To the Graduate Council:

I am submitting herewith a dissertation written by Sathish Vadhiyar entitled “A Preemption-Based Meta-Scheduling System for Distributed Computing.” I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra

Major Professor

We have read this dissertation
and recommend its acceptance:

James Plank

Bradley Vander Zanden

Charles Collins

Accepted for the Council:

Anne Mayhew

Vice Provost
and Dean of Graduate Studies

(Original signatures are on file with official student records.)

A PREEMPTION-BASED META-SCHEDULING SYSTEM FOR DISTRIBUTED COMPUTING

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Sathish Vadhiyar

May 2003

Dedication

This dissertation is dedicated to my parents, V. R. Santhanam and S. Vathsala.

Acknowledgments

The author expresses immense gratitude to the members of his dissertation committee, Dr. Jack Dongarra, Dr. James Plank, Dr. Bradley Vander-Zanden and Dr. Charles Collins for participating in the research and providing valuable comments towards improving the quality of the research.

My profound gratitude to Dr. Jack Dongarra, thesis advisor for his able guidance, motivation and support throughout the process of research. Thanks to Dr. Dongarra for the publicity of the research to the different research communities and providing ample opportunities to the author to share the research ideas in various research forums and workshops. Thanks also to Dr. Dongarra for patiently reviewing the various research related publications of the author.

The author acknowledges the support of the research in part by the National Science Foundation contract GRANT #EIA-9975020, SC #R36505-29200099 and GRANT #EIA-9975015.

The research was conducted as part of the Grid Application Development Software (GrADS) project. The author expresses gratitude to the members of the GrADS team for their valuable comments on the research and their support to the research effort. The author acknowledges the use of machines in the GrADS testbed for the experiments conducted in the research. The author thanks the research teams from different institutions, namely the Pablo research group from the University of Illinois, Urbana-Champaign, the Grid Research and Innovation Laboratory (GRAIL) from the

University of California, San Diego and the Innovative Computing Laboratory (ICL) from University of Tennessee, for the support and maintenance of the machines in the GrADS testbed and enabling the conducting of experiments needed for the research.

Special thanks to Martin Swany from University of California, Santa Barbara for the maintenance and support of Network Weather Service (NWS), a primary component of the research infrastructure. Special thanks also to Brett Ellis, the system administrator of Innovative Computing Laboratory, University of Tennessee, for helping with the reservation of some of the machines in the GrADS testbed. The author also thanks his GrADS research colleague from University of Tennessee, Asim Yarkhan for useful discussions and insights to the various aspects of the research and in reviewing parts of the dissertation.

The author also expresses deep sense of gratitude to various members of his family for their support and sacrifices.

And, of course, thank God !

Abstract

This research aims at designing and building a scheduling framework for distributed computing systems with the primary objectives of providing fast response times to the users, delivering high system throughput and accommodating maximum number of applications into the systems. The author claims that the above mentioned objectives are the most important objectives for scheduling in recent distributed computing systems, especially Grid computing environments.

In order to achieve the objectives of the scheduling framework, the scheduler employs arbitration of application-level schedules and preemption of executing jobs under certain conditions. In application-level scheduling, the user develops a schedule for his application using an execution model that simulates the execution behavior of the application. Since application-level scheduling can seriously impede the performance of the system, the scheduling framework developed in this research arbitrates between different application-level schedules corresponding to different applications to provide fair system usage for all applications and balance the interests of different applications. In this sense, the scheduling framework is not a classical scheduling system, but a *meta-scheduling* system that interacts with the application-level schedulers.

Due to the large system dynamics involved in Grid computing systems, the ability to preempt executing jobs becomes a necessity. The meta-scheduler described in this dissertation employs well defined scheduling policies to preempt and migrate executing applications. In order to provide the users with the capability to make their applications

preemptible, a user-level checkpointing library called *SRS*(Stop-Restart Software) was also developed by this research. The SRS library is different from many user-level checkpointing libraries since it allows reconfiguration of applications between migrations. This reconfiguration can be achieved by changing the processor configuration and/or data distribution.

The experimental results provided in this dissertation demonstrates the utility of the metascheduling framework for distributed computing systems. And lastly, the metascheduling framework was put to practical use by building a Grid computing system called GradSolve. GradSolve is a flexible system and it allows the application library writers to upload applications with different capabilities into the system. GradSolve is also unique with respect to maintaining traces of the execution of the applications and using the traces for subsequent executions of the application.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the years, computing environments have evolved from single-user environments to Massively Parallel Processors (MPPs), networks of workstations, distributed systems and more recently Grid computing systems. The transition from MPPs to networks of workstations helped to increase the ability of common users to solve some large problems. By linking together commodity processors with cheap network connections, it was possible to submit and solve large problems that were previously solvable only in costlier MPPs available only at certain privileged locations like super computing centers.

With the advent of Internet, different networks of workstations with different capabilities were connected to each other to form **distributed computing systems**. These distributed computing systems possessed both hardware and software capabilities from disparate sites and were often heterogeneous in terms of the hardware configurations, operating systems and network connections. Users by utilizing these distributed systems were able to solve problems that required resources that were often not available at their own sites. The smooth functioning and maintenance of these distributed computing systems necessitated the provision of system support tools that provided a range of services like user interfaces, programming environments, programming language tools, operating system services including file services, storage services, process spawning, process management, security infrastructures and most important of all, allocation or scheduling of resources to jobs. The extent and types of theses services provided by a distributed system defined the characteristics of the system. The users' perspectives of problem solving also changed from the traditional sequential programming to parallel programming and distributed computing. New algorithms were developed that leveraged the vast set of resources available in the distributed computing systems.

Grid computing systems are the latest computing environments and have been gaining popularity for the past few years. Grid computing systems can be considered as an extension or abstraction of distributed computing systems, but in which the number and heterogeneity of the systems are much higher. "In the world of high-performance computing, a grid is an infrastructure that enables the integrated, collaborative use of high-end computing systems, networks, data archives, and scientific instruments that multiple organizations operate" [90]. Users, by plugging their systems to Grid computing systems can potentially use the vast number of services that are available in the Grid similar to the way in which electrical appliances can draw power from the electrical power grid. Grid computing also provides an opportunity for collaborative computing in which different users across the grid can collaborate towards solving a large application. Thus, for example, it is possible to start a phase of an application at one site, start another phase of the application at a different site and view the progress of the application through a graphical output on some other site. Due to the large number of resources involved in Grid computing, the load dynamics and the instability of the systems and security violations on the systems can be potentially much higher when compared to those on distributed computing systems. Hence the development of robust Grid system software to maintain the resources and jobs assumes great importance. There have been considerable efforts in developing Grid system software including the services for resource allocation, job management, stable communication infrastructures, information services to maintain records of different states of the Grid systems etc. The Grid has been put to practical use in some situations and have also been demonstrated to solve some Grand Challenge problems.

One of the visions of the distributed and Grid computing systems is to use systems for large-scale computing the way World Wide Web has been successfully used for retrieving information and in some cases solving small-scale problems. In order for this vision to be realized, robust mechanisms for allocating jobs to resources have to be incorporated into these distributed and Grid systems. The problem of *scheduling* jobs to resources has been studied extensively since the advent of time-sharing operating systems and a large number of scheduling disciplines have been developed. Different scheduling disciplines were designed to meet different objectives of scheduling in the systems. With the arrival of dynamic environments like Grid computing, it is necessary to revisit some of the scheduling objectives and to evaluate the applicability of the scheduling methodologies to the distributed and Grid computing systems. The next section presents an overview of the different efforts in the area of scheduling in distributed systems.

1.1 Scheduling in Distributed Systems

Existing scheduling systems for distributed computing can be categorized into different types based on different characteristics including the architecture for scheduling, scheduling objectives, the information used for scheduling decisions, the existence of rescheduling policies etc. In this section, only those categories of scheduling that are relevant to the research are discussed in brief. For a detailed description of different categories of scheduling, the reader is referred to [37].

An important categorization of scheduling systems is based on the objectives of scheduling. Most common objectives of scheduling systems include maintaining load balance of the systems, improving the system utilization, increasing the overall throughput of the system and minimizing the response times for individual jobs. Different scheduling systems are built with different sets of objectives and scheduling systems designed for meeting certain set of objective usually do not meet the other set of objectives. For example, a scheduling system tuned for increasing the throughput of the system may not necessarily meet the objective of minimizing the response times for individual applications.

Most of the scheduling systems for distributed and Grid computing are tuned for maintaining the load balance of the systems [16, 78, 112, 121, 59, 105, 48]. In these systems, the scheduling decisions are made such that the total load due to the jobs entering the system are almost equally distributed among all resources of the system. The assumption is that maintaining load balance in the system will lead to efficient resource utilization among all the jobs and this will in turn increase the throughput of the system. Very few scheduling systems are designed for increasing the throughput of the system [112]. In [112], high system throughput is provided by reducing the interference to the execution of an application by another application. Though this may lead to poor response times for the application, many applications can be accommodated in the system and completed within a given frame of time. There is another category of schedulers called high-performance schedulers [25] where the emphasis is to minimize the response or turnaround times of the applications [95, 59, 27, 113]. In these systems, the application specifies its properties including the total execution cost for the application and total memory and disk spaces consumed by the application etc. to the scheduling system. The properties of the application can be specified either by hard limits or by mathematical formulas or through execution simulation models. The scheduler uses the properties of the application and generates an *application-level schedule* which is a sub-optimal mapping of the application to the system resources aimed to provide high performance for the application. Though this scheduling strategy may not be beneficial to the system, it is useful for scheduling high performance applications.

Another important categorization of schedulers is based on the ability of the scheduling systems to preempt the executing applications in response to dynamic changes in system environments. Most of the scheduling systems for distributed computing follow run-to-completion (RTC) policy for the executing applications. However, the importance of preemption of applications under different conditions have been studied both theoretically [42] and through simulations [72, 88] and few preemptive scheduling systems have been built [80, 77, 46] to preempt the applications in response to changes in system environments. The main motivation for preemption of executing applications is to adapt to load changes both in terms of external load and number of jobs in the job queue thereby increasing throughput of the system. Preemption is also used to minimize the turnaround time of the applications. Although preemption has been widely used in traditional operating systems, it is a relatively new field in distributed systems.

And finally, scheduling systems can also be categorized based on their architecture. Most of the scheduling systems follow a centralized approach in which a central scheduler makes scheduling decisions for all jobs based on global information of the system. Recently, the benefits of multiple levels of scheduling systems have been studied using simulations [60, 68, 101]. While centralized schedulers can generate better schedules and involve less communication than multiple layers of schedulers, multiple levels of scheduling provides flexible scheduling in terms of different levels of scheduling policies and are more fault resilient than centralized scheduling.

1.2 Problem Statement

The focus of this research is to build a preemptive based metascheduling system for distributed and Grid computing that takes into account both application and system level considerations. The main objectives of the metascheduling system are to provide high performance to individual applications within the constraints of the system loads, to accommodate maximum number of applications into the system without overwhelming the system resources and to provide high throughput of the overall system. The dissertation also demonstrates an arbitration system that considers the problem of negotiating between different application schedulers. The scheduling system also deals with a framework for providing adaptive and realistic estimation of execution costs of the applications.

Secondly, this research dwells on building a user-level checkpointing library that will allow the application library writers to build malleable jobs which can migrate across distributed system resources and can change processor configuration and data distribution between migrations. The motivation of this portion of this research is to provide flexibility to the metascheduling system while scheduling different applications and reschedule the applications in the middle of the executions. This portion of the research also discusses the data structure for data map that is necessary for building the checkpointing library.

Finally, the research builds a flexible distributed computing framework that brings together the preemptible applications built by the application library writers using the checkpointing library and the metascheduling system that works with the preemptible jobs. The framework is flexible so as to allow the users to express different capabilities of the applications. The research also ponders over the special considerations that are necessary to deal with problems having roundoff error effect and mechanisms necessary to improve the confidence of the users of such problems in distributed environments.

1.3 Purpose of Study

Of the different objectives for scheduling systems mentioned in an earlier section, the objectives of providing high performance to the applications, accommodating many applications into the system and providing high throughput for the system are considered by our scheduling system. Meeting these objectives is essential for distributed and Grid computing, especially if the vision of using distributed systems for computing similar to the way that World Wide Web is used for information has to be realized. There are few scheduling systems for distributed computing [95, 59, 27, 113] aimed to provide high performance to individual applications. These scheduling systems are attractive for distributed and Grid computing since they allow users to get fast response times for applications that may otherwise execute for long duration on local machines. Also, there are large number of Grand Challenge problems [4] for which large number of instantaneous results are absolutely needed. In addition to collating large number of supercomputing machines for solving these problems, it is also essential for generating scheduling strategies that are tuned for solution of each instance of the problems. The existing high performance schedulers accomplish this task by requiring the application to specify problem constraints and complexities. Hence the mechanism of scheduling in high performance schedulers is called *application-level scheduling* [27].

While the existing high performance schedulers have been experimentally proven to provide fast response times, they can potentially hamper the performance of the system as well as the performance of competing applications especially when a number of applications are submitted simultaneously to the system in a multi-user distributed setting. This happens when the different application-level schedulers lay claim on same set of resources assuming the absence of competing applications. In the worst case, all the applications will be scheduled to the same resources and this leads to frequent swapping out of applications between the CPU and disks. This not only overwhelms the system resources but also defeats the whole purpose of high-performance schedulers to provide fast response times to applications. Hence there is a need for an arbitration mechanism that interacts with the application-level schedulers to oversee the progress of the applications and the smooth functioning of the system. Our research designs and implements this arbitration mechanism in the form of a *metascheduler*. The purpose of our metascheduler is to communicate with the different application-level schedulers to balance the interests of different applications. The metascheduler can also possess knowledge of the memory constraints for problem executions. Based on this knowledge and the total memory available in the system resources at a given point of time, it can either accept or reject an application-level schedule.

The second objective of our scheduling system is to accommodate many applications into the system. This objective is necessary to provide functionalities in distributed computing similar to certain Web services like News services that provide information to many users during certain peak hours. But due to the potential coexistence of different problems with different sizes in distributed computing systems, some large problems can occupy most of the system resources and can force our metascheduler to reject the application-level schedules of the applications that arrive later into the system. This will lead to increase in probability of applications getting rejected at a given point of time. In this situation, preemption of executing large applications to accommodate small incoming applications will be helpful in realizing the second objective of our scheduling system. Hence our metascheduling system is built as a preemption based scheduling infrastructure. Since our objective is also to provide high performance to the applications, preemption of executing applications is based on the remaining execution times of the application. Hence our metascheduler also depends on a monitoring framework that monitors the progress of the applications and reports the remaining execution times of the applications based on the time complexity specified in the application-level schedules and the progress of the application.

The third and final objective of our scheduling system is to provide high system throughput. Though this objective may not be critical for individual users of the system, throughput is considered an attractive measure of the system for drawing users to the distributed systems. Though the implementation of our metascheduler to realize the first two objectives of our scheduling system also leads to high system throughput, this may not necessarily be the case due to the large system dynamics associated with distributed computing resources. Hence preemption of applications is necessary to escape from heavily loaded resources and to utilize free resources due to completion of certain applications. Preemption is also necessary to prevent unfair advantage to large executing applications in the absence of which incoming small applications can be executed much faster. A powerful rescheduling framework is necessary to make appropriate decisions regarding if and when to reschedule the applications. Though the objectives of our metascheduling system are similar to the objectives of the scheduling systems like LSF [121] and PBS [9], our metascheduling system employs more robust policies for preempting and migrating applications than the other scheduling systems.

Since preemption of the executing applications is necessary to meet the different objectives of our scheduling systems, it is also necessary to enable the application library writers to easily build preemptible applications. The existing preemption based schedulers [72, 88, 80, 77, 46] either assume preemptible jobs using simulations [72, 88] or deal with moldable applications in which the applications can be stopped and continued on the same set of resources [80, 77] or work with less popular parallel programming languages [46]. Hence our research also focuses on building a user-level checkpointing library that the library writers can use in their applications to make their applications preemptible. The main focus of our checkpointing library is to help develop malleable jobs where applications can reconfigure in terms of processor configuration and data distribution. This flexibility is necessary for our metascheduler to implement its scheduling policies.

The final focus of our research is to build an actual Grid computing system that consists of our metascheduling framework and deal with preemptible applications. In order for the Grid computing system to be usable, this area of research concentrates on building flexible frameworks where applications with different preemptible capabilities can coexist. The Grid computing system called GrADSolve is unique in that it helps maintain execution traces of the problem runs and use the execution traces for problem runs corresponding to similar problems. This feature is desirable for applications whose results depend on the processor configuration and data distribution.

Thus, to summarize, the contributions of our research are the following:

1. A scheduling system designed with meeting the objectives of providing high performance of the applications, accommodating many applications into the system and providing high system throughput.

2. A performance oriented rescheduling framework for distributed system that employs firm decisions for rescheduling the applications in response to dynamic load changes of the system.

3. A user-level checkpointing library that helps in the development of malleable applications.

4. A flexible Grid computing system that employs the policies of our metascheduler

and deals with application of different capabilities and properties.

Our research work is different from many relevant areas of existing research.

1. The research builds a first known scheduling system that provides a balance between providing high performance for the applications and high system throughput. Most of the existing systems can meet only one of the two objectives.

2. Our research also designs and implements a first known migration framework that takes into account the remaining execution times of the applications.

3. The user-level checkpointing library developed by this research is the first of its kind to allow reconfiguration of the parallel applications in terms of changing the number of processors and data distribution.

4. The Grid computing system implemented in the research allows the application library writers to express more number of application capabilities than most other Grid computing systems.

5. The Grid computing systems is Remote-Procedure-Call (RPC) based system. It is the first RPC system that allows the transportation of application data from the user's machine to the target machine based on the data distribution information provided by the application library writers.

6. And lastly, the Grid computing system built by the research is the first system that attempts to deal with applications whose results depend on the systems used for execution.

Though our research is different from many existing relevant research areas, it also offers scope for extending some research efforts and collaborating with certain research partners. More and more applications in distributed computing systems have realized the importance of using application-level schedulers and have developed execution models for the applications to generate application-level schedules. Since our metascheduling system is not a wholesome scheduling system but interacts with application-level schedules, there is scope for interfacing our metascheduler with existing application level scheduling systems. Also, our user-level checkpointing library allows reconfiguration of applications using certain specific data distribution information provided in the Application Programming Interface (API). This data information can also be gathered from the API provided in other systems [63]. Hence it is possible to collaborate with other systems and enable the applications in those systems to be malleable thus providing different options to library writers to build malleable applications for use with our metascheduler. And lastly, the Grid computing system developed by the research that allows the coexistence of applications with different capabilities is a right start in the direction and can motivate other developers of Grid computing systems to deal with a comprehensive set of capabilities of the applications.

1.4 Assumptions and Limitations of the Research

The design, implementation and testing involved in the research have got few limitations that are worth mentioning.

1. The application-level scheduling involved in the research considers the entire set of resources in making scheduling decisions. In this approach, the local scheduling policies of the different domains containing the resources and the possible overhead caused by the local scheduling policies are not considered. This may be a drawback when the distributed system consists of different supercomputing centers with different administrative domains each implementing its own local scheduling policies and few of the domains may not be able to communicate to each other. We use a Grid computing toolkit with the help of which subprocesses of the application can span across different domains. Currently this Grid computing toolkit supports few kinds of systems although work is being made to interface the toolkit with other systems.

2. Currently, the scheduling framework assumes the existence of only regular applications in which all the sub processes exhibit the same kind of behavior. It does not take into consideration applications with different topologies like Master-Worker topology.

3. The applications must adhere to the single program multiple data (SPMD) programming model.

4. The user-level checkpointing library can be used with only message passing parallel programs based on MPI [98, 7]. It cannot be used in other kinds of parallel applications like shared memory based programs and data parallel applications.

5. The testing of the scheduling framework, the user-level checkpointing library and the Grid computing system were made on the systems available under the GrADS [26] computing testbed. Though the resources involved in the testbed possessed network heterogeneity, the heterogeneity of the computing nodes was limited.

1.5 Definition of Terms

API Application Programming Interface.

Application-level scheduling A kind of scheduling strategy where resources are selected for the execution of end application based on the application characteristics.

Application Manager A driver that is invoked by the user and that invokes various modules leading to the solution of user's problem.

Checkpointing Storing the various application states including the intermediate data and other process states in such a way that the application can be continued from the point when the applications states were stored by retrieving the stored information.

Contract Violations Conditions when the execution of the end application does not progress as expected.

Execution Traces Different information corresponding to executions of problems. The information can include the characteristics of the resources on which the applications are executed, the set of resources chosen for application execution, the data distribution used for the application etc.

GrADS Grid Application Development Software. A collaborative [26] project on Grid computing.

Grid An abstraction of a collection of resources and a set of middleware services to support solution of problems on the resources.

Malleable applications Parallel applications which can be stopped and continued on a different set of processors with different amount of parallelism.

Metascheduling A methodology where may layers of schedulers that interact with each other to determine the set of resources for problem solving.

Moldable applications Parallel applications which can be stopped and continued on the same set of processors.

MPI Abbreviation for Message Passing Interface - a standard for parallel programming using message passing.

Performance Contract A set of parameters that express the expected performance of an application.

Rescheduling To change the resource environments for application execution in the middle of the execution of the application.

Resource Selection The phase where the set of available resources along with the resource characteristics are retrieved from an information system.

RPC Abbreviation for Remote Procedure Call - a model of computing where the user invokes a procedure call that is executed on a set of remote resources.

1.6 Outline of the Dissertation

In the second chapter, all relevant background information necessary for understanding the other chapters in the thesis are presented. An overview of Grid computing is presented and some of the Grid computing projects are described in brief. Then, the different scheduling strategies of the present day schedulers, the definition of the metascheduling, some of the problems in the current scheduling strategies are explained. The rest of the chapter deals with a detailed explanation of the original framework of a Grid Computing project called GrADS, upon which the research is based. The deficiencies of the GrADS architecture are explained by means of experiments and the need for a metascheduler architecture to overcome the deficiencies in the GrADS framework is emphasized.

The third chapter first lists the functions and goals of the metascheduler. The GrADS architecture modified to include the metascheduling strategies is explained. The third chapter also describes in detail the different components of the metascheduler. One of the components of the metascheduler, the rescheduler, employs certain unique policies for rescheduling executing applications and hence described in great detail in a separate section.

To help the library writers to develop malleable applications, the research has developed a checkpointing system called SRS (Stop Restart Software). Chapter 4 first lists the motivations of developing the checkpointing system. The different important functions that constitute the checkpointing API are explained and few examples of usages are given. Finally, the use of the the SRS checkpointing system in the context of the metascheduler is described.

The fifth chapter presents various kinds of experiments conducted in the research and the results obtained. In the first section, the experiments and results for demonstrating the usefulness of the metascheduler components is presented. The comparison between the actual and the predicted redistribution costs are given. Various experiments conducted in the context of the SRS checkpointing system are given. The overhead associated with the checkpointing and the times for reading and writing checkpoints for different problem scenarios are presented. In the final section, some practical experiments conducted with the metascheduler are explained and the encouraging results are shown.

The metascheduling framework used for the experiments in Chapter 5 is based on an ad-hoc infrastructure. The deficiencies of the ad-hoc infrastructure is presented in the sixth chapter. The implementation of a systematic Grid Computing system called GrADSolve to overcome the limitations of the ad-hoc infrastructure is explained. The various entities in the GrADSolve system and the support for the entities in the GrADSolve system are explained. The section also deals with the detailed description of the framework of the GrADSolve system. The support in the GrADSolve system for execution traces is explained.

The contributions of the research work are summarized and the important extensions to the thesis are pondered in the final chapter.

Chapter 2

Background

This chapter is a lightly revised version of a paper published in the Journal of High Performance Computing Applications and Supercomputing, 2001.

A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche and S. Vadhiyar. Numerical Libraries and the Grid: The GrADS Experiments with ScaLAPACK. Journal of High Performance Applications and Supercomputing. 15(4), pages 359-374, Winter, 2001.

My primary contribution of the paper included (1) conducting experiments and obtaining results (2) contibuting sections for describing the framework used and (3) providing analysis of few results. This chapter revises the paper by describing some of the problems in the methodology used in the paper and motivating the need for the extension of the work. The chapter also gives a detailed background information for the dissertation not present in the paper.

In this chapter, relevant background information for our research is presented. The chapter begins with an overview of existing Grid computing projects. In Section 2.2, the scheduling methodologies that are currently practiced for distributed computing and their shortcomings are explained. Our research is based on a Grid computing project called *GrADS*. The overview of the GrADS project and its initial design are presented in Section 2.3. Some of the results that were obtained with the initial design of GrADS are presented in Section 2.4 to demonstrate the practical usefulness of the GrADS Grid computing system. The initial design of GrADS uses plain application-level scheduling. In Section 2.5, some of the limitations of this approach are explained and demonstrated with sample set of experiments. Finally in Section 2.6, the need for an arbitration mechanism or a metascheduler to overcome the limitations is emphasized.

2.1 Grid Computing - An Overview

Computational Grids [58] are powerful abstractions of traditional distributed computing systems and aimed towards achieving multiple objectives of providing seamless access to vast set of distributed resources to users, to allow remote access of hardware and software services, to expand the problem solving capabilities of the users, to allow efficient scheduling of applications to resources, to provide a framework for collaboration of different scientists to solve problems spanning different disciplines etc. While there are many Grid computing projects for realizing different objectives, we review some of the important Grid computing research in this Section.

Globus [57] is one of the pioneering efforts in Grid computing. The Globus project provides a toolkit of various tools which Grid computing developers can use either comprehensively or partially to build Grid computing systems. The various tools provided by Globus include tools for resource management, developing security infrastructure, data management and access and building information services.

Legion [65] is an object-oriented metacomputing system developed at the University of Virginia. Legion represents all kinds of resources including hardware and software as objects. Hence Legions provides an extensible architecture in which the users can implement their own objects. Legion also acts as a platform for high performance computing by its support for parallel applications. The scheduling policy implemented by Legion is based on reservation of system resources.

Condor [75, 116] is one of the oldest Grid computing systems. It supports high throughput computing with the objective of executing large number of long running applications on the system resources. The owners of the workstations can define policies for executing applications on their systems. Condor executes applications on workstations only when the owner is not using them and stops and migrates the applications when the owner wants to reclaim his workstations. Condor supports a flexible Classified Advertisements (ClassAds) mechanism for the expression of the job and resource properties. The Condor system with the help of these properties schedules the jobs to appropriate resources.

NetSolve [34] was developed at University of Tennessee and its main use is to allow contribution and usage of numerical libraries over the Grid. It follows a client-agent-server model and follows a Remote Procedure Call (RPC) mechanism where the users can invoke numerical applications remotely from their C, Fortran or Matlab programs. Ninf [96] is a system developed in Japan that has objectives, design and usage scenarios similar to NetSolve.

Nimrod [13] is a distributed computing system developed at Monash University, Australia. It is a tool for distributed parametric modeling where the users can execute a large number of parametric computational experiments, each specified by different sets of parameters. These parametric experiments play important roles in the area of Bioinformatics, Operations Research, Network simulation etc. Their recent version Nimrod-G [33] follows a model of computational economy for scheduling where costs are associated for each resource in the system.

GrADS [26] stands for **Gr**id **A**pplication **D**evelopment **S**oftware. It is a project involving a number of institutions across United States. It's goal is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. GrADS is explained in greater detail in Section 2.3.

AppLeS [27] provides powerful scheduling concepts for Grid systems. In AppLeS, application library writers provide execution or simulation models for the application that predict the execution cost for the application for a given set of resources with a given set of resource parameters. A search procedure by repeatedly invoking the simulation models determine a near-optimal application-specific schedule for application execution. Thus the approach of AppLeS is aimed to provide high performance to individual applications and hence falls under the category of high-performance schedulers.

2.2 Related Work in Scheduling

There have been number of efforts in devising and/or implementing scheduling strategies for heterogeneous distributed computing systems since the advent of Network of Workstations (NOWs) [37, 95, 121, 59, 111, 112, 27].

Some of the scheduling systems for distributed computing [16, 78, 105] mainly focus on maintaining the load balance of the system resources. In these systems, newly arriving jobs are allocated to a set of resources that are relatively lightly loaded in order to improve the resource utilization. Though these schemes improve system utilization, they may not provide good response times for individual application since heavily loaded supercomputing machines may still give better response times than lightly loaded local workstations.

The work by Khaled Al-Saqabi et. al [95] considers a 2D array of processors and time slices and assigns the Virtual Processes (VPs) of the jobs to the array. The 2D array is updated on processor exit, new virtual process, new processor and virtual process exit events. Scheduling based on time slices will lead to huge overhead for the scheduling system when the scheduling strategies have to be invoked frequently in response to frequent Grid dynamics. Also, this method will involve time consuming updates of the 2D array for large sized problems.

Load Sharing Facility [121] lays emphasis on distributing the jobs among the available machines based on the workload on the machines. The assumption that load sharing leads to good response times is not valid in a Grid scenario where the network heterogeneity can significantly affect the execution time of the application. MARS [59], Prophet [113] and more recently AppLeS [27] provide good approaches for application level scheduling in meta computing environments. The system and application characteristics are collected in a distributed setting and decisions on task allocations are made. MARS also provides for task migration based on changing load conditions. AppLeS is more suitable for Grid environment with its sophisticated NWS [115] mechanism for collecting system information. However, both MARS and AppLeS do not have powerful resource managers that can negotiate with applications to balance the interests of different applications. For e.g., let us assume that the Grid consists of powerful supercomputer M_1 and an ordinary workstation cluster M_2. If an application A_1 that first enters the system, through it scheduler occupies the powerful M_1, a second application A_2, that enters the system, will detect through its scheduler, that the supercomputer M_1 is loaded and will utilize the cluster M_2. If the performance differences between executing A_2 on supercomputer M_1 and on the cluster M_2 is huge and application A_2 can execute in a negligible amount of time if executed on the supercomputer M_1, then a good scheduling strategy will be to stop application A_1 on M_1, accommodate A_2 on M_1, and after A_2 completes on M_1, continue executing A_1 on M_1. Another scenario is when multiple applications are submitted simultaneously to the system, the application-level scheduling decisions made for each application can assume the absence of other competing applications and in the worst case, all applications can claim the same set of resources. Thus few of the resources of the distributed system can become heavily loaded and this in turn can lead to poor performance of both the system and the individual applications. In this case, an arbitration system is needed that interacts with the different application-level schedulers. These kinds of metascheduling decisions play significant roles in scheduling of jobs to Grid and the absence of these decisions will lead to various kinds of problems like the bushel of AppLeS problem [27].

The term metascheduler generally refers to using different levels of schedulers that interact with each other to make scheduling decisions [60]. Such a metascheduling system offers attractive benefits over conventional scheduling systems in terms of fault resilience where termination of few schedulers will not hamper the progress of the entire scheduling system and scalability where the scheduling decisions for jobs arriving in the systems can be distributed among the different schedulers. The metascheduling algorithms that have been studied [60, 68, 101] are mainly motivated by the existence of different administrative domains in the distributed system, each implementing its own local scheduling policy. Thus the top-level scheduler allocates a job to a domain based on the workload information collected from the different domains while the local scheduler in the domain allocates the jobs to its resources based on its own local scheduling policy. The metascheduler investigated in our research [106] is mainly motivated by the presence of different application-level schedulers and the need to interact with them to balance the interests of different applications. Thus the objective and hence the design of our metascheduler is different from those of existing metaschedulers.

Various Grid computing projects including Globus [57], Legion [65], Condor [75], NetSolve [34], Nimrod [33] and Ninf [96] have also considered the process of scheduling jobs to distributed resources. Globus [57] provides tools for information service, resource manager, security infrastructure, communication library etc. Globus does not provide a tool for implementing flexible scheduling policies in the Grid system. Globus team is working on supporting advance reservation and co-allocation but the scheduling does not try to minimize application completion time or increase system throughput. Legion [65] provides scheduling support through its (Collector, Scheduler, Enactor) tuple where a candidate schedule for a given application run is generated by either application level scheduling or general-purpose scheduling algorithms. But Legion does not have a negotiating mechanism to balance the interests of different candidate schedules. The Condor [75] system supports scheduling through the ClassAd mechanism where the application needs are matched with the system conditions. Although the task allocation policies implemented by Condor take into account both application-level and system-level considerations, task reallocation policies of Condor is limited in that it does not take into account the potential performance benefits that can be obtained for the applications due to reallocation. NetSolve [34] implements scheduling policy to reduce the system workload with the assumption that this will lead to improved application's performance. This is not always the case in Grid environment. The objectives of Nimrod-G's [33] scheduling policies are similar to those of our metascheduler where different users' requirements are balanced. Nimrod-G uses grid economies to implement its scheduling policies while our metascheduler uses predicted application time for our scheduling policies. Ninf [96] scheduling through its metascheduler is similar to NetSolve scheduling in that it tries to achieve load balancing. Though the Ninf scheduler had been evaluated when multiple clients run their jobs, no substantial mechanism has been implemented to guarantee performance for each client.

2.3 GrADS

GrADS [26, 3] is a Grid computing research involving number of institutions across United States. The goal of the Grid Application Development Software (GrADS) project is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. The GrADS project is exploring the scientific and technical problems that must be solved to make Grid applications development and performance tuning for real applications an everyday practice.

The University of Tennessee investigates issues regarding integration of numerical libraries in the GrADS system. In our previous work [81], we demonstrated the ease of integration of numerical libraries into the GrADS system and showed some results regarding the usefulness of the approach. In this section, the framework used in [81] is presented. For a more detailed description, the reader is referred to [81].

The primary goals of our effort in numerical libraries are to develop a new generation of algorithms and software libraries needed for the effective and reliable use of dynamic, distributed and parallel environments, and to validate the resulting libraries and algorithms on important scientific applications. To consistently obtain high performance in the Grid environment will require advances in both algorithms and supporting software.

Current numerical libraries for distributed memory machines are designed for heterogeneous computing, and are based on MPI [98, 7] for communication between processes. One such widely used library is ScaLAPACK [31], designed for dense matrix calculations. ScaLAPACK assumes a two-dimensional block cyclic data distribution among the processes. The user must select the number of processes associated with an MPI communicator, and also select the specific routine/algorithm to be invoked. In addition, the ScaLAPACK Users' Guide [31] provides a performance model for estimating the computation time given the speed of the floating point operations, the problem size, and the bandwidth and latency associated with the specifics of the parallel computer. The performance model assumes that the parallel computer is homogeneous with respect to both the processors and communication network. With the Grid both of these assumptions are incorrect and a performance model becomes much more complex. With the dynamic nature of the grid environment, the Grid "scheduler" must assume the task of deciding how many processors to use and the placement of data. This selection would be performed in a dynamic fashion by using the state of the processors and the communication behavior of the network within the grid in conjunction with a performance model for the application. The system would then determine the number and location of the processors for a given problem for the best "time to solution" on the Grid.

