simulator output matches closely with actual data. Figure 5.18 shows the trend in variation of utilization with variation of percentage of malleable jobs in workload. The trend is similar in simulator and real system. In both case the utilization is about 0.873 for rigid workloads. The utilization increases as percentage of malleable job increases and reaches a saturation point of about 0.984 for 33 % malleable jobs in both cases. After the saturation point the utilization increases about 1 and 1.5 percent for simulator and real system respectively for all malleable jobs. Figure 5.19 shows the trend in variation of average turn around time with variation of percentage of malleable jobs in workload. The trend is similar in simulator and real system. The turn around time decreases as percentage of malleable job increases and reaches to a saturation point for 33 % malleable jobs in both cases. After the saturation point the turn around time do not change significantly as the percentage of malleable job increases.

Table 5.5 Mean and standard deviation of distance of start time and completion time

Data Set	$d_e(T_s)$		$d_e(T_c)$	
	Mean	Std. Dev.	Mean	Std. Dev
1	0.0007	0.002	0.001	0.003
2	0.036	0.025	0.042	0.049
3	0.017	0.09	0.016	0.025
4	0.009	0.007	0.01	0.008
5	0.022	0.013	0.024	0.023

Table 5.6 Comparison of utilization and average turn around time between simulator output and real system

%of Job Mix	Utilization			Turn Around Time		
	Real	Simulator	Normalized Distance	Real System	Simulator	Normalized Distance
0	0.873344	0.87367	0.000373	1968.53	1997.89	0.014915
10	0.908738	0.90933	0.000652	1849.26	1863.08	0.007473
33	0.983825	0.98461	0.000798	1674.6	1680.89	0.003756
50	0.984702	0.98407	0.000642	1641.47	1666.1	0.015005
75	0.991332	0.9904	0.00094	1657.91	1684.91	0.016286
100	1.000000	0.9945	0.00550	1657.45	1701.91	0.026824

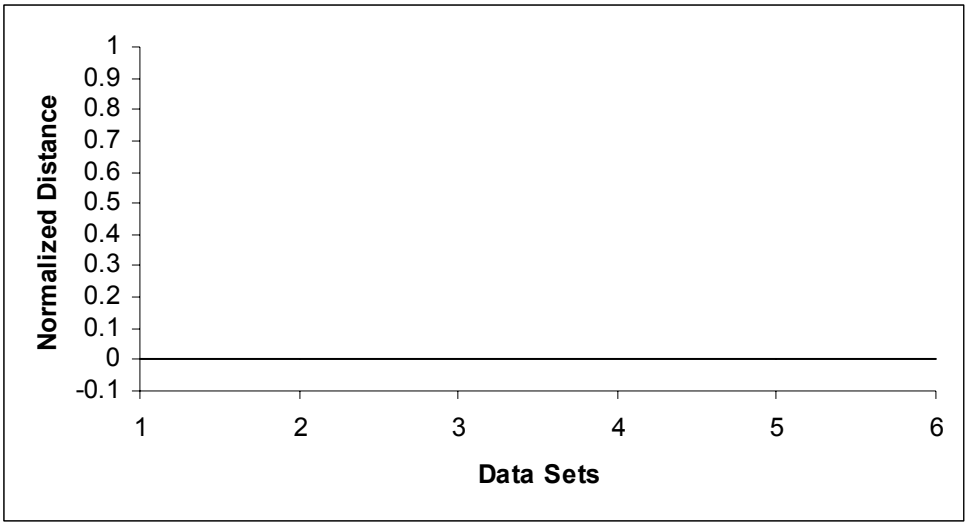


Figure 5.16 Comparison of utilization between simulator output and real system for data sets 1-5

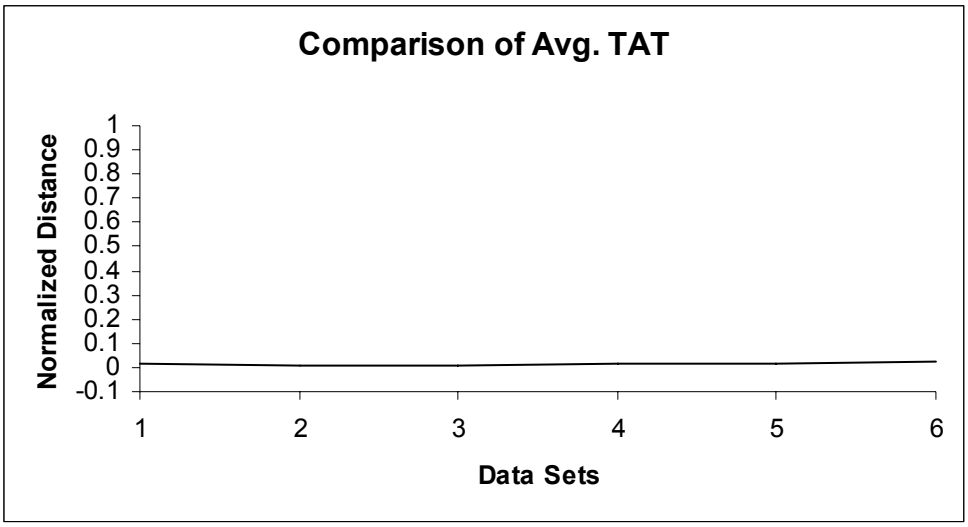
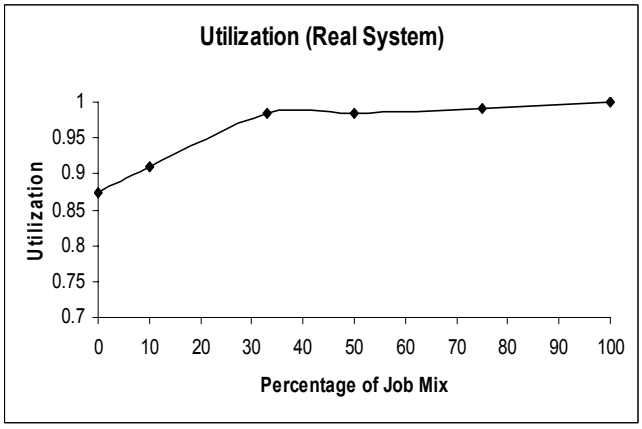
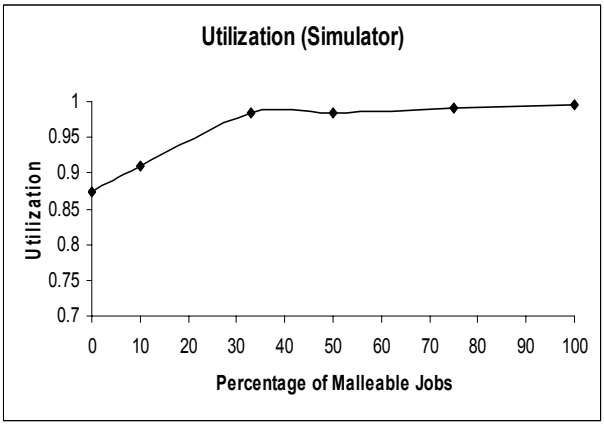


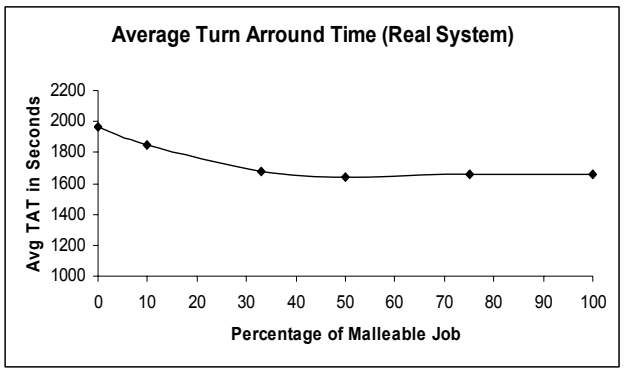
Figure 5.17 Comparison of average turn around time between simulator output and real system for data sets 1-5

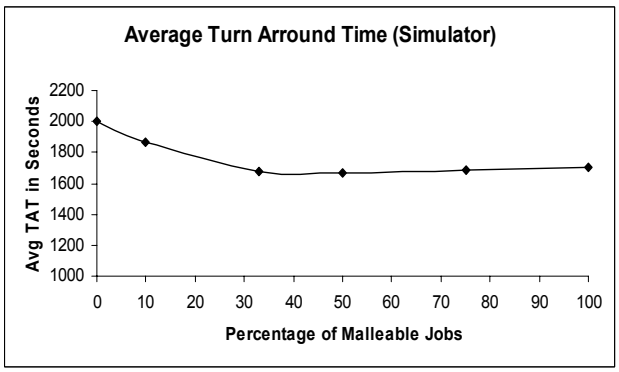




(a) (b)

Figure 5.18 Comparison of utilization between simulator and prototype





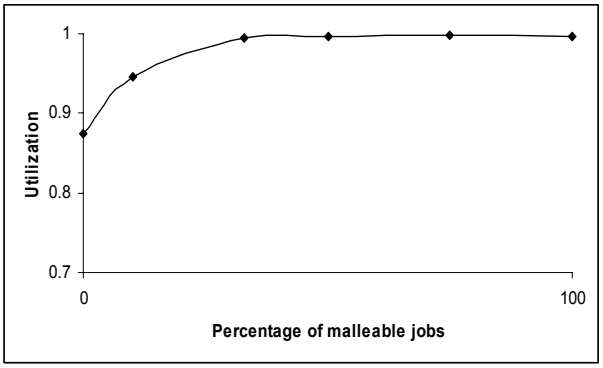
(a) (b)

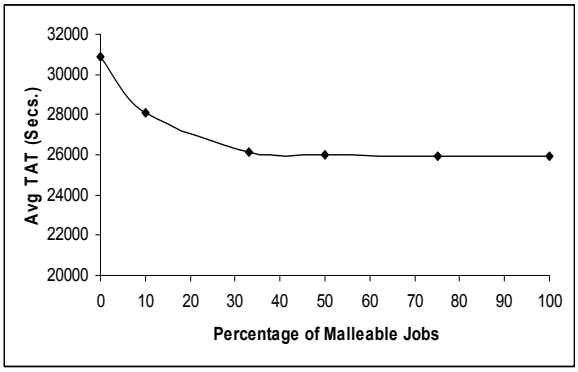
Figure 5.19 Comparison of average turn around time between simulator and prototype

Table 5.7 and figure 5.20 shows the trend in utilization and average turn around time for synthetic malleable workloads presented in table 5.3. From the table and the figure it can be seen that the trends in utilization and average turn around time for the synthetic malleable workloads is similar to those of real system. For all rigid jobs the utilization is about 0.87 which increases with the increase of percentage of malleable jobs and reaches to a saturation point of 0.99 for 335 malleable jobs. The utilization does vary significantly after that point. Similar behavior can be observed in case of average turn around time.

Table 5.7 Trend in utilization and average turn around time for synthetic workload

% of job Mix	Utilization	Average Turn Around Time (secs.)
0	0.874	30850
10	0.946	28088
33	0.995	26112
50	0.996	26026
75	0.998	25938
100	0.997	25938





(a) Utilization (b) Average Turn Around Time

Figure 5.20 Variation of utilization and average turn around time with the variation of percentage of malleable job for 1200 Synthetic workloads

5.4 Summary

From the results presented in section 5.3 it can be seen that the start time and completion for jobs from simulator output closely match with those of simulator outputs for both rigid and malleable workloads. In case of start time the variation is more than 10 percent for one job out 720 jobs. For completion time 10 jobs out of 720 has more than 10 % variation. As explained in section 5.3.2 the reason for these exceptions are unpredictable system variance, variation in negotiation cost, and adaptation cost in real

system. However, the difference in mean and standard deviation of distance between simulator and real system is very low (less than 0.02).

For all workload the distance between simulator and real system for both utilization and average turn around time is very low. Moreover the trend in variation of utilization and average turn around time with the variation of percentage of malleable jobs is similar. Also from the results of experiments with synthetic load it can be observed that the trends in variation of utilization and turn around time are similar to those of the real system, which signifies that the simulator scales properly. From these results it can be concluded that the simulator faithfully approximates a real system.

CHAPTER VI

EXPERIMENTAL RESULTS

In Chapter IV the design and implementation of a simulator for adaptive parallel system has been presented. The validation of the simulator has been discussed in Chapter V. In this chapter, the design, results and analysis of the simulation experiments to investigate the impact of system parameters on performance that were conducted using the validated simulator have been presented. Section 6.1 discussed the design of the experimental setup. A discussion on the dataset that were used for the simulation experiment is presented in section 6.2. The experimental results are discussed in sections 6.3 to 6.6 and finally a summary is presented in section 6.7.

6.1 Experimental Design

The overall purpose of simulation experiments is to determine how the model parameters impact application and system performance in an adaptive parallel system. To achieve this goal a set of model parameter to study must be determined as well as how these parameters will be varied during the simulation experiments and the range of values of these parameters.

An adaptive parallel system consists of an adaptive RMS and a malleable workload. A malleable workload contains rigid as well as malleable jobs and the flexibility of malleable jobs can vary from workload to workload. So it has been decided that impact of the number of malleable jobs in the workload and the flexibility of the malleable jobs would be the parameter to be investigated. The execution of malleable jobs consists of phases and jobs adapt at a phase boundary by shrinking or expanding. The phase change of a malleable job involves negotiation with the RMS and reconfiguration of the job to utilize a new set of processor. As a result it has been decided to investigate the impact of negotiation and adaptation on performance.

The impact of the following parameters on system and application performance has been investigated through simulation experiments. 1) The number of malleable jobs in the workload, 2) flexibility of malleable jobs, 3) cost of negotiation, and 4) cost of adaptation of malleable jobs. To determine the impact of a parameter during the simulation experiments the value of the parameter has been varied while other parameters have been kept constant. For example to measure the impact of number of malleable jobs on performance, the number of malleable jobs in the workload has been varied, while the flexibility of malleable jobs, the cost of negotiation, and cost of adaptation have been kept constant. Then for each variation of number of malleable jobs, simulation experiments have been conducted by varying the cost of negotiation. In this manner one by one all the parameters have been varied while the rest of the parameters has been kept

constant. The experiments have been designed this way to isolate the impact of one parameter on performance. For each simulation run with a workload the following statistics has been collected: system utilization, average turn around time, average wait time, average execution time, total number of negotiation and total number of adaptation.