A framework was developed for the automatic selection of resources when numerical applications like ScaLAPACK are submitted to the GrADS system. The framework is

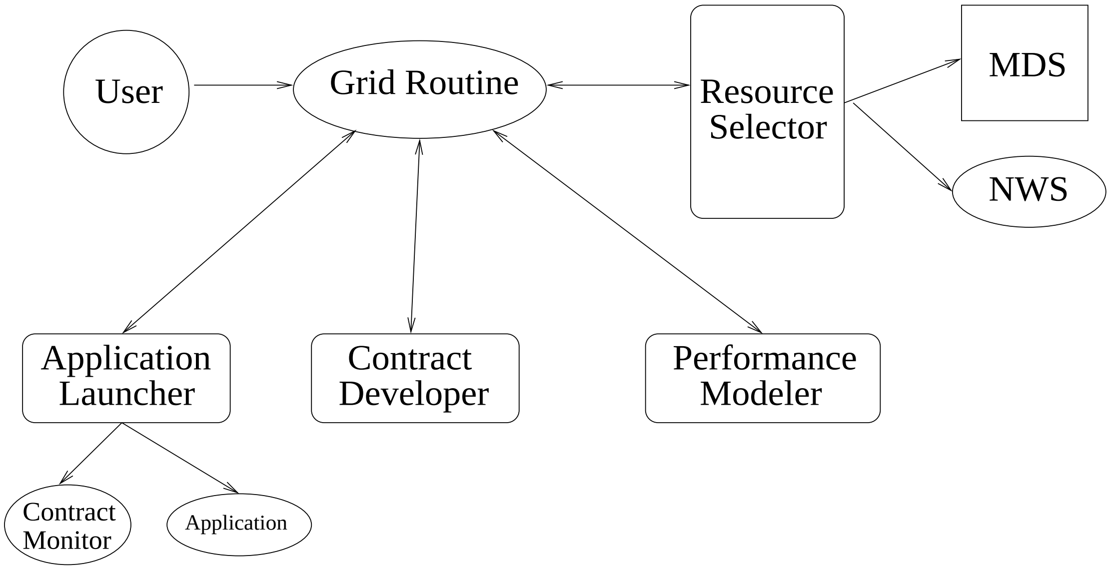

Figure 2.1: GrADS Architecture for Numerical Libraries

illustrated in Figure 2.1.

Before the user can start his application, the Grid system is assumed to have in initialized three components - Globus MDS [55], NWS [115] sensors on all machines in the Globus MDS repository, and the Autopilot Manager/Contract Monitor [93]. We assume that the user has already communicated with the Grid system (Globus) and has been authenticated to use the grid environment. The Globus MDS maintains a repository of all available machines in the Grid, and the NWS sensors monitor a variety of system parameters for all of the machines contained in the Globus MDS repository. This information is necessary for modeling the performance of the application, and for making scheduling decisions of the application on the Grid. Autopilot was designed and is maintained at the University of Illinois, Urbana-Champaign, (UIUC). It is a system for monitoring the application execution and enabling corrective measures, if needed, to improve the performance while the application is executing. . The Autopilot Manager must be running on one of the machines in the Grid prior to the start of the experiment.

After these preliminary steps have been completed, the user invokes a Grid routine with the problem he wants to solve along with the problem parameters. The Grid routine routine performs the following operations:

1. Creates the "coarse grid" of processors and their NWS statistics by calling the resource selector.

2. Refines the "coarse grid" into a "fine grid" by calling the performance modeler.

3. Invokes the contract developer to commit the resources in the "fine grid" for the problem.

   Repeat Steps 1-3 until the "fine grid" is committed for the problem.

4. Launches the application to execute on the committed "fine grid".

2.3.1 Resource Selection

The Grid routine invokes a component called Resource Selector. The Resource Selector accesses the Globus MetaDirectory Service (MDS) to retrieve a list of machines that are alive. The resource selector then contacts the Network Weather Service (NWS) to obtain machine-specific information pertaining to available CPU, available memory, and latency and bandwidth between machines. At the end of the resource selection step, a

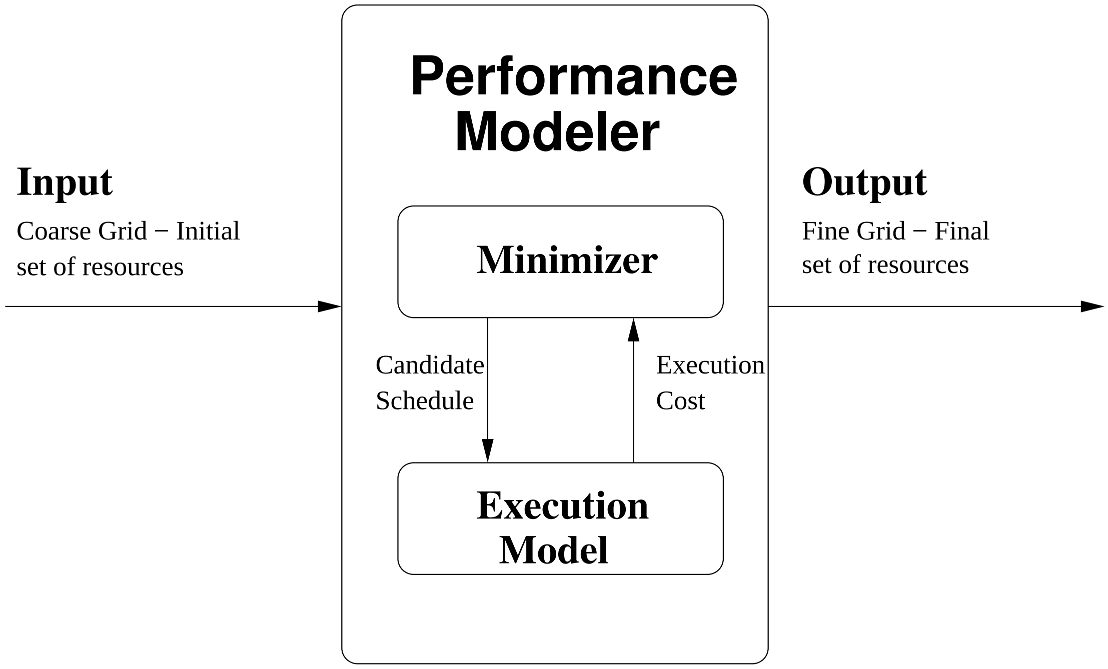

Figure 2.2: Performance Model Interactions

"coarse grid" is formed. This "coarse grid" is essentially all of the machines available along with the statistics returned by NWS.

2.3.2 Performance Modeling

The Grid routine then invokes a component called Performance Modeler with problem parameters, machines and machine information. The Performance Modeler consists of two components, the minimizer and an execution model. The interactions related to the Performance Modeler are illustrated in Figure 2.2.

The minimizer basically adopts a search procedure where it chooses and passes different candidate schedules to the execution model. The execution model is built specifically for the application and returns the time that the application would take if it were to execute on the list of machines passed to the execution model. The execution model uses simulation of the actual application on the sets of resources to determine the approximate execution cost of the application and also indicate if the given set of resources are sufficient to execute the application . Thus, by passing different candidate schedules to the execution model and collecting execution costs corresponding to the different schedules, the minimizer determines a near-optimal schedule for the application and returns a final list of machines for application execution to the Performance Modeler. By employing the application specific execution model, GrADS follows the AppLeS [27] approach to scheduling. The search procedure used in the minimizer can be based either on heuristics or linear programming. The search procedure used in [81] used a ad-hoc scheduling technique for determining the final set of resources for application execution. The details of the ad-hoc scheduling methodology is illustrated in Appendix A.1. At the end of performance modeling, the fine grid, which consists of a subset of machines for application execution, is returned to the Grid routine.

2.3.3 Contract Development

The problem parameters, the final list of machines and the expected execution times are passed as a contract to a component called Contract Developer. The concept of Contract Development was introduced by University of Illinois, Urbana-Champaign. A *performance contract states that given a* set of resources (e.g., processors or networks), with certain *capabilities (e.g., floating point rate or bandwidth), for particular* problem parameters (e.g., matrix size or image resolution), the application will exhibit a specified, measurable and desired *performance (e.g., render* r frames per second or finish iteration *i in* t seconds). For more details regarding the concept of performance contracts, the reader is referred to [110, 108]. The Contract Developer in the original GrADS framework is primitive in that it approves all the contracts that are passed to it.

2.3.4 Application Launching

The Grid routine then passes the problem, its parameters and the final list of machines to Application Launcher. The Application Launcher spawns the job on the given machines using Globus job management mechanism and also spawns a component called Contract Monitor. The function of the Contract Monitor is to monitor if the application execution is meeting its performance guarantees. When the application starts executing, the sensors associated with the application register with the Autopilot manager. The contract monitor looks up the autopilot manager to get information about the sensors, directly gets the application performance data from the sensors and displays the actual and predicted cost for the application.

2.4 GrADS Experiments and Results

An execution model was built for ScaLAPACK LU factorization. ScaLAPACK LU factorization is an iterative parallel application that involves right-looking LU factorization.

In each iteration, a column panel of the input matrix is factored by a process owning the panel. The factored column is communicated or broadcasted to the other processes which then perform updates of the matrix elements possessed by the processes. Hence the primary operations of ScaLAPACK LU factorization are factorizations, broadcasts and updates. These primary operations are simulated by the execution model. Thus the execution model predicts the execution times for factorization, broadcast and updates for each iteration. These times are added to obtain the total predicted execution time for a single iteration. The execution model obtains the sum of the predicted execution times for the iterations to determine the total predicted execution time for the ScaLAPACK LU factorization code. The ScaLAPACK factorization code was also instrumented with calls to Autopilot so that the actual execution times for the iterations are reported to the contract monitor.

The experiments were conducted on the machines in the GrADS testbed. GrADS testbed consists of about 40 machines from University of Tennessee (UT), University of Illinois, Urbana-Champaign (UIUC) and University of California, San Diego (UCSD). The characteristics of the machines are specified in Table 2.1.

The *torc*, *msc* and *cypher* clusters are connected to each other by single 100 Mb Ethernet connections. The rest of the clusters are connected to each other by Internet.

2.4.1 Validation of Execution Model for ScaLAPACK LU

In the first set of experiments, we validate the execution model for ScaLAPACK on a homogeneous cluster. In these experiments, we use 8 machines from *msc* cluster

Table 2.1: GrADS testbed resource characteristics

Cluster name	*Location*	*Nodes*	*Processor type*	*speed (MHz)*	*Memory (MByte)*	*Network*
torc	UT	8	Pentium III	550	512	100 Mb switched Ethernet
msc	UT	8	Pentium III	933	512	100 Mb switched Ethernet
cypher	UT	16	Pentium III	500	512	1 Gbit switched Ethernet
opus	UIUC	16	Pentium II	450	256	1.28 Gbit/sec full duplex myrinet
circus	UCSD	6	2 Pentium III, 4 Pentium II	450, 400	256	100 Mb switched Ethernet

in UT. In Figure 2.3, the loop iteration times predicted by the execution model are compared with the actual execution times for a problem of matrix size 16000. From the shape of the curves in Figure 2.3, we find that the execution model provides a good approximation of the execution times associated with the application.

In the second set of experiments, we show the behavior of the execution model for different problem sizes on the homogeneous cluster *msc*. A total of 8 machines were made available for the experiments. Due to the scheduler mechanism in GrADS, any number of processors ranging from 1-8 can be chosen for the execution of end application. In Figure 2.4, the total predicted execution times are compared with the total actual execution times for different matrix sizes. The number of processors chosen for the

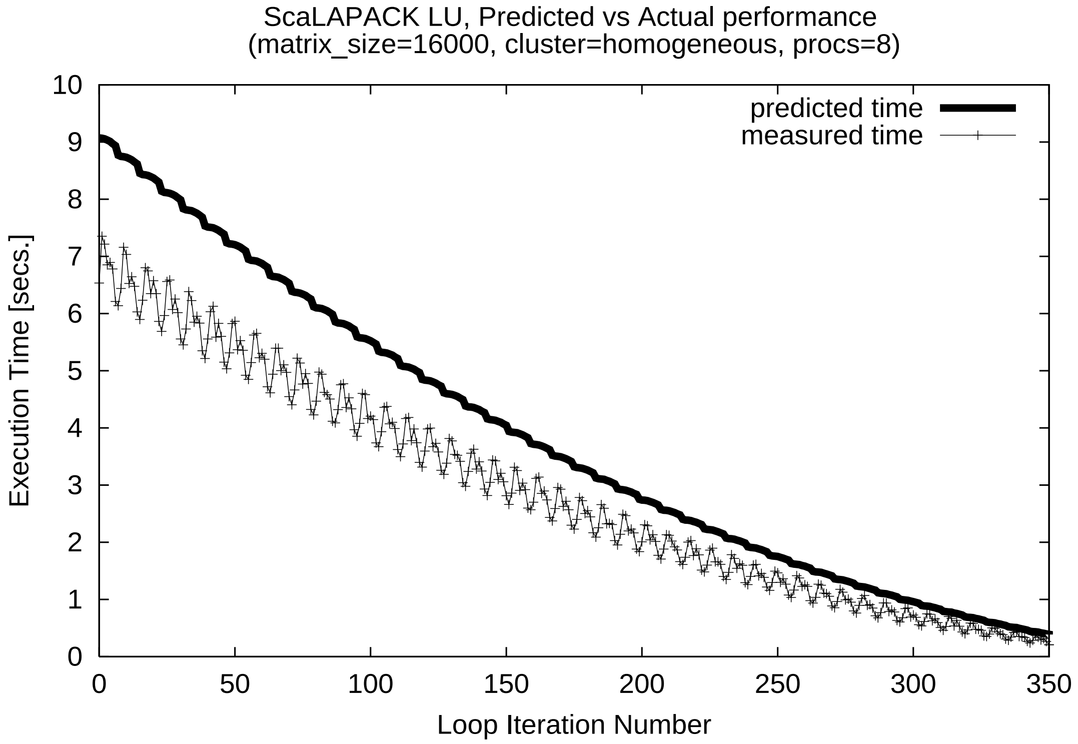

Figure 2.3: Validation of execution model on a homogeneous cluster for matrix size 16000

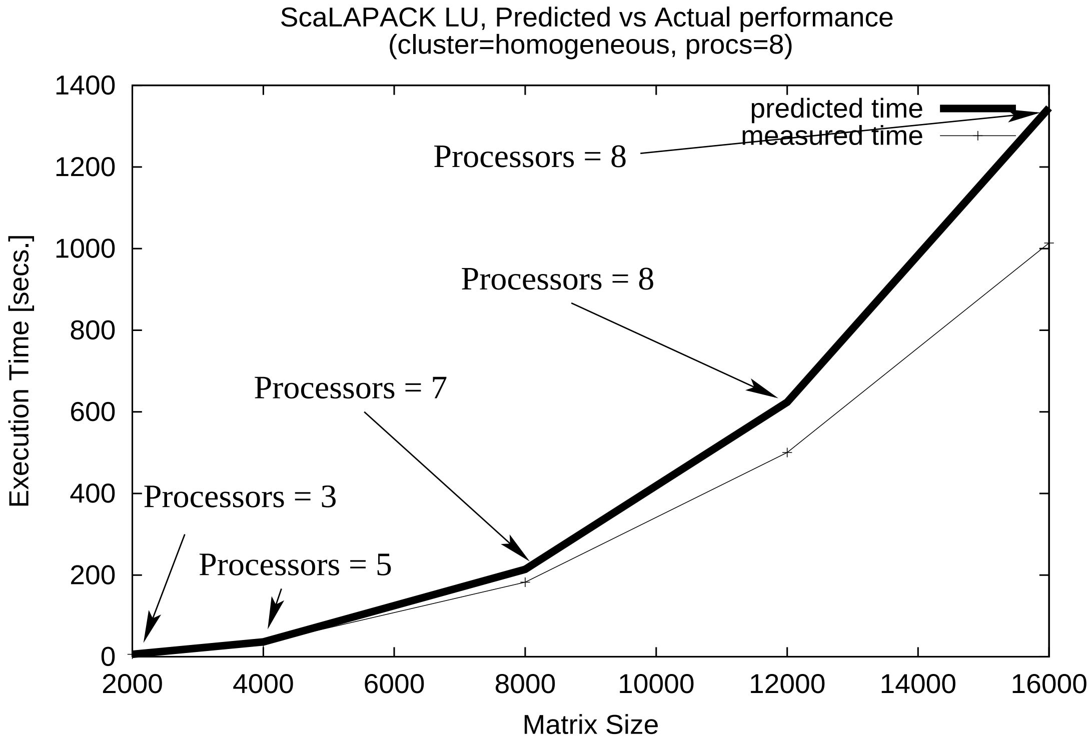

Figure 2.4: Validation of execution model on a homogeneous cluster for different matrix sizes

end application are also indicated in the figure. From Figure 2.4, we observe that the execution model is also able to give a good approximation of the relative execution times for different problem sizes.

In the third set of experiments, the execution model was validated on a heterogeneous set of machines for matrix size 16000. For these experiments 2 *msc*, 2 *torc* and 8 *opus* machines were used. In Figure 2.5, the predicted and actual per-iteration times are compared. We find that the execution model behavior is not satisfactory. This is due to the difficulty in modeling the communications of each iteration in the application.

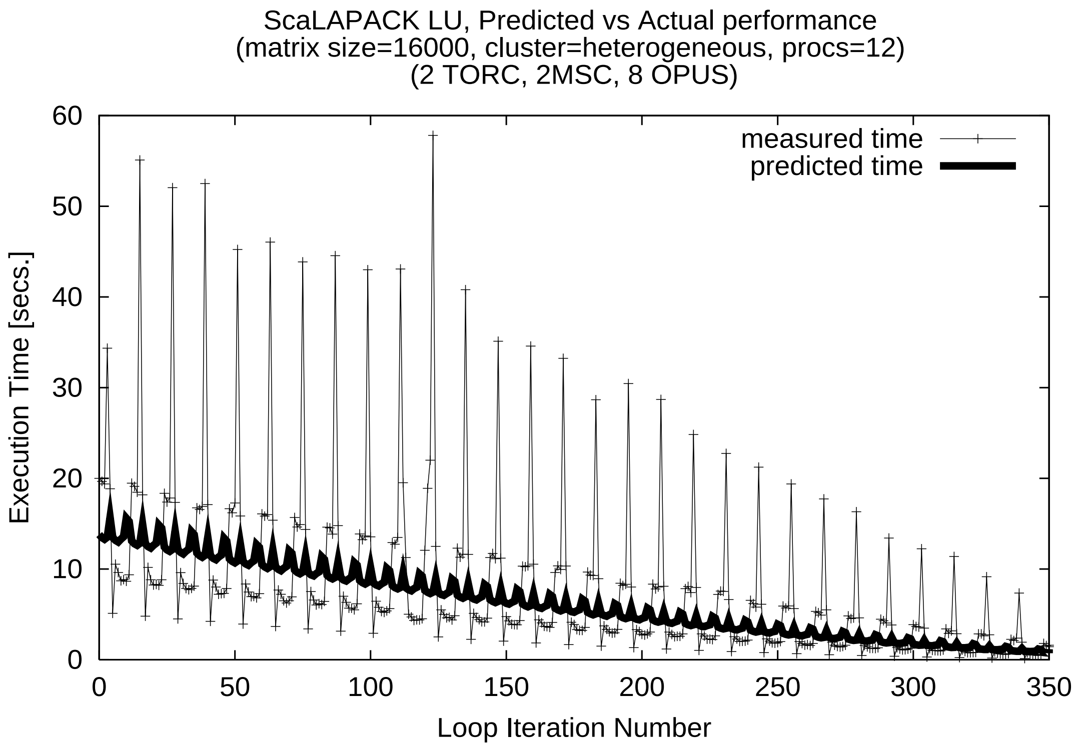

Figure 2.5: Validation of execution model on a heterogeneous environment for matrix size 16000 - Per-Loop iteration times

Most communication mechanisms use buffering schemes and it is difficult to predict the time for storing messages in the buffer. Also, the execution model assumes that all the processors start each iteration at the same time. Since this is not the case in ScaLAPACK application and due to the slow Internet bandwidth involved in the experiments, there are large discrepancies in the per-iteration execution times between the predicted and the actual values.

However, in ScaLAPACK, there is a natural synchronization for every n number of iterations, where n is the number of processes. Since the experiment illustrated in

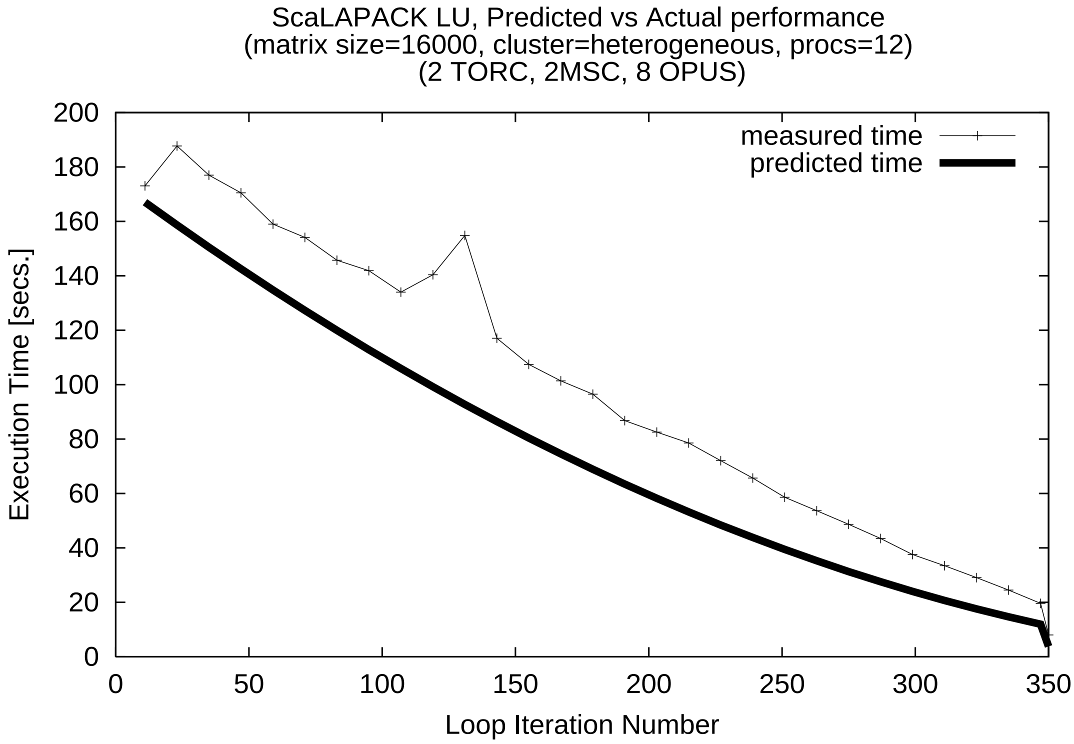

Figure 2.6: Validation of execution model on a heterogeneous environment for matrix size 16000 - Execution times every 12 loop iterations

Figure 2.5 uses 12 processes, the predicted and the actual values for the sums of every 12 iterations are compared in Figure 2.6. As can be observed from Figure 2.6, the execution model provided good approximations on heterogeneous environments. Thus from Figures 2.3 - 2.6, we can conclude that the execution model for ScaLAPACK LU factorization, barring certain deficiencies, mostly provided good approximations of the actual execution times.

2.4.2 GrADS Overhead

The validated execution model for ScaLAPACK LU was integrated into the GrADS framework. The GrADS framework was then used to schedule and execute applications over the resources. In the experiments in this section, 8 *msc* machines were made available to the GrADS executions. The GrADS scheduling mechanism chose a certain number of machines, ranging from 1-8, for the execution of the end application. In Figure 2.7, the non-Grid executions of the ScaLAPACK applications are compared with the Grid executions of the applications for different matrix sizes. For the non-Grid executions, the popular MPICH [66, 67] implementation of MPI was used for the implementation of the ScaLAPACK parallel application. For Grid execution of the applications, MPICH-G [56] implementation of MPI was used for the end application and spawning the processes onto the resources.

The left bars in Figure 2.7 represent the non-Grid execution times and the right bars represent the total Grid execution times. The processors used in the non-Grid and Grid runs are indicated in the figure. For the non-Grid runs, all the 8 machines in the *msc* cluster were made available for execution. For the Grid runs, even though 8 *msc* machines were made available, the GrADS scheduling mechanism chose a subset of the 8 machines for the eventual execution of the end application. For the non-Grid runs, times corresponding to MDS and NWS retrieval, and performance modeling do not exist. We find that the times for starting and executing the applications over the same set of resources, e.g., for matrix sizes 12000 and 16000, are better in the non-Grid runs

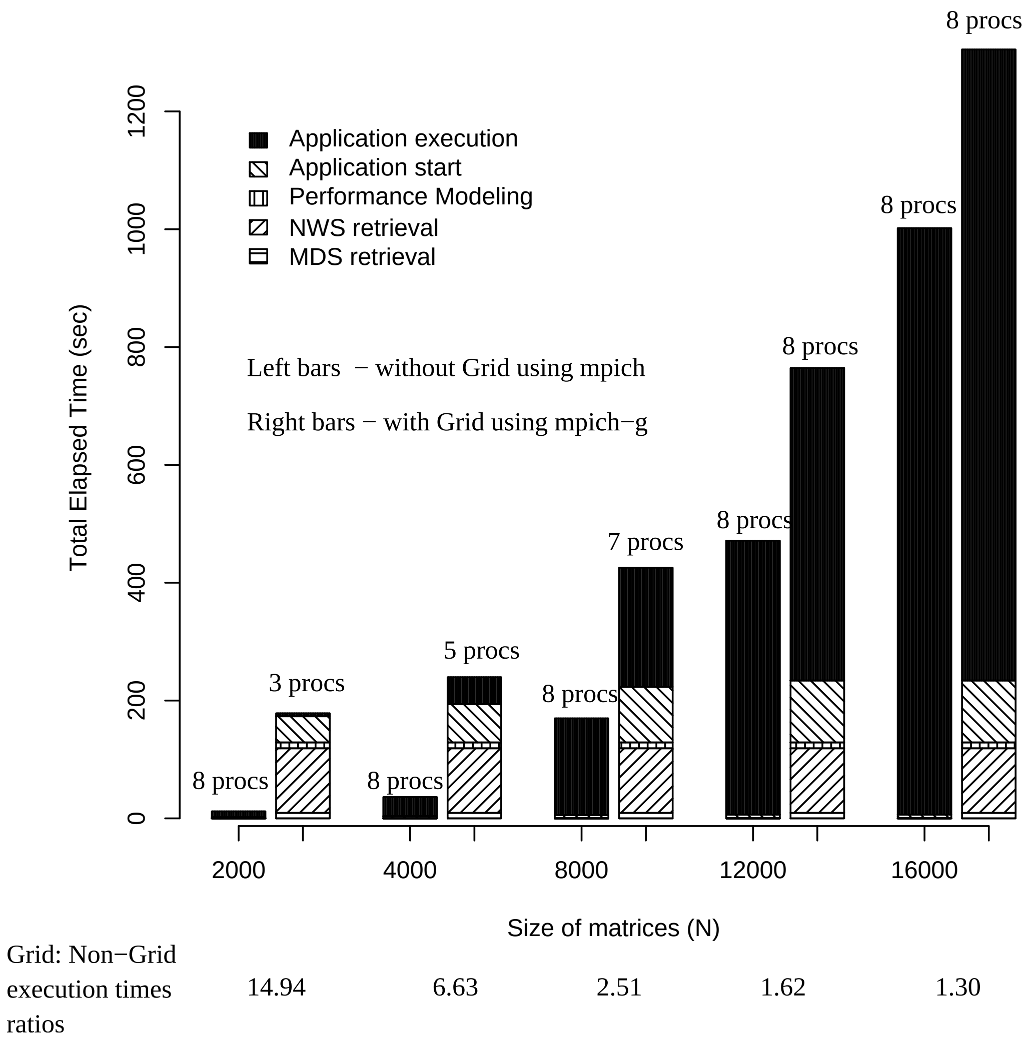

Figure 2.7: GrADS overhead on a homogeneous cluster

than in the Grid runs. This is due to difference in MPI implementations used in the non-Grid and Grid executions. Also, we observe that the time for performance modeling in the Grid runs are negligible. Thus the scheduling mechanism used in GrADS for determining the near-optimal set of resources incur very little overhead.

The ratios between the total execution times for the Grid and the non-Grid runs are also given in Figure 2.7. We observe that for small problem sizes using the GrADS framework is not advisable due to the overhead associated with GrADS. As the problem sizes increase, the impact of the overhead on the Grid runs decreases and hence the ratio between the execution times for the Grid and non-Grid runs also decreases. For the largest problem size that was solvable on the *msc* cluster, matrix size 16000, the overhead was only 30%. For matrix sizes beyond 16000, GrADS did not allow the application into the system since the application execution will involve frequent access to disks and also heavy intrusion into the other processes on the shared set of resources.

The results in Figure 2.7 indicate that it is advisable to use the GrADS framework for only large problem sizes. Though the results indicate that better results can be obtained when using a non-GrADS framework, GrADS relieves the burden of the user to chose the resources for his application. Also, GrADS determines the threshold of problem size for the application beyond which there will be severe degradation in the performance of the application and the system.

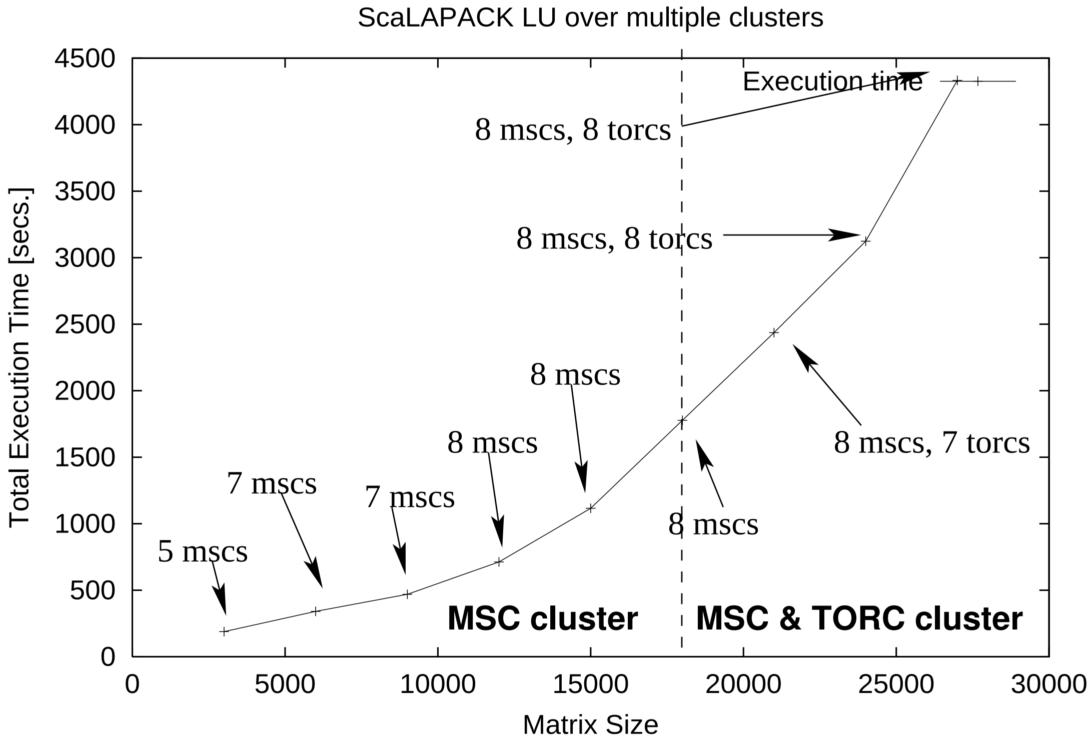

Figure 2.8: GrADS execution across the entire GrADS testbed

2.4.3 GrADS Execution across Multiple Clusters

In this section, all the machines available in the GrADS testbed, namely machines in *msc*, *torc*, *opus* and *circus* clusters were made available for the execution of the ScaLAPACK LU application. Different problems corresponding to different matrix sizes were input to GrADS framework. The GrADS scheduler determined the final set of resources for the execution of the end application for each run. Figure 2.8 indicates total execution times including the GrADS overhead for different problem sizes.

Since the *msc* machines have high processing power, the *msc*s were chosen for small problem sizes. For ScaLAPACK LU application of matrix size 21000 to be executed on 8 machines, the per-machine memory required is about 440 MB. Though the memory capacity of the *msc* machines is 512 MB, the memory available at the time of the experiments was less than 440 MB due to the presence of other processes on the shared resources. Execution of problem size of 21000 in this scenario will lead to frequent disk accesses and will result in severe degradation of the system performance for the other processes using the system. Hence, beyond matrix size 18000, GrADS chose machines from both *msc* cluster and the next best cluster in terms of computing power, *torc*. The maximum problem size that was solvable on the entire GrADS testbed was for matrix size 27000. Beyond this problem size, for e.g.. matrix size of 30000, the per-processor memory needed for the execution of the application on 16 processors is 450 MB which was not available at the time of conducting the experiments. We also find that the *opus* and *circus* machines were not used for any of the problem runs. This is due to the superior computing power of the *msc* and *torc* clusters. For problems of matrix sizes greater than 27000, e.g., matrix size of 30000, the per-processor memory needed to execute the application on the entire GrADS testbed, i.e. 29 machines, is 248 MB. This memory capacity was not available on some *opus* and *circus* machines due to the presence of other applications on the resources.

2.5 Deficiencies of the GrADS Architecture

Although GrADS has been proven to be useful in the previous sections, there are some deficiencies in the GrADS framework that prevent it from being completely useful in Grid environments. The major deficiency is the use of only application level scheduling by employing an execution model for the application. The application level scheduling does not take into account other competing application due to the lack of knowledge of the existence of the other applications. Thus when applications are executed in a competing environment, the resulting actual performance of the applications may be much less than the expected performance. Also, large number of competing applications can severely degrade the performance of the entire system.

To prove the deficiencies in the GrADS architecture, an experiment was conducted where two problems corresponding to matrix size 16000 were input to the GrADS system at almost the same time. For this experiment, 8 *msc* machines were utilized. Thus the experimental setup is similar to the setup used for obtaining the Figure 2.3. Figure 2.9 plots the actual and predicted values for per-iteration times for the ScaLAPACK LU factorization of matrix size 16000 in the presence of a competing application whose matrix size was also 16000.

Unlike in Figure 2.3, the predicted per-iteration execution times in Figure 2.9 do not correspond satisfactorily with the actual per-iteration execution times. This is because, when both the applications are input to the GrADS system at the same time, the GrADS framework obtains information about the machines from NWS at the same

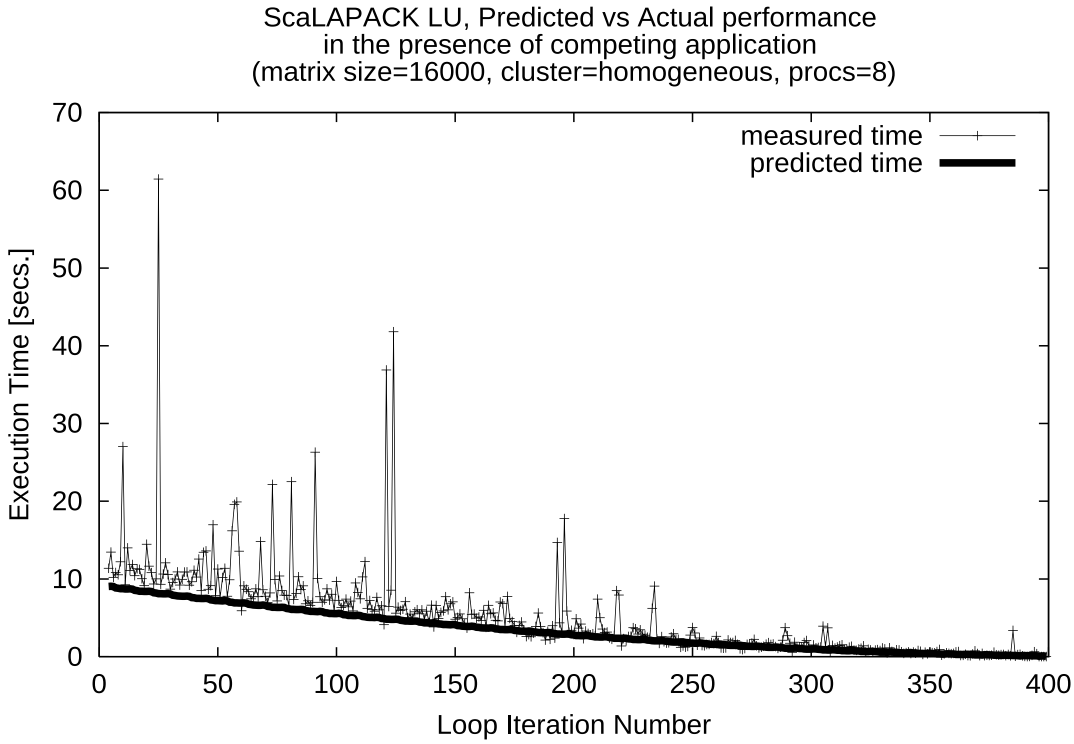

Figure 2.9: Result of deficiency of the GrADS architecture

time. Hence the performance modelers for the two applications use the same information about the resources and make the same decisions for determining the final schedule for the execution of end application. The machine information obtained from the NWS and used by the performance modeler correspond to the scenario when the resources do not execute either of the two applications. When the Application Launchers corresponding to the two applications launch the applications on the same set of resources at about the same time, the two applications execute in the presence of each other and hence contend for the resources. Since the memory capacity of the *msc* machines is less than the memory capacity needed to execute two problems of matrix sizes 16000, the two applications frequently access disks for data. This leads to severe performance loss for both the applications and the system resources. This loss in performance is manifested as the spikes seen in Figure 2.9 for the measured per-iteration execution times. If the contract monitor employs a rescheduler to migrate applications when it notices performance loss for the application, unnecessary overhead is incurred in invoking the rescheduling decisions.

Another deficiency of the GrADS framework, as illustrated in Figure 2.5 is that the execution model provided by the application library writers may not closely match the behavior of the application. In these cases, the contract monitor may assume the interference of the external load on the executing applications and may invoke the rescheduler, leading to unnecessary rescheduling overhead.

2.6 Need for a Metascheduler in GrADS

The problems with the GrADS architecture mentioned in the previous section illustrate the need for a robust contract development and arbitration mechanism that balances the interests of different applications. For the experiments in the previous section, one possibility is for the contract development system to accept the contract of one application and to reject the contract of the other application. This gives satisfactory performance for one of the applications without degrading the system and present a true picture of the system resources to the other application.

Besides simply acting as a queuing system, the arbitration mechanism can also make intelligent decisions for accepting or rejecting the application contracts. For e.g., the arbitrator can reject the contract of the applications if it determines that the addition of the application to the system can severely degrade the performance of the already executing applications on the system. Also, to meet the original objectives of the scheduling system in our research, i.e. to provide high performance to individual applications within the constraints of the system loads, to accommodate maximum number of applications into the system without overwhelming the system resources and to provide high throughput of the overall system, preemption of executing applications is desirable. In this case, firm scheduling policies can be built into the arbitration mechanism for preempting executing applications.

In spite of the arbitration mechanism, performance loss of the executing applications due to the interference by executing applications is unavoidable in Grid systems. The

Contract Monitor can be extended to contact a rescheduler on noticing performance losses of the applications and the rescheduler can employ firm rescheduling policies to migrate executing applications from heavily loaded systems or to newly available resources. The Contract Monitor can also be extended to dynamically adjust the expected performance levels in order to avoid contacting the rescheduler frequently and to reduce the resulting rescheduling overhead. The dynamic adjusting of contract limits is necessary especially in cases when better resources are not available for rescheduling or when the execution model of the application is not accurate.

These arbitration, contract development and rescheduling policies are implemented in the form of a *metascheduler* discussed in the next chapter.

Chapter 3

Metascheduling Framework

This chapter includes lightly revised sections of the following three papers.

Sathish S. Vadhiyar and Jack J. Dongarra. A Metascheduler For The Grid. Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing. pages 343-351. July, 2002.

Sathish S. Vadhiyar and Jack J. Dongarra. A Performance Oriented Migration Framework for the Grid. To appear in the Proceedings of The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003).

Sathish S. Vadhiyar and Jack J. Dongarra. Self Adaptivity in Grid Computing. Submitted to the special issue of Concurrency: Practice and Experience on Grid Performance, 2003.

I was the primary contributor of the papers and was involved in the design and implementation of the frameworks and verification by experiments. This chapter revises the papers by providing a detailed motivation of the work and more detailed descriptions of the sections in the papers.

In order to overcome the potential problems in the GrADS architecture and to realize the objectives of providing high performance to individual applications within the constraints of the system loads, accommodating maximum number of applications into the system without overwhelming the system resources and providing high throughput for the overall Grid system, a *metascheduler* was developed for the GrADS framework. Conventional schedulers take as inputs, problem parameters and machine characteristics and generate a schedule which is the final list of machines on which the application will execute. The metascheduler designed and developed in this research is different from conventional schedulers in that it takes as inputs, schedules generated for individual applications by conventional schedulers, especially application-level schedulers and accepts or rejects the schedules based on current system conditions and the presence of other competing applications in the system. Thus the metascheduler negotiates between different application-level schedulers to balance the interests of different applications.

The functions and goals of the metascheduler include

1. **trying to accommodate new applications by waiting for long running applications to complete.**

   An individual application-level scheduler for an application can determine by means of the problem constraints expressed for the application that the current resources are not sufficient for the application execution. Hence the application may be prevented from making further progress in the Grid system. But a large application that occupies the resources may complete its execution within a reasonable amount of time and the completion of the large application can facilitate the execution of the new application for which the application-level schedule is determined. In this case, waiting for the large application to complete and accommodating the new application will help increase the number of jobs that are accommodated into the Grid system.

2. **accommodating new applications by stopping long running jobs in the presence of which new applications will not be able to execute.**

   There may be some long running jobs in the system in the presence of which the system resources may be insufficient for the execution of a new application. The long running jobs may have large remaining execution times. Waiting for the long running jobs to complete in this case will lead to poor response time to the user who has submitted the new job to the system. If the new application has a relatively shorter execution time compared to the remaining execution times of the long running jobs, then few of the long running jobs can be preempted from the system, the new application can be allowed to execute and after the short job completes, the long running jobs can be reaccommodated into the system and continued from the previous point in execution. Since, the new application has a short execution time, the performance loss incurred for the long running application due to preemption will be minimal. The pro-active preemptive strategy allows to increase the number of short jobs that can be accommodated into the system.

3. **verifying that the applications made their scheduling decisions based on conditions of the system when competing applications are executing.**

   Since the application-level schedulers make their scheduling decisions independently, the information about resource properties used for the different application-level schedules may be the same in situations when multiple applications are executed simultaneously. This may lead to contention for resources by different applications due to the lack of knowledge of the existence of competing applications by the application-level schedulers. Eventually, this will result in overall performance degradation of the system. Hence an arbitration mechanism is needed that accepts few application-level schedules, accommodates few applications onto the system, and rejects the other application-level schedules prompting the corresponding applications to generate new application-level schedules taking into account the change in resource information caused by the execution of applications accommodated into the system by the arbitration mechanism.

4. **facilitating new applications to execute faster by stopping certain competing applications.**

   A large application may be executing on the system. The system resources may be sufficient to accommodate a small application that enters the system. But the execution time of the small application if it were to execute in the presence of the large application may be much larger than the execution time of the same small application if it were to execute in the system free of the large application. This may be due to the allocation of inferior system resources to the small application due the occupation of superior resources by the large application. In this scenario, the large application may be removed from the resources and the small application can be accommodated on the superior system resources. After the small application completes, the large application can be allowed to continue. Since the large application has long remaining execution time, the performance loss for the large application due to preemption may be minimal when compared to the performance loss for the small execution if it were to execute in the presence of large application. Thus penalizing large applications in favor of small applications will lead to increased throughput for the overall system.

5. **minimizing the impact that new applications can create on already running applications.**

   An application-level scheduler for an application may generate an application-level schedule which when approved will lead to severe performance losses of some executing applications. An arbitration mechanism can suggest a new application-level schedule for the application that is beneficial for the application and also reduces the interference caused by the application on executing resources. This may be accomplished by removing few resources that are used by few executing applications, from the original application-level schedule of the new application.

6. **migrating executing applications.**

   In some cases, migration is necessary when performance expectations are not being met for an executing application due to the change in resource characteristics of the machines on which the application is executing. The change in resource characteristics may happen due to sudden increase in external load on the resources. In these situations, the application may be stopped, a new application-level schedule may be developed for the application and the application may be migrated to the set of resources determined by the new application-level schedule.

In other cases, an executing application may complete thereby making available few free system resources. Some of the executing applications may be migrated to make use of the free resources to improve the performance of the applications. These kinds of migrating decisions help to maintain load balance in the system thereby resulting in high system throughput.

The metascheduling and arbitration mechanisms are implemented by the addition of four components, namely *database manager*, *permission service*, *contract negotiator* and *rescheduler* to the GrADS architecture. In this chapter, the modified GrADS architecture is discussed and the behavior of the GrADS applications in the modified setup is elaborated. The components of the metascheduler are described in detail. One of the components, the *rescheduler* for migrating executing applications is robust and unique in many ways. A separate section is devoted to the description of the rescheduling framework.

3.1 Modified GrADS Architecture

The modified GrADS architecture with the metascheduling components is depicted in Figure 3.1. The metacheduling components are shown shaded in the figure.

As in the original architecture, the user submits his problem to the Grid routine or Application Manager. The Application Manager registers the problem with the

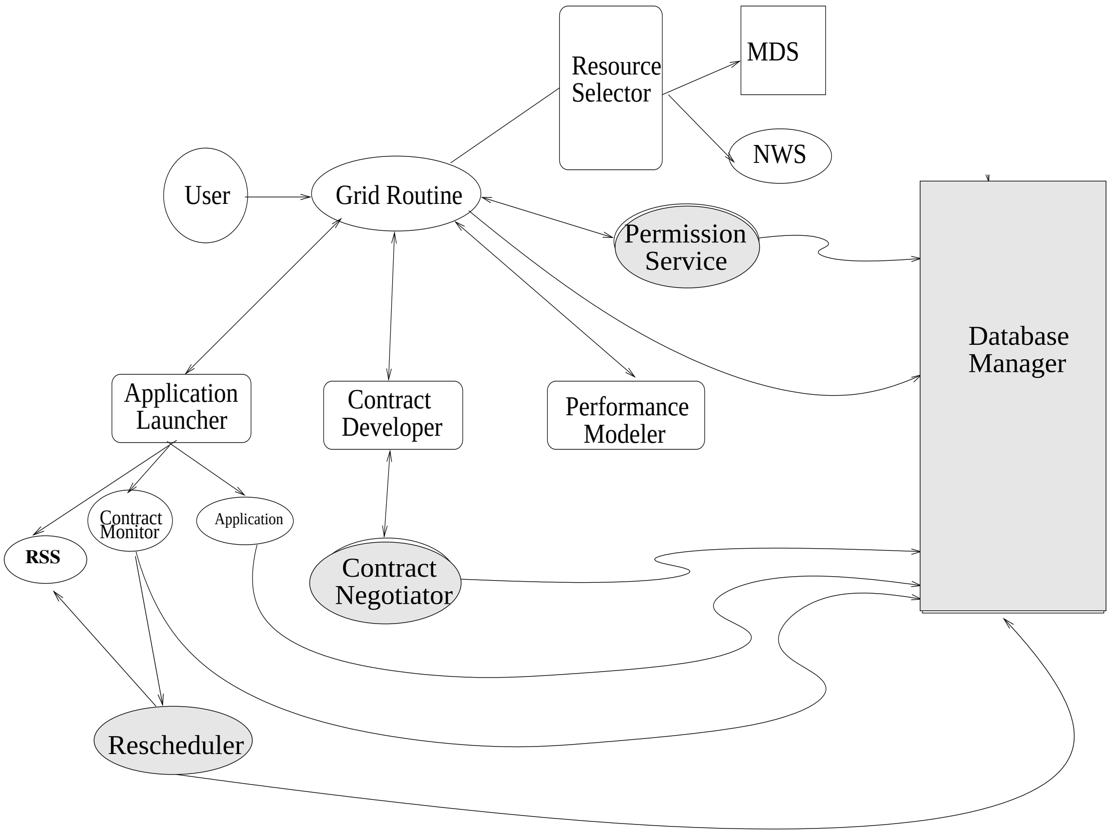

Figure 3.1: Modified GrADS Architecture

Database Manager. The Application Manager then invokes the Resource Selector. The Resource Selector invokes the Globus Metadirectory Service (MDS) to get a list of available resources. The Application Manager retrieves the information about resource characteristics from Network Weather Service (NWS). The list of machines and the resource information of the machines are stored in the DataBase Manager corresponding to the entry for the application.

The Application Manager passes the problem parameters and resource information to a metascheduler component called Permission Service. The Permission Service can grant permission for the application to proceed to the next stages of the GrADS application or can reject permission to the application in which case the Application Manager aborts. The Permission Service can also prompt the application to pass through the Resource Selection phase again.

If the permission is granted, the Application Manager proceeds to Performance Modeling phase. The Performance Modeler accepts problem parameters and resource characteristics from the Application Manager. It then uses an application-specific execution model to generate an application schedule which is the final list of machines for application execution. The application schedule along with the predicted performance cost are returned to the Application Manager. These parameters are stored by the Application Manager in the Database Manager.

The Application Manager passes the application schedule and the predicted performance cost as a *contract* to the Contract Developer. Unlike the original GrADS architecture, the Contract Developer does not approve all contracts. It passes the incoming contract to a metascheduling service component called Contract Negotiator. The Contract Negotiator can either approve or reject the contract. If the contract is approved, the Application Manager proceeds to the next stages of the GrADS execution. If the contract is rejected, the Application Manager goes back to the Resource Selection phase. In either case, the Application Manager stores the Contract Development result in the Database Manager.

The Application Manager passes the problem, its parameters and the final list of machines to Application Launcher. The Application Launcher spawns three components - the end application on the final set of machines using Globus job management mechanism, a component called Contract Monitor and a supporting component for the end application called Runtime Support System (RSS). The Contract Monitor monitors the progress of the application and compares with the predicted behavior. If the actual behavior of the application differs from its predicted behavior, the Contract Monitor contacts a metascheduling component called *Rescheduler*. The Rescheduler can decide to migrate the application in which case it contacts the RSS to stop the application. The various application states are stored by the end application in the Database Manager.

After spawning the numerical application through the Application Launcher, the application manager waits for the job to complete. The job can either complete or suspend its execution due to intervention by Permission Service, Contract Negotiator or Rescheduler. These application states are passed to the application manager through the Database Manager. If the job has completed, the Application Manager exits, passing success values to the user. If the application is stopped, the Application Manager waits for a resume signal and then collects new machine information by starting from the Resource Selection phase again.

The life cycle of an application and its interactions with the metascheduler is shown in Figure 3.2.

3.2 Metascheduler Components

The Database Manager, the Permission Service, the Contract Negotiator and the Rescheduler together form the Metascheduler. The interactions between these different metascheduler components and the interactions between the applications and the metascheduler are illustrated by Figure 3.3.

Most of the metascheduling components can possibly preempt executing application. Before preempting an executing application, the metascheduler checks if the application is preemptible. An executing application is preemptible if it has made atleast 20% progress since its last preemption or the beginning of its execution. Also, for the sake of simplicity, applications that are executing due to preemption of other applications are not preempted.

The following subsections describe each of the metascheduler components.

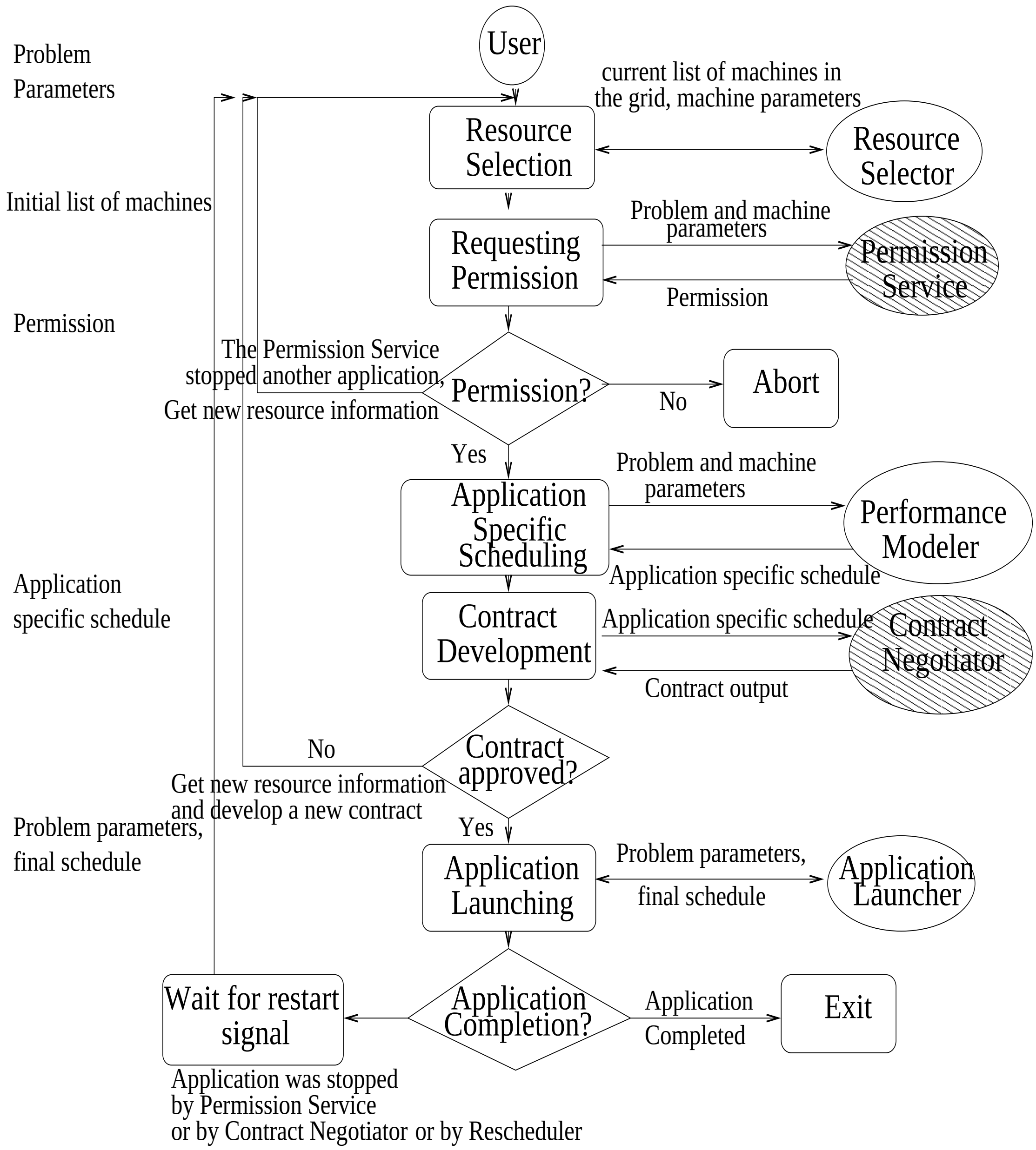

Figure 3.2: Life cycle of an application in the Grid

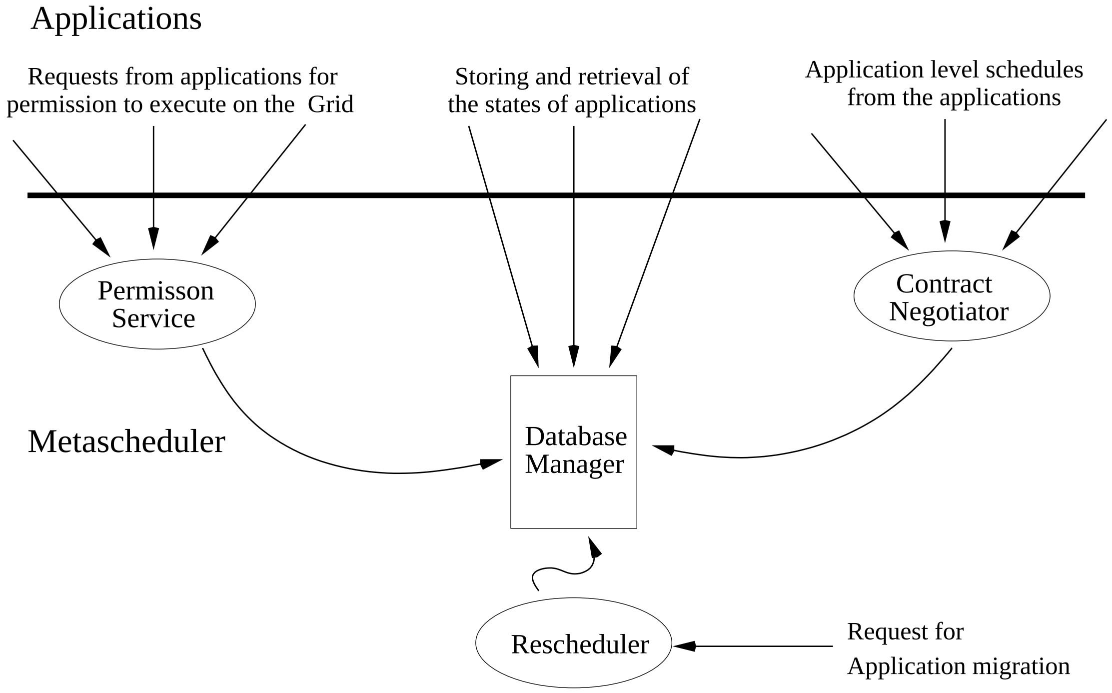

Figure 3.3: Metascheduler and interactions

3.2.1 Database Manager

Database Manager is a daemon listening for requests from the clients. The various requests correspond to storing and retrieving of information for the GrADS applications. The Database Manager also maintains a global clock and sends the global time maintained by the clock to clients on requests. The Database Manager also has event notification capabilities where the clients can register their interests on the occurrence of particular events and the Database Manager notifies the clients when the events occur. In addition to the database for storing information about GrADS applications, the Database Manager also maintains a database to keep track of the activities in the metascheduler.

The GrADS application and the metascheduler components store and retrieve various information to and from the Database Manager respectively. The information include the different states of the application as it passes through the different phases of the GrADS execution, the problem parameters, the constraints for the problem, the initial set of machines available for the application execution, the resource information including the memory, current load, peak performance and network characteristics of the machines, the predicted performance cost for the end application, the location of the application-level scheduler, the Contract Monitor and the Runtime Support System (RSS) and the performance behavior information of the application including the average of the ratios between the measured and expected performance and the number of times the measured performance values crossed the thresholds of expected performance or the number of contract violations. In addition to the above information passed by the clients to the Database Manager for a GrADS application, the Database Manager stores its own information for the applications including the global times when the end application started and completed. When an application stops or completes, the Database manager calculates the percentage completion time, *percent_completion_time* for the application from the average of the ratios of the actual and predicted performance costs for the application, *avg_ratio*, the difference between the current time and the time when the application started execution, *time_diff* and the predicted execution time for the application, *predicted_time* using the following equation.

$$percent\_completion\_time = percent\_completion\_time + \frac{time\_diff \times 100.0}{predicted\_time \times avg\_ratio} \tag{3.1}$$

The various states of the GrADS application stored in the Database Manager include the states corresponding to when the application initially registers with the Database Manager, after the application performs Resource Selection, the result of Contract Development, i.e. approval or rejection of the contract and the states of the end application including when it started, stopped and resumed. The Database Manager also stores information regarding if another GrADS application is waiting for the current GrADS application to complete.

In addition to storing and retrieving capabilities, the Database Manager also possesses event notification capabilities. The events mostly include the change of states for the GrADS application (for e.g., when the end application completes). The clients, mainly the GrADS application and the metascheduler components register their interest in certain change of states of the GrADS applications and the end applications. During this process, the Database Manager stores the information necessary for communication to the clients. The clients wait for notification from the Database Manager. Later, when the state of a GrADS application is updated, the Database Manager checks if there is any client waiting for notification of the particular update of the state. If it finds such a client, it communicates with the client using the communication information previously stored. The client gets the notification from the Database manager and resumes its operation. As an example, a metascheduler component may inform the Database manager that it wants to be notified when a particular GrADS application completes. The GrADS application may be currently executing on the system resources. Later, the GrADS application will complete and store the completion state in the Database Manager in the entry corresponding to the application. When the Database Manager stores the completion state, it finds that a metascheduler component has expressed interest in the completion state of the application, retrieves the communication information for the metascheduler component previously stored and communicates to the metascheduler component. The metascheduler component on receiving the notification information resumes its operation.

The Database Manager also maintains a global clock. Since the resources are distributed, there can be potential *out-of-synchronization* of the local clocks for the individual resources. To perform some calculations based on times, for e.g., when a client wants to calculate the total time elapsed in an executing end application by subtracting the current time and the start time of the application, a component may want to determine the current time. Since the current time cannot be gathered from local clocks, the component requests the global current time from the Database Manager and retrieves it.

Lastly, the Database Manager keeps track of the states of the metascheduler component activities. This information acts as database locks to prevent race conditions among the metascheduler components. For e.g., the Contract Negotiator may be making decisions regarding preempting an executing application. At that time, it is desirable that the Rescheduler does not make preempting decisions for the same executing applications. By retrieving the lock information from the database, the Rescheduler may find that the Contract Negotiator is making decisions for the end application and hence not make its own decisions.

3.2.2 Permission Service

Permission Service is a daemon that receives requests from the GrADS applications to grant them permission to proceed with the usage of the Grid system. The Application Manager, after getting resource information from NWS, passes the resource information, the problem requirements and the problem parameters to the Permission Service.

The Permission Service, based on the constraints of the problem , for e.g., the memory needed for the GrADS application, and the resource parameters of all the resources available in the GrADS system, checks if the resources are sufficient for the execution of the problem. If the resources are sufficient, the Permission Service grants permission by returning PERMISSION to the GrADS application. In this case, the GrADS application proceeds to the next phases in GrADS execution.

If the resources are not sufficient for the execution of the end application, the Permission Service retrieves a list of end applications that have started executing on the GrADS resources from the Database Manager. If no problems are executing on GrADS resources, then the Permission Service rejects permission to the GrADS application by returning NO_PERMISSSION_NO_RS. Thus the Permission Service ensures that applications whose memory requirements exceed the resource capacity of the resources do not execute on the resources since this will lead to frequent accesses to disks and eventual degradation of the entire system. The GrADS application immediately aborts, displaying information about lack of sufficient resources for execution of end application to the GrADS user.

If some GrADS problems are executing on the resources, the Permission Service tries to find a list of executing applications, *sub_list*, in the absence of which the resources will be sufficient enough to execute the current GrADS application for which permission decision is being made. The Permission Service sorts the list of executing applications based on the starting time of the applications. For each application in the list, the Permission Service retrieves the initial resource parameters used for deriving the application-level schedule for the GrADS application. For application i in the list, the Permission Service calculates the change in resource parameters caused by the execution of application i, $resource\_change_i$ by

$$resource\_change_i = resource\_parameters_{i+1} - resource\_parameters_i \tag{3.2}$$

where resource_parameters$_{i+1}$ is the set of resource parameters used for scheduling in GrADS application *i+1* and resource_parameters$_i$ is the set of resource parameters used for scheduling in GrADS application i. The Permission Service then determines the resource parameters for the present conditions assuming the absence of application

i, $resource\_absence_i$ by

$$resource\_absence_i = resource\_parameters\_current + resource\_change_i \tag{3.3}$$

where resource_parameters_current is the set of resource parameters passed to the Permission Service by the current GrADS application. The Permission Service then determines if resources with parameters indicated by $resource\_absence_i$ are sufficient for the execution of end application for the current GrADS application. If the resources are sufficient, then the end application will be able to execute in the absence of application i. If no executing application, whose absence will enable the execution of the current application, is found, the Permission Service sends NO_PERMISSION_NO_RS to the application.

For each executing application i in the *sub_list*, the Permission Service determines the remaining execution time of application i, $remaining\_exec_i$, and the predicted execution time of the current application in the absence of application i, *predicted_time* by contacting the respective application-level schedulers. The Permission Service then calculates, $ratio_i$ as

$$ratio_i = \frac{remaining\_exec_i}{predicted\_time} \tag{3.4}$$

$ratio_i$ indicates the relative execution times between the executing application i and the current application. More the ratio, greater the remaining execution time of application i and/or smaller the predicted cost for the execution of current application in the absence of application i.

The Permission Service checks if any of the executing applications in the *sub_list* has short remaining execution times. For the sake of simplicity, the time for short remaining time is fixed at 5 minutes. If such an executing application exists, the Permission Service waits for the application to complete by registering its interest for the notification of application completion in the Database Manager. The completion of such an application will free the resources for the execution of current application. When the application with the short remaining execution time completes, the Permission Service sends NO_PERMISSION_RS to the current GrADS application for which permission decision is being made. This prompts the current GrADS Application Manager to pass through Resource Selection phase and retrieve new resource information from NWS.

If no executing application in *sub_list* has short remaining execution time, the Permission Service chooses the application, *big_application*, from the *sub_list* that has the maximum of the ratios determined in equation 3.4 for all executing applications. The Permission Service determines if the *big_application* is preemptible. If the *big_application* is not preemptible, the Permission Service sends NO_PERMISSION_NO_RS to the GrADS application for which the permission decision is being made. If the *big_application* is preemptible, the Permission Service stops the application by sending STOP signal to the Runtime Support System (RSS) for the application and waits for the application to stop by waiting for a notification message from the Database Manager. After the *big_application* stops, the Permission Service sends NO_PERMISSION_RS to the

GrADS application for which the permission decision is being made. The GrADS application gets new resource information by passing through Resource Selection phase again, gets permission for the new set of resources from the Permission Service, determines a final schedule using Performance Modeler and executes the end application on the final set of resources. When the end application completes, the Permission Service sends a RESUME signal to the preempted application. This prompts the preempted application to start from Resource Selection phase again and continue its execution on possibly a new configuration of machines.

At the core of the metascheduling decisions made in the Permission Service and in the other metascheduling components is the ability to determine the approximate remaining execution time of an application. The Contract Monitor maintains the ratios between the actual and predicted costs of the end application at different points of the execution of the application. When a metascheduling component for e.g., the Permission Service, wants to determine the approximate remaining execution time for an application, it retrieves the location of the Contract Monitor corresponding to the GrADS application from the Database Manager. It contacts the Contract Monitor and requests for the ratios of the actual and predicted costs of the end application. It then calculates the remaining execution time for the application from the total percentage completion time of the application, predicted execution cost for the application and the average of the ratios between the actual and predicted costs for the end application. The details of the algorithm for calculating the remaining execution time for the application is given in Appendix A.2.

The Permission Service uses heuristics in its metascheduling decisions rather than any definite criteria. For.e.g., equations 3.2 and 3.3 assume a dedicated and quiet environment where no external non-GrADS applications execute on the resources between the submissions of applications i and $i+1$. This assumption may lead to over-estimation of *resource_change*, i.e. the change in resource characteristics caused by the execution of application i. In Grid environments, there can be change in resource characteristics caused by external loads. Even though this over-estimation is mitigated to a certain extent by the calculation of $ratio_i$ in equation 3.4, there may be situations when the Permission Service can cause unnecessary preemptions of executing applications. Also, the approximations in calculating the remaining execution times of the executing applications can cause further deficiencies in the metascheduling decisions of the Permission Service. By requiring that the applications make atleast 20% progress between preemptions, the probability of faulty preemptions of the executing applications due to the deficiencies in the Permission Service is greatly reduced.

In general, by requiring that the resources meet the criteria expressed by the constraints for GrADS applications, the Permission Service meets our metascheduling objective of maintaining a consistent system performance. Also, by pro-actively preempting large executing applications and accommodating small applications, the Permission Service increases the number of requests serviced by the system and maintain high throughput of the overall system.

The functions of the Permission Service are summarized by the pseudo code in Appendix A.3.

3.2.3 Contract Negotiator

The Contract Negotiator component of the metascheduler is a daemon that receives application level schedules from the GrADS applications. An application level schedule of an application is the final list of machines that the GrADS application obtains from the Performance Modeler through the employment of the application specific execution model. These are the list of machines on which the end application can potentially execute. The application passes the problem parameters, the application level schedule and predicted execution cost for the end application as a *contract* to the Contract Developer. The Contract Developer, instead of approving the contracts of the applications under all conditions, contacts the Contract Negotiator for obtaining approval of the application contract.

The Contract Negotiator is implemented as a threaded program containing atmost three threads of execution at any point in time. These are the main thread, input control thread and process thread. The Contract Negotiator maintains two global data structures, namely input queue and process queue. The implementation of the Contract Negotiator is illustrated by Figure 3.4.