The range of values of the parameters that have been selected for the simulation experiments are as follows:

Number of malleable jobs: The number of malleable jobs in a workload has been has been selected as percentage of total number of jobs in the workload. The number of malleable jobs has been varied from 10% to 100% in steps of 10.

Flexibility of Malleable jobs: The flexibility of malleable jobs in the workload has been varied in two ways. In one approach the minimum number of processors has been kept constant while the maximum number of processors has been varied. This has been done to measure impact of flexibility on performance. Experiments have been conducted with following values of flexibility range: 2-16, 2-32, 2-64, 2-80, 2-96, 2-112, and 2-128. In the second approach the flexibility has been kept constant while the minimum number processors of the range have been varied. This has been done to determine the impact of the values of the minimum processor on performance for same flexibility. Experiments have been conducted with following values of flexibility range: 4-130, 8-134, 12-138, and 16-142.

Negotiation Cost: Experiments have been conducted with the following values of negotiation costs: 0.0015 second, 0.003 second, 0.006 second, 0.006 second, 0.0012 second, 0.012 second, 0.024 second, 0.048 second, 0.96 second, 0.2 second, 0.4 second, 0.8 second, 2 seconds, 4 second and 8 seconds. From the experiments with the prototype system in 16 processor dedicated cluster, it has been found that the average negotiation cost is 0.0015 seconds on average. In the experiments all negotiations were two round negotiations and applications responded to a negotiation without delay. For this reason 0.0015 second has been selected as the lower limit of the negotiation cost. The other values of the negotiation cost have been selected by doubling the previous values. The upper limit of the negotiation cost has been selected as 8 seconds, which is 5333 times larger than the lower limit. It has been assumed that this is large enough value that may occur in a real system.

Adaptation Cost: Experiment have been conducted with the following values of adaptation costs: 0.002 second, 0.004 second, 0.008 second, 0.01 second, 0.02 second, 0.04 second, 0.08 second, 0.2 second, 0.4 second, 0.8 second, 1 second, 2 seconds, 4 seconds, and 8 seconds. Like the negotiation cost the lower limit of the adaptation cost has been selected as the adaptation cost measured in the prototype system and the upper limit has been selected as 4000 times larger than the lower limit.

6.2 Experimental Data

As explained in Chapter III (sub section 3.4.3) for this research, a workload model has been developed by extending Downy's model. Synthetic data for the simulation experiments has been generated using this model. To generate a workload the model takes the following parameters as input: 1) number of jobs in the workload, 2) the minimum execution time and maximum execution time of the jobs, 3) the range of the default processors, 4) the number of malleable jobs in the workload, and 5) the flexibility range (minimum and maximum number of processors) of malleable jobs. Each job in a workload generated by the model contains the following information.

1. Arrival time of the job
2. Type of the jobs (Rigid/Malleable)
3. Number of processors required.
4. Execution time of the job on the required number of processors.
5. Flexibility range (for malleable jobs)

A workload with all rigid jobs has been generated with 1000 jobs. The minimum run time and maximum run time has been fixed to 100 second and 3600 seconds respectively, and default processor range has been fixed to 16-128 processors. The rigid workload has been used as the baseline workload. Several malleable workloads have been generated from the base line workload by varying the flexibility range and number of malleable jobs with in the range mention in section 6.1.

There are two justifications for selecting the size of the workload as 1000 jobs. In most super computer centers on average about 200 jobs are submitted in a day in one cluster [65]. Workloads with 1000 jobs have been selected to simulate about five days of workload of a cluster in a typical super computer center, which seems reasonable. Simulation experiments with larger workload size of 1000 jobs, 3000 jobs and 5000 jobs have been conducted. Table 6.1 and figure 6.1 show the result of these experiments. It has been found that increasing the workload size do not impact the performance in any significant way. Increasing workload size would increase the simulation time, data file size and result processing time.

Table 6.1 Utilization for workload size 1000 jobs, 3000 jobs and 5000 jobs.

% of Malleable Job	Utilization		
	1000	3000	5000
0	0.84381	0.84693	0.84489
10	0.92574	0.92725	0.92658
20	0.99747	0.99286	0.99134
30	0.9993	0.99969	0.9997
40	0.99873	0.99973	0.99989
50	0.9998	0.9996	0.99955
60	0.9998	0.9999	0.99994
70	0.99849	0.9994	0.99989
80	0.99874	0.99924	0.99988
90	0.99979	0.99955	0.99977
100	0.99974	0.99988	0.99978

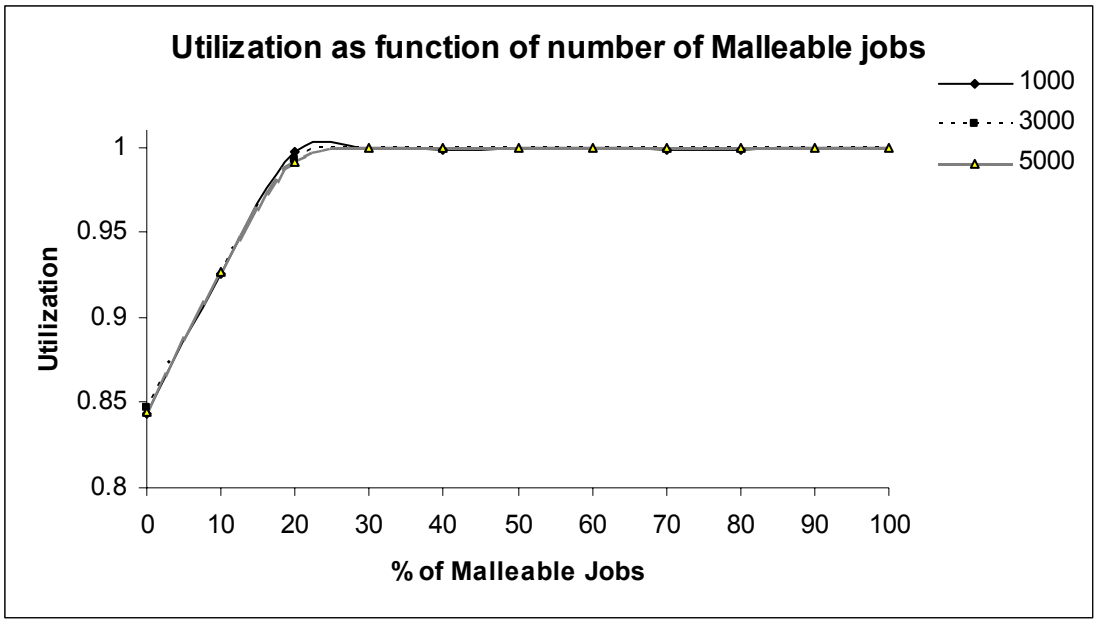


Figure 6.1 Utilization for workload size 1000 jobs, 3000 jobs and 5000 jobs.

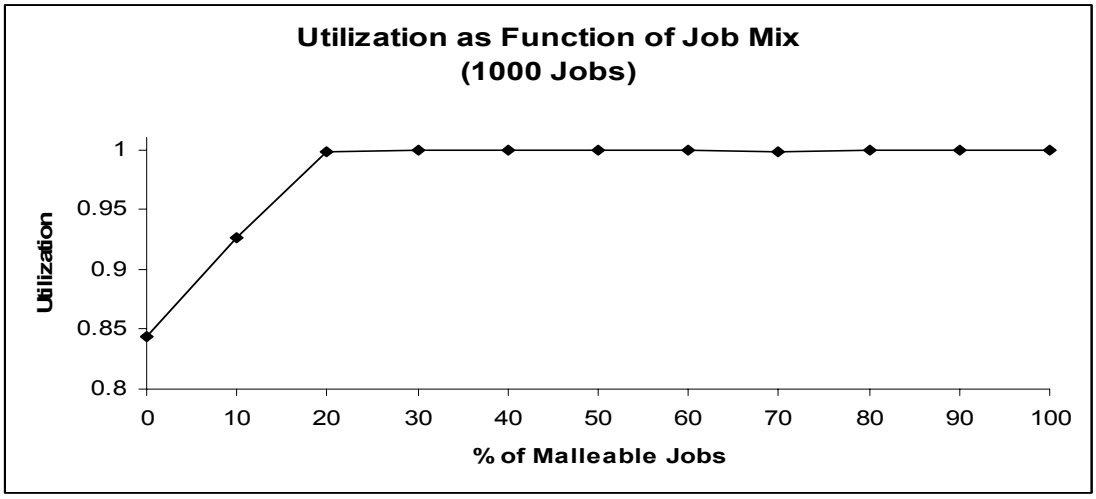
6.3 Performance with the Variation of Number of Malleable Jobs in Workload

To investigate the impact of number of malleable jobs in the workload on performance, simulation experiments with all rigid workload, and workloads containing different percentage of malleable jobs have been conducted. The rigid workload is used as baseline to compare performance with malleable workloads. Each workload is simulated twice, once on a cluster with 256 nodes and the second time on a cluster with 512 nodes. Table 6.2 shows the variation of performance as the percentage of malleable jobs increases in the workload. Figure 6.2 and 6.3 graphically shows the information presented in Table 6.2. From the results presented in table 6.2 and figure 6.2 it can be seen that the system utilization increases as the number of malleable jobs in the workload

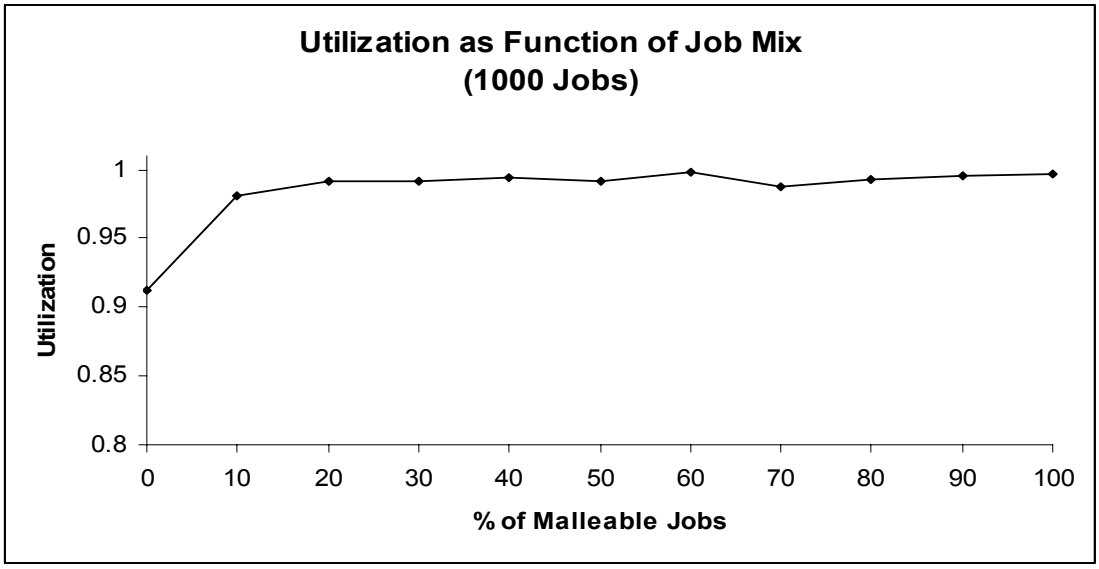
increases. The utilization saturates at one point and it does not vary significantly after the saturation.

Table 6.2 Variation of performance with change in number of malleable jobs in the workload. Negotiation cost: 1.5 ms, adaptation cost: 2 ms.

% of Malleable jobs	Cluster Size : 256 Nodes		Cluster Size: 512 Nodes	
	Utilization	Avg. TAT (Secs)	Utilization	Avg. TAT (Secs)
0	0.84381	109741	0.91165	47580
10	0.92574	101443	0.98121	44222
20	0.99747	93002	0.99186	42997
30	0.99930	92724	0.99112	43387
40	0.99873	91822	0.99447	42441
50	0.99980	91913	0.99178	42601
60	0.99980	91607	0.99860	42293
70	0.99849	89184	0.98803	40295
80	0.99879	88069	0.99327	38929
90	0.99979	89398	0.99488	38909
100	0.99974	86115	0.99629	38235



(a) Cluster Size: 256 Nodes



(b) Cluster size 512 nodes

Figure 6.2 Variation of utilization with the number of malleable jobs in the workload. Negotiation cost: 1.5 ms, adaptation cost: 2 ms

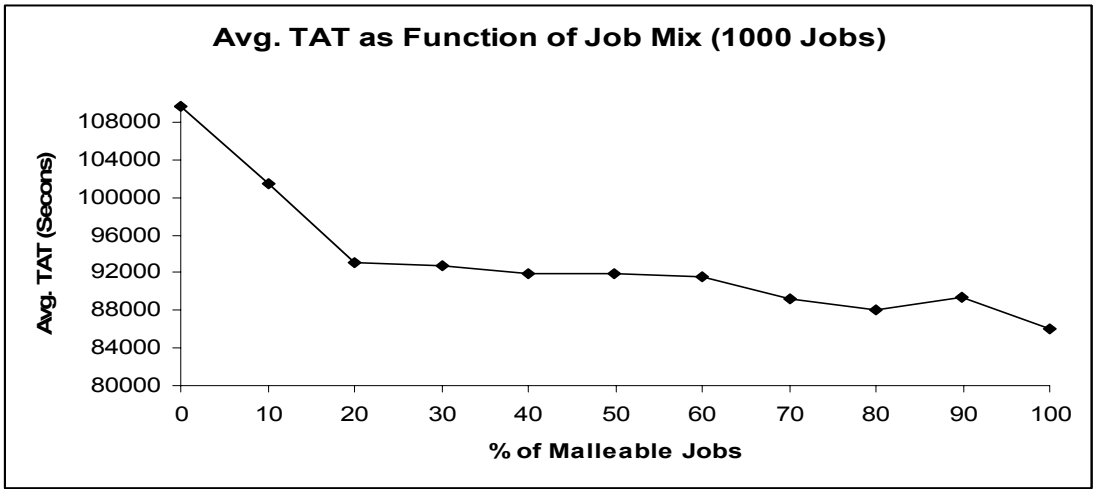
The utilization at saturation is above 99%, which indicates that the cluster is fully utilized after saturation. The saturation occurs at 20% job mix. The improvement in utilization over all rigid workload is about 15% in a 256 node cluster and 8% in a 512 node cluster. From the results it can be seen that the improvement over the same rigid workload may vary from cluster to cluster. The most important finding of these experiments is that irrespective of cluster size or base line utilization with all rigid workload, it is possible to achieve maximum utilization with a malleable workload. The maximum possible utilization can be achieved with relatively few malleable jobs (20% in our experiments).

In this research we are interested in the gap between the performance of an all rigid workload and the maximum achievable performance. We want to investigate what parameters of a workload and RMS impact this difference and how these parameters impact the performance.

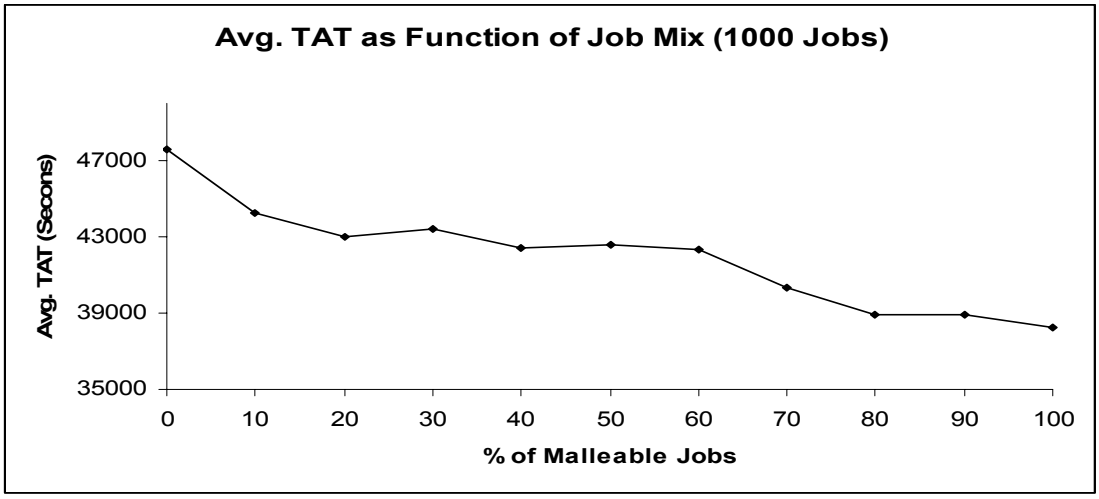
Table 6.2 and Figure 6.3 present the impact of malleable jobs on average turn around time. From the results it can be seen that the average turn around time decreases as the number of malleable jobs in the workload increases. The decrease is initially high and gradually the decrease becomes low as the number of malleable job increases further. Unlike utilization no saturation can be observed in case of average turn around time. For a 256 node cluster, the average turn around time improves about 15% for a 20% job mix, while the improvement was about 10% for a 512 node cluster. For all malleable jobs the improvement was 27% for 256 nodes cluster and 24 % for 512 nodes cluster.

The improvement in performance in the presence of malleable jobs in the workload comes at the expense of the execution time of the malleable jobs. In malleable mode, the execution time of a job increases in general. Since the scheduler has to shrink a malleable job to accommodate the pending jobs, on average, a job runs on a lower number of processors in malleable mode than it does in rigid mode. The graph of figure 6.4 shows the average execution time, the average turn around time, and the average wait time as function of number of malleable jobs in the workload. From the figure it can be seen that as the number of malleable jobs increases, the job execution time increases on

average. However, as the number of malleable job increases, the average turn around time decreases. This is because with higher number of malleable jobs, a job has to wait less in the pending queue.



(a) Cluster Size: 256 Nodes



(b) Cluster Size: 256 Nodes

Figure 6.3 Variation of average turn around time with the number of malleable jobs for data set 1. Negotiation cost: 1.5 ms, adaptation cost: 2 ms

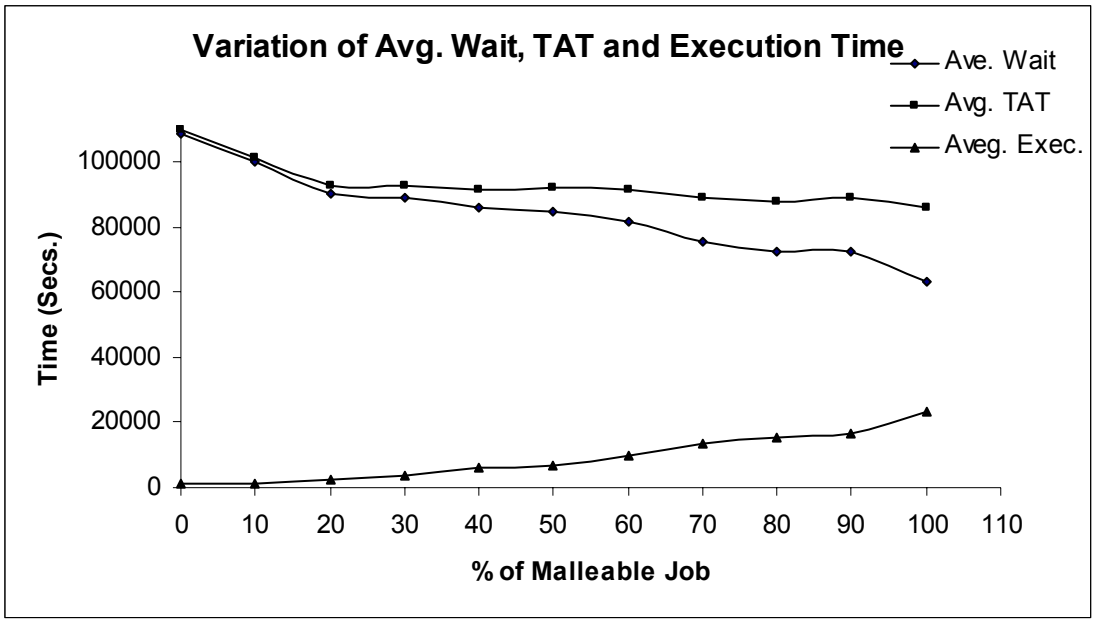


Figure 6.4 Variation of average execution, wait, and turn around time as the number of malleable jobs increases. Flexibility range: 2- 128, negotiation cost: 1.5ms, adaptation cost: 2ms, cluster size: 256 nodes.

6.4 Performance with the Variation of Flexibility of Malleable Jobs

To measure impact of flexibility of malleable jobs on performance, two sets of experiments were conducted. In the first set of experiments, the minimum number of processors was varied keeping the flexibility (difference between the maximum and minimum number of processors) constant. The goal was to investigate the impact of the minimum number of processors on performance for constant flexibility. For example if there are two malleable workloads, one has malleable jobs with flexibility range 10 - 30 processor, and other has malleable jobs with flexibility range 20 -40 processors. The malleable jobs in both workloads have flexibility of 20 processors, but in one the

minimum number of processors is 10 and in the other it is 20. The goal of these experiments is to determine if the performance of these two workloads is different?

In the second set of experiments the minimum number of processors was kept constant and the flexibility was varied by changing the maximum number of processors. The goal of these experiments was to measure the impact of variation of flexibility on performance while the minimum number of processors is constant. The two sets of experiments were conducted with 256 nodes cluster.

Table 6.3 shows the minimum processors for first set of experiments. Table 6.4 presents the impact of minimum number processors of malleable jobs on performance for constant flexibility.

Table 6.3 Flexibility range for experiment set one

Minimum Processor	Maximum Processors	Flexibility
2	128	126
4	130	126
8	134	126
12	134	126
16	134	126

Table 6.4 Impact of minimum number of processor on performance. Flexibility: 126 processors, negotiation cost: 1.5 ms, adaptation cost: 2ms.

Min. Proc.	Utilization					Average Turn Around Time (secs)				
	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%
2	0.93	0.997	0.999	0.999	0.999	101443	93002	92724	91822	91913
4	0.92	0.993	0.999	0.999	0.999	101744	93733	92739	92330	92231
8	0.92	0.982	0.997	0.999	0.999	102379	94956	93043	92599	92526
12	0.91	0.973	0.991	0.999	0.995	102688	95712	93537	92742	93050
16	0.90	0.965	0.983	0.998	0.993	103083	96507	94111	92804	93188

The results of table 6.4 are presented graphically in Figure 6.5 and Figure 6.6. From the results it can seen that utilization decreases as the number of minimum processors of malleable jobs increases, while the percentage of malleable jobs in the workload is kept constant. The variation of utilization is more pronounced for 10% and 20% job mixes. For job mixes above 40%, the utilization is relatively unaffected by the number of minimum processors of malleable jobs. From figure 6.6 similar trends can be observed in case of average turn around time.

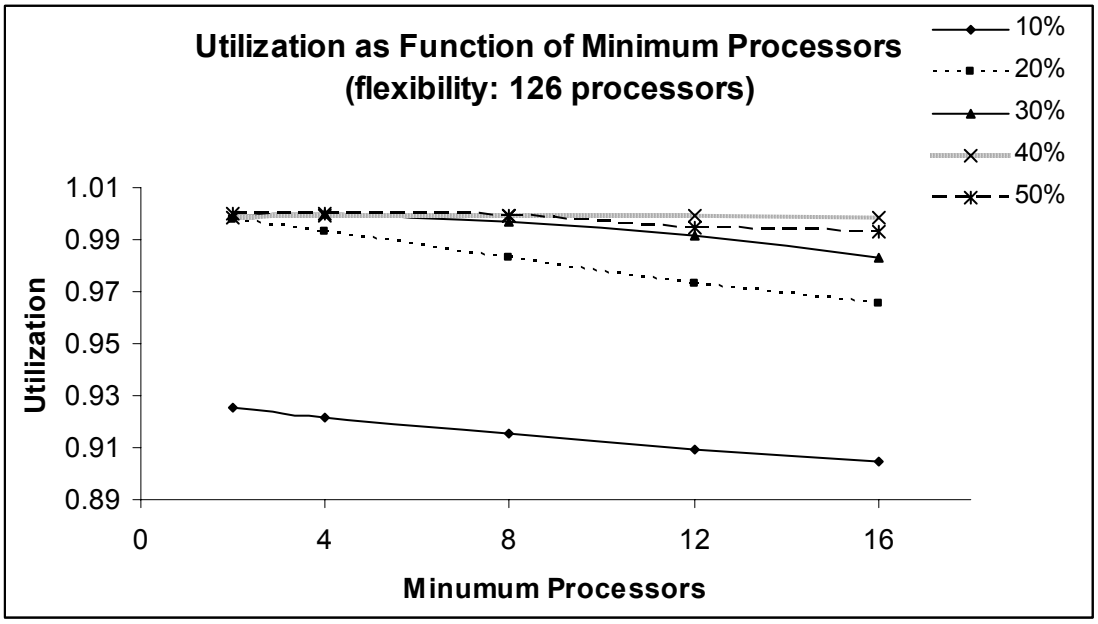


Figure 6.5 Variation of utilization with minimum number of processor on performance for data set 1. Flexibility: 126 processors, negotiation cost: 1.5 ms, adaptation cost: 2ms.

The reason for this behavior is that with a lower minimum processor requirement, the scheduler has a higher probability of scheduling a pending malleable job. For example, consider the scenario that there are 2 idle processors, all the running jobs are either rigid and/or malleable running on minimum processors, and the job at the head of the pending list is malleable. In such case, if the flexibility range of the pending malleable job is 2-6, it could be started immediately, but if the flexibility range is 4-8, the job cannot be started. It has to wait till a running job exits and releases at least 2 or more processors making the number of idle processors 4 or more. As the number of malleable jobs increases, the probability of a malleable job running on more than its minimum

processors increases, which in turn increases the probability of releasing 2 processors by shrinking running malleable to start the pending malleable jobs.

In case of workloads with a low number of malleable jobs, at any particular time there are more rigid jobs running and waiting, compared to malleable jobs. In this situation to utilize any idle processor by scheduling a pending rigid job, the scheduler needs to preempt required number of processor by shrinking running malleable jobs. Since there are few malleable jobs running, if the lower bound of the flexibility range is high, the scheduler has less opportunity of preempting required number of processors, compared to the situation when the lower bound is low. As a result for workloads with low number of malleable jobs, the utilization decreases as the lower bound of flexibility range of malleable job increases. However, as the number of malleable jobs in the workload increases, the number of running malleable job also increases. In this situation the scheduler has higher probability of preempting required number of processors to accommodate the pending job, and thereby utilizing the idle processor. As a result the lower bound of flexibility range of malleable jobs doesn't have any significant impact on performance when the workload contains a high number of malleable jobs.