The main thread initially spawns the input control thread and waits for entries in the input queue. The Contract Monitors of the GrADS applications contact the input control thread, passing to it the application-level schedule and the predicted performance

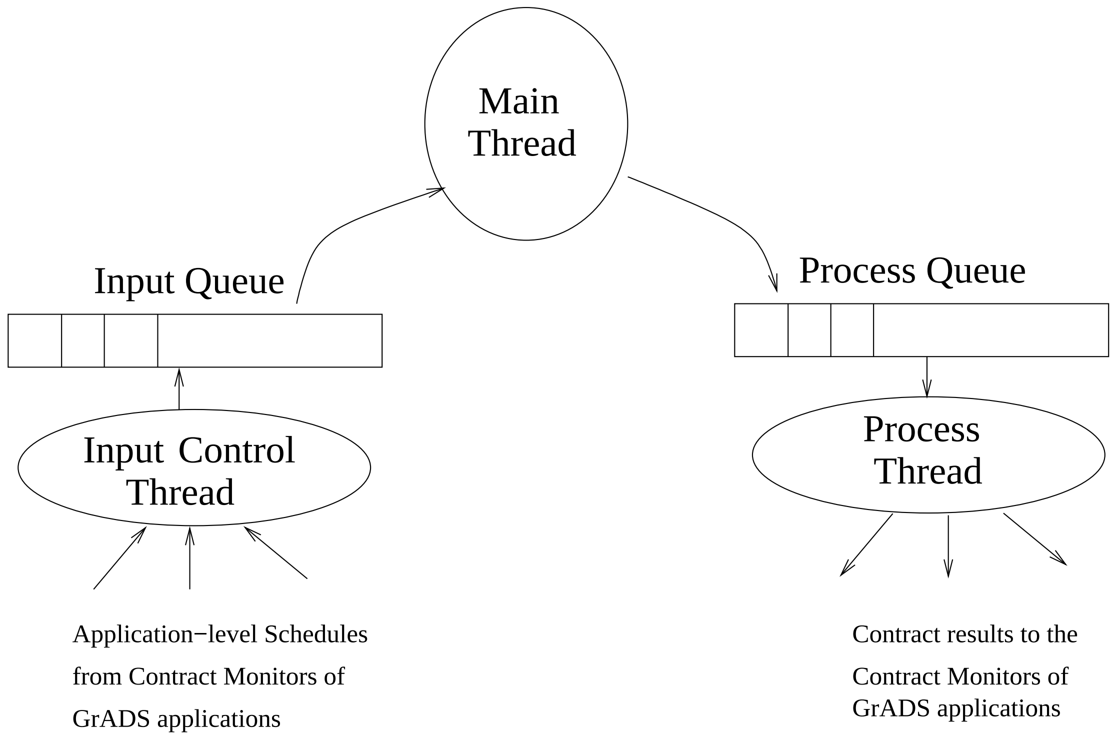

Figure 3.4: Implementation of Contract Negotiator

cost for the application. The input control thread stores these parameters in the input queue. The main thread, on noticing entries corresponding to GrADS applications in the input queue, moves an entry corresponding to a GrADS application to the process queue and spawns a process thread for processing the contract corresponding to the stored entry in the process queue.

The Contract Negotiator first tries to find GrADS applications for which the contracts have been approved and for which the application-level schedules consists of some machines in the application-level schedule of the current GrADS application for which contract is being processed. If such GrADS applications exist and if the end applications corresponding to the GrADS applications have not started executing on the resources, the Contract Negotiator waits for the end applications to start executing and then sends CONTRACT_NOT_OK as a contract result to the Contract Monitor of the current GrADS application. This prompts the current GrADS Application Manager to start from the Resource Selection phase again and evolve a new performance contract. Thus when competing applications are submitted to the GrADS system simultaneously and claims the same set of resources due to the lack of knowledge of the competition from other GrADS applications, the Contract Negotiator ensures an ordering of the applications whereby contract is approved for one GrADS application allowing the end application for the GrADS application to use the system resources and possibly modify the resource characteristics while contacts are rejected for other GrADS applications thereby making those applications to generate a new application-level schedule using the latest resource characteristics for scheduling decisions.

The Contract Negotiator then tries to retrieve a list of executing applications, *executing_list* executing on some of the machines that are in the application-level schedule of the current GrADS application. It then tries to determine a subset of *executing_list* of applications that started executing after the current GrADS application completed its Resource Selection phase. In this case, the current GrADS application generated its application-level schedule with outdated resource characteristics. Hence the Contract Negotiator sends CONTRACT_NOT_OK to the Contract Monitor of the current GrADS application, prompting the Application Manager to generate a new application-level schedule. As in the precious case, this behavior of the Contract Negotiator ensures that applications do not make conflicting claims on the same sets of resources.

The Contract Negotiator sorts the applications in *executing_list* in the order of their starting times to form a sorted list, *sorted_list*. For each GrADS application, i in the *sorted_list*, the Contract Negotiator retrieves the resource characteristics, $resource_i$ with which the application-level schedule for the application i was generated. The Contract Negotiator contacts the application-level scheduler of the current GrADS applications and retrieves the predicted execution cost, $predicted_i$ of the end application using $resource_i$ as the initial set of resources. Thus $predicted_i$ gives the approximate predicted execution cost of the end application corresponding to the current GrADS application if it were to execute in the absence of application i. The Contract Negotiator then calculates $ratio_i$ corresponding to GrADS application i. $ratio_i$ is the ratio between $predicted_{i+1}$ and $predicted_i$ corresponding to GrADS applications *i+1* and *i* respectively in the *sorted_list*. If $ratio_i$ is greater than 1.5, then the Contract Negotiator chooses executing application *i* as the application, *big_application*, whose absence will lead to significant reduction in the execution time of the end application corresponding to the current GrADS application for which contract decision is being made.

The Contract Negotiator calculates the remaining execution time, *remaining_exec_time*, of the executing application, *big_application*. The Contract Negotiator also calculates *impact_time* which is the increase in execution time that can be incurred by *big_application* if the end application corresponding to the current GrADS application is allowed to execute on the resources proposed by the application-level schedule for the application. The *impact_time* is calculated using the slowdown model by Figueira [54, 53, 52, 51]. In this model, the impact in execution time of a parallel application executing on a set of resources is modeled using the impact in execution time of sub processes of the parallel applications executing on an individual resource. Processes executing on a single resource are assumed to be scheduled in a round-robin fashion. Though the assumption used in the model is not valid for all environments, it has been proven to be valid for most of the environments.

The Contract Negotiator calculates average of the completion times of the executing application, *big_application*, and the end application corresponding to the current GrADS application for three different scenarios. t_1 is the average of the completion times when the *big_application* is preempted from the resources, the end application for the current GrADS application is executed, and the *big_application* is continued after the completion of the current application. t_2 is the average of the completion times when the Contract Negotiator waits for the *big_application* to complete and then schedules the current application. t_2 is the average of the completion times when the current application is executed in the presence of the *big_application*. These times are calculated as:

$$t_1 = \frac{(predicted\_absence) + (predicted\_absence + remaining\_exec\_time)}{2} \tag{3.5}$$

$$t_2 = \frac{(remaining\_exec\_time) + (remaining\_exec\_time + predicted\_absence)}{2} \tag{3.6}$$

$$t_3 = \frac{(predicted\_presence) + (remaining\_exec\_time + impact\_time)}{2} \tag{3.7}$$

where *predicted_absence* is the predicted execution cost of the current application if it were to execute in the absence of the *big_application* and *predicted_presence* is the predicted execution cost of the current application if it were to execute in the presence of the *big_application*.

If t_1 is less than 25% of the minimum of t_2 and t_3, the Contract Negotiator stops *big_application*, sends CONTRACT_NOT_OK to the current GrADS application prompting it to generate a new application-level schedule, approves the contract of the new schedule, waits for the approved application to complete its application and continues *big_application* on possibly a new set of resources. If t_2 is less than t_1 and t_3, the Contract Negotiator waits for the *big_application* to complete, sends CONTRACT_NOT_OK to the current GrADS application prompting it to generate a new application-level schedule and approves the contract of the new schedule. If t_3 is less than t_1 and t_2, then the Contract Negotiator can possibly approve the contract of the current application thereby allowing the application to execute on the resources indicated by the application-level schedule.

If during the previous operations, the Contract Negotiator has not sent CONTRACT_NOT_OK to the Contract Monitor for the current GrADS application, the Contract Negotiator approves the contract for the current GrADS application by sending CONTRACT_OK to the Contract Monitor. Before approving the contract, the Contract Negotiator tries to reduce the impact on already executing applications that can be caused by the execution of the end application. It uses the slowdown model by Figueira to calculate the maximum of the percentage increase in execution times of the executing applications, *max_percent* due to the addition of the end application for the current GrADS application. If *max_percent* is greater than 30%, the Contract Negotiator removes a resource from the application-level schedule for the current GrADS application and calculates *max_percent* again. It continues to remove resources from the application-level schedule for the current GrADS application until the percentage increase in predicted cost of the end application for the current GrADS application becomes greater than twice the *max_percent*. Finally, the Contract Negotiator approves the contract of the current GrADS application by sending CONTRACT_NOT_OK to the Contract Monitor.

Thus the Contract Negotiator acts as a major metascheduling component balancing the interests of the different applications. To summarize, its main functions are:

1. verifying if the GrADS applications generated application-level schedules with most recent resource characteristics,

2. preempting executing applications if the presence of those applications can severely degrade the performance of the end application for the current GrADS application for which contract decision is being made and

3. reducing the impact that can be caused by the execution of end application on executing applications.

These metascheduling decisions help to increase the overall throughput of the system and providing high performance to the individual applications.

The actions of the Contract Negotiator are summarized by the pseudo code in Appendix A.4.

3.2.4 Rescheduler

Rescheduler is a service that maintains load balance in the system and improves the performance of executing end applications. It performs migration of executing applications both when the executing applications do not meet the expected performance levels and when few system resources are freed due to the completion of certain GrADS applications. The executing applications can be migrated to a new set of resources and executed on possibly different number of resources. This ability to dynamically reconfigure the application in terms of the number of resources that the application is using makes the Rescheduler adopt flexible rescheduling mechanisms.

Though the Contract Negotiator balances the interests of different GrADS applications to ensure that the performance contracts for the end applications are met, the presence of external loads in the system in the form of non-GrADS applications can degrade the performance of the end applications. The Contract Monitor that monitors the actual performance of the end applications is initially set with tolerance limits for performance degradation of the end application it monitors. On noticing the drop in performance of the executing applications, the Contract Monitor compares the ratios between the actual and the predicted performance with the tolerance limits that were previously set. *Contract violations* occur when the ratios become greater than the tolerance limits. On noticing few contract violations, the Contract Monitor contacts the Rescheduler requesting for rescheduling the application. The Rescheduler evaluates the benefit of rescheduling the application and if it determines that potential performance benefits can be obtained for the application, migrates the application by sending STOP signal to the end application and storing RESUME flag in the Database Manager prompting the Application Manager to evolve a new schedule where the application can be continued.

The Rescheduler also proactively preempts executing applications to utilize free resources that were made available by the completion of few end applications. The

Rescheduler continuously queries the Database Manager for completed applications. If an application completes, the Rescheduler retrieves a list of executing applications from the Database Manager. For each application in the list, it determines the new application-level schedule and predicted remaining execution cost with the recent resource conditions by contacting the application-level scheduler. The rescheduler then determines the potential rescheduling gain that can be obtained by migrating the executing application. It chooses the executing application for which maximum rescheduling gain can be obtained and migrates the application if the rescheduling gain is greater than an acceptable rescheduling threshold. Thus by utilizing free resources in the system, the Rescheduler tries to maintain load balance of the system resources.

The working of the Rescheduler is summarized by the pseudo code in Appendix A.5.

The framework used for monitoring the executing applications, conditions for contacting the Rescheduler and the policies used in the Rescheduler for migrating the end applications make the Rescheduler a unique and robust metascheduling component. The next Section gives a detailed description of the Rescheduling framework comparing the migration decisions used in the Rescheduler with relevant work in the area of rescheduling.

3.3 Rescheduling Framework

Migration of executing applications onto different sets of resources is an interesting research area since it involves issues regarding techniques for application migration and also regarding scheduling decisions for migration. There have been many research efforts that built migration systems which migrate applications under different conditions including load changes on machines, availability of new machines, non-availability of existing machines due to reclaiming by owners, providing fault tolerance etc. At least three factors in the existing migrating systems make them less suitable in Grid systems especially when the goal is to improve the response times for individual applications - separate policies for suspension and migration of executing applications employed by these migration systems, the use of pre-defined conditions for suspension and migration and the lack of knowledge of the remaining execution time of the applications. The Rescheduling framework developed in this research implements a migration framework for performance oriented Grid systems that implements tightly coupled policies for both suspension and migration of executing applications. The suspension and migration policies take into account both the load changes on systems as well the remaining execution times of the applications thereby taking into account both system load and application characteristics. The main goal of the migration framework is to improve the response times for individual applications.

Computational Grids [58] involve large system dynamics that the ability to migrate executing applications onto different sets of resources assumes great importance. Specifically, the main motivations for migrating applications in Grid systems are to provide fault tolerance and to adapt to load changes on the systems. The main focus of the migration framework in this research is on migration of applications executing on the distributed and Grid systems when the loads on the system resources change.

4 issues have to be dealt to build efficient migration systems.

1. *When* - The scheduling and migrating systems have to define the conditions under which migration of executing applications will take place. These conditions can be few key strokes on the executing systems, sudden non-availability of the systems on which the applications are executing, availability of new sets of resources, load imbalance on the systems etc.

2. *Where* - After the decision to migrate, the scheduling system should determine the new sets of resources on which the applications will be migrated. These new sets of resources can be determined based on different sets of criteria.

3. *How* - Different migrating systems employ different methods for migrating applications for different kinds of applications. Some migrations can be simple context switches while some migrations can involve complex checkpointing mechanisms.

4. *Who* - The migration decisions and the migration process can be implemented by the system automatically or can be specified by the user.

There are at least two disadvantages in using the existing migration systems [78, 47, 75, 107, 121, 59, 30] for improving the response times of executing applications. Due to the separate policies employed by these migration systems for suspension of executing applications and migration of the applications to different systems, the applications can incur lengthy waiting times between when they are suspended and when they are restarted on new systems. Secondly, due to the use of pre-defined conditions for suspension and migration and due to the lack of knowledge of the remaining execution time of the applications, the applications can be suspended and migrated even when they are about to finish execution in a short period of time. This is certainly less desirable in performance oriented Grid systems where the large load dynamics will to lead to frequent satisfaction of the pre-defined conditions and hence will lead to frequent invocation of suspension and migration decisions.

The Rescheduler implements a migration framework that defines and implements scheduling policies for migrating applications executing on distributed and Grid systems in response to system load changes. In the framework, the migration of applications depends on

1. the amount of increase or decrease in loads on the resources,
2. the time of the application execution when load is introduced into the system,
3. the performance benefits that can be obtained for the application due to migration.

Thus the migrating framework takes into account both the load and application characteristics. The policies are implemented in such a way that the executing applications are suspended and migrated only when better systems are found for application execution thereby invoking the migration decisions as infrequently as possible. In the following subsections, the related work in the field of migration is described and the migration architecture is explained in detail.

3.3.1 Related Work in the Field of Migration of Applications

Different systems have been implemented to migrate executing applications onto different sets of resources. These systems migrate applications either to efficiently use under-utilized resources [78, 95, 36, 35, 120, 107, 46], to provide fault resilience [16] or to reduce the obtrusiveness to workstation owner [16, 75].

The work by Mirchandaney et. al. [78] deals with migration of executing applications to efficiently use under-utilized resources. The Dome system [16] performs data redistribution for load balancing and migrates executing applications to provide fault resilience. Khaled Al-Saqabi et. al [95] discusses migration of applications in the context of gang scheduling. MPVM/MIST [36], [35] projects and the work by Zhang et. al. [120] have built migration systems that uses the concept of gang scheduling to utilize system resources. MIST also deals with migration under increasing loads but the scheduling policy has not been defined clearly. The HMF system [30] uses a graph model to define migration policies. The efficiency of this model in Grid systems is still to be proven.

The particular projects that are closely related to our work are Dynamite [107], MARS [59], LSF [121] and Condor [75]. The Dynamite system [107] based on Dynamic PVM [46] migrates applications when the loads of certain machines gets under-utilized or over-utilized as defined by application-specified thresholds. Although this method takes into account application-specific characteristics it does not necessarily evaluate the remaining execution time of the application and the resulting performance benefits due to migration. MARS [59] migrates applications taking into account both the system loads and application characteristics. But the migration decisions are made only at different phases of the applications unlike our migration framework where the applications are continuously monitored and migration decisions are made whenever the applications are not making sufficient progress.

In LSF [121], jobs can be submitted to queues which have pre-defined migration thresholds. A job can be suspended when the load of the resource increases beyond a particular limit. When the time since the suspension becomes higher than the migration threshold for the queue, the job is migrated and submitted to a new queue. Thus LSF suspends jobs to maintain the load level of the resources while our migration framework suspends jobs only when it is able to find better resources where the jobs can be migrated. By adopting a strict approach to suspending jobs based on pre-defined system limits, LSF gives less priority to the stage of the application execution whereas our migration framework suspends an application only when the application has large enough remaining execution time so that performance benefits can be obtained due to migration. And lastly, due to the separation of the suspension and migration decisions, a suspended application in LSF can wait for a long time before it restarts executing on a suitable resource. In our migration framework, a suspended application is immediately restarted due to the tight coupling of suspension and migration decisions.

Of the Grid computing systems, only Condor [75] seems to migrate applications under workload changes. Condor provides powerful and flexible ClassAd mechanism by means of which the administrator of resources can define policies for allowing jobs to execute on the resources, suspending the jobs and vacating the jobs from the resources. The fundamental philosophy of Condor is to increase the throughput of long running jobs and also respect the ownership of the resource administrators. The main goal of our migration framework is to increase the response times of individual applications. Similar to LSF, Condor also separates the suspension and migration decisions and hence has the same problems mentioned for LSF in taking into into account the performance benefits of migrating the applications. Unlike our metascheduler framework, the Condor system does not possess the knowledge about the remaining execution time of the applications. Thus suspension and migrating decisions can be invoked frequently in Condor based on system load changes. This may be less desirable in Grid systems where system load dynamics are fairly high.

3.3.2 The Migration Framework

The ability to migrate applications in the GrADS system is implemented by adding the metascheduling component called *Rescheduler* to the GrADS architecture. The migrating numerical application, *migrator*, the *contract monitor* that monitors the application's progress and the *rescheduler* that decides when to migrate, together form the core of the migrating framework. The interactions between the different components involved in the migration framework is illustrated in Figure 3.5. These components are described in detail in the following subsections.

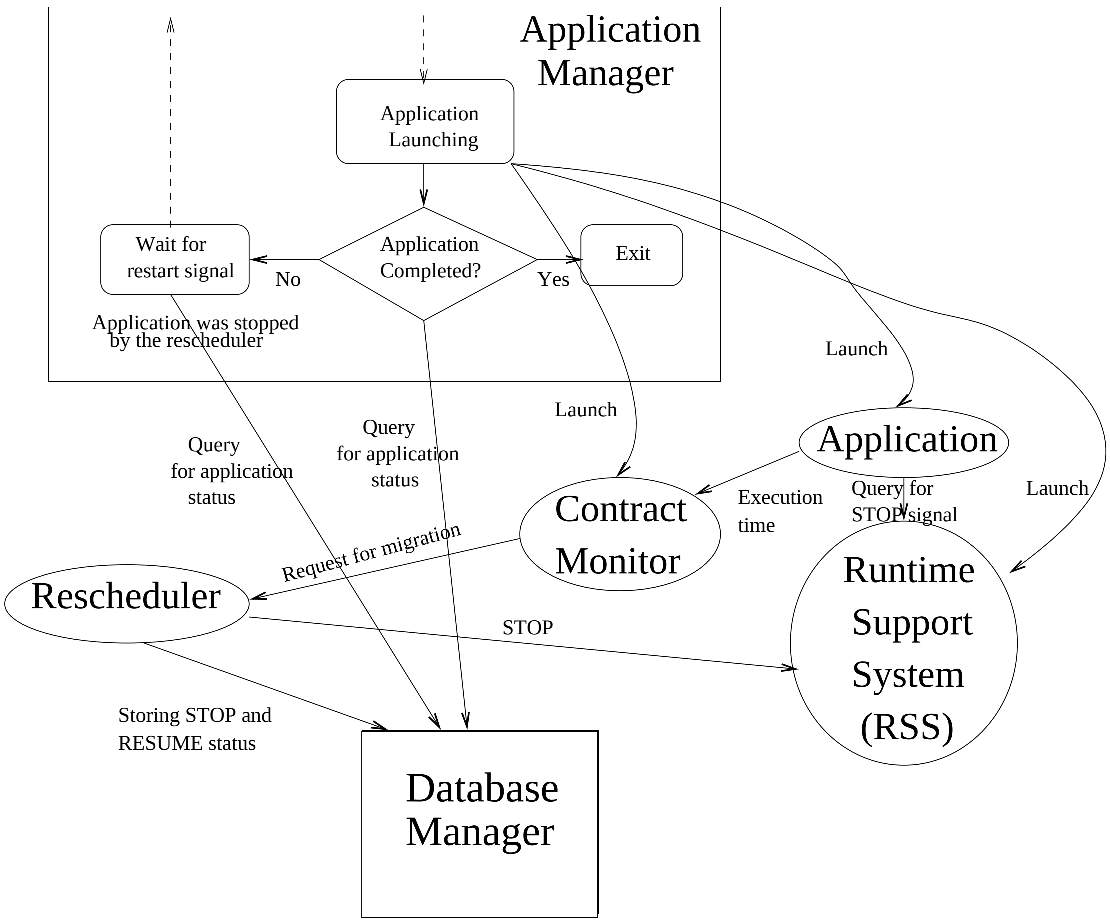

Figure 3.5: Interactions in Migration Framework

The Migrator

We have implemented a user-level checkpointing library called SRS (**S**top **R**estart **S**oftware). The application by making calls to SRS gets the ability to checkpoint data, to be stopped at a particular point in execution, to be restarted later on a different configuration of processors and to be continued from the previous point of execution. The SRS library is implemented on top of MPI and hence can be used only with MPI based parallel programs. Since checkpointing in SRS is implemented at the application layer and not at the MPI layer, migration is achieved by clean exit of the entire application and restarting the application over a new configuration of machines. Due to the clean exit of the application during migration, no interaction with the resource allocation manager is necessary during rescheduling. The application interfaces for SRS look similar to CUMULVS [63], but unlike CUMULVS, SRS does not require a PVM virtual machine to be setup on the hosts. Also, SRS allows reconfiguration of applications between migrations.

The SRS library consists of 6 main functions:

1. SRS_Init()
2. SRS_Finish()
3. SRS_Restart_Value(),
4. SRS_Check_Stop()
5. SRS_Register()

6. SRS_Read().

The user calls SRS_Init() and SRS_Finish() in his application after MPI_Init() and before MPI_Finalize() respectively. Since SRS is a user-level checkpointing library, the application may contain conditional statements to execute certain parts of the application in the start mode and certain other parts in the restart mode. In order to know if the application is executed in the start or restart mode, the user calls SRS_Restart_Value() that returns 0 and 1 on start and restart modes respectively. The user also calls SRS_Check_Stop() at different phases of the application to check if an external component wants the application to be stopped. If the SRS_Check_Stop() returns 1, then the application has received a stop signal from an external component and hence can perform application-specific stop actions.

SRS library uses Internet Backplane Protocol(IBP)[82] for storage of the checkpoint data. IBP depots are started on all the machines of the GrADS testbed. The user calls SRS_Register() in his application to register the variables that will be checkpointed by the SRS library. When an external component stops the application, the SRS library checkpoints only those variables that were registered through SRS_Register(). The user reads in the checkpointed data in the restart mode using SRS_Read(). The user, through SRS_Read(), also specifies the previous and current data distributions. By knowing the number of processors and the data distributions used in the previous and current execution of the application , the SRS library automatically performs the appropriate data redistribution. Thus, for example, the user can start his application on 4 processors with block distribution of data, stop the application and restart it on 8 processors with block-cyclic distribution. The details of the SRS API for accomplishing the automatic redistribution of data is explained in Chapter 5.

An external component(e.g., the rescheduler) wanting to stop an executing application interacts with a daemon called Runtime Support System (RSS). RSS exists for the entire duration of the application and spans across multiple migrations of the application. Before the actual parallel application is started, the RSS is launched by the Application Launcher on the machine where the user invokes the GrADS Application Manager. The actual application through the SRS library knows the location of the RSS from the Database Manager and interacts with RSS to perform some initialization, to check if the application needs to be stopped during SRS_Check_Stop(), to store pointers to the checkpointed data, to retrieve pointers to the checkpointed data and to store the present processor configuration and data distribution used by the application.

The SRS library is explained in detail in Chapter 5.

Contract Monitor

As mentioned in the previous sections, Contract Monitor is a component that uses the Autopilot infrastructure to monitor the progress of the applications in GrADS. Autopilot [93] is a real-time adaptive control infrastructure built by the Pablo group at University of Illinois, Urbana-Champaign. An autopilot manager is started before the launch of the numerical application. The numerical application is instrumented with calls to register to autopilot. The Contract Monitor retrieves the registration information of the application through the autopilot. The numerical applications are also instrumented with calls at different points of the program to send the times taken for the different phases of the execution to the Contract Monitor. The Contract Monitor compares the actual execution times with the predicted execution times and calculates the ratio between them. The tolerance limits of the ratio are specified as inputs to the Contract Monitor.

When a given ratio is greater than the upper tolerance limit, the Contract Monitor calculates the average of the computed ratios. If the average is greater than the upper tolerance limit, it contacts the rescheduler, requesting for migrating the application. The average of the ratios is used by the Contract Monitor to contact the rescheduler due to the following reasons:

1. A competing application of short duration on one of the machines may have increased the load on the machine and hence the loss in performance of the application. Contacting the rescheduler for migration on noticing few losses in performance will result in unnecessary migration in this case since the competing application will end soon and the application's performance will be back to normal.

2. The average of the ratios also captures the history of the behavior of the machines on which the application is running. If the application's performance on most of the iterations has been satisfactory, then few losses of performance may be due to sparse occurrences of load changes on the machines.

3. The average of the ratios also takes into account the percentage completed time of application's execution.

4. Contacting the rescheduler for migration only when the average of ratios is greater than the upper tolerance limit significantly reduces the overhead of migrating decisions.

If the rescheduler refuses to migrate the application, the Contract Monitor adjusts its tolerance limits to new values. Similarly when a given ratio is less than the lower tolerance limit, the Contract Monitor calculates the average of the ratios and adjusts the tolerance limits if the average is less than the lower tolerance limit. The dynamic adjusting of tolerance limits serves three purposes:

1. It reduces the overhead involved in Contract Monitor when the ratios between actual and predicted times are not the original expected ratios.

2. It reduces the amount of communication between the Contract Monitor and the rescheduler.

3. It hides the deficiencies in the application-specific execution time model.

Rescheduler

Rescheduler is the metascheduling component that evaluates the performance benefits that can be obtained due to the migration of an application and initiates the migration of the application. The rescheduler is a daemon that operates in two modes: *migration* *on request* and *opportunistic migration*. When the Contract Monitor detects intolerable performance loss for an application, it contacts the rescheduler requesting it to migrate the application. This is called migration on request. In other cases when no Contract Monitor has contacted the rescheduler for migration, the rescheduler periodically queries the Database Manager for recently completed applications. If a GrADS application was recently completed, the rescheduler determines if performance benefits can be obtained for an executing application by migrating it to use the resources that were freed by the completed application. This is called opportunistic rescheduling.

In both cases, the rescheduler first contacts the Network Weather Service (NWS) to get the updated information for the machines in the Grid. It then contacts the application-specific Performance Modeler to evolve a new schedule for the application. Based on the total percentage completion time for the application and the total predicted execution time for the application with the new schedule, the rescheduler calculates the remaining execution time, *ret_new*, of the application if it were to execute on the machines in the new schedule. The rescheduler also calculates *ret_current*, the remaining execution time of the numerical application if it were to to continue executing on the original set of machines. The rescheduler then calculates the rescheduling gain as

$$rescheduling\_gain = \frac{(ret\_current - (ret\_new + rescheduling\_cost))}{ret\_current}$$

The rescheduling_cost is the cost of rescheduling and includes cost for redistribution of data and other fixed overhead. If the application uses conventional data distributions like block-cyclic data distribution, the Rescheduler by interaction with the Runtime Support System (RSS) retrieves the parameters used for the data distribution and uses these parameters to determine the data mapping for the new schedule. By the knowledge of data distributions in the old and new schedules and the network information between the resources in the old and new schedule, the Rescheduler calculates the time for data redistribution from the old to the new schedule.

In cases when the end application uses its own data distribution strategies, the Rescheduler uses 900 seconds for the rescheduling cost. This time is the worst case time in seconds needed to reschedule the application. The various times involved in rescheduling is given in Table 3.1. The times shown in Table 3.1 were obtained by conducting a number of experiments with ScaLAPACK QR factorization problems of different problem sizes and obtaining the maximum times for each phases of rescheduling. Thus the rescheduling strategy adopts pessimistic approach for rescheduling where migration of applications will be avoided in certain cases where migration can yield performance benefits.

If the rescheduling gain is greater than 30%, the rescheduler sends STOP signal to the application, and stores the stop status in the Database Manager. The Application Manager then waits for the RESUME signal. The Rescheduler stores the RESUME value in the Database Manager thus prompting the Application Manager to evolve a new schedule and restart the application on the new schedule. If the rescheduling gain is less than 30% and if the rescheduler is operating in the *migration on request* mode, the

Table 3.1: Times for rescheduling phases for ScaLAPACK QR application

Rescheduling Phase	*Time (seconds)*
Writing checkpoints	40
Waiting for NWS to update resource information	90
Time for application manager to get new resource information from NWS	120
Evolving new application-level schedule	80
Other grid overhead	10
Starting application	60
Reading checkpoints and Data redistribution	500
Total	900

rescheduler contacts the Contract Monitor prompting the Contract Monitor to adjust its tolerance limits.

The rescheduling threshold [114] which the performance gain due to rescheduling must cross for rescheduling to yield significant performance benefits depends on the load dynamics of the system resources, the accuracy of the measurements of resource information and may also depend on the particular application for which rescheduling is made. Since the measurements made by NWS are fairly accurate, the rescheduling threshold for our experiments depended only on the load dynamics of the system resources. By means of trail-and-error experiments we determined the rescheduling threshold for our testbed to be 30%. Rescheduling decisions made below this threshold may not yield performance benefits in all cases.

Chapter 4

SRS Checkpointing System

This chapter includes lightly revised version of a paper submitted to a journal.

Sathish S. Vadhiyar and Jack J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. Submitted to Parallel Processing Letters, 2003.

I was the primary contributor of the paper and was involved in the design and implementation of the frameworks and verification by experiments. This chapter revises the paper by providing a more detailed description of the framework developed and also adds a section to the paper describing the relevance of the effort to the overall research.

The metascheduling framework described in the previous chapter assumes the existence of parallel applications that can be stopped and continued on a different set of processors. Due to the existence of multiple applications and the high failure rate of the resources in the Grid framework, the parallel application that was stopped may not be able to continue on the same number of processors. Hence it is necessary that the parallel applications have the ability to be stopped and continued on a possibly different number of processors. Such reconfigurable parallel applications are called **malleable** applications.

The ability to produce malleable parallel applications that can be stopped and reconfigured during the execution can offer attractive benefits for both the system and the applications. The reconfiguration can be in terms of varying the parallelism for the applications, changing the data distributions during the executions or dynamically changing the software components involved in the application execution. In distributed and Grid computing systems, migration and reconfiguration of such malleable applications across distributed heterogeneous sites which do not share common file systems provides flexibility for scheduling and resource management in such distributed environments. For e.g., the Rescheduler in the metascheduling framework can *shrink* an executing parallel application to run on fewer machines, if some of the machines on which the application was executing become heavily loaded. The present reconfiguration systems do not support migration of parallel applications to distributed locations. In this chapter, we discuss a framework for developing malleable and migratable MPI message-passing parallel applications for distributed systems. The framework includes a user-level checkpointing library called SRS and a runtime support system that manages the checkpointed data for distribution to distributed locations.

4.1 Motivation

Distributed systems and computational Grids [58] involve large system dynamics that it is highly desirable to reconfigure executing applications in response to the change in environments. Specifically, reconfiguration of executing applications is useful in the following cases:

1. **Application migration**

   The machines in a cluster on which the application is currently executing may become unavailable after a period of time. After knowing this information from the system administrators, the user may determine that the application will not be able to complete within the period of time. Hence he may want to stop the application and move the application to another cluster and continue the application from the point where the application was stopped.

   Application migration is also useful for resource management systems like Condor [75] where an application has to be migrated when the workstation owner returns to using the machine. Also, parallel applications execute on large number of shared systems in distributed environments. The performance of the applications will be degraded if there is increase in external load on the resources caused by other applications. In this situation, the scheduling system may want to migrate the executing application to a different site to avoid the impact in performance of the application caused by the heavy loads on the machines.

2. **Trial-and-Error experiments**

   In many cases, it is difficult for users of parallel applications to determine the amount of parallelism to be used for their applications. The users may want to determine the amount of parallelism by means of trial-and-error experiments. Hence he can start the application on initial set of processors, determine that his application is not running at sufficient speed, stop the application, restart and continue it with more number of processors, stop the application again and so on.

Like with the number of processors, the user of parallel programs is at a loss regarding the type of data distribution he has to use for the data in his program. The user can use a initial data distribution, e.g., block data distribution, and execute his application. If the performance of his application is not satisfactory, he can stop his application, compile his application with a new data distribution, e.g., block cyclic, restart the application and continue from the point when it was stopped, but this time with the block-cyclic data distribution, note the performance change, stop his application again and so on.

3. **Reducing the processor set**

   The user may want to reduce the number of processors he is using for the application either to increase the performance of the application or due to non-availability of some resources.

4. **Fault tolerance**

   Due to the large number of machines involved in the distributed computing systems, the mean single processor failure rate and hence the failure rate of the set of machines where parallel applications are executing are fairly high [24]. Hence, for long running applications involving large number of machines, the probability of successful completion of the applications is low. In this case, a mechanism in the application for withstanding the failures is needed.

In the above situations, it will be helpful for the users or the scheduling system to stop the executing parallel application and continue it possibly with a new configuration in terms of the number of processors used for the execution. In cases of the failure of the application due to non-deterministic events, restarting the application on a possibly new configuration also provides a way of fault tolerance. Reconfigurable or malleable and migratable application provide added functionality and flexibility to the scheduling and resource management systems for distributed computing.

In order to achieve starting and stopping of the parallel applications, the state of the applications have to be checkpointed. Elonazhy [49] and Plank [83] have surveyed several checkpointing strategies for sequential and parallel applications. Checkpointing systems for sequential [84, 104] and parallel applications [46, 35, 16, 99, 63] have been built. Checkpointing systems are of different types depending on the transparency to the user and the portability of the checkpoints. Transparent and semi-transparent checkpointing systems [84, 39, 99] hide the details of checkpointing and restoration of saved states from the users, but are not portable. Non-transparent checkpointing systems [70, 64, 79, 63] involves the users to make some modifications to their programs but are highly portable across systems. Checkpointing can also be implemented at the kernel level or user-level.

In this research, a checkpointing infrastructure was developed that helps in the development and execution of malleable and migratable parallel applications for distributed systems. The infrastructure consists of a user-level semi-transparent checkpointing library called SRS (**S**top **R**estart **S**oftware) and a Runtime Support System (RSS). Our SRS library is semi-transparent because the user of the parallel applications has to insert calls in his program to specify the data for checkpointing and to restore the application state in the event of a restart. But the actual storing of checkpoints and the redistribution of data in the event of a reconfiguration are handled internally by the library. Also, SRS library provides for modifying the data distribution from one application run to another. Here a single application run refers to the period in the application from when the application began or when it was continued to when the application was stopped or terminated execution. Any native MPI versions can be used with SRS library. Though there are few checkpointing systems that allow changing the parallelism of the parallel applications [64, 79], our system is unique in that it allows for the applications to be migrated to distributed locations with different file systems without requiring the users to manually migrate the checkpoint data to distributed locations. This is achieved by the use of a distributed storage infrastructure called IBP [82] that allows the applications to remotely access checkpoint data. Our checkpointing infrastructure provides both proactive preemption and restarts of the applications and tolerance in the event of failures.

The contributions of our checkpointing infrastructure are:

1. providing an easy-to-use checkpointing library that allows reconfiguration of parallel applications.
2. allowing checkpoint data to be ported across heterogeneous machines and
3. providing migration of the application across locations that do not share common file systems without requiring the user to migrate data.

4.2 Related Work

Checkpointing parallel applications have been widely studied in [49, 83, 73] and checkpointing systems for parallel applications have been developed [39, 35, 94, 118, 86, 46, 63, 99, 15, 70, 63, 16, 69, 64, 79]. Some of the systems were developed for homogeneous systems [39, 36, 94, 99] while some checkpointing systems allows applications to be checkpointed and restarted on heterogeneous systems [46, 63, 15, 16, 21, 70, 64, 79]. Calypso [21] and Plinda [70] require application writers to write their programs in terms of special constructs and cannot be used with third-party software. Systems including Dynamic PVM [46] and CUMULVS [63] use PVM [23, 61, 62, 11] mechanisms for fault detection and process spawning and can only be used with PVM environments. Cocheck [99] and Starfish [15] provide fault tolerance with their own MPI implementations and hence are not suitable for distributed computing and Grid systems where the more secure MPICH-G [56] is used. CUMULVS [63], Dome [16, 24], the work by Hofmeister [69] and Deconick [40, 41, 29], DRMS [79] and DyRecT [14, 64] are closely related to our research in terms of the checkpointing API, the migrating infrastructure and reconfiguration capabilities.

The CUMULVS [63] API is very similar to our API in that it requires the application writers to specify the data distributions of the data used in the applications and it provides support for some of the commonly used data distributions like block, cyclic etc. CUMULVS also supports stopping and restarting of applications. But the applications can be stopped and continued only on the same number of processors. Though CUMULVS supports MPI applications, it uses PVM as the base infrastructure and hence poses the restriction of executing applications on PVM.

Dome [16, 24] supports reconfiguration of executing application in terms of changing the parallelism for the application. But the data that can be redistributed for reconfiguration have to be declared as Dome objects. Hence it is difficult to use Dome with third-party software like ScaLAPACK where native data is used for computations. Also Dome uses PVM as the underlying architecture and cannot be used for message passing applications.

The work by Hofmeister [69] supports reconfiguration in terms of dynamically replacing a software module in the application, moving a module to a different processor and adding or removing a module to and from the applications. But the package by Hofmeister only works on homogeneous systems. The work by Deconinck [40, 41, 29] is similar to SRS in terms of the checkpointing API and the checkpointing infrastructure. Their checkpoint control layer is similar to our RSS in terms of managing the distributed data and the protocols for communication between the applications and the checkpoint control layer is similar to ours. By using architecture-independent checkpoints, the checkpoints used in their work are heterogeneous and portable. But the work by Deconick does not support reconfiguration of application in terms of varying the parallelism of the applications.

The DyRecT [14, 64] framework for reconfiguration allows dynamic reconfiguration of applications in terms of varying the parallelism by adding or removing the processors during the execution of parallel application. The user-level checkpointing library in DyRecT also supports the specification of data distribution. The checkpoints are system-independent and MPI applications can use the checkpointing library for dynamic reconfiguration across heterogeneous systems. But DyRecT uses LAM MPI [6] for implementing the checkpointing infrastructure to use the dynamic process spawning and fault detection mechanisms provided by LAM. Hence DyRecT is mainly suitable for workstation clusters and not distributed and Grid systems where the more secure MPICH-G is used [56]. Also, DyRecT requires the machines to share a common file system and hence applications cannot be migrated and reconfigured to distributed locations that do not share common file systems.

The DRMS [79] checkpointing infrastructure uses DRMS programming model to support checkpointing and restarting parallel applications on different number of processors. It uses powerful checkpointing mechanisms for storing and retrieving checkpoint data to and from permanent storage. It is the closest related work to SRS in that it supports a flexible checkpointing API for reconfiguring MPI message passing applications implemented on any MPI implementations to be reconfigured on heterogeneous systems. But DRMS also does not support migrating and restarting applications on environments that do not share common file systems with the environments where the applications initially executed.

A more recent work by Kalé et. al [71] achieves reconfiguration of MPI-based message passing programs. But reconfiguration is achieved by using a MPI implementation called AMPI [28] that is less suitable to Grid systems than MPICH-G.

4.3 SRS Checkpointing Library

SRS (**S**top **R**estart **S**oftware) is a user-level checkpointing library that helps to make iterative parallel MPI message passing applications reconfigurable. Iterative parallel applications cover a broad range of important applications including linear solvers, heat-wave equation solvers, partial differential equation (PDE) applications etc. The SRS library has been implemented in both C and Fortran and hence SRS functions can be called from both C and Fortran MPI programs. The SRS library consists of 6 main functions:

1. SRS_Init,
2. SRS_Restart_Value,
3. SRS_Read,
4. SRS_Register,

5. SRS_Check_Stop and

6. SRS_Finish.

The user calls SRS_Init after calling MPI_Init. SRS_Init is a collective operation and initializes the various data structures used internally by the library. SRS_Init also reads various parameters from a user-supplied configuration file called *srs.config*. These parameters include the location of the Runtime Support System (RSS), a flag indicating if the application needs periodic checkpointing and the location of the Database Manager. SRS_Init, after reading these parameters, contacts the RSS and sends the current number of processes that the application is using. It also receives the previous configuration of the application from the RSS if the application has been restarted from a previous checkpoint. SRS_Init, then contacts the Database Manager registering the status of the end application as STARTED.

In order to stop and continue an executing application, apart from checkpointing the data used by the application, the execution context of the application also needs to be stored. For e.g., when the application is initially started on the system, various data needs to be initialized, whereas when the application is restarted and continued, data needs to be read from a checkpoint and the initialization phase can be skipped. Most checkpointing systems [84] restore execution context by storing and retrieving execution stack. This solution compromises on the portability of the checkpointing system. Since the main goal of the SRS library is to provide heterogeneous support, the task of restoring the execution context is implemented by the user by calling SRS_Restart_Value.

SRS_Restart_Value returns 0 if the application is starting its execution and 1 if the application is continued from its previous checkpoint. By using these values returned by SRS_Restart_Value, the user can implement conditional statements in his application to execute certain parts of the code when the application begins its execution and certain other parts of the code when the application is continued from its previous checkpoint.

SRS library uses Internet Backplane Protocol(IBP)[82] for storage of the checkpoint data. IBP depots are started on all the machines the user wants to use for the execution of his application. SRS_Register is used to mark the data that will be checkpointed by the SRS library during periodic checkpointing or when SRS_Check_Stop is called. Only the data that are passed in the SRS_Register call are checkpointed. The user specifies the parameters of the data including the size, data type and data distribution when calling SRS_Register. The data distributions supported by the SRS library include common data distributions like block, cyclic and block-cyclic distributions. For checkpointing data local to a process of the application or for data without distribution, a distribution value of 0 can be specified. SRS_Register stores the various parameters of the data in a local data structure. SRS_Register does not perform actual checkpointing of the data.

SRS_Read is the main function that achieves reconfiguration of the application. When the application is stopped and continued, the checkpointed data can be retrieved by invoking SRS_Read. The user specifies the name of the checkpointed data, the memory into which the checkpointed data is read and the new data distribution when calling SRS_Read. The data distribution specified can be conventional distributions or

0 for no distribution or SAME if the same data has to be propagated over all processes. The value SAME is useful for retrieving iterator values when all the processes need to start execution from the same iteration. The SRS_Read contacts the RSS and retrieves the previous data distribution and the location of the actual data. If no distribution is specified for SRS_Read, each process retrieves the entire portion of the data from the corresponding IBP depot used in the previous execution. If SAME is used for the data distribution, the first process reads the data from the IBP depot corresponding to the first process in the previous execution and broadcasts the data to the other processes. If data distribution is specified in SRS_Read, SRS_Read determines the data maps for the old and new distributions of the data corresponding to the previous and the current distributions. Based on the information contained in the data maps, each process retrieves its portion of data from the IBP depots containing the data portions. Thus reconfiguration of the application is achieved by using different level of parallelism for the current execution and specifying a data distribution in SRS_Read that may be different from the distribution used in the previous execution.

SRS_Check_Stop is a collective operation and called at various phases of the program to check if the application has to be stopped. If SRS_Check_Stop returns 1, then an external component has requested for the application to stop, and the application can execute application-specific code to stop the executing application. SRS_Check_Stop contacts the RSS to retrieve a value that specifies if the application has to be stopped. If an external component has requested for the application to be stopped, SRS_Check_Stop stores the various data distributions and the actual data registered by SRS_Register to the IBP [82] depots. Each process of the parallel application stores its piece of data to the local IBP depot. By storing only the data specified by SRS_Register and requiring each process of the parallel application to store the data to the IBP depot on the corresponding machine, the overhead incurred for checkpointing is significantly low. SRS_Check_Stop sends the pointers for the checkpointed data to RSS and deletes all the local data structures maintained by the library. SRS_Check_Stop also contacts the Database Manager specified in the srs.config file and stores the application status as STOPPED.

SRS_Finish is called collectively by all the processes of the parallel application before MPI_Finish in the application. SRS_Finish deletes all the local data structures maintained by the library and contacts the RSS requesting the RSS to terminate execution. SRS_Finish also contacts the Database Manager specified in the srs.config file storing the status of the application as DONE.

Fundamental to the reconfiguration capability provided by the SRS framework is the representation of data distributions by internal data structures called *data maps*. The data distributions stored in the IBP depots are in the form of the data maps. SRS_Read performs data redistribution by generating data maps for the old and the new distributions for the data. A data map contains the sizes and the locations of the different blocks of data. Figure 4.1 illustrates a sample data map. In the figure, the first data block consisting of 1000 units of data reside in processor 0, the second block

offset	size	processor
0	1000	0
1000	500	1
1500	800	2
2300	1000	3
3300	700	4

Figure 4.1: A Data Map representing a data distribution of a data of size 4000 units

of data consisting of 500 units of data reside in processor 1 etc. Although the data map is restricted in that it will not be able to express complex data distributions, it is useful for expressing most of the common data distributions like block, block-cyclic, circular etc.

Apart from the 6 main functions, SRS also provides 3 auxiliary functions: SRS_StoreMap, SRS_DistributeFunc_Create and SRS_DistributeMap_Create.

SRS_StoreMap is a collective operation to store the data maps of the various data to the IBP depots. The first processor gets the data distributions of all the data that were specified in all the SRS_Registers from all processors. For data with data distributions, it then calls the appropriate data distribution functions to generate data maps. The distribution functions also return an encoding of the input information used by the functions. For e.g., for a block-cyclic data distribution, the encoding is the block size of the data. The first process then stores the data maps to the IBP depots and sends pointers to the location of the IBP depots to the RSS. It also sends the encodings used in the distribution functions, the type and size of the data, the process number of the processor holding the local data without distributions and other relevant information to the RSS. The user can call SRS_StoreMap after all the SRS_Register calls in his code to store the data maps. This is done so that an external component like the Rescheduler can retrieve the data maps and other information stored by SRS_StoreMap from the RSS to make rescheduling decisions. SRS_Check_Stop also calls SRS_StoreMap if the user has not explicitly called SRS_StoreMap.

SRS_DistributeFunc_Create and SRS_DistributeMap_Create allow the user to specify his own data distributions instead of using the data distributions provided by the SRS library. The user can create his own distribution function that returns a data map and register his function to the SRS system using SRS_DistributeFunc_Create. The handle returned by SRS_DistributeFunc_Create can then be passed to the SRS_Register and SRS_Read calls. The user can also explicitly construct the data map structure and register the data map to the SRS library using SRS_DistributeMap_Create.
SRS_DistributeMap_Create also returns a handle that can be used in SRS_Register and SRS_Read calls.

Figure 4.2 shows a simple MPI based parallel program. The global data indicated by *global_A* is initialized in the first process and distributed across all the processes in a block distribution. The program then enters a loop where each element of the global data is incremented by a value of 10 by the process holding the element.

```
int main(int argc, char** argv){
int *global_A, int* local_A;
int global_size, local_size;
MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  local_size = global_size/size;

  if(rank == 0){

    for(i=0; i<global_size; i++){
      global_A[i] = i;
    }
  }

  MPI_Scatter (global_A, local_size, MPI_INT, local_A, local_size,
              MPI_INT, 0, comm );

  for(i=0; i<global_size; i++){

    proc_number = i/local_size;
    local_index = i\%local_size;

    if(rank == proc_number){
      local_A[local_index] += 10;
    }
  }

  MPI_Finalize();

  exit(0);

}
```

Figure 4.2: Original code

Figure 4.3 shows the same code instrumented with calls to the SRS library. The application shown in Figure 4.3 is reconfigurable in that it can be stopped and continued on a different number of processors.

4.4 Runtime Support System (RSS)

RSS is a sequential application that can be executed on any machine with which the machines used for the execution of actual parallel application will be able to communicate. RSS exists for the entire duration of the application and spans across multiple migrations of the application. Before the actual parallel application is started, the RSS is launched by the user. The RSS prints out a port number on which it listens for requests. The user fills a configuration file called *srs.config* with the name of the machine where RSS is executing and the port number printed by RSS and makes the configuration file available to the first process of the parallel application. When the parallel application is started, the first process retrieves the location of RSS from the configuration file and registers with the RSS during SRS_Init. The RSS maintains the application configuration of the present as well as the previous executions of the application.

The RSS also maintains an internal flag, called *stop_flag* that indicates if the application has to be stopped. Initially, the flag is cleared by the RSS. A utility called *stop_application* is provided and allows the user to stop the application. When the utility is executed with the location of RSS specified as input parameter, the utility contacts the RSS and makes the RSS set the *stop_flag*. When the application calls

```
int main(int argc, char** argv){
int *global_A, int* local_A;
int global_size, local_size;
MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);
  SRS_Init();

  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  local_size = global_size/size;
  restart_value = SRS_Restart_Value();

  if(restart_value == 0){
    if(rank == 0){

      for(i=0; i<global_size; i++){
        global_A[i] = i;
      }
    }

    MPI_Scatter (global_A, local_size, MPI_INT, local_A, local_size,
                 MPI_INT, 0, comm );

    iter_start = 0;
  }
  else{
    SRS_Read("A", local_A, BLOCK, NULL);
    SRS_Read("iterator", &iter_start, SAME, NULL);
  }

  SRS_Register("A", local_A, GRADS_INT, local_size, BLOCK, NULL);
  SRS_Register("iterator", &i, GRADS_INT, 1, 0, NULL);
```

Figure 4.3: Modified code with SRS calls

```
  for(i=iter_start; i<global_size; i++){
    stop_value = SRS_Check_Stop();
    if(stop_value == 1){
      MPI_Finalize();
      exit(0);
    }
    proc_number = i/local_size;
    local_index = i\%local_size;

    if(rank == proc_number){
      local_A[local_index] += 10;
    }
  }

  SRS_Finish();
  MPI_Finalize();

  exit(0);

}
```

Figure 4.3. Continued

SRS_Check_Stop, the SRS library contacts the RSS and retrieves the *stop_flag*. The application either continues executing or stops its execution depending on the value of the flag.

When the SRS_Check_Stop checkpoints the data used in the application to IBP depots, it sends the location of the checkpoints and the data distributions to the RSS. When the application is later restarted, it contacts the RSS and retrieves the location of the checkpoints from the RSS. When the application finally calls SRS_Finish, the RSS is requested by the application to terminate itself. The RSS cleans the data stored in the IBP depots, deletes its internal data structures and terminates.

The interactions between the different components in the SRS checkpointing architecture is illustrated in Figure 4.4.

Appendix B gives a detailed description about the SRS API.

4.5 Steps for Developing and Executing Malleable Applications

Following is the summary of the actions needed for developing and executing malleable and migratable MPI message passing applications with the SRS library.

1. The user starts IBP depots on all machines where he may execute his application.

2. The user converts his parallel MPI application into a malleable application by inserting calls to SRS library. He then compiles and links with the SRS library.

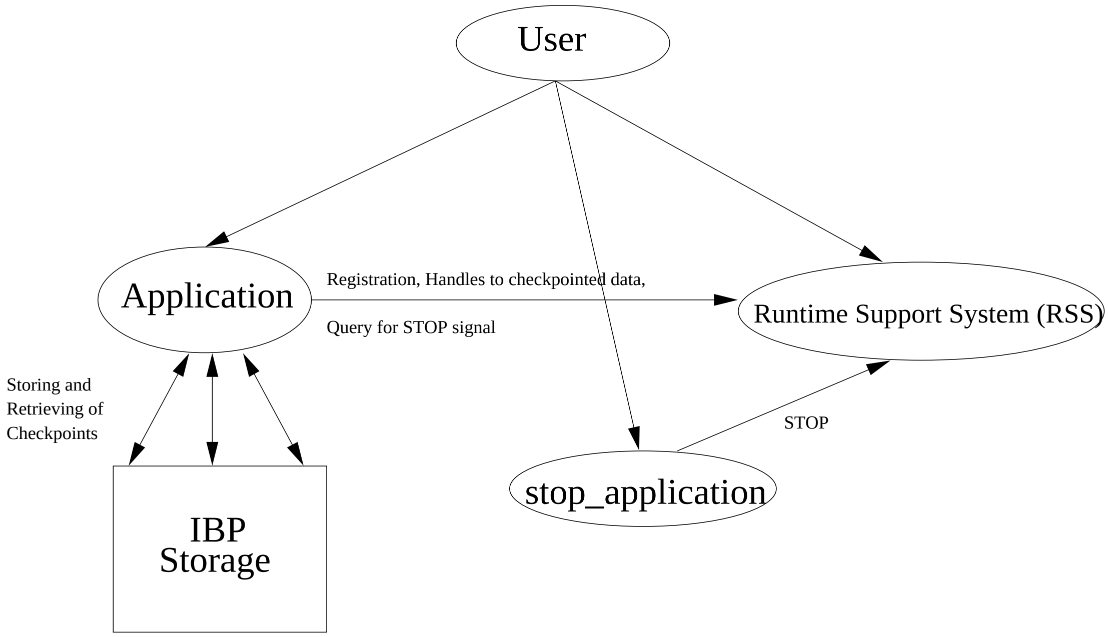

Figure 4.4: Interactions in SRS

3. The user then executes RSS on a machine with which the machines for application execution will be able to communicate. The RSS will output a port number on which it listens for requests.

4. The user creates a configuration file specifying the machine and the port number of RSS.

5. The user stores the configuration file in the working directory of the first process of the parallel application.

6. The user starts his parallel application on a set of machines. The application, through the SRS library communicates with the RSS.

7. In the middle of the application execution, the user can stop the application by using the *stop_application* utility. The user specifies the location and the port number of the RSS to the *stop_application* utility.

8. The user can restart his application on possibly a different number of processors in the same way he initially started his application. After the application completes, the RSS terminates.

4.6 Limitations

Although the SRS framework is robust in supporting migration of malleable parallel applications across heterogeneous environments, it has certain limitations in terms of the checkpointing library and the kind of applications it can support.

1. Although the SRS library can be used in a large number of parallel applications, it is most suitable to iterative applications where SRS_Check_Stop can be inserted at the beginning or at the end of the loop. The SRS library is not suitable for applications like multi-component applications where different data can be initialized and used at different points in the program.

2. Currently, the execution context is restored by the user by the use of appropriate conditional statements in the program. This approach is cumbersome and difficult for the users when programs where multiple nested procedures are involved.

3. The SRS library supports only native data types like single and double precision

floating point numbers, integers, characters etc. It does not support checkpointing of complex pointers, files and structures.

4. Although the main motivation of the SRS library is to help the user proactively stop an executing application and restart and continue it with a different configuration, SRS also allows fault tolerance by means of periodic checkpointing. However, the fault tolerance supported by SRS is limited in that it can tolerate only application failures due to non-deterministic events and not total processor failures. This is because the IBP depots on which the checkpoints are stored also fail when the machines on which the IBP depots are located fail.

4.7 SRS and Metascheduler

After the end application has been instrumented with SRS calls, it can be started, stopped, restarted and continued a number of times by the different components in the GrADS framework. The GrADS Application Launcher, after starting the Contract Monitor, launches the Runtime Support System (RSS) on the machine where the user initiated the GrADS application execution. The Application Launcher then creates the configuration file, srs.config needed by the SRS library. The Application Launcher fills the srs.config file with information about the location of the RSS and the Database Manager and stages the configuration file to the machine that will hold the first process of the executing application determined by the Performance Modeler. The Application Launcher finally launches the end application.

When the end application executes, the SRS library associated with the end application reads the parameters from the srs.config file, registers its state as STARTED with the Database Manager and communicates with the RSS. When a metascheduling component like the Permission Service, Contract Negotiator or the Rescheduler decides to stop the application, it contacts the RSS corresponding to the application and sends a STOP signal. This signal is conveyed to the end application, when the application executes its next SRS_Check_Stop call. The end application stops execution and stores its status as STOPPED in the Database Manager. The Contract Monitor that monitors the end application also stops execution. The GrADS Application Manager that launched the end application through the Application Launcher, reads the status of the end application from the Database Manager, learns that the application has stopped and waits for the status of the application to change to RESUME.

When the metascheduler component decides to continue the end application, it stores RESUME for the status of the application. This prompts the Application Manager to restart from the Resource Selection phase in its life cycle and ultimately relaunch the end application and the Contract Monitor with the new schedule and performance prediction.

Chapter 5

Experiments and Results

This chapter includes experiments and results presented in the following papers.

Sathish S. Vadhiyar and Jack J. Dongarra. A Metascheduler For The Grid. Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing. pages 343-351. July, 2002.

Sathish S. Vadhiyar and Jack J. Dongarra. A Performance Oriented Migration Framework for the Grid. To appear in the Proceedings of The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003).

Sathish S. Vadhiyar and Jack J. Dongarra. Self Adaptivity in Grid Computing. Submitted to the special issue of Concurrency: Practice and Experience on Grid Performance, 2003.

Sathish S. Vadhiyar and Jack J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. Submitted to Parallel Processing Letters, 2003.

I was the primary contributor of the papers and was involved in conducting experiments and obtaining the results. In this chapter, more experiments and results are added to the paper versions.

In this chapter, experimental results corresponding to different experiments are provided. Some of the experiments are intended to demonstrate the usefulness of the metascheduling components. Some experiments were conducted to verify the accuracy in predicting the cost associated with redistribution of data during rescheduling. The experiments in the third section were conducted to study the overhead associated with the SRS checkpointing library. The experiments in the final section were conducted to verify the robustness of our metascheduler when large number of problems are submitted to the system. The experiments in the final section also help in studying the different characteristics of the metascheduler and in comparing with the situations when the metascheduler was not used.

5.1 Usefulness of Metascheduling Components

The following experiments were conducted for demonstrating the usefulness of different metascheduling components.

ScaLAPACK LU and QR factorization codes were instrumented such that the time taken for each iteration corresponding to a block of the matrix is measured and monitored. IBP [82] depots, where storage can be allocated, are started on the processors of the Grid System. The experimental testbed consists of the machines shown in Table 2.1. For all the experiments, the simulated annealing scheduler by Yarkhan [117] was used for the GrADS Performance Modeler.

For the easy demonstration of the usefulness of the metascheduling components, only a subset of the machines in the entire GrADS testbed shown in Table 2.1 was used for the experiments in this section. The total execution times reported in the following subsections include the time for Grid overhead and not just the time taken by the end application. The times for the Grid overhead was reported in Chapter 2 and explained in great detail in our previous work [81].

5.1.1 Permission Service

In this experiment, we demonstrate the functionality of the Permission Service. For the experiments in this section, ScaLAPACK LU factorization code was used. A large application, app_1, was introduced into the system consisting of 4 *opus* machines, 1 *torc* machine and 2 *cypher* machines. Ten minutes after app_1 started, a relatively small application, app_2, that intended to use only the 4 *opus* machines was introduced into the system. app_2 was chosen such that its memory requirements were greater than the memory available in the *opus* system when app_1 was executing. In the following experiment, a linear algebra problem with matrix size 13000 was chosen for app_1. The Permission Service evaluated the performance benefits of stopping app_1, accommodating app_2, and restarting app_1 after the completion of app_2. The functionality of the Permission Service, when the matrix size of the linear algebra problem, app_2, is 5000, is illustrated on a single *opus* machine in Figure 5.1.

In Figure 5.2, we observe the percentage performance loss incurred by app_1 due to the accommodation of app_2 in the system. The x-axis represents different matrix sizes for app_2 and the y-axis represents the percentage performance loss incurred by app_1. Two points can be observed from Figure 5.2. First, for less than 20% of performance loss

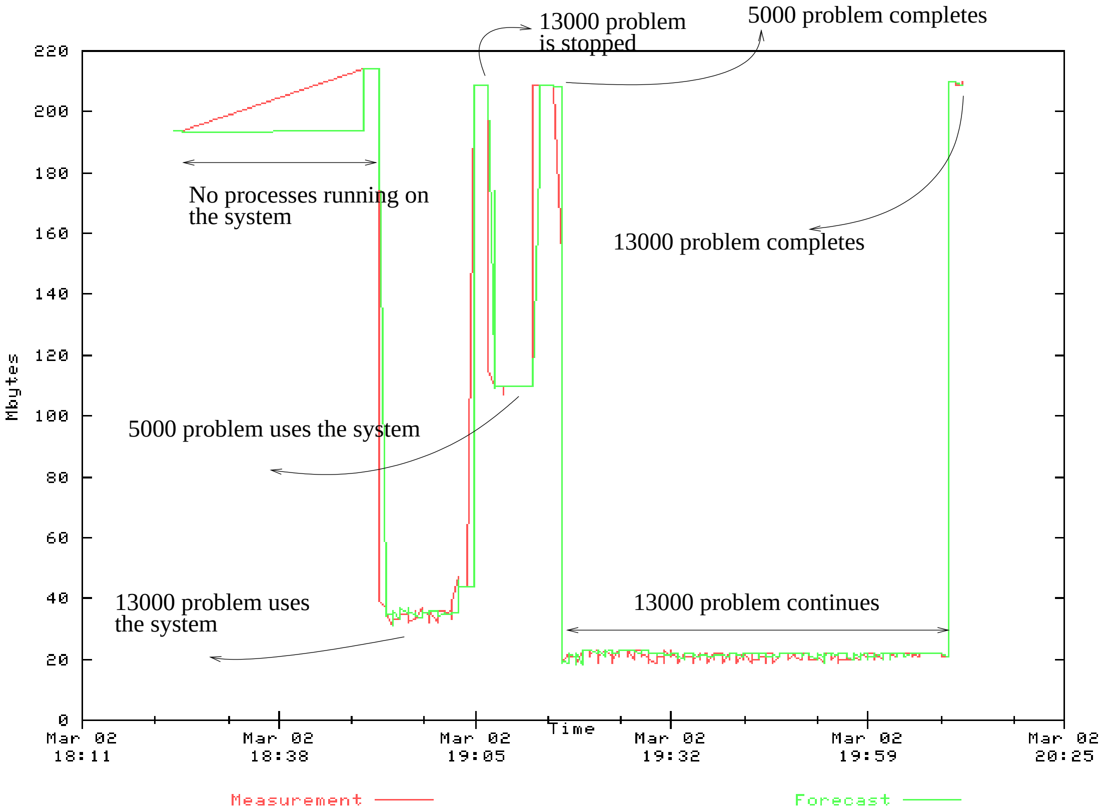

Figure 5.1: Free memory available on a opus machine during the execution of app_1 and app_2
Source: The NWS interactive query website [8].

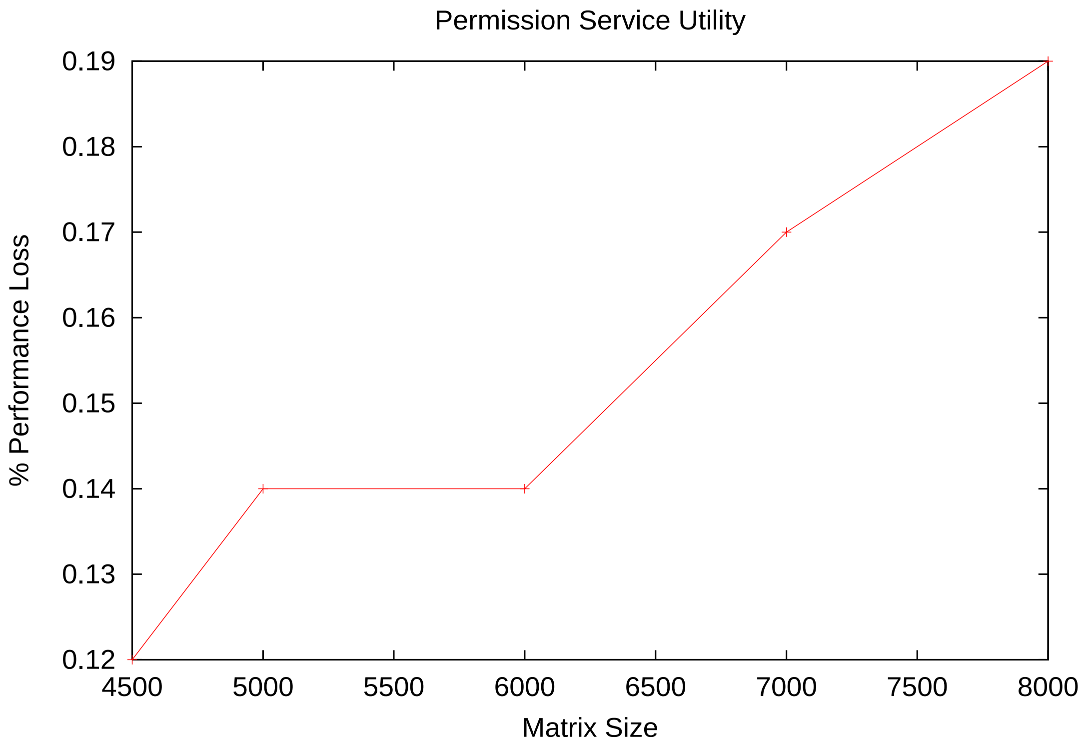

Figure 5.2: Performance loss for app_1

for app_1, the system was able to accommodate app_2. Without the Permission Service mechanism, app_2 would not have been able to use the system. Second, the performance loss increases with the increase in problem size of app_2. When the problem size of app_2 is comparable with the problem size of app_1, the Permission Service determines that performance benefits cannot be achieved for the system by accommodating app_2.

5.1.2 Contract Negotiator

In this experiment, we demonstrate the utility of the Contract Negotiator in accommodating a new application by stopping an already running application, if significant performance benefits can be obtained for the new application. The stopped application is restarted after the new application completes its execution. For this experiment, ScaLAPACK LU factorization code was used on only *cypher* machines. In this experiment, an application, app_1 is executed on N processors. 3 minutes after app_1 started its execution, an application, app_2 is introduced in the Grid system. app_2 is intended to use (N+1) processors. Since N of the processors were occupied by app_1, only a single processor is available for app_2. The Contract Negotiator analyzes the performance benefits that can be obtained by stopping app_1 and making (N+1) processors available for app_2. In the experiments, matrix size 7500 was used for app_2. The total execution time of a 7500 matrix size ScaLAPACK problem when executed on a single processor is 818.11 seconds.

We define

1. Execution time of app_1 without rescheduling, $exec1_{without\_re}$
2. Execution time of app_1 with rescheduling, $exec1_{with\_re}$
3. Execution time of app_2 without rescheduling, $exec2_{without\_re}$
4. Execution time of app_2 with rescheduling, $exec2_{with\_re}$
5. Performance loss for app_1, perf_loss

$$perf\_loss = \frac{exec1_{with\_re} - exec1_{without\_re}}{exec1_{without\_re}}$$

6. Performance gain for app_2, perf_gain

$$perf\_gain = \frac{exec2_{without\_re} - exec2_{with\_re}}{exec2_{without\_re}}$$

7. Utility value, util_val

$$util\_val = \frac{perf\_gain}{perf\_loss}$$

util_val > 1 indicates that the rescheduling strategy is useful for the entire system. util_val < 1 indicates that the rescheduling strategy can cause an overall loss in performance for the entire system. Greater the value of util_val, more the usefulness of the rescheduling strategy.

Table 5.1 shows the matrix sizes of app_1, the number of processors N, the number of processors eventually used by app_2 and the util_val. Note that the eventual number of processors used by app_2 depends on system conditions and execution time model and is not always the (N+1) processors available to app_2.

We observe from Table 5.1, that the values of util_val are consistently high for the above experiments. This proves that the scheduling strategy of compromising long running jobs for short running jobs is beneficial to the entire system. The value of util_val depends on a number of factors including the times for the long and short jobs and the times for checkpointing the states of the long job. For e.g., even though the 7500

Table 5.1: Utility of Contract Negotiator

Matrix Size of app_1	*Processors N*	*Number of Processors used by* app_2	*util_val*
15000	4	5	2.13
17000	5	6	5.11
18500	6	7	2.27
20000	7	8	2.04
21000	8	9	2.05
22500	9	9	2.36
24000	10	9	1.72

matrix problem uses 9 processors when matrix sizes of 21000, 22500 and 24000 were used for app_1, the performance benefits due to the scheduling strategy obtained for the cases when matrix sizes of app_1 were 21000 and 22500, were higher than the performance benefit obtained for the case when matrix size of app_1 was 24000. This was due to the longer time incurred for checkpointing the state of the 24000 matrix problem when compared to checkpointing the states of 21000 and 22500 matrix problems. Also, there is an optimal combination of long and short job sizes when the performance benefits due to scheduling strategy can be high. In the above experiments, we observe that matrix size 17000 for app_1 leads to such high performance for the overall system.

5.1.3 Rescheduler

In the experiments in this section, ScaLAPACK QR factorization was used as the end application. For all the experiments in this section, the worst cast rescheduling cost of 900 seconds as shown in Table 3.1 was used for rescheduling decisions.

Migration on Request

In all the experiments in this section, 4 *msc* machines and 8 *opus* machines were used. A given matrix size for the QR factorization problem was input to the Application Manager. Since the *msc* machines were faster than the *opus* machines, the Application Manager by means of the performance modeler chose the 4 *msc* machines for the end application run. A few minutes after the start of the end application, artificial load is introduced into the 4 *msc* machines. This artificial load is achieved by executing a certain number of loading programs on each of the *msc* machines. The loading program used was a sequential C code that consists of a single looping statement that loops forever. This program was compiled without any optimization in order to achieve the loading effect.