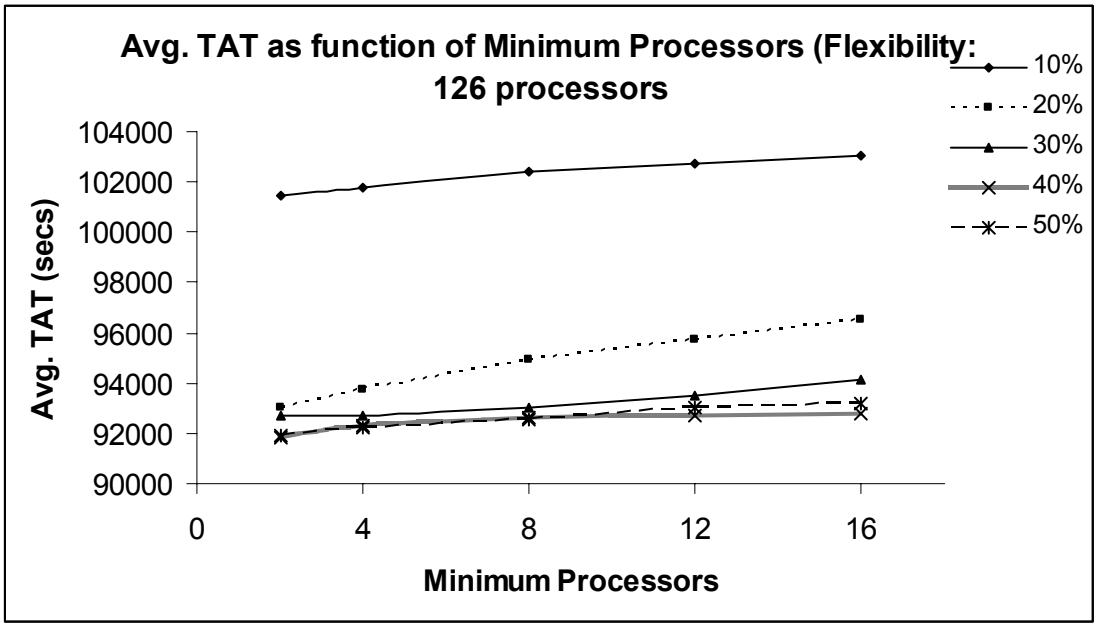


Figure 6.6 Variation of avg. TAT with minimum number of processor on performance for data set 1. Flexibility: 126 processors, negotiation cost: 1.5 ms, adaptation cost: 2ms.

Experimental result presented in Tables 6.5 and 6.6 show the impact of flexibility on performance, for constant minimum number of processors. Figures 6.7, 6.8 and 6.9 present the results of Tables 6.5 and 6.6 graphically. Figure 6.7 shows the utilization as function of percentage of malleable jobs in the workload for different flexibility. Figure 6.8 shows utilization as function of flexibility for different job mix. From the results presented in Figure 6.7 it can be seen that the utilization increases and saturate at 20% job mix. The utilization at the saturation point increases as the flexibility increases. The utilization at the saturation point reaches the maximum possible at a flexibility of 62 processors. Figure 6.8 (showing the same result in a different way) clearly demonstrates

that, for a given job mix, the utilization improves and reaches a saturation point at a certain flexibility. Increasing flexibility further does not improve the utilization.

The reason for this behavior is that at lower flexibility, the running malleable can release fewer processors compared to higher flexibility. As a result the probability of accommodating a pending job to utilize idle processors becomes low at lower flexibility. As the flexibility increases the running malleable jobs can release more processors and they can also expand more to utilize idle processors.

From the result presented in Table 6.6 and Figure 6.9 it can be seen that the change in flexibility does not impact the average turn around time in any significant way. For a low minimum number of processors (2 in these experiments), only the number of malleable jobs in the workload impacts the average turn around time.

Table 6.5 Impact of flexibility of malleable job on utilization. Minimum processors: 2 negotiation cost: 1.5 ms, adaptation cost: 2ms.

%of Malleable Jobs	Utilization							
	14 (2-16)	30 (2-32)	46 (2-48)	62 (2-64)	78 (2-80)	94 (2-96)	110 (2-112)	126 (2-128)
10	0.86461	0.90117	0.91404	0.91949	0.92277	0.92494	0.92574	0.92574
20	0.92978	0.97413	0.98862	0.99095	0.99454	0.99745	0.99683	0.99747
30	0.94403	0.97482	0.98663	0.99048	0.99409	0.99677	0.99813	0.9993
40	0.94996	0.97972	0.98922	0.9941	0.9985	0.99813	0.99824	0.99873
50	0.94501	0.97416	0.9877	0.99301	0.99636	0.99802	0.99976	0.9998
60	0.9504	0.98193	0.9912	0.99749	0.99825	0.99873	0.9994	0.9998
70	0.94572	0.98252	0.98595	0.98888	0.99312	0.99574	0.99723	0.99849
80	0.94963	0.97515	0.98865	0.99431	0.99825	0.99858	0.9984	0.99874
90	0.94424	0.97468	0.98505	0.99232	0.99708	0.99821	0.99948	0.99979
100	0.9555	0.98035	0.98943	0.99612	0.99893	0.99856	0.99935	0.99974

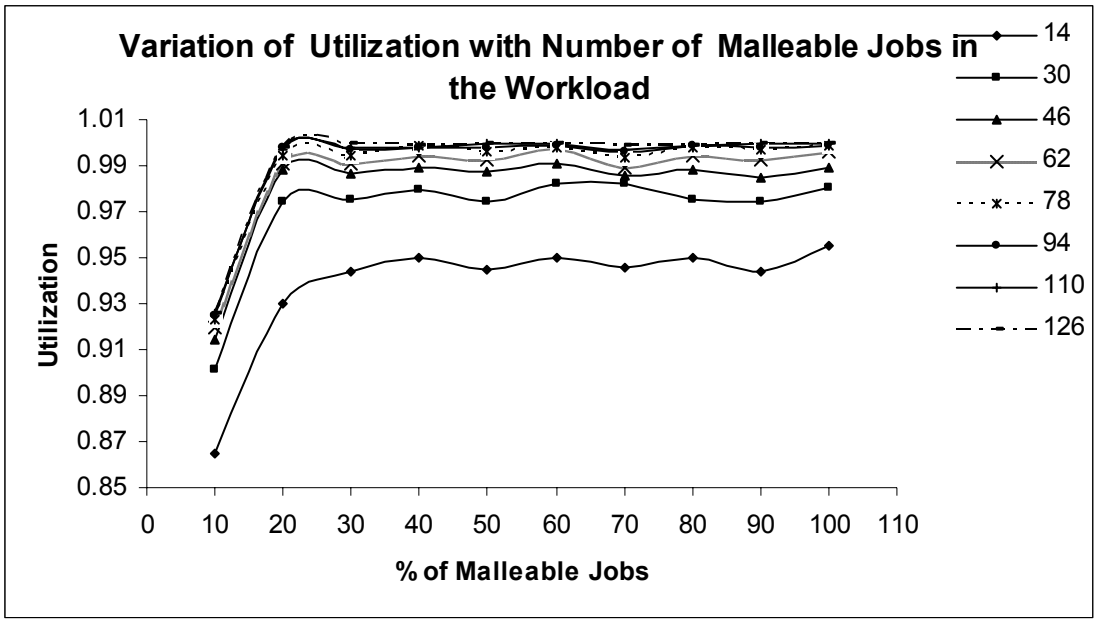


Figure 6.7 Variation of utilization as the number of malleable jobs changes in the workload for different flexibility. Minimum processors: 2, negotiation cost: 1.5 ms, adaptation cost: 2ms.

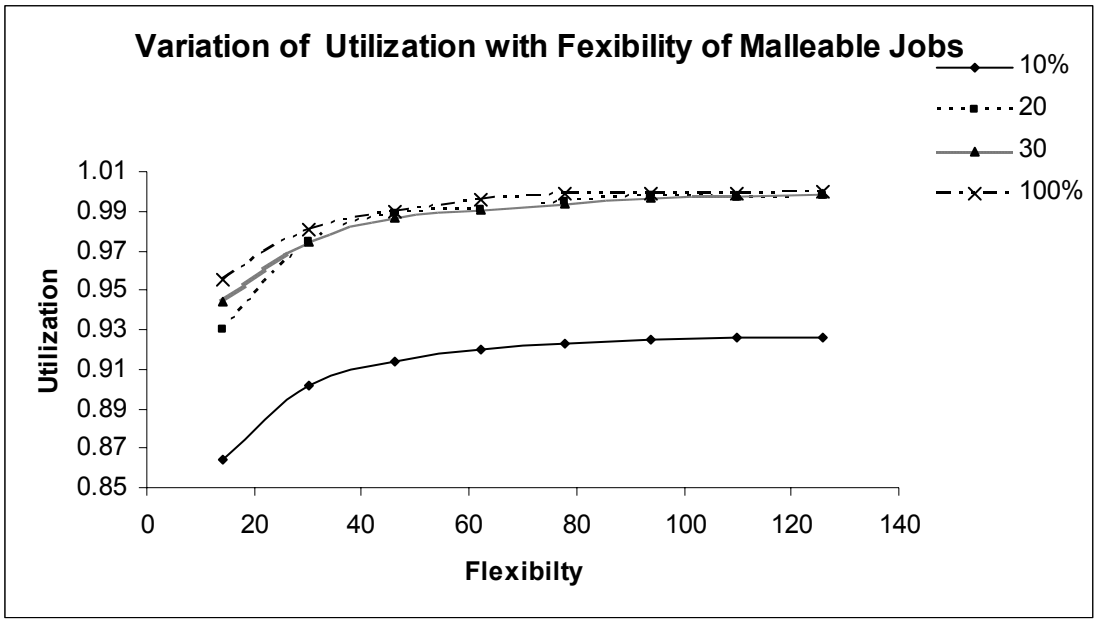


Figure 6.8 Variation of utilization with flexibility for different job mix. Minimum processors: 2, negotiation cost: 1.5 ms, adaptation cost: 2ms

Table 6.6 Impact of flexibility of malleable job on avg. TAT. Minimum processors: 2 negotiation cost: 1.5 ms, adaptation cost: 2ms.

%of Malleable Jobs	Average Turn Around Time (Seconds)							
	14 (2-16)	30 (2-32)	46 (2-48)	62 (2-64)	78 (2-80)	94 (2-96)	110 (2-112)	126 (2-128)
10	102390	101677	101409	101487	101463	101444	101443	101443
20	93540	92894	92898	92919	92900	92917	93008	93002
30	92640	92701	92699	92698	92717	92726	92719	92724
40	91831	91748	91843	91797	91788	91813	91813	91822
50	91965	91893	91885	91914	91872	91911	91915	91913
60	91400	91590	91423	91385	91458	91442	91553	91607
70	88826	89074	89240	89422	89177	89150	89306	89184
80	88184	88360	88256	87996	88074	88262	88015	88069
90	89063	89088	89546	88863	88814	89248	89224	89398
100	86293	86160	86136	86124	86125	86122	86118	86115

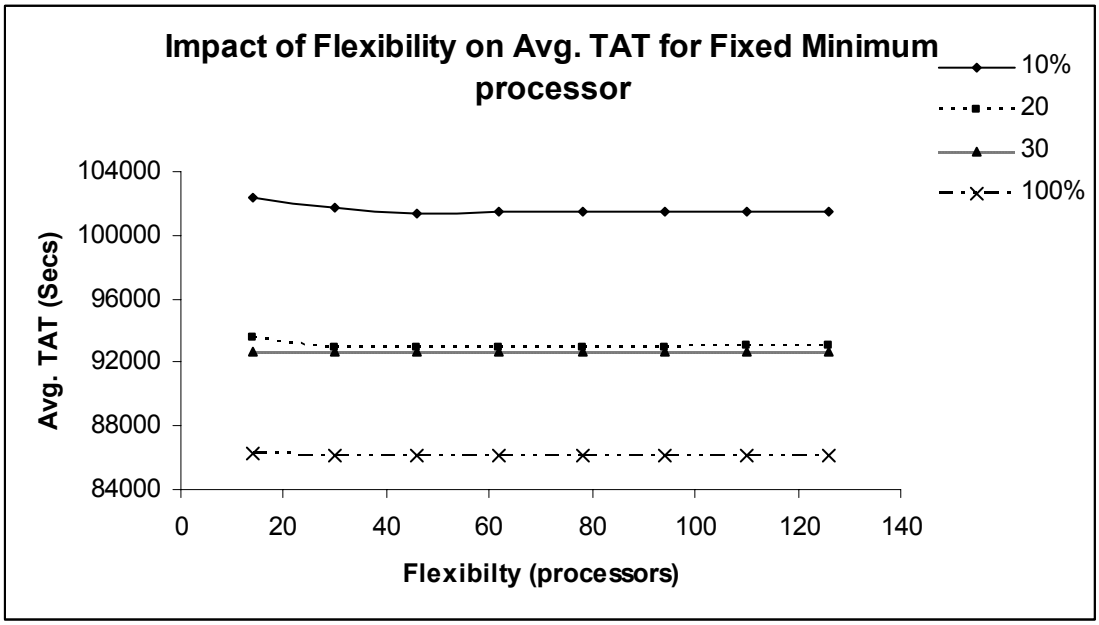


Figure 6.9 Impact of flexibility of malleable jobs on utilization. Minimum processors: 2, negotiation cost: 1.5 ms, adaptation cost: 2ms.

6.5 Performance with the Variation of Negotiation Cost

To investigate the impact of negotiation cost on performance, simulation experiments were conducted by varying the negotiation cost while keeping the adaptation cost constant. The adaptation cost per processor was fixed to 2 milliseconds. The negotiation cost was varied from 1.5 milliseconds to 8 seconds. Table 6.7 shows the impact of negotiation cost on utilization. Figures 6.10 and 6.11 graphically show the impact of negotiation cost on utilization. From Table 6.7 and Figure 6.10 it can be seen that the negotiation costs up to 0.8 second do not have any significant effect on utilization. As the negotiation cost increases further, the utilization decreases as the

number of malleable increases and after a point then the utilization starts increasing again as the number of malleable job increases further. Figure 6.11 shows variation of utilization as the negotiation cost increases for different job mixes. From the Figure 6.11 and Table 6.7, it can be seen that for job mixes 10% and 100% the negotiation cost has no significant effect. For the other job mixes, the utilization decreases as the negotiation cost increases. The impact of negotiation cost is most profound in job mixes between the range 40%-60%. From Table 6.8 and Figures 6.12 and 6.13 similar trends can be observed for average turn around time. Table 6.9 shows the decrease in utilization and average turn around time for increasing negotiation cost from 1.5 milliseconds to 8 seconds.