Due to the loss in predicted performance caused by the artificial load, the contract monitor requested the Rescheduler to migrate the application. The Rescheduler evaluated the potential performance benefits that can be obtained by migrating the application to the 8 *opus* machines and either migrated the application or allowed the application to continue on the 4 *msc* machines. The Rescheduler was operated in two modes - a default and a non-default mode. The normal operation of the Rescheduler is its default mode and the non-default mode of the Rescheduler is when the Rescheduler code was modified to force the application to either migrate or continue on the same set of resources. Thus in cases when the default mode of the Rescheduler was to migrate the application, the non-default mode was to continue the application on the same set of resources and in cases when the default mode of the Rescheduler was to not migrate the application, the non-default mode was to force the Rescheduler to migrate the application by adjusting the rescheduling cost parameters. For each experimental run, results were obtained for both when Rescheduler was operated in the default and non-default mode. This allowed us to compare both scenarios and to verify if the Rescheduler made the right decision.

Three parameters were involved in each set of experiments - the size of the matrices, the amount of load and the time after the start of the application when the load was introduced into the system. The following three sets of experiments were obtained by fixing two of the parameters and varying the other parameter.

In the first set of experiments, the artificial load consisting of 10 loading programs was introduced into the system 5 minutes after the start of the end application. The bar chart in Figure 5.3 was obtained by varying the size of the matrices, i.e. the problem size on the x-axis. The y-axis represents the execution time in seconds of the entire problem including the Grid overhead. For each problem size, the bar on the left represents the execution time when the application was not migrated and the bar on the right represents the execution time when the application was migrated.

Several points can be observed from Figure 5.3. The time for reading checkpoints occupied most of the rescheduling cost since it involves moving data across the Internet from Tennessee to Illinois and redistribution of data from 4 to 8 processors. On the other hand, the time for writing checkpoints is insignificant since the checkpoints are

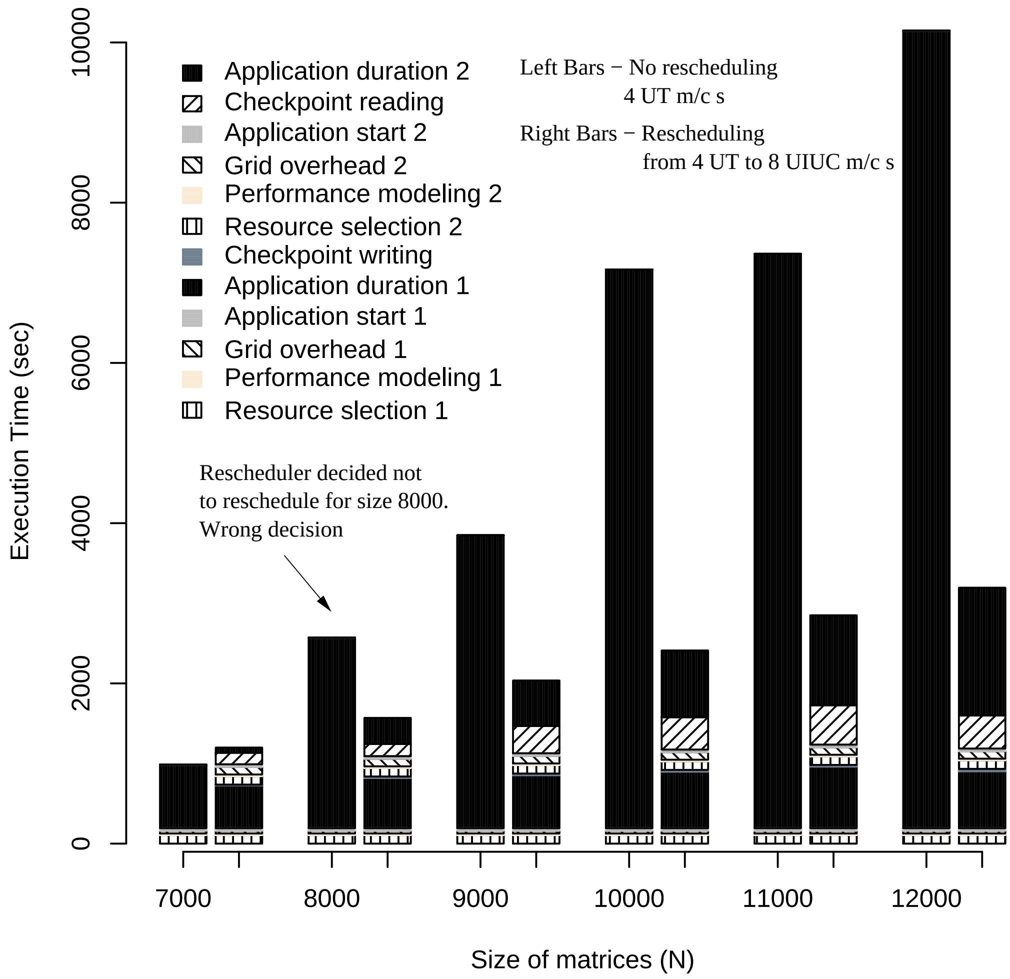

Figure 5.3: Effect of Problem Sizes on Migration

written to local disks. The rescheduling benefits are more for large problem sizes since the remaining lifetime of the end application when load is introduced is larger for larger problem sizes. There is a particular size of the problem below which the migrating cost overshadows the performance benefit due to rescheduling. Except for matrix size 8000, the Rescheduler made the correct decision for all matrix sizes. For matrix size 8000, the Rescheduler assumed a worst-case rescheduling cost of 900 seconds while the actual rescheduling cost was close to about 420 seconds. Thus the Rescheduler evaluated the performance benefit to be negligible while the actual scenario points to the contrary. Thus the pessimistic approach followed by using a worst-case rescheduling cost in the Rescheduler will lead to underestimating the performance benefits due to rescheduling in some cases.

In the second set of experiments, matrix size 12000 was chosen for the end application and artificial load was introduced 20 minutes into the execution of the application. In this set of experiments, the amount of artificial load was varied by varying the number of loading programs that were executed. In Figure 5.4, the x-axis represents the number of loading programs and the y-axis represents the execution time in seconds. For each amount of load, the bar on the left represents the case when the application was continued on 4 *msc* machines and the bar on the right represents the case when the application was migrated to 8 *opus* machines.

Similar to the first set of experiments, we find only one case when the Rescheduler made incorrect decision for rescheduling. This case, when the number of loading pro-

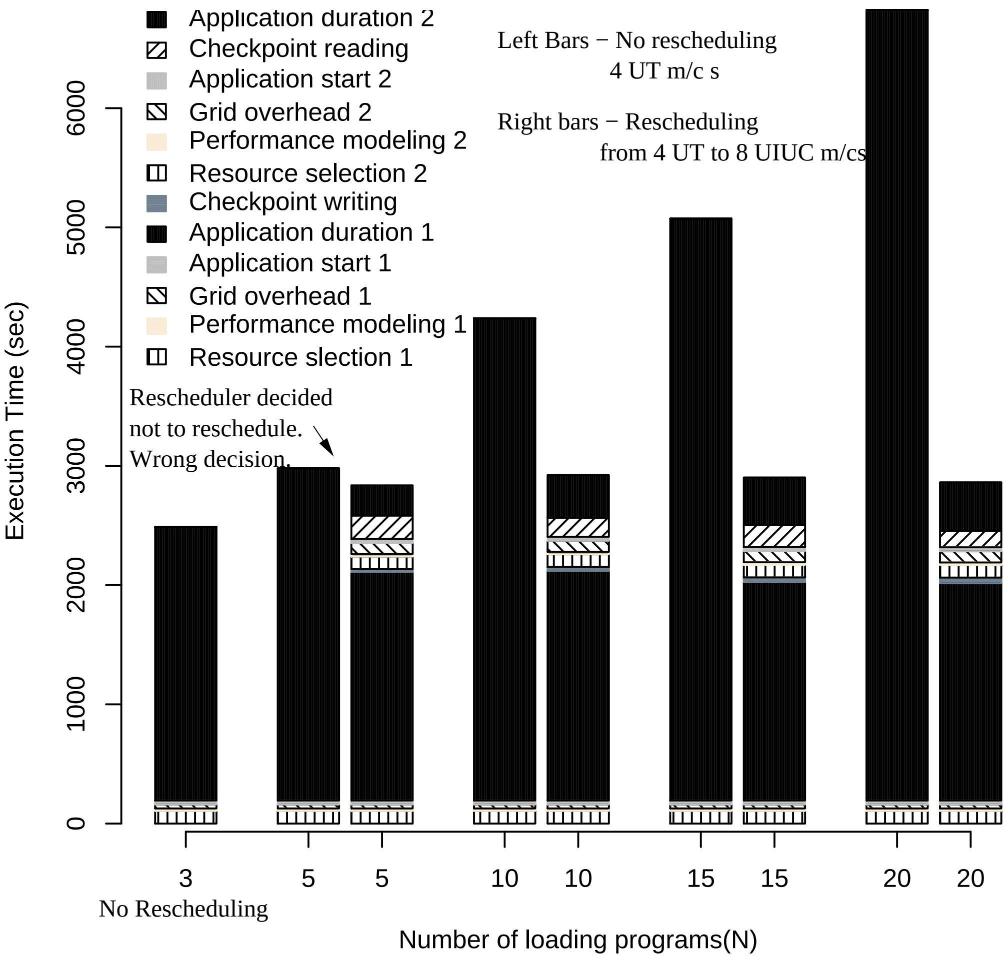

Figure 5.4: Effect of Amount of Load on Migration

grams was 5 was due to the insignificant performance gain that can be obtained due to rescheduling. When the number of loading programs was 3, we were not able to force the Rescheduler to migrate the application since the application completed at the time of rescheduling decision. Also, more the amount of load, the more the performance benefit due to rescheduling because of larger performance losses for the application in the presence of heavier loads. But the most significant result in Figure 5.4 was that the execution times when the application was rescheduled remained almost constant irrespective of the amount of load. This is because, as can be observed from the results when the number of loading programs was 10 and when the number was 20, the more the amount of load, the earlier the application was rescheduled. Hence our rescheduling framework was able to adapt to the external load.

In the third set of experiments, shown in Figure 5.5, equal amount of load consisting of 7 loading programs was introduced at different points of execution of the end application for the same problem of matrix size 12000. The x-axis represents the elapsed time in minutes of the execution of end application when the load was introduced. The y-axis represents the total execution time in seconds. Similar to the previous experiments, the bars on the left denote the cases when the application was not rescheduled and the bars on the right represent the cases when the application was rescheduled.

As can be observed from Figure 5.5, there are diminishing returns due to rescheduling as the load is introduced later into the program execution. The Rescheduler made wrong decisions in two cases - when the load introduction times are 15 and 20 minutes after

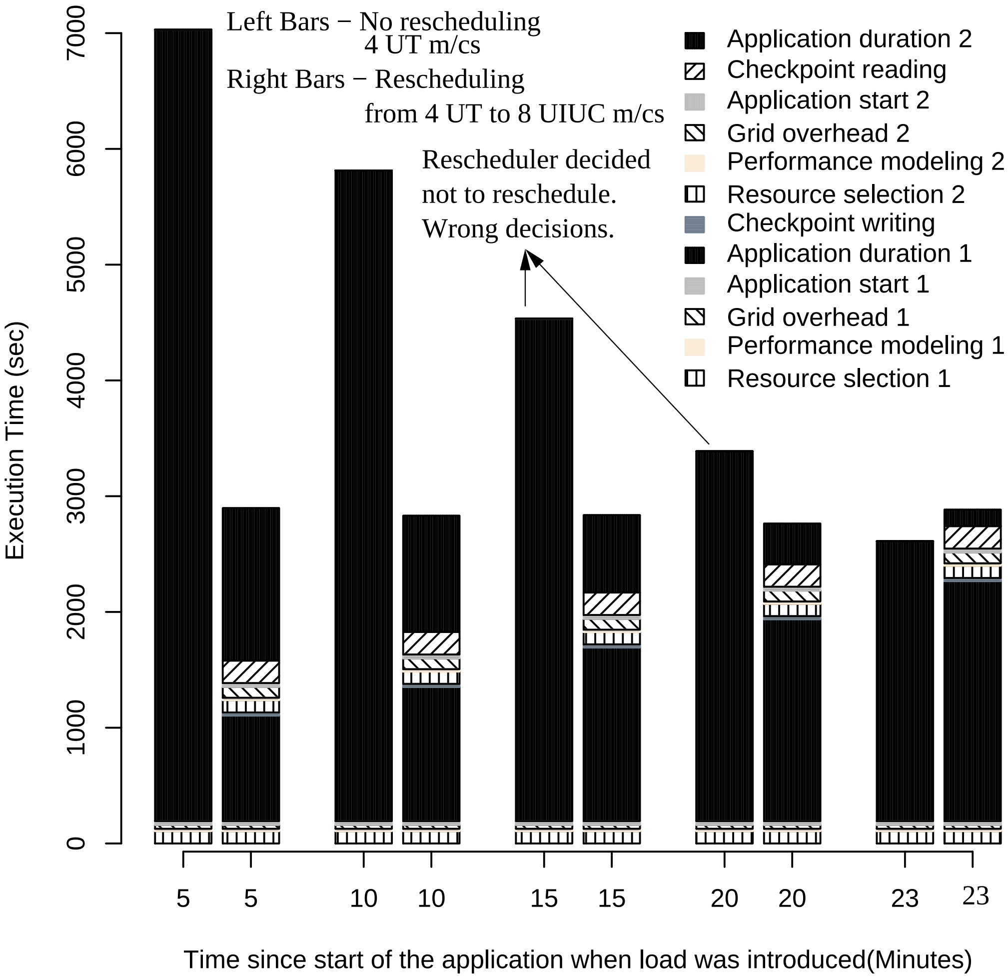

Figure 5.5: Effect of Load Introduction Time on Migration

the start of end application execution. While the wrong decision for 20 minutes can be attributed to the pessimistic approach of rescheduling, the wrong decision of the Rescheduler for 15 minutes was determined to be due to the faulty functioning of the performance model for the ScaLAPACK QR problem for UIUC machines. The most startling result in Figure 5.5 is when the load was introduced 23 minutes after the start of the end application. At this point, the program almost completed and hence rescheduling will not yield performance benefits for the application. The Rescheduler was able to evaluate the scenario and avoid unnecessary rescheduling of the application. Most rescheduling frameworks will not be capable of achieving this since they do not possess the knowledge regarding remaining execution time of the application.

Opportunistic Migration

In this set of experiments, we illustrate opportunistic migration in which the Rescheduler tries to migrate an executing application when some other application completes. For the experiments in this section, ScaLAPACK QR factorization code was used. Two experiments were conducted to demonstrate the usefulness of the Rescheduler.

In the first experiment, an application, app_1, was introduced into the system such that it consumed most of the memory of 8 *msc* machines. During the execution of app_1, an app_2, that intended to use 11 machines, 3 *torcs* and 8 *mscs* was introduced into the system. Since the 8 *msc* machines were occupied by app_1, app_2 was able to utilize only the 3 *torc* machines. When app_1 completed, the 8 *msc* machines were freed and app_2 was able to utilize the extra resources to reduce its remaining execution time. The

Rescheduler evaluated the performance benefits of allowing app_2 to utilize the extra 8 processors.

ScaLAPACK problems of sizes 20000 and 21000, depending on the available memory on *mscs* when the experiments were run, were used for app_1. ScaLAPACK problem of size 11000 was used for app_2.

We define

1. Total execution time of app_2 on 3 *torcs* without rescheduling, $exec_{without\_re}$

2. Total execution time of app_2 with rescheduling, $exec_{with\_re}$

3. Percentage rescheduling gain for app_2, $percentage_gain$

$$percentage\_gain = \frac{exec_{without\_re} - exec_{with\_re}}{exec_{without\_re}}$$

app_2 was introduced at various points of time after the starting of app_1. Hence additional resources will be available for app_2 at various points of time into its execution. The total number of iterations needed by the ScaLAPACK problem of size 11000 was 275. Figure 5.6 illustrates the utility of rescheduling as a function of the remaining number of iterations left for app_2 when app_2 was rescheduled. We observe that the percentage rescheduling gain for app_2 increases when the remaining execution time left for app_2 at the time of rescheduling increases. The rescheduling gain depends on a number of parameters like the time taken for redistribution of data and the number of additional resources available etc. These parameters depend on the specific application

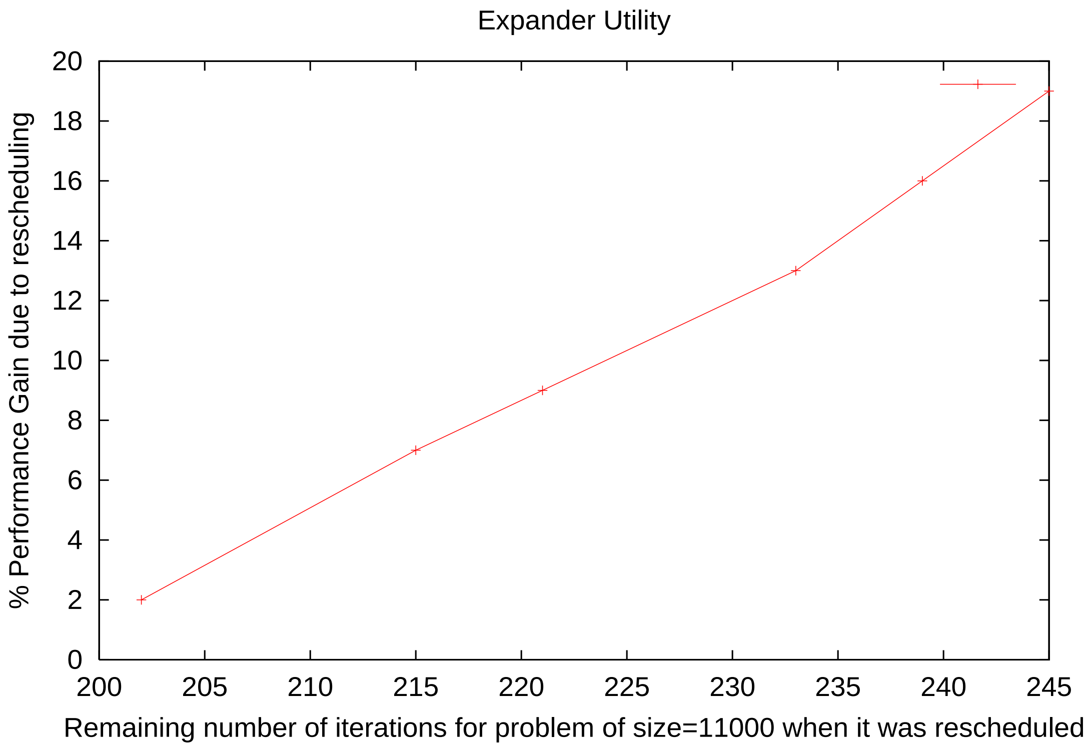

Figure 5.6: Rescheduling gain for app_2

for which rescheduling is done.

For the second experiment, two problems were involved similar to the first experiment. For the first problem, matrix size of 14000 was input to the Application Manager and 6 *msc* machines were made available. The Application Manager, through the Performance Modeler chose the 6 machines for the end application run. Two minutes after the start of the end application for the first problem, a second problem of a given matrix size was input to the Application Manager. For the second problem, the 6 *msc* machines on which the first problem was executing and 2 *opus* machines were made available. Due to the presence of the first problem, the 6 *msc* machines alone were insufficient to accommodate the second problem. Hence the performance model chose the 6 *msc* machines and 2 *opus* machines for the end application and the actual application run involved communication across the Internet.

In the middle of the execution of the second application, the first application completed and hence the second application can be potentially migrated to use only the 6 *msc* machines. Although this involved constricting the number of processors of the second application from 8 to 6, there can be potential performance benefits due to the non-involvement of Internet. The Rescheduler evaluated the potential performance benefits due to migration and made an appropriate decision.

Figure 5.7 shows the results for two illustrative cases when matrix sizes of the second application were 13000 and 14000. The x-axis represents the matrix sizes and the y-axis represents the execution time in seconds. For each application run, three bars are shown. The bar on the left represents the execution time for the first application that was executed on 6 *msc* machines. The middle bar represents the execution time of the second application when the entire application was executed on 6 *msc* and 2 *opus* machines. The bar on the right represents the execution time of the second application, when the application was initially executed on 6 *msc* and 2 *opus* machines and later migrated to execute on only 6 *msc* machines when the first application completed.

In both problem cases, matrix sizes 13000 and 14000, for the second problem, the Rescheduler made the correct decision of migrating the application. We also find that for both problem cases, the second application was almost immediately rescheduled

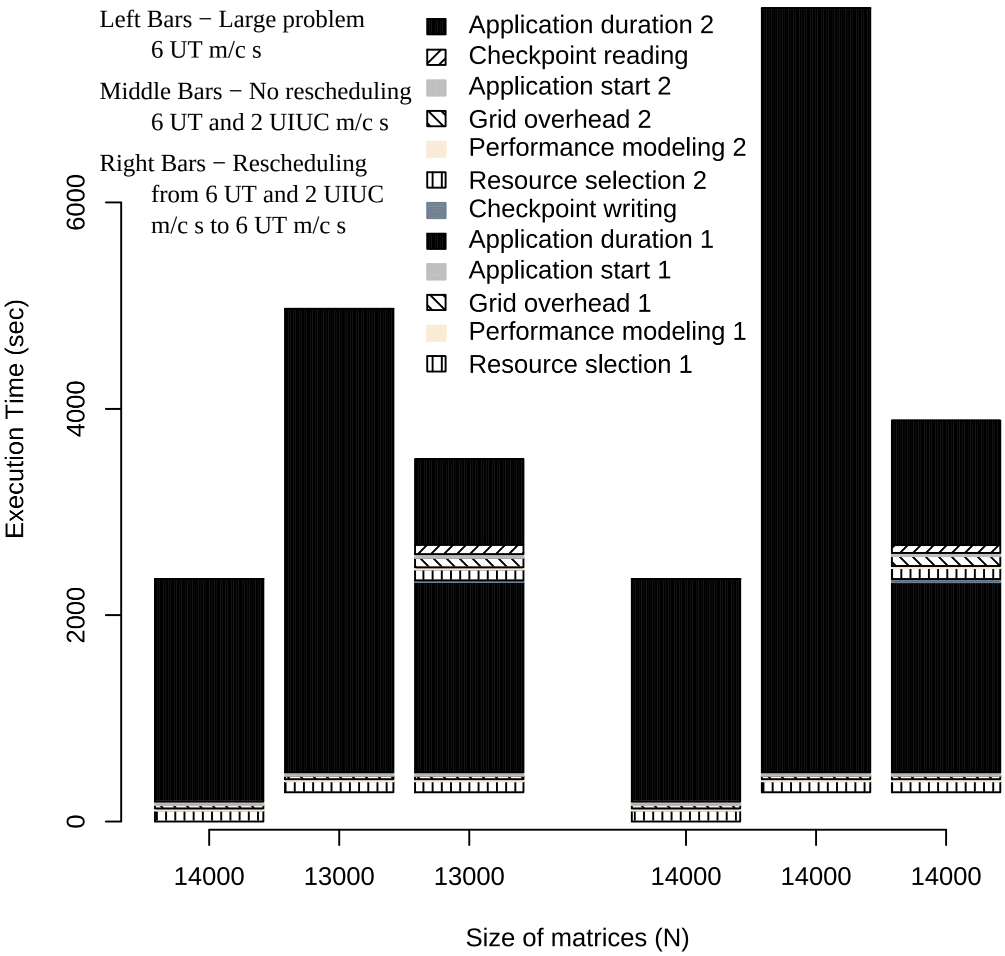

Figure 5.7: Opportunistic Migration

after the completion of the first application.

5.2 Predicting Redistribution Cost

As observed in Figures 5.3, 5.4, 5.5, the Rescheduler makes wrong decisions for rescheduling in certain cases. In cases where the Rescheduler made the wrong decisions, the Rescheduler decided that rescheduling the executing application will not yield significant performance benefits for the application while the actual results point to the contrary. This is because the Rescheduler used the worst case times shown in Table 3.1 for different phases of rescheduling while the actual rescheduling cost was less than the worst case rescheduling cost for cases when the Rescheduler made the wrong decisions.

As shown in Table 3.1, of the various costs involved in rescheduling, the cost for reading and redistribution of data is the highest. The data redistribution and reading the checkpoints are performed in a single operation where the processes determine the portions and locations of data needed by them and read the checkpoints directly from the IBP [82] depots. The data redistribution cost depends on a number of factors including the number and amount of checkpointed data, the data distributions used for the data , the current and future processors sets for the application used before and after rescheduling respectively, the network characteristics, particularly the latency and bandwidth, of the links between the current and future processor sets etc. The rescheduling framework was extended to predict the redistribution cost and use the predicted redistribution cost for calculating the gain due to rescheduling the executing application. Though the time for writing the checkpoints also depends on the size of the checkpoints which in turn depends on the problem size, the checkpoint writing time is insignificant due to the design of the rescheduling architecture where the processes write checkpoint data to the local disks. Hence the time for checkpoint writing is not predicted in the rescheduling framework.

Similar to the SRS library, the Rescheduling framework has also been extended to support common data distribution algorithms like block, cyclic and block-cyclic distributions. When the end application calls SRS_Register to mark the checkpointed data, it also specifies the data distribution used for the data. If the data distribution is one of the common data distributions, the input parameter used for the distribution is stored in an internal data structure of the SRS library. For e.g., if block-cyclic data distribution is specified for the data, the block size used for the distribution is stored in the internal data structure. When the application calls, SRS_StoreMap, the data distributions used for the different data along with the parameters used for the distribution are sent to the Runtime Support System (RSS).

When the Rescheduler wants to calculate the rescheduling cost of an executing application, it contacts the RSS of the application, and retrieves various information about the data that were marked for checkpointing including the total size and data types of the data, the data distributions used for the data and the parameters used for the data distributions. For each data that uses one of the common data distributions supported by the Rescheduler, the Rescheduler determines the data maps for the current processor configuration on which the application is executing and the future processor configuration where the application can be potentially rescheduled. A data map indicate the total number of panels of the data and the size and location of each of the data panel. The Rescheduler calculates the data map using the data distribution and the parameters used for data distribution, it collected from RSS. Based on the data maps for the current and future processor configuration and the properties of the networks between the current and future processor configuration it collected from NWS, the Rescheduler simulates the redistribution behavior. The end result of the simulation is the predicted cost for reading and redistribution of checkpointed data if the application was rescheduled to the new processor configuration. The Rescheduler uses this predicted redistribution cost for calculation the potential rescheduling gain that can be obtained due to rescheduling the application.

An experiment was conducted in which the simulation model for predicting the redistribution cost was validated. In this experiment, 4 *msc* and 8 *opus* machines were used. A ScaLAPACK QR factorization problem was submitted to the GrADS Application Manager. Since the *msc* machines are faster than the *opus* machines, the 4 *msc* machines were chosen by the Performance Modeler for the execution of the end application. 5 minutes after the start of the execution of the end application, artificial loads are introduced in the *msc* machines by the execution of 10 loading programs on each of the *msc* machines. When the Contract Monitor contacted the Rescheduler requesting for rescheduling the application, The Rescheduler dynamically predicted the the

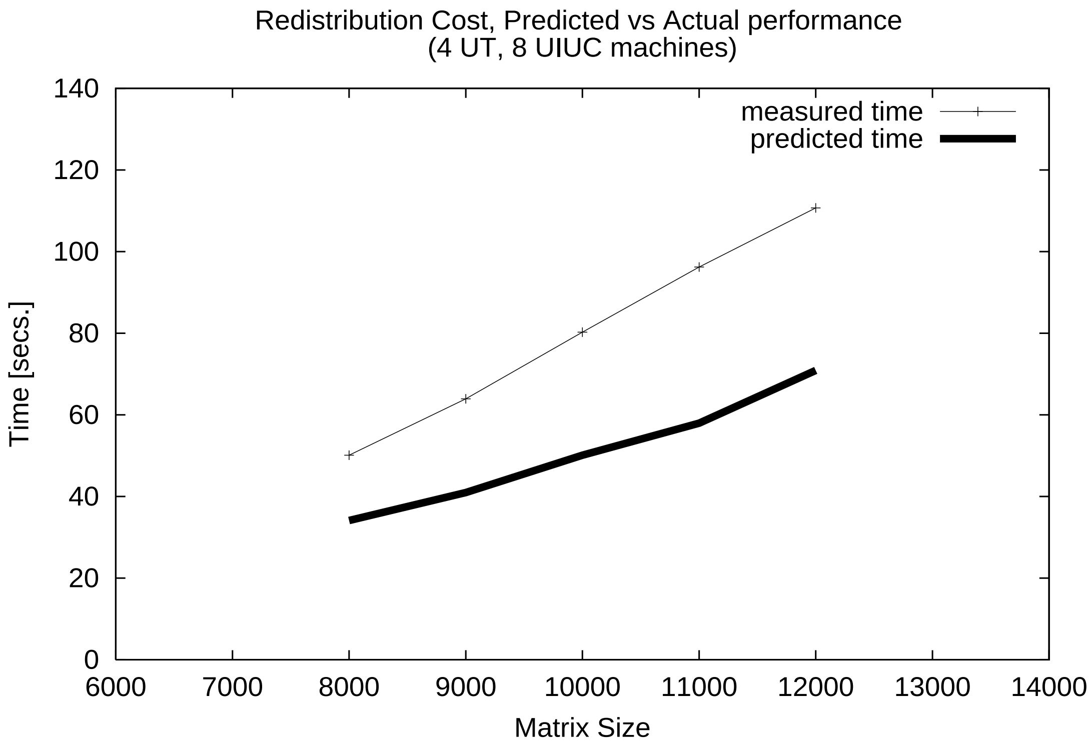

Figure 5.8: Redistribution Cost Prediction

redistribution cost involved in rescheduling the application from *msc* to *opus* machines. Figure 5.8 compares the predicted and the actual cost for redistribution of data in the application for different problem sizes. The x-axis denoted the matrix sizes used for the QR factorization problem and the y-axis represents the redistribution time.

From Figure 5.8, we find that the Rescheduler was able to perform a reasonable simulation of the redistribution of data. The actual redistribution cost was greater than the predicted redistribution cost by only 30-40 seconds. The difference is mainly due to the unpredictable behavior in the network characteristics of the Internet connection between Tennessee and Illinois, Urban-Champaign. By employing the predicted redistribution cost, the Rescheduler was able to make the right decisions for rescheduling for cases in Figures 5.3, 5.4 and 5.5 when it previously made wrong decisions.

5.3 SRS Checkpointing Experiments

In the experiments in this section, *msc* and *opus* clusters were used. The application used for SRS checkpointing was ScaLAPACK QR factorization. The experiments were conducted on non-dedicated machines.

5.3.1 SRS Overhead

In the first experiment, the overhead of SRS library was analyzed when checkpointing of data is not performed. Thus the application instrumented with SRS library simply connects to a RSS daemon and runs to completion. Figure 5.9 compares the execution of the factorization application on 8 *msc* machines when operated in three modes. The "Normal" mode is when the plain application without SRS calls is executed. In the second mode, the application instrumented with SRS library was executed connecting to a RSS daemon started at UT. In the third mode, the application instrumented with SRS library was executed connecting to a RSS daemon started at UIUC. The x-axis represents the matrix sizes used for the problem and the y-axis represents the total elapsed execution time of the application.

The maximum overhead of using SRS when RSS was started at UT was 15% of the overall execution time of the application. This is close to the 10% overhead that

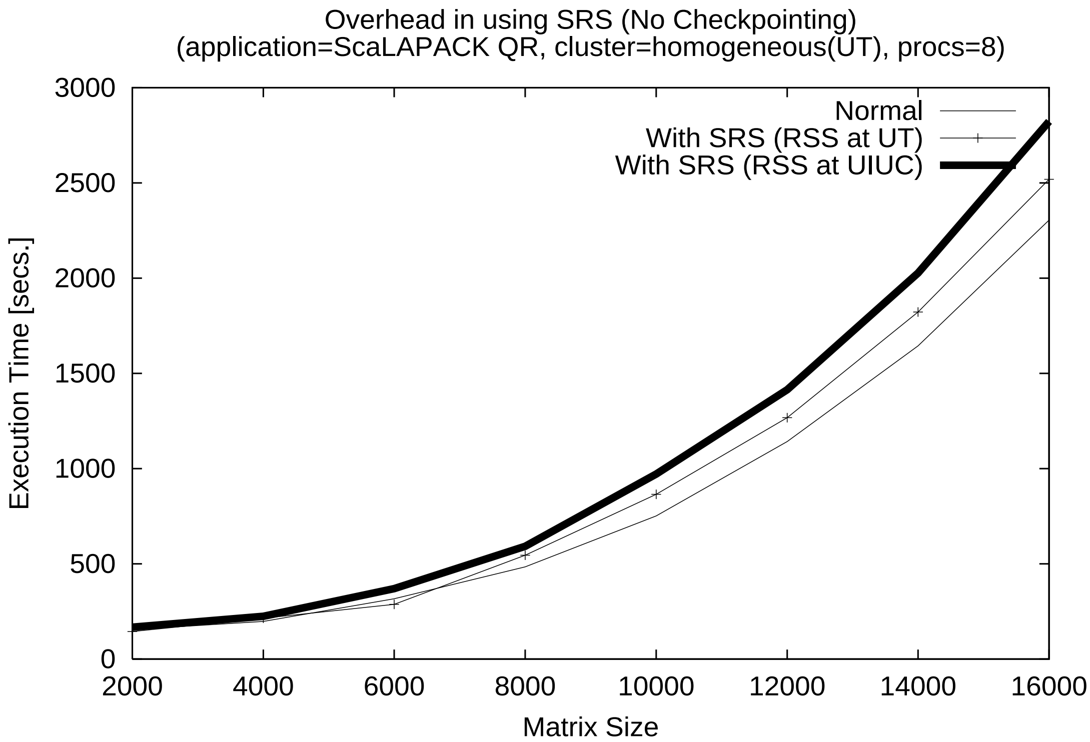

Figure 5.9: Overhead in SRS on a homogeneous cluster (No Checkpointing)

Table 5.2: Details of Periodic Checkpointing used for Figure 5.10

Matrix Size	*Number of Checkpoints*	*Size per Checkpoint (MBytes)*	*Time for storing a checkpoint (Seconds)*
8000	1	64	6.51
10000	2	100	10.06
12000	3	144	13.68
14000	4	196	32.34
16000	5	256	93.25

is desired for checkpointing systems [74]. The worst-case overhead of using SRS when RSS was started at UIUC was 29% of the overall execution time of the application. The increased overhead is due to the communication between SRS and RSS during initialization and at different phases of the application. Since RSS was located at UIUC, the communications involved slow Internet bandwidth between UT and UIUC. The large overhead can be justified by the benefits the SRS library provide in reconfiguring applications across heterogeneous sites.

Figure 5.10 shows the results of an experiment similar to the first experiment, but with the periodic checkpointing option turned on. In the periodic checkpointing mode, the SRS library checkpoints the application data to IBP depots every 10 minutes.

The worst-case SRS overheads in this experiment was high - 23% of the application time when RSS was located at UT and 36% of the application time when RSS was located at UIUC. The details of the periodic checkpointing used for Figure 5.10 is given in Table 5.2.

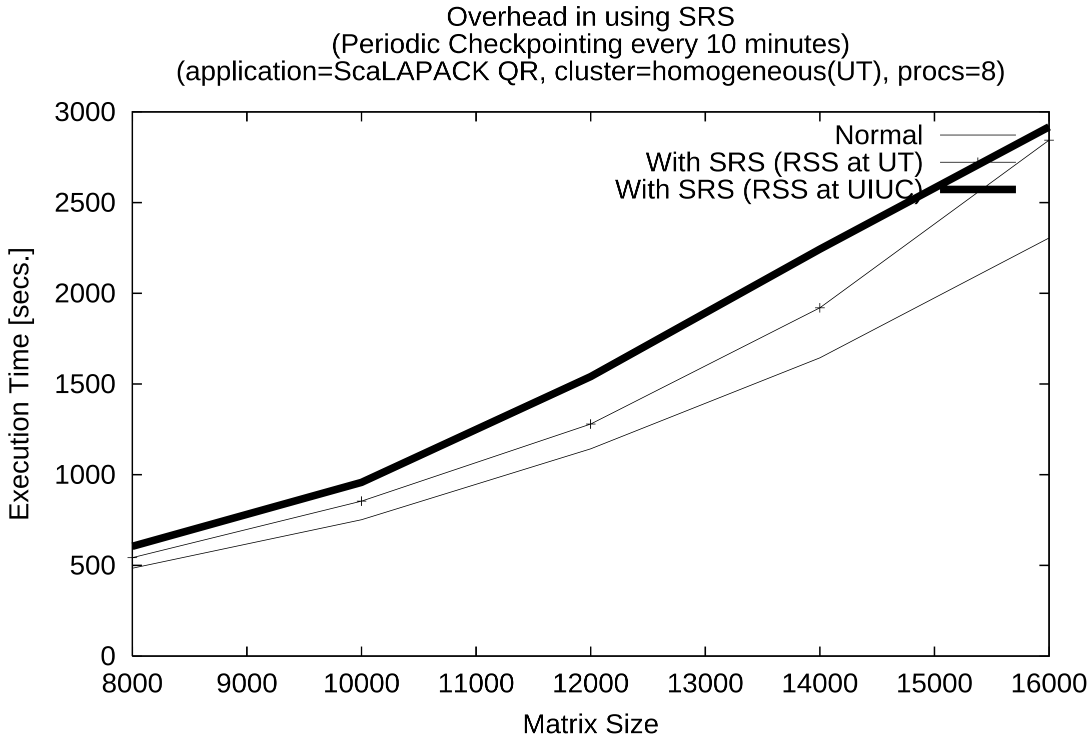

Figure 5.10: Overhead in SRS on a homogeneous cluster (Periodic Checkpointing)

Table 5.3: Details of Periodic Checkpointing used for Figure 5.11

Matrix Size	*Number of Checkpoints*	*Size per Checkpoint (MBytes)*	*Time for storing a checkpoint (Seconds)*
2000	1	2.5	4.59
4000	1	10.6	9.34
6000	2	24	11.22
8000	3	42.7	13.50
10000	5	66.7	18.51

From Table 5.2, it is clear that the high worst-case SRS overheads seen in Figure 5.10 are not due to the time taken for storing checkpoints. We suspect that the overheads are due to the transient loads on the non-dedicated machines.

In the third experiment in this subsection, the application was executed in a heterogeneous environment comprising 8 *opus* and 4 *msc* machines. The application was operated in 3 modes. "Normal" was when the plain application was executed. In the second mode, the application instrumented with SRS calls was executed without checkpointing. In the third mode, the application instrumented wit SRS calls was executed with periodic checkpointing of every 10 minutes. In the SRS mode, the RSS was started at UT. Figure 5.11 shows the results of the third experiment. The worst-case SRS overhead for the application was 15% and hardly noticeable in the figure. The details of the periodic checkpointing used in the third mode for the figure is given in Table 5.3.

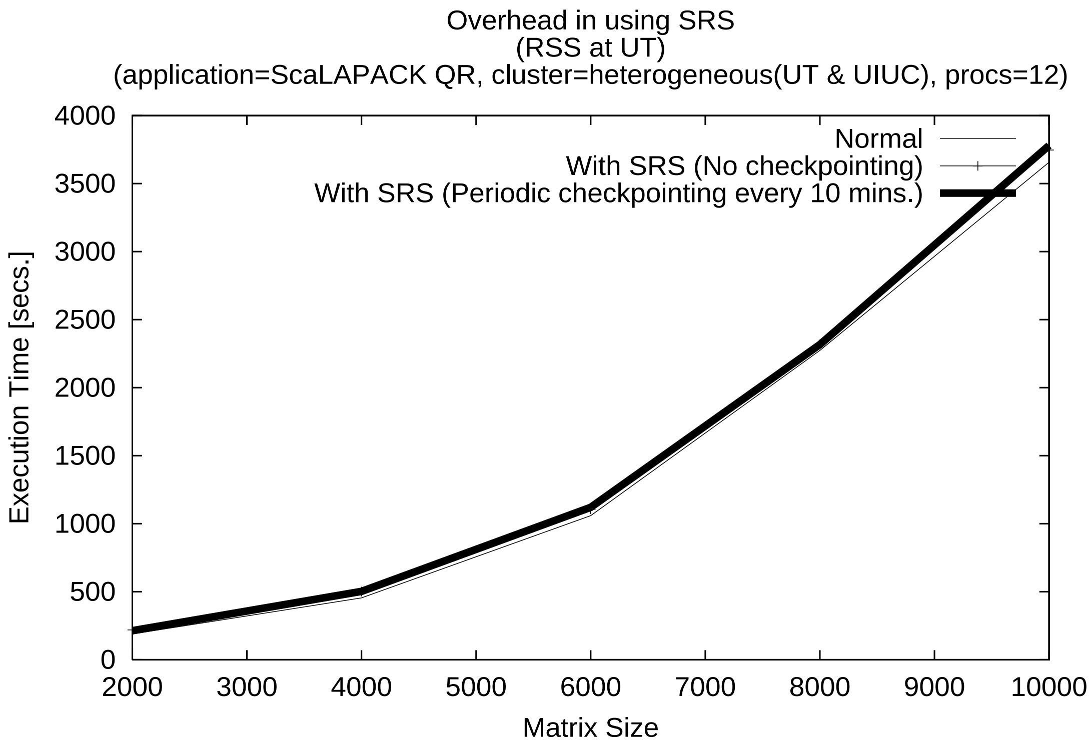

Figure 5.11: Overhead in SRS on a heterogeneous cluster

5.3.2 SRS for Moldable Applications

In this subsection, results for experiments where the application is stopped and restarted on the same number of processors are shown. The application instrumented with SRS calls was initially executed at 8 *msc* machines. 3 minutes after the start of the application, the application was stopped using the *stop_application* utility. The application was restarted on the same number of machines. In this scenario, the processes of the parallel application read the corresponding checkpoints from the IBP storage without performing any redistribution of data. The RSS daemon was started at UT.

Figure 5.12 shows the times for writing and reading checkpoints when the application was restarted on the same 8 *msc* machines on which it was originally executing. From the figure, we find that the times for writing and reading checkpoints are very low and in the range of 7-10 seconds. Thus the application can be removed from a system and restarted later on the same system for various reasons without much overhead. The time between when the stop signal was issued to the application and when the application actually stops depends on the moment when the application calls SRS_Check_Stop after the stop signal. Table 5.4 gives the checkpoint sizes used in Figure 5.12.

Figure 5.13 shows the results when the application was started at 8 *msc* machines, stopped and restarted at 8 *opus* machines. The increased times in reading checkpoints is due to the communication of checkpoints across the Internet from *msc* to *opus* machines.

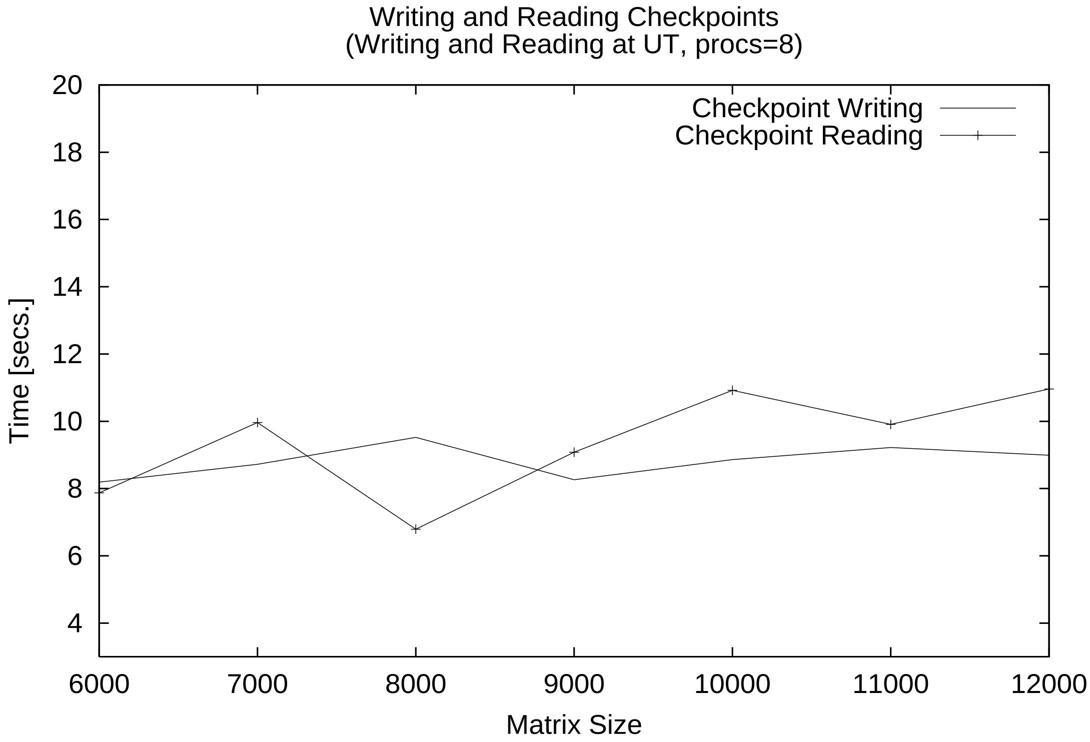

Figure 5.12: Times for Checkpoint Writing and Reading when the application was restarted on *msc* machines

Table 5.4: Details of Checkpointing used in Figure 5.12

Matrix Size	*Size per Checkpoint (MBytes)*
6000	36
7000	49
8000	64
9000	81
10000	100
11000	121
12000	144

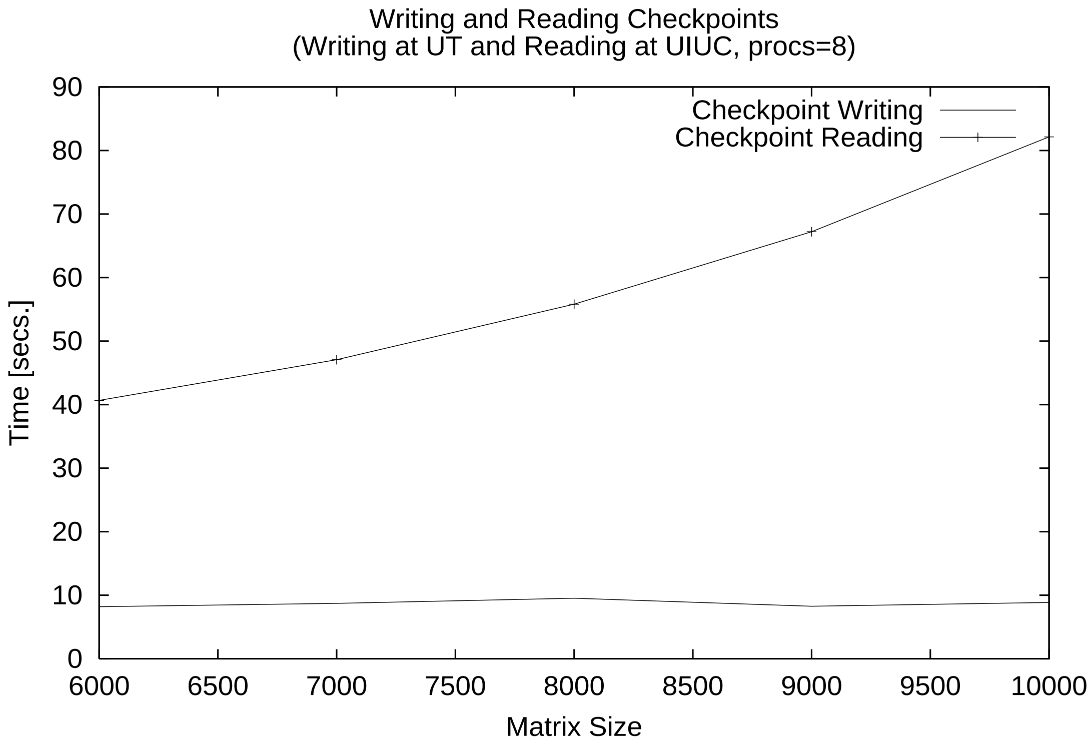

Figure 5.13: Times for Checkpoint Writing and Reading when the application was restarted on *opus* machines

5.3.3 SRS for Malleable Applications

In the experiments in this section, the application was started on 8 *msc* machines and restarted on a different number of machines spanning UT and UIUC. In this case, the restarted application, through the SRS library, determines the new data maps for the processors and redistributed the stored checkpoint data among the processors. The RSS daemon was started on UT.

In Figure 5.14, results are shown when the ScaLAPACK QR application corresponding to matrix size 8000 was restarted on different number of processors (3 *opus* machines - 8 *opus* + 2 *msc* machines). The size of a single stored checkpoint was 64 MBytes. The time for data redistribution depends on the number and size of the data blocks that are communicated during redistribution and the network characteristics of the machines between which the data are transferred. When the application is restarted on a smaller number of processors, the size of the data blocks are large and hence the redistribution time is large. For larger number of processors, the redistribution time decreases due to the reduced size of data blocks communicated between the processors.

Figure 5.15 shows the dependence of the redistribution times on the problem size. For this experiment, the application was initially started on 8 *msc* machines and restarted on 8 *opus* and 2 *msc* machines.

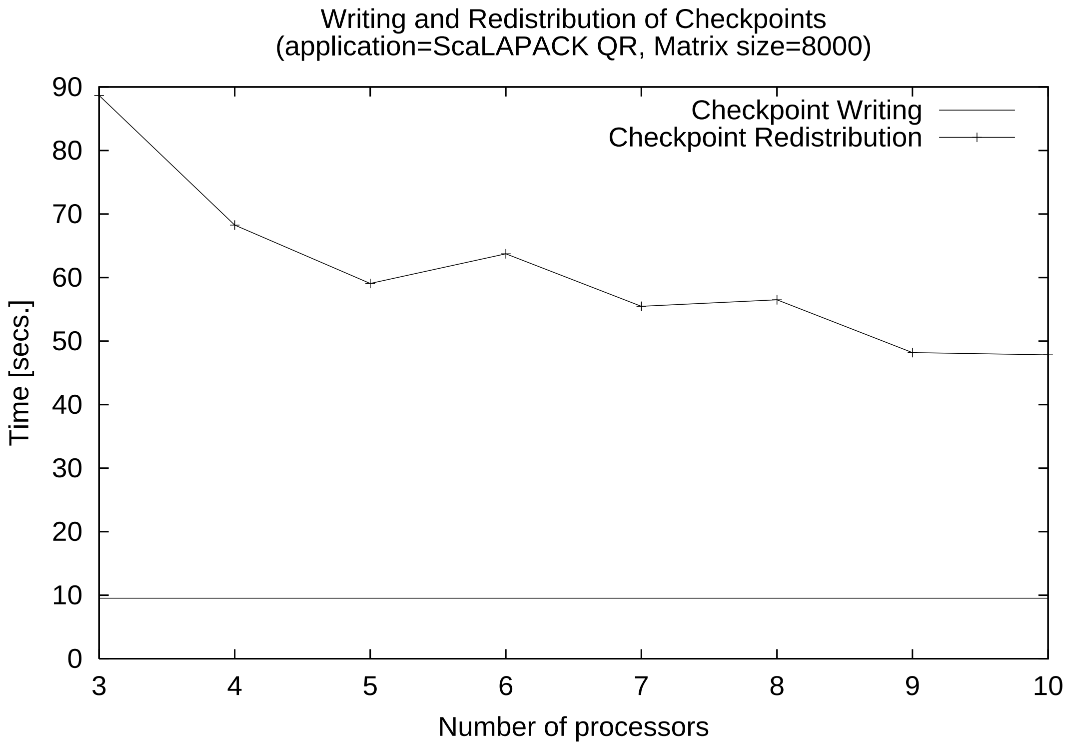

Figure 5.14: Times for Checkpoint Writing and Redistribution when the application was restarted on different number of processors

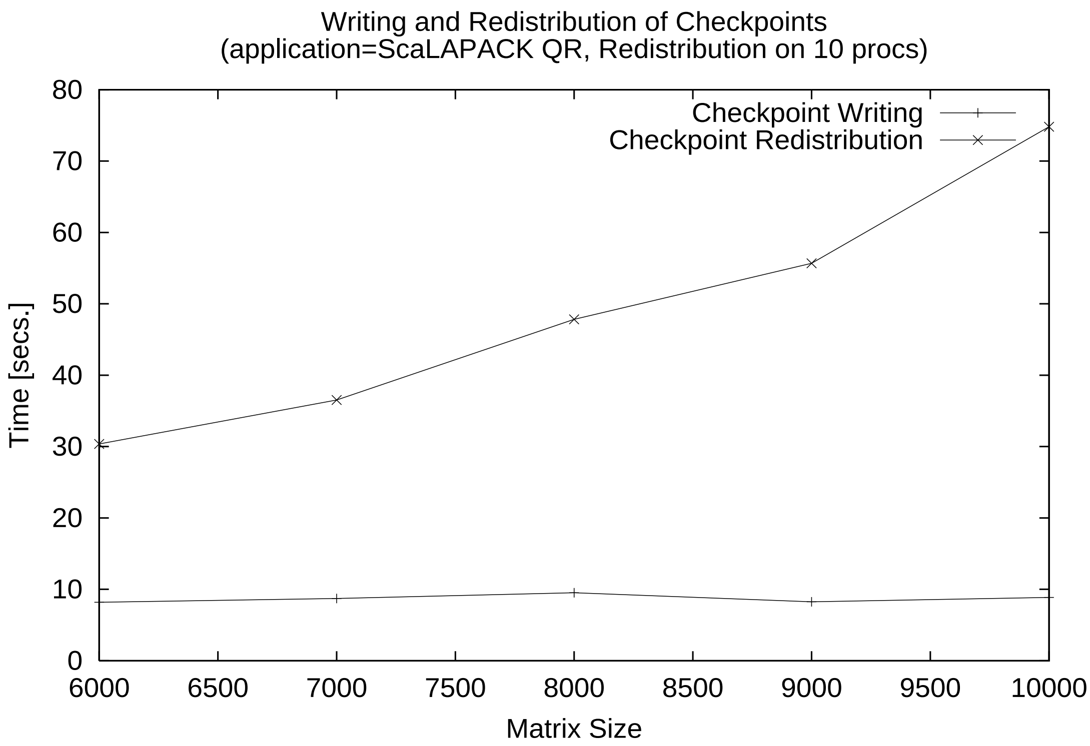

Figure 5.15: Times for Checkpoint Writing and Redistribution for different problem sizes

5.4 Practical Experiments

For the experiments in this section, 5 iterative applications were integrated into the GrADS framework - ScaLAPACK LU and QR factorizations, ScaLAPACK eigenvalue problem, PETSC [20, 19, 18] Conjugate Gradient (CG) application and heat equation application using finite difference stencil method. The integration involved developing execution models for the Performance Modeler and invoking SRS calls from the applications for rescheduling.

50 problems were submitted to the GrADS system with different arrival rates. Poisson distributions with different mean interval times in minutes were used for job submissions. Uniform distributions were used for the type of the applications corresponding to the problems and the problem sizes. At the end of the problem runs, different statistics including the total throughput of the system, the number of jobs rejected by the metascheduler, the mean response times of the different kinds of jobs, the number of instances of the different kinds of metascheduling decisions etc. were collected.

5.4.1 Comparison with Plain Application-level Scheduling

Two sets of experiments were conducted for different mean inter-arrival times. In the first set of experiments, 50 GrADS applications were executed in the presence of the metascheduling components for different mean inter-arrival times and different statistics were collected. In the second set of experiments, the same 50 GrADS applications with the same inter-arrival times were executed in the absence of the metascheduler.

The metascheduler was disabled by disabling the Contract Negotiator and the Rescheduler. The Contract Developers corresponding to the GrADS applications, instead of contacting the Contract Negotiator, approved the contracts passed by the Application Managers. Also, the Permission Service rejected permission for the GrADS applications if the resources were not sufficient for the end applications. It did not try to stop executing applications to accommodate new applications.

Figure 5.16 shows the number of applications that were rejected permission in the presence and absence of the metascheduler. For mean inter-arrival times 2, 4 and 6, the number of rejected applications are much higher when the metascheduler was not used. This is because, few applications that were submitted to the system at about the same time occupied the same set of resources and hence prevented the accommodation of the applications that arrived later in the system. Also, when the metascheduler was enabled, the metascheduler tried to accommodate new applications by stopping executing applications. For mean inter-arrival times 1, 8 and 10, though few applications that were submitted to the system at about the same time claimed the same set of resources, the jobs that arrived later into the system had small resource requirements and were able to be accommodated into the system. The metascheduler adopted a conservative approach and rejected few of the applications that were submitted simultaneously. Hence when the mean inter-arrival times were 1, 8 and 10, more number of applications were rejected permission when the metascheduler was used.

Figure 5.17 shows the total times in minutes, taken for all the 50 applications in

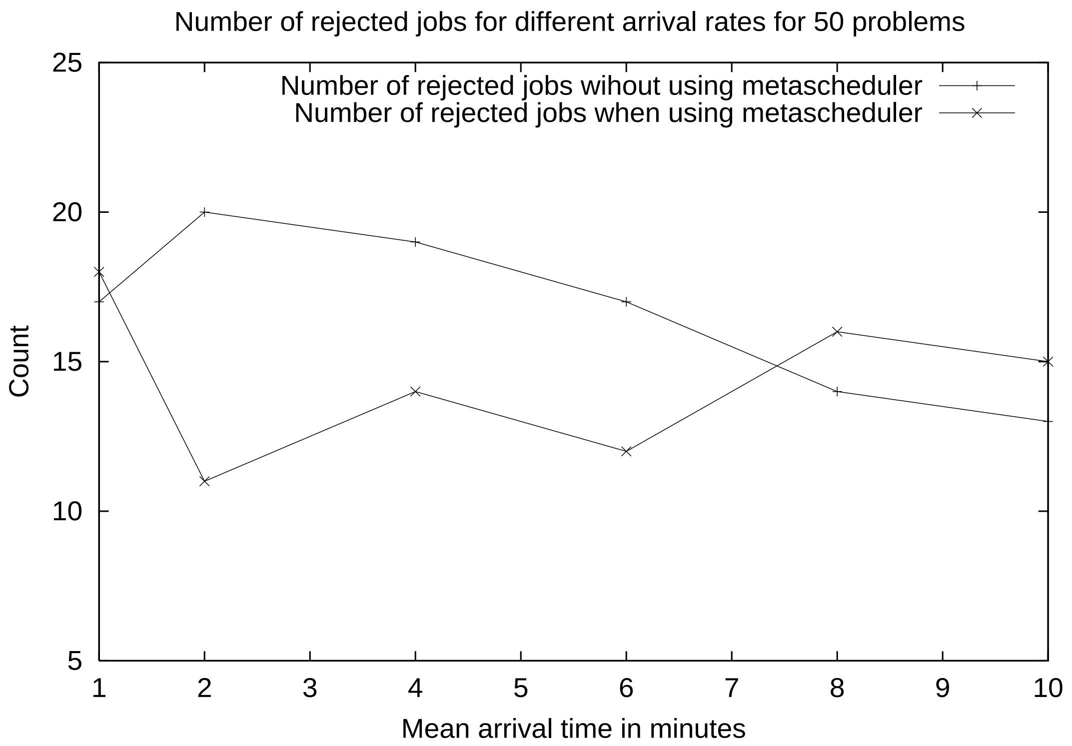

Figure 5.16: Number of rejected applications in the presence and absence of metascheduler

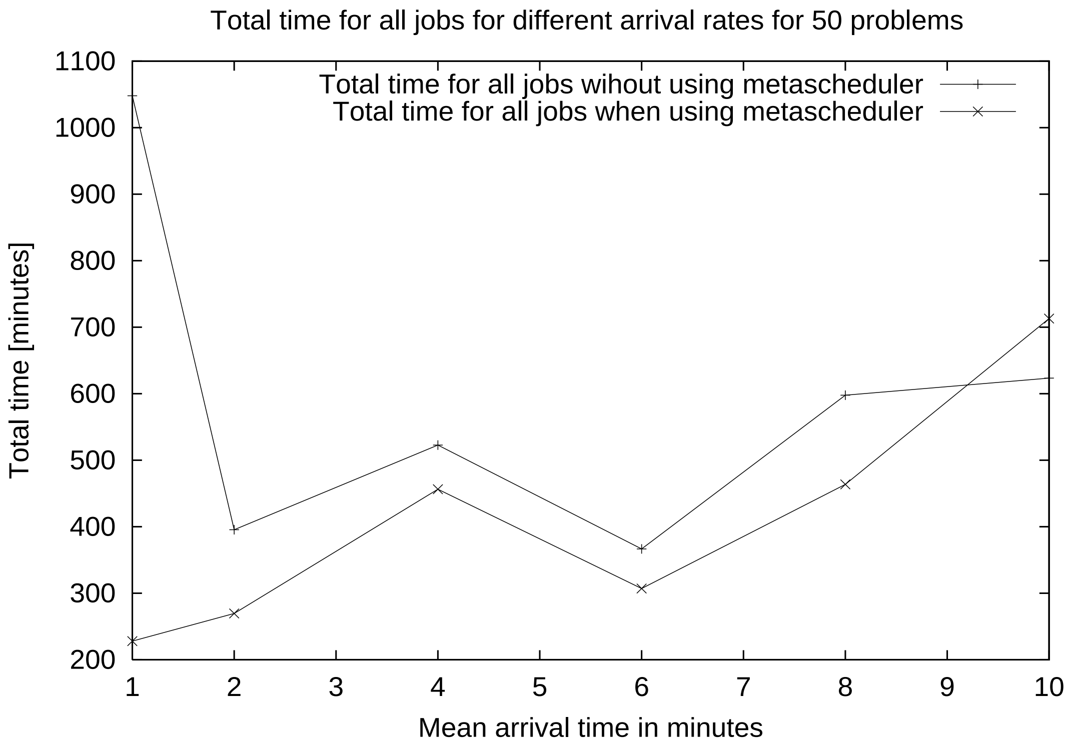

Figure 5.17: Total times of all applications with and without the metascheduler

the presence and absence of the metascheduler. We observe that the presence of the metascheduler facilitates the faster completion of the jobs even for cases when more applications were accommodated into the system. This is because, in the absence of the metascheduler, large applications that were submitted to the system at about the same time occupied the same set of resources. Hence, these applications had to frequently access the disks, thus significantly increasing the response times.

Figures 5.18 and 5.19 show the extent of contract violations with different mean inter-arrival times in the presence and absence of metascheduler.

Contract violation is defined as an event when the ratio between the measured and

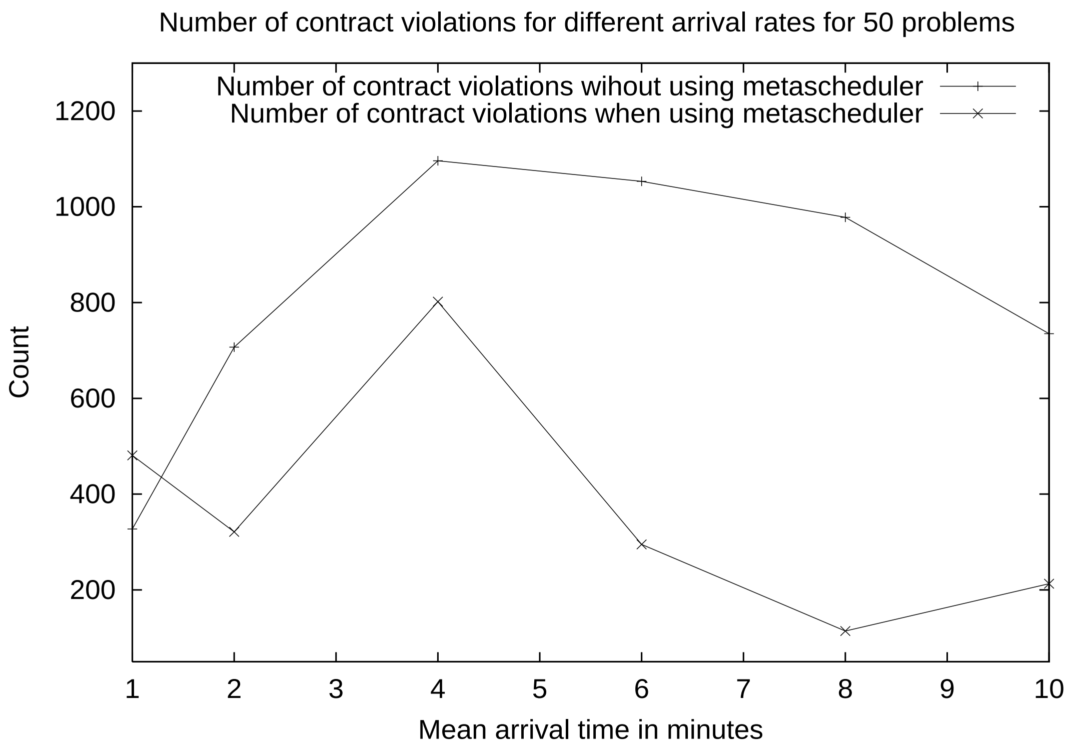

Figure 5.18: Number of contract violations with and without the metascheduler

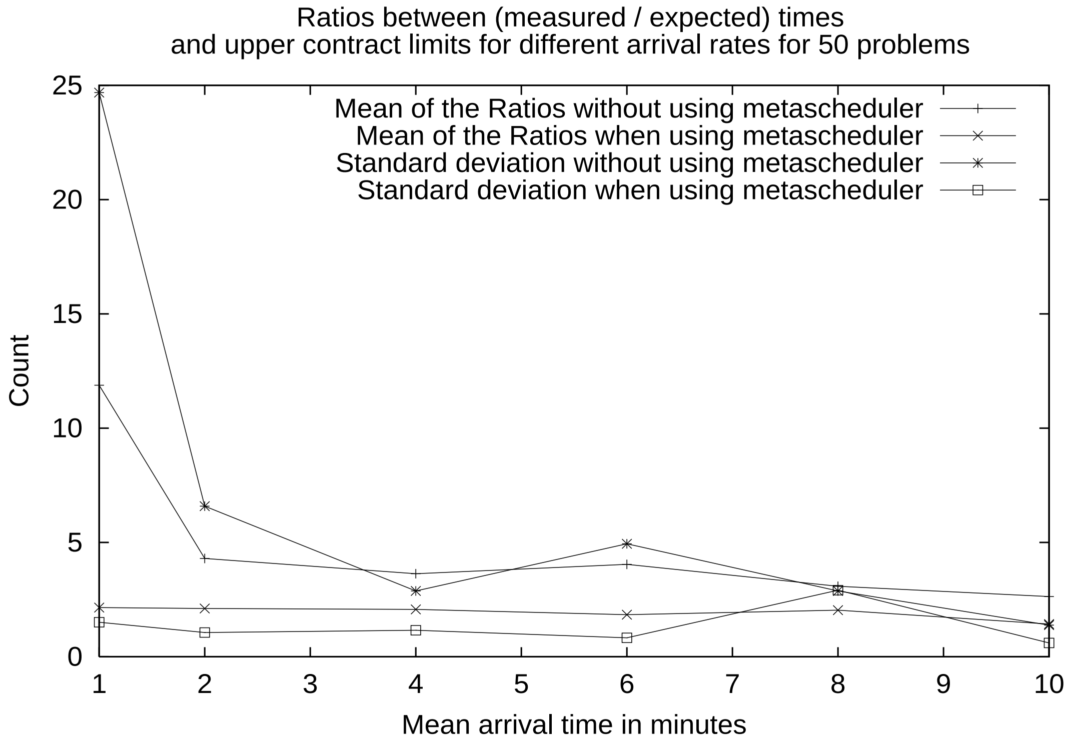

Figure 5.19: Extent of contract violations with and without the metascheduler

the expected costs of the application is greater than the specified tolerance limit. Figure 5.18 shows the number of contract violations for different mean inter-arrival times. Figure 5.19 takes into account the ratios between the ratios of the measured and expected costs of the applications and the upper tolerance limits of the ratios. We find that in most of the cases, the number of contract violations and the mean and standard deviation of the contract violation ratios are much higher when the metascheduler was not used. The small extents of the contract violations, when the metascheduler was used, are achieved by the rescheduling of the executing applications for which contract violations are noticed and dynamically adjusting the upper tolerance limits by the rescheduling framework. Thus for applications, for which performance guarantees are desired, the use of the metascheduler is advisable.

5.4.2 Behavior of the Metascheduler

Figure 5.20 shows the behavior of the metascheduler for different job submission rates. The number of jobs accommodated into the system by the Permission Service, the number of times the Contract Negotiator act as queue manager allowing few applications to execute while prompting other applications to retrieve new resource information and the number of times different metascheduling components stopped executing applications are shown in the figure. The x-axis represents the mean inter-arrival times of the Poisson distribution in minutes.