The reason for this behavior is that as the number of malleable jobs in the workload increases the number of negotiation increases and reaches a maximum at a certain job mix. If the number of malleable jobs in the workload increases further the number of negotiation decreases and reaches a minimum at 100% malleable jobs. Table 6.10 show the variation in the number of negotiations as the percentage of malleable jobs varies. Figure 6.14 presents the result graphically. When percentage of malleable job in the workload is low the need for negotiation is high, as there are many rigid jobs waiting in the pending queue. Because there are very few malleable jobs running, the scope of negotiation is low. As a result the number of negotiations is low. As the number of malleable jobs in the workload increases, the number of running malleable jobs also

increases. Consequently, the number of negotiation increases. However, as the number of malleable jobs in the workload increases further, there are more malleable jobs in the pending queue compared to rigid jobs. Because of it flexibility, a pending malleable job can be scheduled for execution as long as the number the idle processors is equal or more than the minimum processors requirement for the job. For this reason as the number of malleable job in the pending queue increases the need for negotiation also decreases.

Table 6.7 Impact of negotiation cost on utilization

Neg. Cost (secs.)	Utilization									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.0015	0.92574	0.99747	0.9993	0.99873	0.9998	0.9998	0.99849	0.99874	0.99979	0.99974
0.003	0.92574	0.99739	0.9993	0.99873	0.99979	0.99879	0.99849	0.99873	0.99979	0.99974
0.006	0.92573	0.99738	0.99928	0.99949	0.99978	0.99983	0.999848	0.99873	0.99978	0.99974
0.012	0.92573	0.99746	0.99926	0.99869	0.99976	0.99978	0.99846	0.99871	0.99977	0.99974
0.024	0.92574	0.99739	0.99927	0.99866	0.99973	0.99974	0.99842	0.99868	0.99976	0.99974
0.048	0.92571	0.99738	0.99923	0.99849	0.99964	0.99968	0.99817	0.99864	0.9993	0.99973
0.096	0.92569	0.99725	0.99913	0.99862	0.99948	0.99948	0.9982	0.99537	0.99924	0.99972
0.2	0.92564	0.99704	0.99866	0.99662	0.99866	0.99874	0.99758	0.99825	0.99951	0.9997
0.4	0.92554	0.996	0.99697	0.99679	0.99729	0.99669	0.9968	0.997	0.99903	0.99965
0.8	0.92512	0.99553	0.99642	0.9929	0.99527	0.99544	0.99472	0.99074	0.99729	0.99956
2	0.92433	0.99307	0.99279	0.98726	0.98848	0.98816	0.9877	0.99096	0.99528	0.99923
4	0.9221	0.9864	0.9816	0.97445	0.96888	0.97615	0.97734	0.98686	0.98974	0.99875
8	0.92051	0.97961	0.96659	0.94554	0.94841	0.94637	0.95367	0.96609	0.96609	0.9967

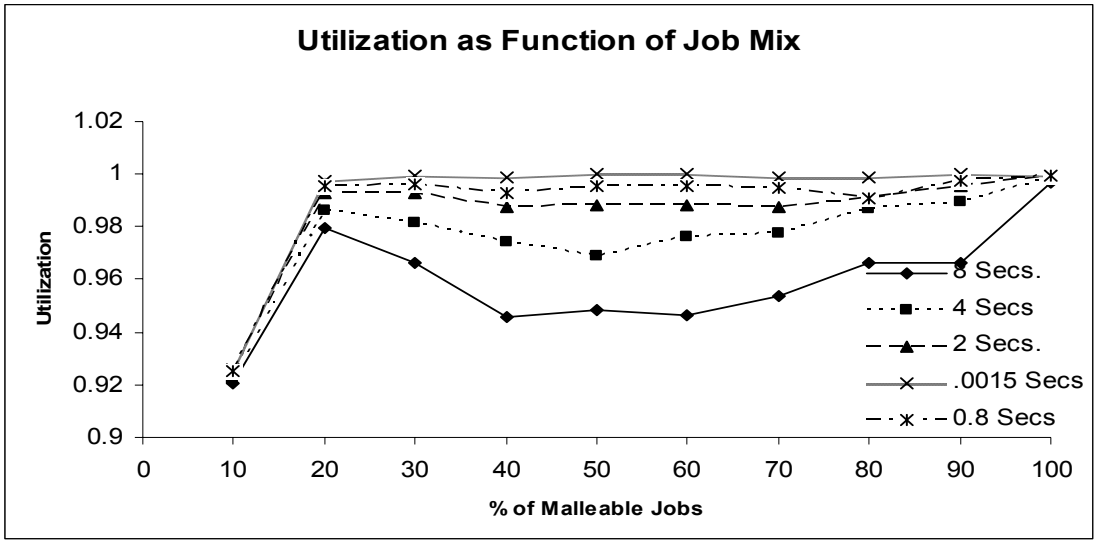


Figure 6.10 Impact of negotiation cost on utilization. Flexibility range: 2- 128, adaptation cost: 2ms

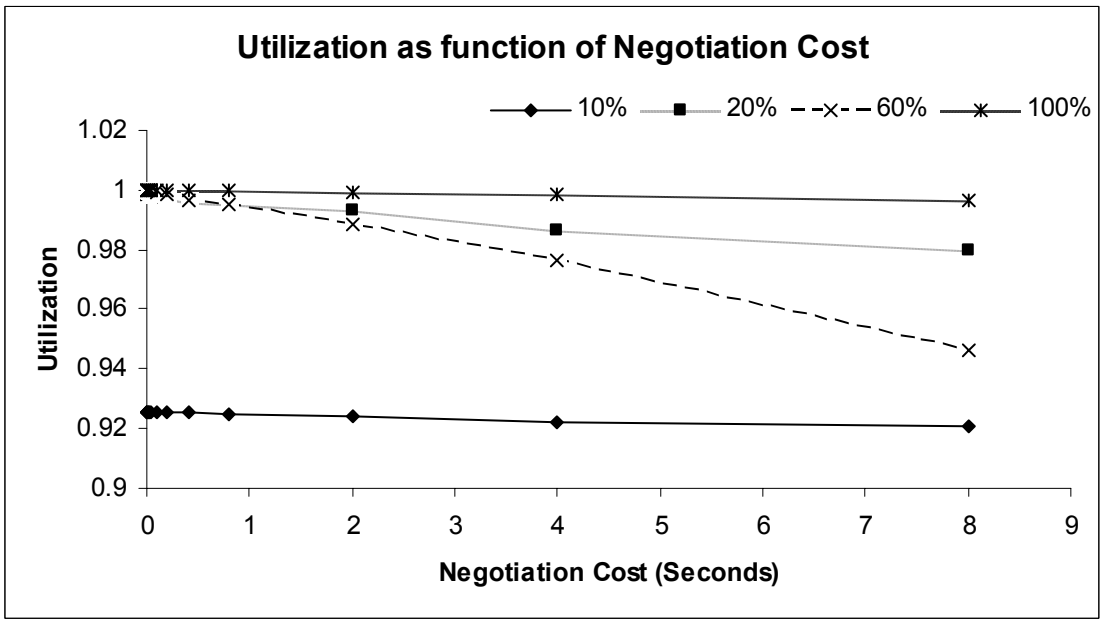


Figure 6.11 Impact of negotiation cost on utilization. Flexibility range: 2- 128, adaptation cost: 2ms

Table 6.8 Impact of negotiation cost on average turn around time

Neg Cost (Secs.)	Average Turn Around Time (Seconds)									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.0015	101443	93002	92724	91822	91913	91607	89184	88069	89398	86115
0.003	101443	93003	92725	91822	91914	91608	89185	88070	89398	86115
0.006	101443	93003	92725	91817	91915	91554	89186	88071	89399	86115
0.012	101444	93005	92726	91843	91917	91590	89187	88073	89399	86115
0.024	101445	93006	92742	91829	91918	91592	89192	88077	89401	86115
0.048	101446	93007	92747	91852	91913	91523	89345	88090	89327	86115
0.096	101447	92977	92740	91829	91959	91567	89385	88144	89332	86115
0.2	101452	93061	92758	92078	92060	91549	89384	88424	89420	86115
0.4	101461	93083	92939	91933	92117	91827	89448	88372	89363	86116
0.8	101485	93156	92856	92477	92342	91915	89595	88570	89396	86117
2	101581	93367	93278	92846	93026	92776	90359	88679	90057	86159
4	101710	93921	94559	94777	94821	94181	90910	89306	90387	86144
8	101902	94691	95637	97250	97339	96759	93732	92122	90857	86374

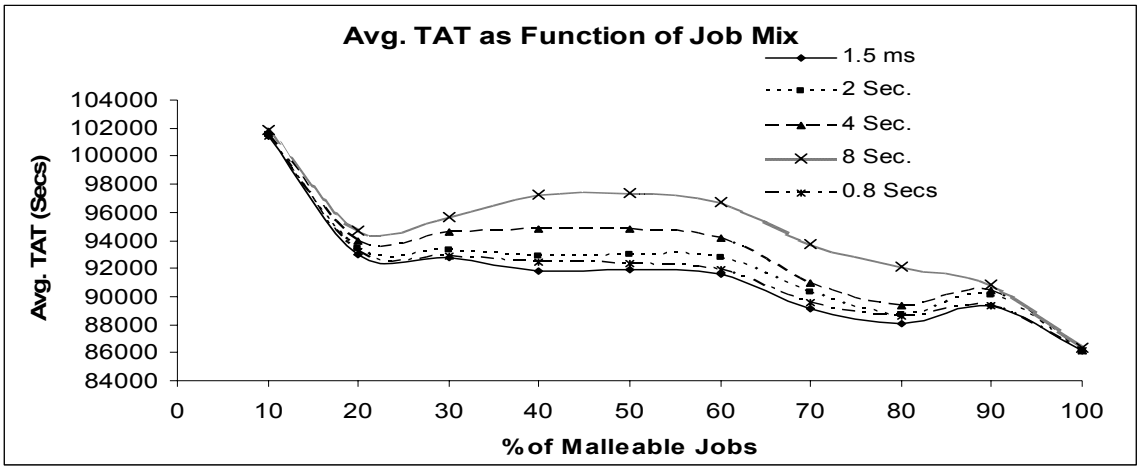


Figure 6.12 Impact of negotiation cost on avg. TAT. Flexibility range: 2- 128, adaptation cost: 2ms

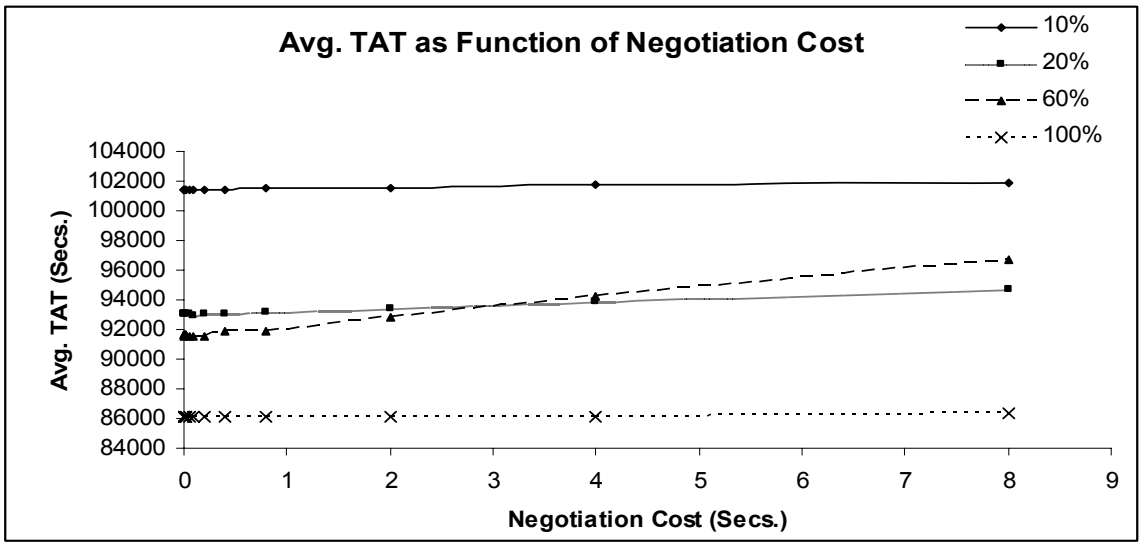


Figure 6.13 Impact of negotiation cost on avg. TAT. Flexibility range: 2- 128, adaptation cost: 2ms

Table 6.9 Decrease in performance as negotiation cost increased from 1.5 ms to 8 seconds

% of Mal. Jobs	10	20	30	40	50	60	70	80	90	100
% Decrease in Utilization	0.52	1.79	3.27	5.32	5.14	5.34	4.48	3.27	3.37	0.30
% Decrease in avg.TAT	0.45	1.82	3.14	5.91	5.90	5.62	5.10	4.60	1.63	0.30

Table 6.10 Variation of number of negotiation with the variation of number of malleable jobs in the workload

% of Malleable Jobs	Number of Negotiation
10	599
20	1699
30	2750
40	3709
50	4072
60	6090
70	6106
80	4636
90	2882
100	306

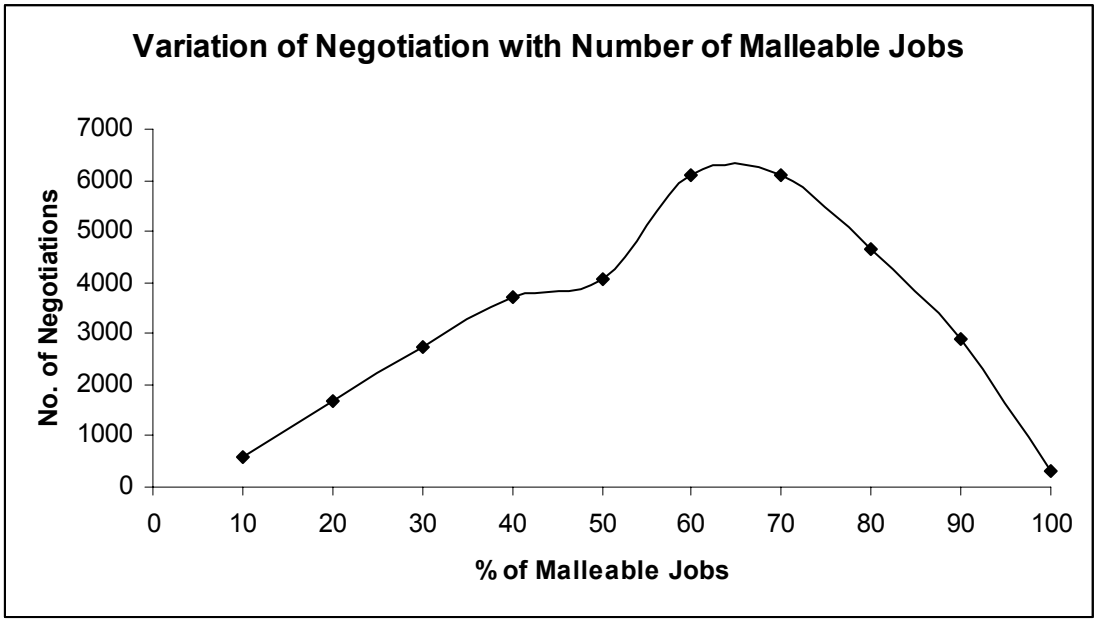


Figure 6.14 Variation of number of negotiation with the variation of percentage of malleable jobs in the workload

6.6 Performance with the Variation of Adaptation Cost

To investigate the impact of adaptation cost on performance, simulation experiments were conducted by varying the adaptation cost while keeping the negotiation cost constant. The negotiation cost was fixed at 1.5 milliseconds. The adaptation cost was varied from 2 milliseconds to 8 seconds. Table 6.11 shows the impact of adaptation cost on utilization. Figures 6.15, 6.16 and 6.17 graphically show the impact of adaptation cost on utilization. From the simulation results it can be seen that for adaptation costs up to 0.2 second, the utilization does not vary significantly for any job mixes. For 10% job mixes the adaptation cost does not effect utilization up to 1 second. For 100% job mixes the adaptation cost doesn't effect utilization up to 0.4 second. In general as the adaptation cost increases beyond 0.2 seconds the utilization decreases. The impact of adaptation cost is more profound on job mixes between 70% - 90%. From Table 6.11 and figure 6.15 it can be seen that for adaptation costs beyond 2 seconds, as the number of malleable jobs increases, the utilization increases initially but then it decreases. As number of malleable jobs further increases the utilization increases again. The reason for this behavior is that as the number of malleable jobs in the workload increases the number of negotiation increases, and consequently the number of adaptation increases and reaches a maximum at certain job mix. If the number of malleable jobs in the workload increases further the number of adaptation decreases and reaches a minimum at 100% malleable jobs. Table 6.14 show the variation of number of adaptation as the percentage of malleable job

varies. From Figures 6.16 and 6.17 it can be seen that for all job mixes as the adaptation cost increases the utilization decreases more or less linearly. A job mix 80% shows highest decrease.

Table 6.12 and Figures 6.18, 6.19 and 6.20 show the impact of adaptation cost on average turn around time. The adaptation cost has no significant impact on average turn around time for job mixes 10% and 100%. For other job mixes trends similar to those seen in utilization can be observed. For all job mixes, the impact of adaptation cost on average turn around time is much less compared to the impact on utilization.

Table 6.13 shows the decrease in utilization and average turn around time for increasing adaptation cost from 2 milliseconds to .08, 1 and 8 seconds. From the table it can be seen that, in general the decrease increases as the percentage of malleable jobs increases and reaches a maximum. After that the performance degradation improves as the number of malleable jobs increases further. The reason for this behavior is that as the number of malleable jobs in the workload increases the number of adaptation increases and reaches a maximum at certain job mix. If the number of malleable jobs in the workload increases further the number of adaptation decreases and reaches a minimum at 100% malleable jobs.

Table 6.11 Impact of adaptation cost on utilization

Adapt. Cost	Utilization									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.002	0.92574	0.99747	0.9993	0.99873	0.9998	0.9998	0.99849	0.99874	0.99979	0.99974
0.004	0.92574	0.99733	0.99907	0.99894	0.99973	0.99974	0.99844	0.99879	0.99973	0.99971
0.008	0.92574	0.99737	0.99882	0.99874	0.99961	0.99963	0.99852	0.99863	0.9996	0.99964
0.01	0.92574	0.99727	0.99867	0.99867	0.9995	0.99958	0.99861	0.99847	0.99954	0.99961
0.02	0.92574	0.99645	0.99879	0.99894	0.99923	0.99931	0.99797	0.99816	0.9989	0.99945
0.04	0.92568	0.99618	0.99829	0.99801	0.99858	0.99882	0.99753	0.99769	0.99822	0.99913
0.08	0.9267	0.99544	0.99737	0.99652	0.99747	0.99736	0.99668	0.99588	0.99708	0.99849
0.2	0.92542	0.9899	0.99263	0.99259	0.99387	0.99505	0.99346	0.99192	0.99353	0.99658
0.4	0.92512	0.98514	0.98681	0.98795	0.98917	0.98933	0.98866	0.98433	0.98802	0.99328
0.8	0.9233	0.97772	0.97528	0.97671	0.97995	0.97863	0.98058	0.97214	0.97787	0.98677
1	0.92256	0.97392	0.97156	0.97412	0.97773	0.97598	0.97581	0.96689	0.97184	0.98436
2	0.91823	0.95577	0.95832	0.94937	0.95843	0.95465	0.95107	0.94149	0.94814	0.97086
4	0.91612	0.94205	0.93835	0.92872	0.93403	0.9289	0.92023	0.89598	0.90425	0.94731
8	0.9032	0.90535	0.90638	0.89453	0.90023	0.88113	0.87146	0.84255	0.86391	0.91249

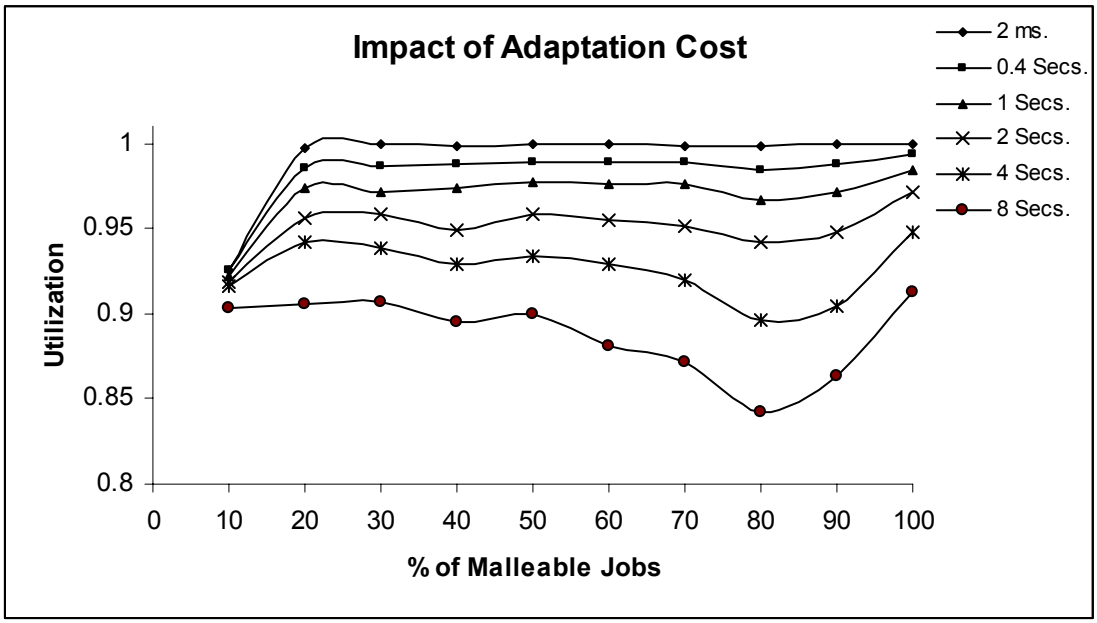


Figure 6.15 Impact of adaptation cost on utilization. Flexibility range: 2- 128, negotiation cost: 1.5ms

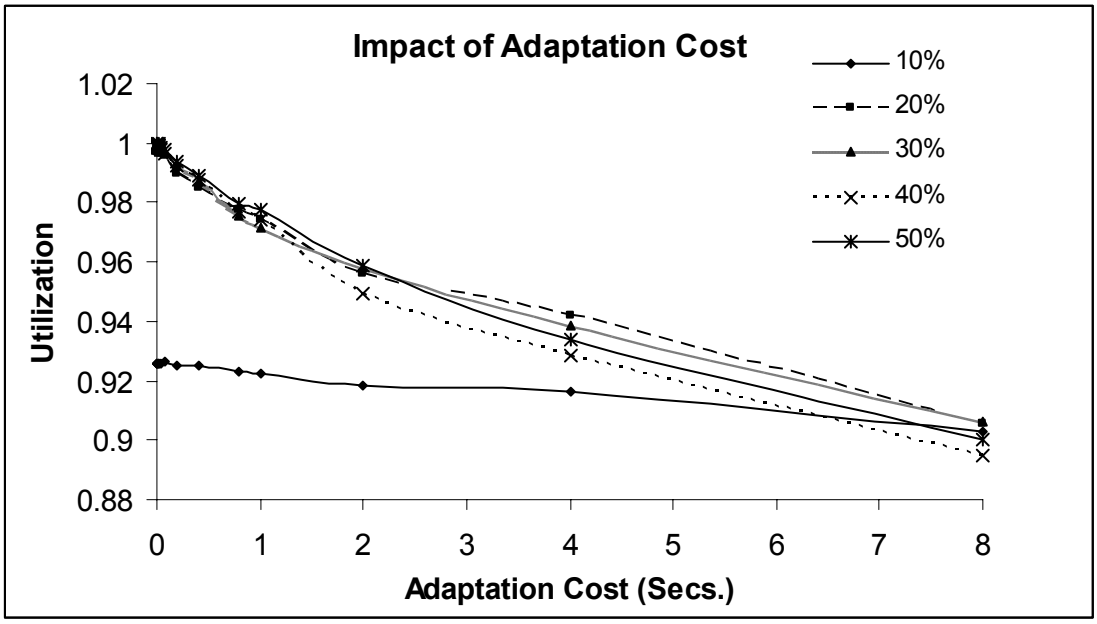


Figure 6.16 Impact of adaptation cost on utilization. Flexibility range: 2- 128, negotiation cost: 1.5ms

Table 6.12 Impact of adaptation cost on average turn around time

Adapt. Cost	Average Turn Around Time (Seconds)									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.002	101443	93002	92724	91822	91913	91607	89184	88069	89398	86115
0.004	101443	93008	92741	91815	91918	91609	89187	88226	89402	86115
0.008	101443	92995	92696	91817	91874	91612	89261	88109	89408	86116
0.01	101443	93003	92756	91820	91968	91614	89008	88025	89411	86117
0.02	101444	93017	92733	91859	91946	91540	89045	88244	89288	86119
0.04	101452	93069	92828	91897	92019	91577	89265	88399	89300	86123
0.08	101456	93134	92865	91959	92097	91567	89400	88376	89259	86133
0.2	101475	93610	93197	92211	92279	91932	89493	88489	89670	86159
0.4	101476	94007	93668	92605	92591	92057	89797	88964	90222	86200
0.8	101391	94545	94761	93129	93081	92767	90145	89974	90609	86259
1	101397	94722	94954	93262	93149	92852	90308	89866	90560	86267
2	101566	96088	95352	94754	94143	93787	91494	91340	92057	86378
4	101557	96500	96069	95469	95430	94421	92818	93935	94133	86556
8	101793	97737	97504	96220	96064	95308	94386	95679	95812	86788

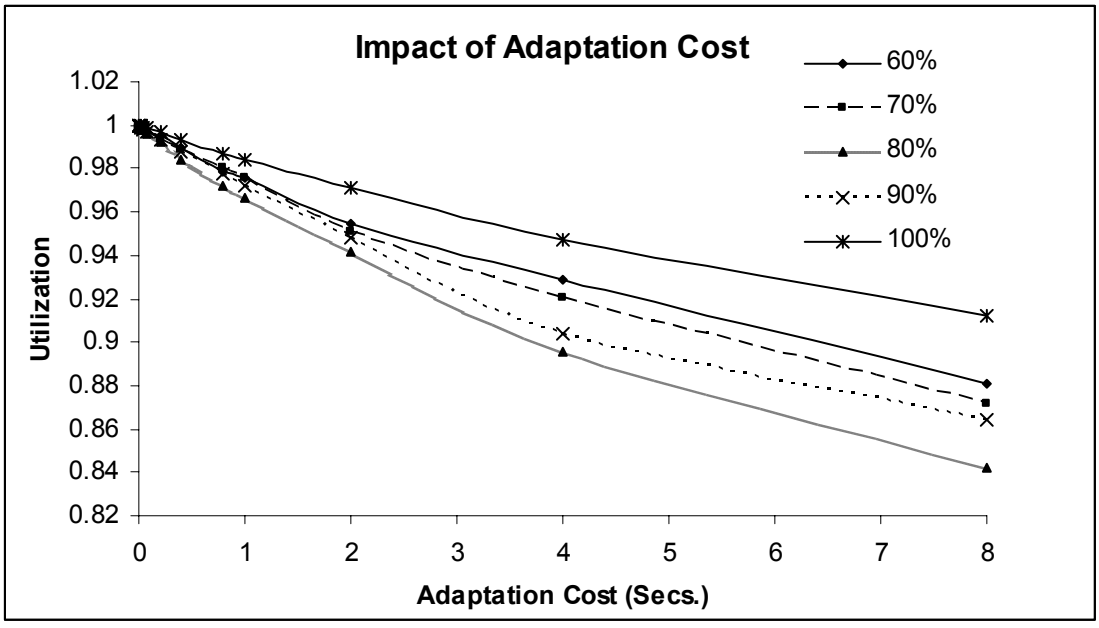


Figure 6.17 Impact of adaptation cost on utilization. Flexibility range: 2- 128, negotiation cost: 1.5ms

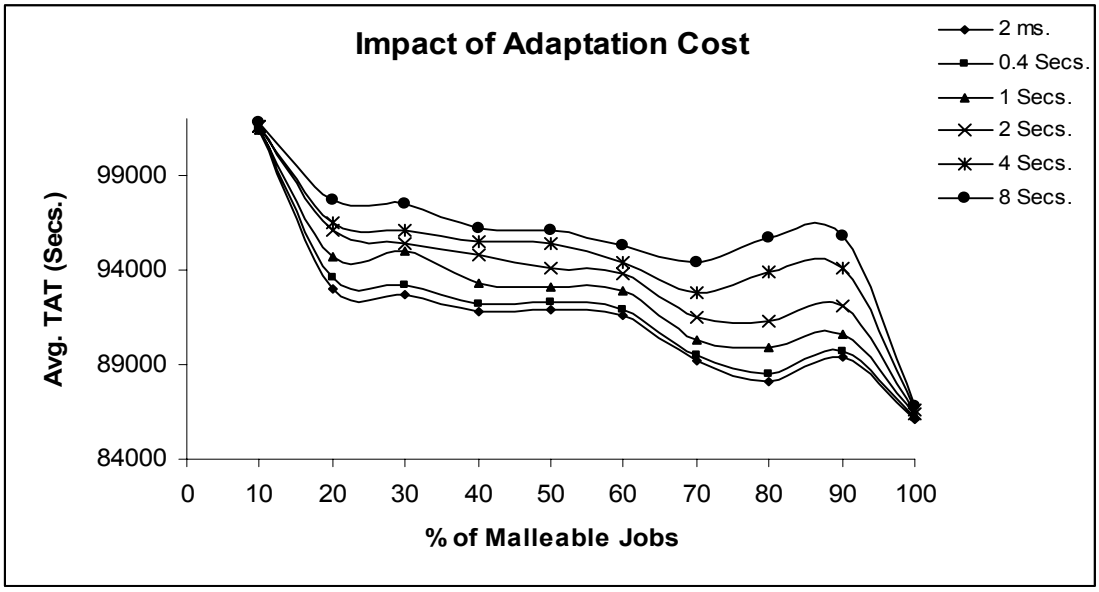


Figure 6.18 Impact of adaptation cost on average turn around time. Flexibility range: 2-128, negotiation cost: 1.5ms

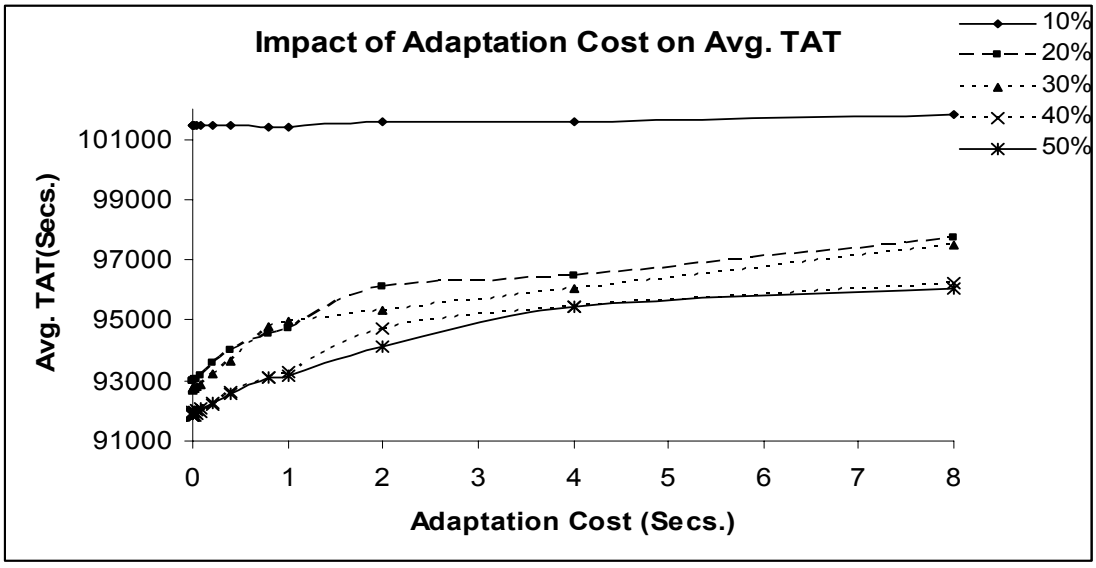


Figure 6.19 Impact of adaptation cost on average turn around time. Flexibility range: 2-128, negotiation cost: 2ms

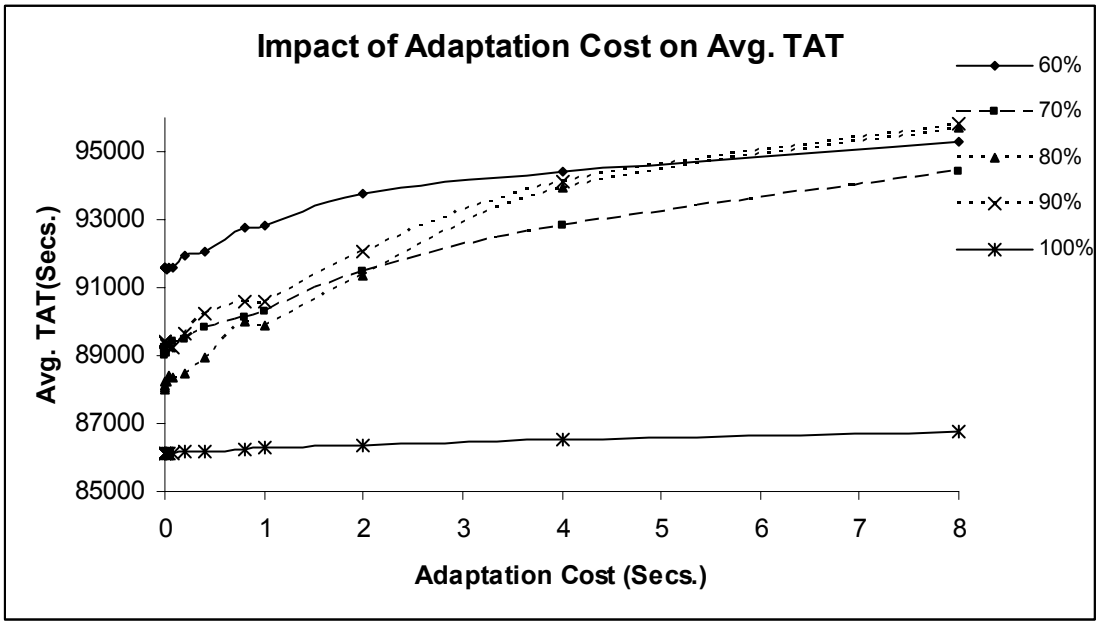


Figure 6.20 Impact of adaptation cost on average turn around time. Flexibility range: 2-128, negotiation cost: 1.5ms

Table 6.13 Decrease in performance as adaptation cost increased from 1.5 ms to 8 seconds

% of Malleable Jobs	10	20	30	40	50	60	70	80	90	100
% Decrease in Utilization (0.08 Sec)	0.03	0.767	0.68	0.61	0.59	0.48	0.50	0.68	0.63	0.316
% Decrease in Utilization (1 Sec)	0.32	2.36	2.77	2.46	2.21	2.38	2.27	3.19	2.80	1.538
% Decrease in Utilization (8 Sec)	2.25	9.21	9.29	10.42	9.96	11.87	12.70	15.62	13.59	8.725
% Decrease in Avg. TAT (0.08 Sec)	0.01	0.14	0.15	0.15	0.20	-0.05	0.24	0.35	-0.16	0.021
% Decrease in Avg. TAT (1 Sec)	-0.05	1.85	2.40	1.57	1.34	1.36	1.26	2.04	1.30	0.18
% Decrease in Avg. TAT (8 Sec)	0.35	5.09	5.16	4.79	4.52	4.04	5.83	8.64	7.17	0.78

Table 6.14 Variation of number of Adaptation with the variation of number of malleable jobs in the workload

% of Malleable Jobs	Number of Adaptation
10	599
20	1699
30	2750
40	3709
50	4072
60	6090
70	6106
80	4636
90	2882
100	306

6.7 Summary

Results of simulation experiments to investigate the impact of RMS and workload parameters on system and application performance have been presented in this chapter. The impact of the following parameters on system and application performance has been investigated through simulation experiments. 1) The number of malleable jobs in the workload, 2) flexibility of malleable jobs, 3) cost of negotiation, and 4) cost of adaptation of malleable jobs. During the simulation experiments one of the parameters has been varied while other parameters have been kept constant.

The performance in general improves with increase of percentage of malleable jobs in the workload and saturates at a certain job mix and it increases very little after saturation. The most important finding is that irrespective of cluster size, or base line utilization with an all rigid workload, it is possible to achieve maximum utilization with a

malleable workload. The maximum possible utilization can be achieved with relatively few malleable jobs (20% in our experiments).

The performance in general improves as the flexibility increases up to certain flexibility. The performance than saturates. Increasing flexibility further does not improve performance. The impact of minimum processor of flexibility range has more impact on performance than the flexibility range itself. Decreasing the minimum number of processors for same the flexibility range increases the performance.

Both negotiation cost and adaptations cost impact the performance. As these costs increases the performance decreases. Negotiation costs up to 0.8 second had no significant impact on performance. Negotiation cost does not impact 10% and 100% job mixes. For the same negotiation cost as the number of malleable job increases the utilization decreases, and then the utilization increases as the number of malleable increases further. The number of adaptations and the number of negotiations increase as the number of malleable job increases and they reach a maximum. After the maxima the number of negotiations and the number of adaptations decreases as the number of malleable jobs increases further. The impact of adaptation cost on performance more pronounced compared to the impact of negotiation cost.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Current resource management systems for clusters support mostly rigid applications. A few systems support moldable applications, where there is some flexibility in the amount of resources that can be assigned before the applications start. The dynamic nature of adaptive applications requires a new paradigm for cluster resource management. If workloads contain rigid applications only, some of the processors may remain idle even though there are applications waiting in the queue to be executed. That is due to the fact that the available number of processors is not enough to satisfy the requirements of the waiting applications to be executed. However, if the workload contains malleable applications, they can utilize otherwise idle resources and improve performance. Conversely, malleable applications can shrink at a scheduler's request to relinquish resources that can be allocated to evolving applications asking for resources, or to applications waiting in the pending queue.

Existing resource management systems are not capable of handling malleable applications. The absence of RMS support is one of the major obstacles for application developers to write malleable applications. On the other hand, since there is an absence of malleable applications, researchers have not been sufficiently motivated to research and

develop RMS support for malleable applications. Management of malleable applications in a distributed environment is a multi-faceted problem and the research issues are complex and interrelated. The nature, complexities and relationship of these issues are not well researched and understood. Before developing malleable applications or infrastructure support for malleable applications, these issues needs to be investigated and studied in detail. One approach to address this problem is to develop a model for adaptive parallel systems and investigate and understand the behavior of these systems by numerically simulating the model.

7.1 Contributions and Summary

This dissertation makes several contributions to the research in the area of adaptive parallel systems. Specific contributions of this dissertation are described below.

A conceptual model and subsequently, a semi-formal mathematical model have been developed for an adaptive parallel system. The system consists of a RMS capable of managing rigid as well as malleable applications, and workloads containing both rigid and malleable applications. The model of this system consists of three components: a model for the resource management system, a model for malleable and rigid applications, and a model for generating malleable workloads. The model can be used to investigate and understand the behavior of the system qualitatively and quantitatively under different conditions.

A discrete event simulator has been developed which can be used to numerically simulate the model of adaptive parallel systems. In particular the simulator can be used to determine the impact of the RMS, the application and the workload parameters on system and application performance. The simulator has been developed following the standard architecture for discrete even simulators. As a result the, simulator is modular and flexible enough to accommodate modification in the model with minimum rework. For example, to investigate the impact of different scheduling algorithms, one needs to modify the scheduler module without making changes to any other parts of the simulator.

In this dissertation, the first detailed empirical evaluation of the impact of the RMS and the workload parameters on system and application performance has been reported. The key contribution of this dissertation is discovering the following new knowledge about adaptive parallel systems with malleable applications.

1. The performance in general improves with an increase in the percentage of malleable jobs in a workload. The performance saturates at a certain rigid/malleable job mix and it increases very little after saturation. Also, a high percentage of malleable jobs is not necessary to make significant improvement in performance.

2. The presence of malleable jobs in a workload decreases the average turn around time and the average wait time compared to a workload with all

rigid jobs. However, the presence of malleable applications increases the average execution time.

3. In general the performance improves as the flexibility increases up to a certain point, than it saturates. The minimum number processors in the flexibility range has more impact on performance than the flexibility range itself. Decreasing the minimum number of processors for the same flexibility range increases the performance.

4. The negotiation cost has a small impact on the performance. Small negotiation costs (costs up to one second) do not have any significant impact on the performance. If negotiation costs increase further, the performance decreases.

5. For negotiation costs beyond 2 seconds, as the number of malleable jobs in a workload increases, the performance increases and reaches a maximum point. Increasing the number of malleable jobs further results into a decrease in the performance and it reaches a minimum. The performance starts increasing again as the number of malleable jobs increases further.

6. The number of negotiations for a given workload increases as number of malleable jobs increases up to a certain point. As the number of malleable jobs increases further the number of negotiations decreases, and it reaches a minimum as the percentage of malleable jobs reaches 100.

7. The performance degrades as the application adaptation cost increases. The impact of the application adaptation cost is much more profound compared to that of the negotiation cost.

Another contribution of this dissertation is the development of a prototype RMS system capable of managing malleable as well as rigid applications. Even though the prototype RMS is not robust enough to be of production quality, it provides valuable information regarding difficulties of developing an RMS for adaptive applications. As part of developing the prototype RMS, a negotiation protocol has also been developed. The following lessons have been learned from this exercise.

1. Developing a communication infrastructure to manage negotiation between malleable applications and RMS is the most critical and difficult part.

2. A negotiation mechanism that can handle ill-behaved malleable applications (such as an application that does not respond to a negotiation offers in a timely manner) is important.

3. Multistage scheduling is required to take advantage of malleable application to the full extent.

7.2 Future Work

The paradigm of adaptive applications is relatively new and not well understood. There are many issues that have not been addressed in this dissertation that are worth investigating in the future.

In this dissertation, the algorithm adopted for scheduling is very simple. In particular, the selection of candidates for processor preemption is on a first start first candidate basis, which results into a high number of negotiations, and consequently a high number of adaptations. Further research in the scheduling algorithm is required, especially regarding a candidate selection policy with the goal of reducing the number of negotiations and adaptations. A scheduling algorithm involving a candidate selection policy, such as selecting a candidate which can give up the maximum number of processors and multistage scheduling, can be investigated.

Further research to investigate the impact of failed negotiations, which has not been investigated in this research, also needs to be performed. In modeling a malleable application, some simplified assumptions have been made which are mostly applicable to embarrassingly parallel applications with no data dependency. Further research is required in modeling malleable applications which are not embarrassingly parallel, or have data dependency among tasks.

One area of future work is to investigate simulation outputs to discover relationship and dependencies among model parameters using statistical techniques. Another area of future research is to develop a programming model for adaptive applications. Moreover, future research is required in modeling of an adaptive system that includes evolving applications along with malleable applications.

REFERENCES

1. D. G. Feitelson and L. Rudolph, "Towards convergence in job scheduling for parallel super computers," in Job Scheduling Strategies for Parallel Processing, Vol. 1162, Lecture Notes in Computer Science D. G. Feitelson and L. Rudolph Eds. Springer-Verilag, 1996, pp 1-26.

2. C. R. Anderson, An Implementation of the Fast Multipole Method SIAM J. Sci. Stat. Comput.,1992, 923-947.

3. I. Banicescu. Load Balancing and Data Locality in the Parallelization of the Fast Multipole Algorithm, Ph.D. Dissertation, Polytechnic University, 1996 January.

4. K. Droegemeier, "Transforming the Sensing and Prediction of Intense Local Weather Through Dynamic Adaptation: People and Technologies Interacting with the Atmosphere", Keynote Presentation in the 6th International Conference on Linux Cluster: The HPC Revolution 2005, April 25-28, Chapel Hill, North Carolina, http://www.linuxclustersinstitute.org/Linux-HPC-revolution/Archive/2005presentations.html

5. K. Droegemeier, K. Brewster, M. Xue, D. Weber, D. Ganon, B. Plale, D. Reed, L. Ramakrishnan, J. Almedia, R. Wilhelmson, T. Baltzer, B. Dominico, D. Murray, M. Ramamurthy, A. Wilson, R. Clark, S. Yalda, S. Graves, R. Ramachandran, J. Rushing, E. Joseph, and V. Morris, "Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather", Computing in Science and Engineering, 7:6 pp 12-29, 2005.

6. P.-F. Dutot, and D. Trystram, Scheduling on hierarchical clusters using malleable tasks, Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, p.199-208, July 2001, Crete Island, Greece

7. R. Lepere, G. Mounie, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. European Journal of Operational Research, (142):242-249.

8. R. Lepere, D. Trystram, and G.J. Woeginger. Approximation scheduling for malleable tasks under precedence constraints. International Journal of Foundation in Computer Science, 13(4):613-627, 2002.

9. G. Mounie, C. Rapine, and D. Trystram, "Efficient Approximation Algorithms for scheduling Malleable Tasks. In the Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, Saint Malo, France, pp: 23 – 32, 1999.

10. J. Hungershofer, "On the Combined Scheduling of Malleable and Rigid Jobs", in Proceedings of the 16th Synposium on Computer Architecture and High Performance Computing, 2004.

11. Jan Hungershofer, Achim Streit, and Jens-Michael Wierum. Efficient Resource Management for Malleable Applications. Technical Report PC2, TR-003-01, December 2001, http://wwwcs.upb.de/pc2/papers/files/394.pdf

12. S.S. Vadhiyar and J. Dongarra, "SRS: A framework for developing malleable and migratable parallel applications for distributed systems", Parallel Processing Letters, Vol. 13, No. 2 (2003) 291-312

13. J. E. Moreira and V. K. Naik. Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications. IBM Journal of Research and Development, Vol. 41, No. 3, May 1997, pp 303 – 330.

14. L. V. Kale, S. Kumar, and J. DeSouza, "A Malleable-Job System for Timeshared Parallel Machines", L,2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), May 21-24, 2002, Berlin, Germany.

15. L. V. Kale, S. Kumar, and J. DeSouza, "An Adaptive Job Scheduler for Timeshared Parallel Machines", L, PPL Technical Report 00-02, University of Illinois at Urbana-Champaign, Sep 2000.

16. S.K. Ghafoor, T.A. Haupt, I. Banicescu, R.L. Carino, "A Resource Management System for Adaptive Parallel Applications in Cluster Environments,**"** to appear in proceedings of *the 6th International Conference on Linux Cluster: The HPC Revolution 2005,* April 25-28, Chapel Hill, North Carolina.

17. S.K. Ghafoor, T.A. Haupt, and S.N. Gasula, "A Communication Protocol for Adaptive Parallel Applications in Cluster Environments," to appear in

Proceedings of the High Performance Computing Symposium (HPC 2005), April 3- 7, 2005, San Diego, 2005.

18. S.K. Ghafoor, T.A. Haupt, M. Rashid, N. Ammari, "Impact of Malleable Jobs on System and Application Performance," to appear in *Proceedings of the High Performance Computing Symposium (HPC 2006)*, April 2- 6, 2006, Huntsville, Al, 2006.

19. The Condor Project Homepage, http://www.cs.wisc.edu/condor/

20. DQS-Distributed Queuing System. http://www.scri.fsu.edu/~pasko/dqs.html

21. M.L. Massie, B.N. Chun, D.E. Culler, "The Ganglia distributed monitoring system: Design, implementation and experience" Parallel Computing, vol. 30, Issue 7, July 2004.

22. S. Angaluri. "ClusterController - An interoperable scheduler architecture for windows and linux", Presented at the 2nd Cluster Computing in the Sciences Conference, University of Utah, Salt Lake city. Feb 9, 2001.

23. MPI Software Technology, Inc., ClusterController™ User Guide Version 1.0.1, 2001.

24. Portable Batch System. http://www.openpbs.org

25. Platform LSF Family of Products. http://www.platform.com/products/LSFfimily.

26. Grid Engine. http://www.gridengine.sunsource.net

27. Maui Cluster Scheduler. http://www.clusterresources.com/products/maui

28. Moab Cluster Suit. http://www.cluterresources.com/products/moabclustersuit.shtml

29. S.S. Vadhiyar and J. Dongarra, "SRS: A framework for developing malleable and migratable parallel applications for distributed systems", Parallel Processing Letters, Vol. 13, No. 2 (2003) 291-312

30. L. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In Proceedings of the conference object oriented programming systems, languages and applications, September 1993.

31. L. Kale and S. Krishnan. CHARM++: Parallel programming with message-driven objects. In G.V. Wilson and P. Lu, editors, Parallel programming using C++. Pages 175-213. MIT Press, 1996.

32. L. V. Kale, M. Bhandarkar, N. Jagathsen, S. Krishnan, and J yelon. Converse: An interoperable framework for parallel. In proceedings of the 1oth international parllel processing symposium, pages 212-217, April 1996.

33. M. Bhandarkar, L. V. Kale, E de Sturler, and J. Hoefinger. Object-based adaptive load balancing for MPI programs. In proceedings of the international conference on computational science, Sanfarancisco, CA LNCS 2074, pages 108-117, May 2001.

34. R. B. Konuru, J. E. Moreira, and V. K. Naik, "Application-Assisted Dynamic scheduling on Large-Scale Multi-computer Systems," in Proceedings of the Second International Euro-Par conference, Lyon, France, Volume 1124 of Lecture Notes in Computer Science, august 1996, pp II:621-630.

35. S. Midkiff, J. E. Moreira, and V. K. Naik, "Run-Time Support for Dynamic Processor Allocation in HPF programs," in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997.

36. R. Jha, M. Muhammad, S. Yalamanchili, K. Schwan, D. Ivan Rosu, and C. de Castro, "Adaptive resource allocation for embedded parallel applications", in Proceedings of the 3rd International Conference on High Performance Computing", Trivandrum India, December 1996.

37. D. Ivan Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications", in Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, December 1997.

38. G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz, ``Runtime and Compiler Support for Programming in Adaptive Parallel Environments", in Journal of Scientific Programming, vol. 6, no. 2, pp. 215-227, 1997.

39. G. Edjlali, G. Agrawal, A. Sussman and J. Saltz. Data Parallel Programming in an Adaptive Environment. In the Proceedings of the Ninth International Parallel Processing Symposium. April 1995. pages 827-832. IEEE Computer Society Press.

40. J. Turek, J. Wolf, and P. Yu, "Approximation algorithms for scheduling parallelizable tasks", *Proceeding of the 4th ACM Annual Symposium on Parallel Algorithms and Architectures*, 1992, pp. 323-332.

41. P.-F. Dutot, G. Mounie and D. Trystram, "Scheduling Parallel Tasks – Approximation Algorithms", Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Edited by Joseph Y-T. Leung, Published by CRC Press, Boca Raton, FL, USA, 2004.

42. D. Gelernter and D. Kaminsky, "Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha," in Proceedings of international Conference on Supercomputing, ACM July 19-23, pp. 417-427.

43. N. Carriero and D. Gelernter, How to write parllel Programs: Afirst course, The MIT Press, Cambridge, MA, 1990.

44. Linda User Guide, HTTP://WWW.LINDASPACES.COM/DOWNLOADS/LINDAMANUAL.PDF

45. K. A. Robins and S. Robins, The Cray X-MP/Model 24, Volume 374 of Lectures Notes in Computer Science, Springer-Verilag, New York, 1989.

46. A. Gupta, A. Tucker, and L. Stevens, "Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach," Technical Report CSL-TR-91-475A, Computer Systems Laboratory, Stanford University, Stanford, CA, 1991.

47. C. McCann, R. Vaswami, and J. Zahorjan, " A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors," ACM Transaction on Computer System, 11, No. 2, 146-178 (May 1993).

48. J. E. Moreira, On the Implementation and Effectiveness of Autoscheduling for Shared Memory Multiprocessors, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1995.

49. C. Polychronopoulos, "Auto-scheduling: Control Flow and Data Flow Come Together," Technical Report 1058, Center for Supercomputer Research and Development, University of Illinois at Urbana-Champaign, December 1990.

50. A. Streit, A self-tuning job scheduler family with dynamic policy switching. Lecture notes in computer science; vol. 2537, PAGES: 1 - 23 , Springer-Verlag London, Uk.

51. Y. Zhang, A. Sivasubramaniam, J. Moreira, H. Franke. Impact Of Workload And System Parameters On Next Generation Cluster Scheduling Mechanisms, IEEE

Transactions On Parallel And Distributed Systems, 12(9):967-985, September 2001.

52. Allen B. Downey, ``A Parallel Workload Model and Its Implications for Processor Allocation". 6th *Intl. Symp. High Performance Distributed Comput.*, Aug 1997.

53. Dror G. Feitelson and Larry Rudolph, ``Metrics and Benchmarking for Parallel Job Scheduling". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Springer-Verlag, 1998, Lect. Notes Comput. Sci. vol. 1459, pp. 1-24.

54. T. Murata, "Petri Nets: Properties, Analysis and Applications," an invited surve y paper, Proceedings of the IEEE, Vol.77, No.4 pp.541-580, April, 1989.

55. L.M. Kristensen, S. Christensen, K. Jensen:T*he Practitioner's Guide To Coloured Petri Nets*. International Journal On Software Tools For Technology Transfer, 2 (1998), Springer Verlag, 98-132.

56. A. M. Law and W. D. Kelton, Simulation Modeling and Analysis, Third Edition, McGraw-Hill Higher Education, 2000, ISBN 0-07-059292-6.

57. J. Hungershofer, “On the Combined Scheduling of Malleable and Rigid Jobs.” In Proceedings of the 16th Synposium on Computer Architecture and High Performance Computing, 2004.

58. Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson, ``Modeling User Runtime Estimates". 11th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), pp. 1-35*, Jun 2005. Lecture Notes in Computer Science Vol.3834.

59. Uri Lublin and Dror G. Feitelson, *The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs*. *J. Parallel & Distributed Comput.* 63(11), pp. 1105-1122, Nov 2003.

60. D. G. Feitelson, ``Packing schemes for gang scheduling". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), Springer-Verlag, 1996, Lect. Notes Comput. Sci. vol. 1162, pp. 89-110.

61. Maria Calzarossa and Giuseppe Serazzi, ``A Characterization of the Variation in Time of Workload Arrival Patterns". *IEEE Trans. Comput.* C-34(2), pp. 156-162, Feb 1985.

62. Open System for Earthquake Engineering Simulations, http://opensees.Berkeley.edu.

63. PVM: Parallel Virtual Machine http://www.csm.ornl.gov/pvm/pvm_home.html

64. Schimdt J. W. and R. Taylor, Simulation and analysis of industrial system, Richard D. Irwin, Homewood, Illinois (1970).

65. Logs of Real Parallel Workloads from Production Systems http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.