We find that the number of applications accommodated into the system by the Permission Service depends primarily on the job sizes and independent of the mean arrival

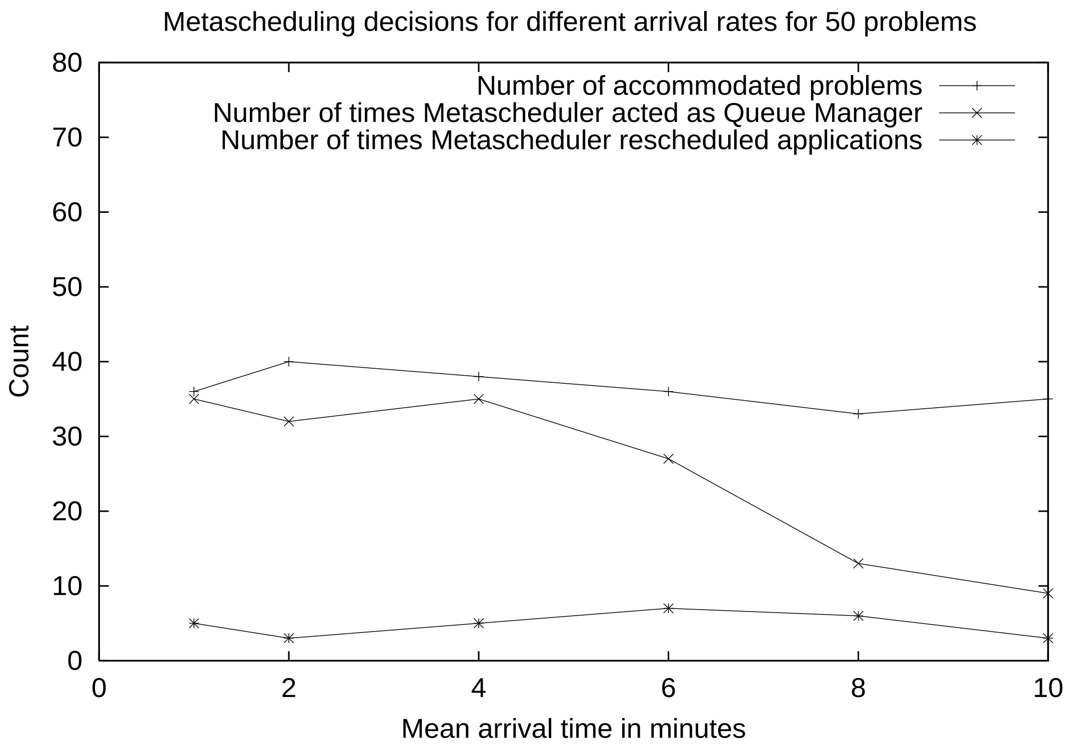

Figure 5.20: Different kinds of metascheduling decisions based on the amount of contention

times of the jobs. Similarly, the number of rescheduling decisions by the Metascheduler depends on the job mix and the contention for the resources by the large sized jobs and is independent of the mean inter arrival times. The number of times the Contract Negotiator acts as queue manager depends on the mean inter-arrival times of the jobs. Smaller the time difference between the submission of the jobs, greater the number of times the Contract Negotiator rejects the contracts of the application, thus prompting the applications to restart from the resource selection phase. We also find that the number is almost constant for inter-arrival times of 1, 2 and 4 minutes. This is due to the constant overhead associated with the GrADS applications and the metascheduling.

The graph in Figure 5.21 shows the mean response times of all the jobs, the accommodated jobs and the rejected jobs. We find that the mean response times decrease with the increase in mean inter-arrival times. This is due to the reduction in the contention among the jobs for large mean inter-arrival times. The mean response times for inter-arrival times of 1, 2 and 4 minutes are almost constant due to the constant overhead associated with the GrADS applications and the metascheduler. The mean response times for rejected jobs is about 2 hours for small mean inter-arrival times. This is due to the longer durations spent by all the jobs in the various metascheduling queues. Our future metascheduling systems will give higher priority to jobs that can be potentially rejected and respond sooner than 2 hours for such jobs.

Figures 5.22 and 5.23 show the extent of contract violations with different mean inter-arrival times. Figure 5.22 shows the number of contract violations for different

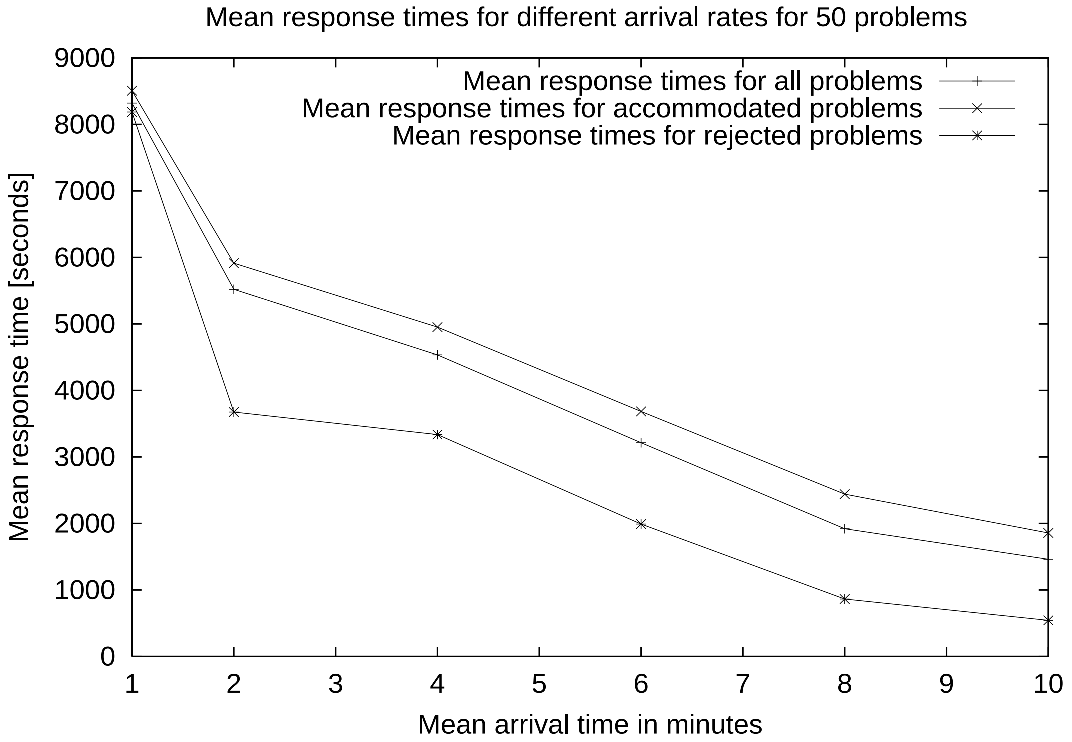

Figure 5.21: Mean response times of the jobs for different mean inter-arrival times

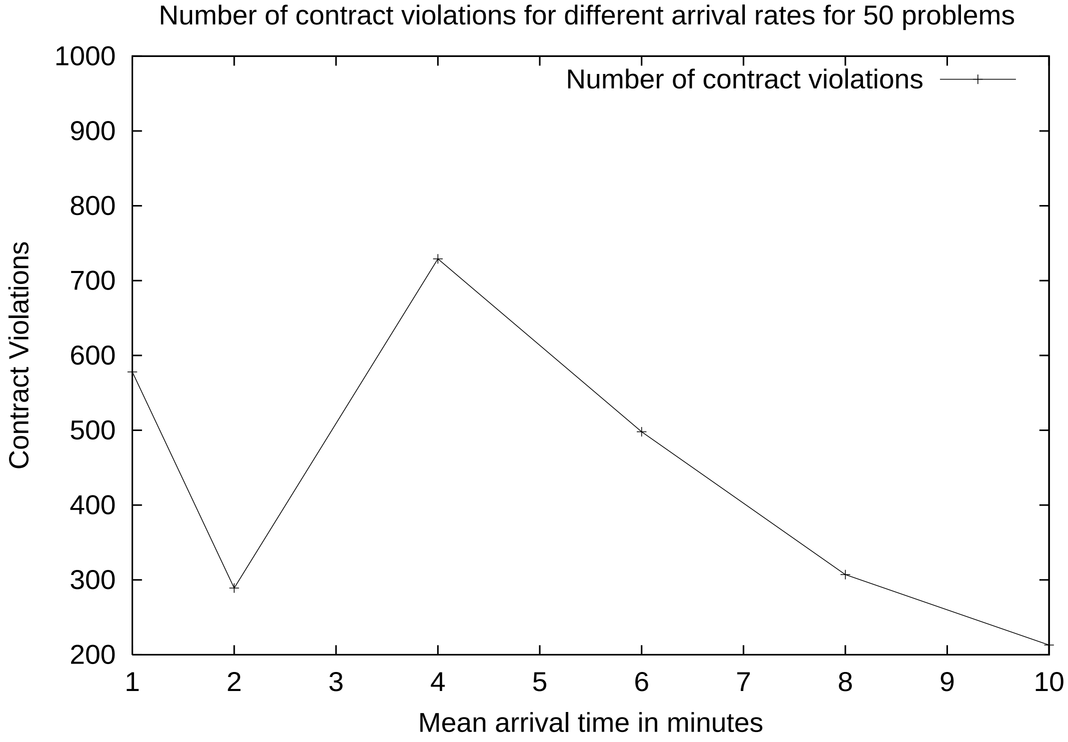

Figure 5.22: Number of contract violations

mean inter-arrival times. We find that the number of contract violations primarily depends on the job sizes of the different jobs and the contention for the resources by the jobs and is independent of the mean inter-arrival times. Figure 5.23 takes into account the ratios between the ratios of the measured and expected costs of the applications and the upper tolerance limits of the ratios. We find that the mean of the ratios is small and lie in the range of 2-3 and the standard deviation of the ratios are small and are about 1.5.

Following is the summary of the conclusions from the practical experiments in this section.

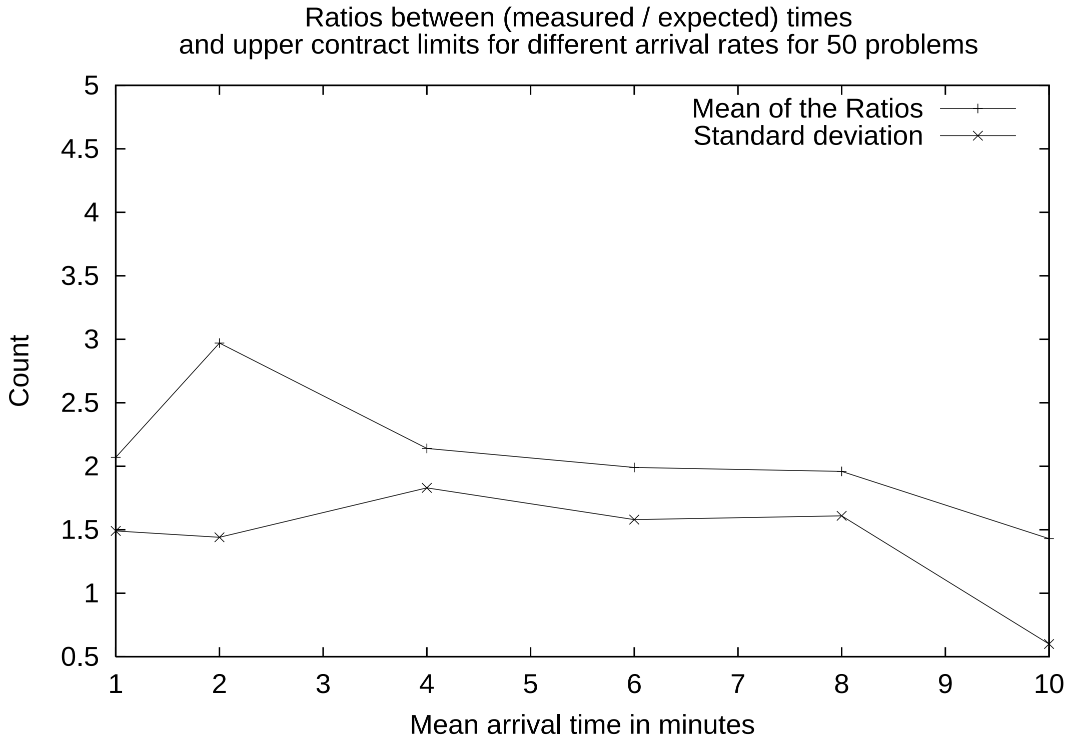

Figure 5.23: Extent of contract violations

1. The metascheduler is particularly useful in situations when large number of large applications are submitted to the system at about the same time.

2. In general, the use of the metascheduler facilitates the accommodation of more number of jobs.

3. The metascheduler is definitely useful when performance guarantees need to be met for the applications.

4. When the job mix and the job arrival rates are known and there are large gaps between the submissions of the large jobs, plain application-level scheduling without metascheduling is sufficient.

5. The number of metascheduling decisions tend to reduce as the mean inter-arrival times of the jobs increase.

6. The metascheduling strategies have to be enhanced to improve the mean response times of the rejected jobs.

7. The extent of the performance contract violations are almost constant and independent of the mean inter-arrival times of the jobs, when the metascheduler was used.

Chapter 6

GrADSolve - A Metascheduling-Based Distributed Computing System

This chapter includes lightly revised sections of the following papers.

Sathish S. Vadhiyar and Jack J. Dongarra. GrADSolve - A Grid-based RPC system for Remote Invocation of Parallel Software. Submitted to Journal of Parallel and Distributed Computing, 2003.

Sathish S. Vadhiyar and Jack J. Dongarra. GrADSolve - RPC for High Performance Computing on the Grid". Submitted to EuroPar 2003.

I was the primary contributor of the papers and was involved in the design and implementation of the frameworks and verification by experiments. This chapter revises the papers by providing a more detailed descriptions of the sections in the papers.

In the previous chapters, the metascheduling architecture was explained and the usefulness of metascheduler was demonstrated by means of experiments. The experiments and the results for the metascheduler shown in Chapter 5 were obtained using ad-hoc implementations of the the metascheduler, the Grid Application Manager and the interactions between them. Though the ad-hoc implementations are powerful for conducting demonstrative and practical experiments, they present various obstacles in the seamless use of the Grid components both by the service providers and the service consumers. The ad-hoc implementations of the system components are not easy to use due to the following reasons:

1. **Lack of distinction between service providers and consumers.**

   The users, while trying to solve a problem using the ad-hoc components specifies the location of the Performance Modeler and the Contract Monitor in an input configuration file. These components are application-specific and are generated by the application library writers. Hence the binaries corresponding to these components need to be communicated from the library writers to the users. This leads to large consumption of storage resources and results in poor scalability of the Grid system in the sense that more the number of applications that the users intends to solve over the Grid, more the amount of storage is required by the user. The user also specifies the location of the end applications in all the resources of the Grid system. Due to the absence of an Information Repository (IR) in the ad-hoc infrastructure, the user needs to retrieve this information explicitly from the library writers. Also, due to the absence of error checking mechanisms in the ad-hoc infrastructure, the user needs to retrieve the input and output specifications for the end applications from the library writers.

2. **Absence of programming constructs for the users.**

   In most cases, users desire to use remote services with minimal changes to their application codes. These application codes are written in conventional programming languages like C, Fortran, Matlab etc. Hence it is desirable to have a layer in the Grid architecture that translates the programming language function calls into corresponding Grid requests. In the ad-hoc implementation, such a layer is absent resulting in the expression of service requests by means of cumbersome command-line arguments and configuration file. Also, service requests by means of a programming language provides greater flexibility to the user in terms of input and output data management. The users will also be able to harness other powerful native features provided by the programming language.

3. **Absence of well-defined interfaces for the service providers.**

   In the ad-hoc infrastructure, the application library writers perform several difficult steps before uploading their applications to the Grid resources for use by the end users. This is due to the absence of easy-to-use interfaces for the library writers in the ad-hoc infrastructure. First, the library writers physically upload their applications to all the resources on the Grid. In most cases, the information about the Grid resources are not available to the library writers. Secondly, due to the absence of generic templates, the library writers have to be extra careful while writing the execution models for the applications. These execution models depend on the internal structures in the infrastructure and due to the absence of error checking mechanisms in the infrastructure, there are no means to validate the prototypes used by the library writers for the execution models. Also, the library writer replicates most of the work from the execution model in the Contract Monitor for monitoring the application. Transparent template generation and remote storage mechanisms can mitigate most of the problems faced by the library writer in the ad-hoc infrastructure.

4. **Lack of support for user data management.**

   In the ad-hoc infrastructure, the data needed by the end applications are generated within the application itself. The end users have control over only the parameters of the date including the data size, block size etc. A more realistic scenario is for the applications to operate on the data passed by the end-users. Also, the users by means of programming constructs may vary few data for the end applications or may use the same data while invoking different remote services. For these reasons, powerful and transparent data movement and data handling mechanisms are necessary in the Grid infrastructure. When the applications involved are parallel applications and the end-users operate in sequential environments, data have to be partitioned among the different remote resources used for the execution of the parallel application. The data partitioning should be performed corresponding to the data distribution used by the application. For the transparent movement of data from the user to the remote resources, matching the user's data with the different parameters needed by the end application and for checking error conditions due to mismatch between the size and the type passed by the user and the data types and sizes needed by the application, an Interface Definition Language (IDL) is necessary for the application writers to convey information about their applications to the Grid system. The IDL mechanism can also be used for advertisement purposes. Performance model wrappers also need to be developed for conveying the information about the data distribution used by the application.

5. **Rigidity in terms of the capabilities of the applications.**

   The ad-hoc infrastructure assumes the existence of end applications that can be stopped and continued from previous execution on possibly different number of processors. Though the SRS library allows the applications to possess such capabilities, there can be certain kinds of applications that cannot be stopped and continued from previous executions. In these cases, the applications have to be restarted from the beginning of the execution. The metascheduling policies have to be modified while dealing with such applications such that the cost incurred due to restarting the applications from the beginning of the execution are taken into account while making rescheduling decisions. Also, some applications may be able to continue from the previous point in execution once stopped, but may not be able to continue on a different number of processors. The GrADS Application Manager and the Performance Modeler have to be modified so that such applications are restarted on the same number of processors.

Also, the ad-hoc framework depends excessively on the application-level schedules generated for the end applications. These application-level schedules are generated based on the execution models written by the library writers for the applications. In some cases, the library writers may not possess enough information to predict the execution cost of the end applications and to write the execution models for the applications. In these cases, default scheduling strategies that are independent of execution model of the end applications need to be employed in the Application Manager and the metascheduler. In some cases, the library writers may have provided execution models for the applications but may have failed to provide data distribution information for some of the critical data used by the application. In these situations, the Grid framework should be able to employ robust default data distribution strategies for handling most of the common data distributions used in the applications.

Finally, the ad-hoc framework deals with only parallel end applications while it is a desirable property of the Grid framework to deal with both sequential and parallel applications. In general, the Grid framework must be flexible to deal with the above mentioned different capabilities of the end applications. While some properties of the applications like the ability to continue and reconfigure from the previous point in execution are desirable for the metascheduling strategies, the Grid framework need not mandate the existence of these properties in the applications. More the number of capabilities of the applications and more the information passed to the system regarding the capabilities, more the robustness and service performance the system will be able to provide for the individual service requests.

The above mentioned obstacles have to be removed for the ease of use of the Grid framework - one of the important goals for Grid computing systems [58]. In this chapter, a flexible Grid computing framework based on the metascheduling strategies discussed in the previous sections is explained. The system is called *GrADSolve* since it combines the easy-to-use interfaces of the NetSolve system [34] and the powerful GrADS scheduling strategies employed by the metascheduler. The system is flexible since it allows the library writers to express different capabilities of the applications and possess mechanisms to deal with the different capabilities. GrADSolve also possess simple and powerful Interface Definition (IDL) mechanisms for the library writers to convey information about the input and output parameters used by the application. GrADSolve supports users programs written in C to invoke Grid services. It also supports the notion of separate domains for the end users and the library writers. It also provides powerful mechanisms for partitioning the user's data among different remote resources used for problem solving.

In addition to the above capabilities, GrADSolve also investigates the concepts and provides fundamental framework for maintaining and using *execution traces*. In many cases, it is desirable for users to execute their applications on a set of Grid resources and replicate the execution at a later point of time. This is extremely useful for testing the applications over the Grid framework and in projects where large number of collaborators are involved. GrADSolve provides a framework whereby the user can request the system to maintain the trace for the current execution. The user can then execute his Grid program at a later point of time with a system-returned key. GrADSolve bypasses the scheduling and data distribution phases that it used for the initial execution of the application. Instead, it executes the application with the same processors used in the trace and with the same data distributions. The ability to maintain traces and data partitioning strategies are unique features of the GrADSolve system.

In th next section, the related work in the field of RPC systems for invoking remote parallel applications is presented. The overview of NetSolve system is discussed in the Section 6.2 and the disadvantages of the NetSolve framework are highlighted. The overview of the GrADSolve system is explained in Section 6.3 and compared with the NetSolve system. The enhanced features in GrADSolve are described and its advantages over the NetSolve system are elucidated. The various entities in the GrADSolve system and the support for the entities in the GradSolve system are explained in Section 6.4. Section 6.4 also deals with the detailed description of the framework of the GrADSolve system. One of the unique features of the GrADSolve is the ability to store, maintain and use *execution traces* for problem runs. The support in the GrADSolve system for execution traces is explained in Section 6.5. The changes needed in the metascheduler of the ad-hoc infrastructure to support the flexibility provided to the end applications by the GrADSolve system is dealt in Section 6.6. In Section 6.7, the experiments conducted in GrADSolve are explained and results are presented to demonstrate the usefulness of the data staging mechanisms and execution traces in GrADSolve.

6.1 Related Work

Few RPC systems contain mechanisms for the execution of remote parallel software.

MRPC [38] is a RPC system tuned for providing high performance for MPMD applications on homogeneous clusters. The RPC communications are implemented on top of Active Messages (AM) [109] and the user's client programs are written in Compositional C++ (CC++). The work by Maassen et. al [76] extends Java RMI [5] for efficient communications in solving high performance computing problems. Both MRPC [38] and the Java RMI extension [76] require the end user's programs to be parallel programs.

NetSolve [34], Ninf [96], RCS [17] and DFN-RPC [87] support task parallelism by the asynchronous execution of number of remote sequential applications. OmniRPC [97] is an extension of Ninf and supports asynchronous RPC calls to be made from OpenMP programs. But similar to the approaches in NetSolve, Ninf, RCS and DFN-RPC, OmniRPC supports only master-worker models of parallelism. NetSolve, Ninf and RCS also support remote invocation of MPI applications, but the amount of parallelism and the locations of the resources to be used for the execution are fixed at the time when the applications are uploaded to the systems and hence are not adaptive to dynamic loads in the Grid environments.

The efforts that are very closely related to GrADSolve are PaCO [91, 92] and PaCO++ [44, 43] from the PARIS project in France. The PaCO systems are implemented within the CORBA [2] framework to encapsulate MPI applications in RPC systems. The data distribution and redistribution mechanisms in PaCO are much more robust than in GrADSolve and support invocation of remote parallel applications either from sequential or parallel client programs. Recently, the PARIS project has been investigating coupling multiple applications of different types in Grid frameworks [45, 85]. Although the PARIS project aims to improve the performance of CORBA for high performance computing, the RPC mechanisms provided in CORBA by the use of client stubs and server skeletons have not found to be favorable for high performance computing according to a previous study [103]. Also, the PaCO projects do not support dynamic selection of resources for application execution as in GrADSolve. Also, GrADSolve supports Grid related security models by employing Globus mechanisms. And finally, GrADSolve is unique in maintaining execution traces that can help bypass the resource selection and data staging phases.

6.2 NetSolve - A Brief Overview

NetSolve [34] is a Grid computing system developed at University of Tennessee. It is a Remote Procedure Call (RPC) based system used for solving numerical applications over remote machines. The NetSolve system consists of 3 main components - agent, server and client. The working of the NetSolve system is illustrated in Figure 6.1.

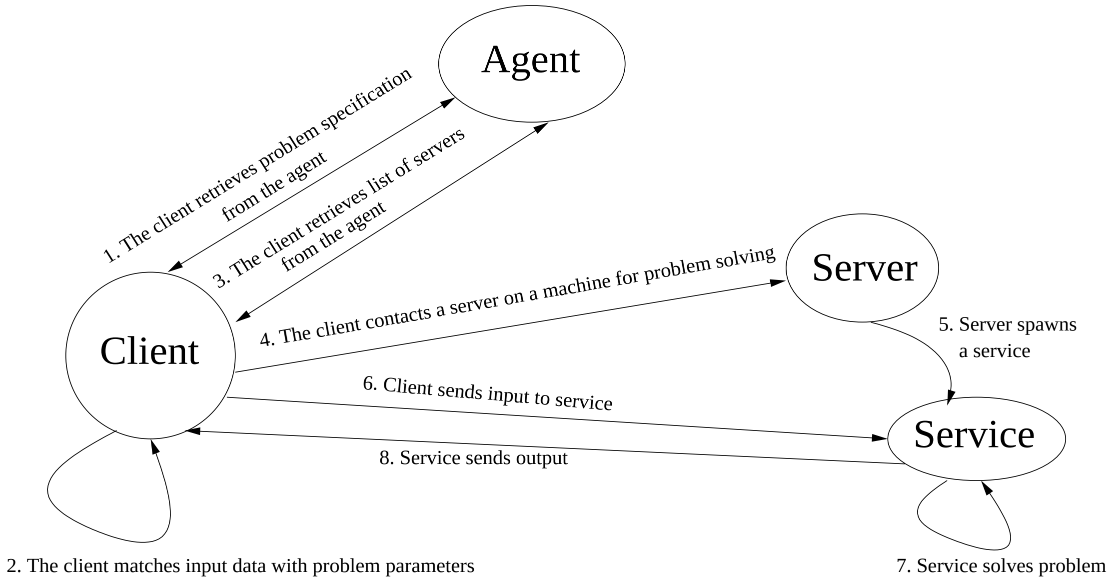

Figure 6.1: Overview of NetSolve system

The library writers upload their applications into the NetSolve system by writing a Problem Description File (PDF). The PDF for the application is translated into a wrapper and compiled into a server stub. The server containing different applications are started as server daemons. When the server daemons are started the problem descriptions of the different applications in the servers are sent to the agent and stored in an internal database maintained by the agent. Thus the agent maintains global information about all the servers and the applications supported by the servers.

The end users solve their numerical problems remotely over NetSolve servers by writing client codes. The user client codes can be written in C, Fortran or Matlab languages. The client code invokes a function call provided by the NetSolve API. The function call specifies the name of the remote application and the input and the output data needed by the remote application. The client contacts the agent and retrieves the problem description of the application. The client, after performing error checking procedures, matches the user's data with the different parameters needed by the application. The client then contacts the agent to obtain a list of servers that can solve the problem. The agent periodically receives workload of the different servers from the servers. Based on the computational capacities and the workloads of the servers, the agent prepares a list of servers sorted by the computational capacities and workloads. The servers with maximum computational capacities and minimal workloads are placed at the top of the list. Thus the agent performs dual-roles of maintaining information about the various components and scheduling the servers for problem solving.

After retrieving the list of servers from the agent, the client contacts the first server in the list. The server daemon spawns a service specific for the application run and passes the location of the service to the client. The client contacts the service and passes the input data for the application. The service, then solves the problem with the input data and passes the output back to the client.

NetSolve system is mostly suitable for sequential applications. Though NetSolve supports remote execution of parallel applications, the amount of parallelism is fixed at the time the server daemons are started. Also, the format of the Problem Description Files (PDFs) are cumbersome to be written by the library writers. These problems are rectified and more enhancements are added in the GrADSolve system.

6.3 The GrADSolve System

At the core of the GrADSolve system is a XML database implemented with Apache Xindice [1]. Since XML is mostly useful for storing metadata and transferring compatible documents across the network, GrADSolve uses XML as a language for storing information about different Grid entities. This database maintains three kinds of tables - *users*, *resources* and *applications*. The *users* table contains information about the different users of the Grid system, namely the home directories of the users on different resources. The *resources* table contains information about the different machines in the Grid, namely the name of the machines, the clusters to which the machines belong, the architecture and the operating system in the machines, the peak performance of the machines etc. The *applications* table contains information about different applications, namely the name and owner of the application, if the application is sequential or parallel, the language in which the application is written, if the application can continue from the previous point in application once stopped, if the application is reconfigurable once stopped, the number of input and output arguments, the data type and size of each arguments, the location of the binaries of the applications on each of the resources etc. All the above mentioned information are stored in the XML database in the form of XML documents. The Xindice implementation of the XML-RPC standard [12] was used for storing and retrieving information to and from the XML database.

Of the metascheduling components, the DataBase Manager is implemented with the popular PostgreSQL [10] Database mechanisms. Apart from the normal database storage and retrieval mechanisms, PostgreSQL also provides *triggers* for execution of certain procedures on the occurrence of certain events and event notification capabilities. Event notification mechanisms is one of the important capabilities needed by the Database Manager in the metascheduler. When the PostgreSQL Database Manager is initialized, a table for storing information related to different problem runs is created. As new GrADS applications are executed in the system, entries for the applications are created in the table and different information regarding the application, including the problem name, user name, the problem status, capabilities of the problem, the resource information of the resources used in the application-level scheduling, the final application-level schedule, the locations of the Contract Monitor, the Performance Modeler and the Runtime Support System (RSS), the ratios between the measured and the predicted performance cost for the end application, the number of contract violations, the information about the execution trace if the application is executed in the trace mode etc., are stored in the table as the applications pass through different phases of the GrADS execution.

A number of PostgreSQL triggers are implemented for recording certain entries in the table on the occurrence of certain events. A PostgreSQL trigger is implemented such that the times corresponding to the Resource Selection, start of the end application and completion of the application, and the accumulated percentage completion time are recorded internally by the Database Manager as the GrADS application generates the corresponding events. Another trigger is employed to implement policies for deleting some of the records in the table as new entries are inserted. Another trigger is used to record the last usage of a stored execution trace. Also, the PostgreSQL event notification mechanism is used such that components including GrADS Application Manager and metascheduling components are notified on the occurrence of certain events, for example, when the status of an application is changed.

Other metascheduling components, namely the Permission Service, the Contract Negotiator and the Rescheduler perform PostgreSQL queries against the Database Manager. The XML database and the metascheduling components together correspond to the agent in the NetSolve system where the XML database forms the database part of the agent while the metascheduling components form the scheduling layer of the agent. The general functioning of the GrADSolve system is illustrated by Figure 6.2.

Unlike the NetSolve system, there are no server daemons in GrADSolve. The library writer uploads his application into the Grid system specifying the problem description of the application using an Interface Definition Language (IDL). The GrADSolve system creates a wrapper for the application, compiles the wrapper along with the application and transports the executable application to the different resources of the Grid system using the Globus GridFTP mechanisms. The library writer also has the option of adding an execution model for the application. The information regarding the locations of the end applications on the resources are stored in the Xindice XML database. The end user writes a NetSolve-like client program to execute applications over the Grid. The GrADSolve client accesses the XML database, retrieves the problem specification

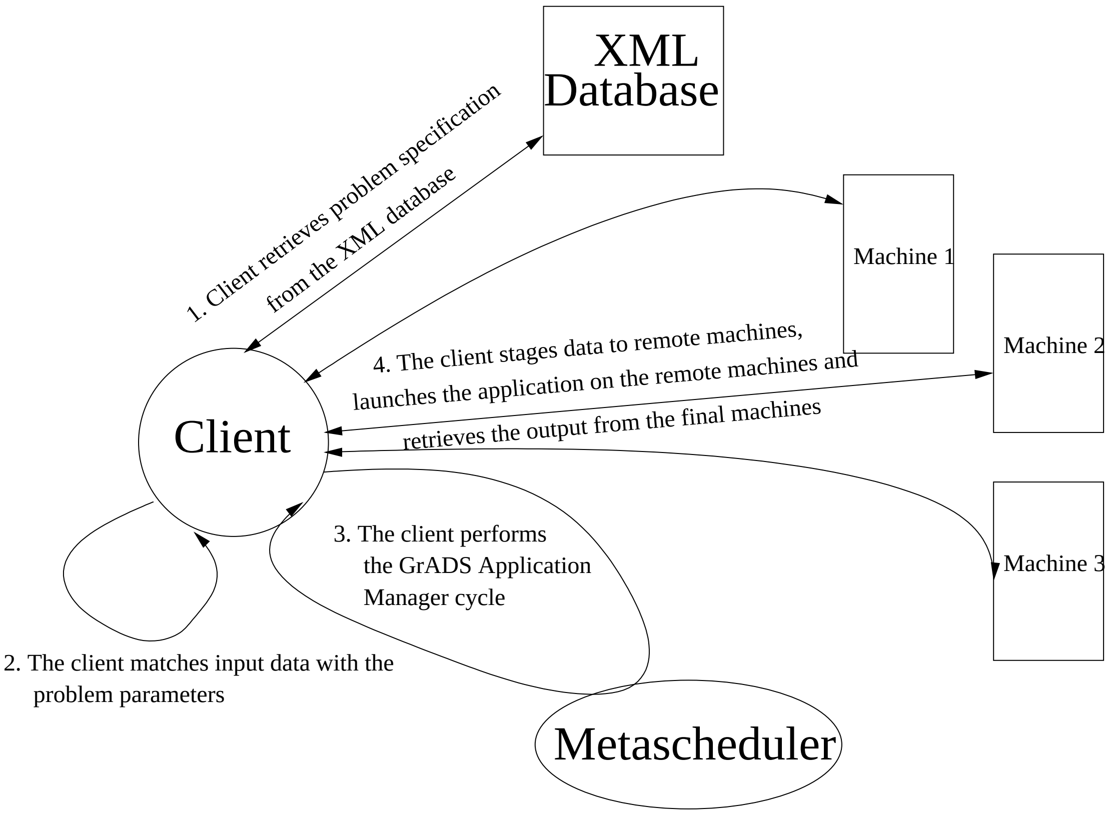

Figure 6.2: Overview of GrADSolve system

for the application and matches the user's data with the parameters of the problem. The GrADSolve client then passes through different stages of the GrADS application manager including the Resource Selection, Performance Modeling and Contract Development. For the Performance Modeling, the client retrieves the execution model if the application possesses an execution model. After determining the final application-level schedule, the GrADSolve client partitions the user's input data and stages the appropriate blocks of data to the different resources using the Globus GridFTP mechanisms. The client then spawns the application on the set of resources using MPICH-G [56]. Similar to the staging of the input data, the client gathers the different blocks of output data from different resources using GridFTP and copies the data to the user's memory. During the different stages, the GrADSolve client stores different information to the Database Manager. The GrADSolve client also interacts with the metascheduling components to validate the performance contracts.

6.4 GrADSolve Entities

There are three human entities involved in GrADSolve - *administrators*, *library writers* and *end users*. The role of these entities in GrADSolve and the functions performed by the GrADSolve system for these entities are explained in the following sub sections.

6.4.1 Administrators

The GrADSolve administrator is responsible for managing the users and resources of the GrADSolve system. The administrator initializes the XML database and starts the various metascheduling components - namely, the Database Manager, the Permission Service, the Contract Negotiator and the Rescheduler. During the initialization of the Database Manager, the administrator also creates various database triggers and notification events. The administrator also creates entities for different users in the XML database by specifying a *user configuration file*. The user configuration file contains information for different users, namely the user account names for different resources and the location of the home directories on different resources in the GrADSolve system. These information are translated into XML documents and stored in the *users* table of the Xindice database. Finally, the administrator creates the *resources* table in the Xindice database and adds entries for different resources in the GrADSolve system by specifying a *resource configuration file*. The various information in the configuration file, namely the names of the different machines, their computational capacities, the number of processors in the machines and other machine specifications, are stored as XML documents. The translation of the configuration files into XML documents are automatically handled by the GrADSolve system.

6.4.2 Library Writers

The library writer uploads his application into the GrADSolve system by specifying an Interface Definition Language (IDL) file for the application. The Backus Normal Form (BNF) of the GrADSolve IDL is given in Figure 6.3.

In the IDL file, the library writer specifies the name of the problem suite and the description of the problem suite. A problem suite consists of a set of functions that the user can invoke remotely. For each function, the library writer specifies in the IDL file, the programming language in which the function is written, the name of the function, the set of input and output arguments in the function, the description of the function, the names of the object files and libraries needed for linking the function with other functions, if the function is sequential or parallel, if the function can continue from its previous point in execution once stopped, if the function can restart and continue on a different set of processors once stopped etc. GrADSolve supports the library functions to be written in the popular C or Fortran languages. For each input and output arguments, the library writer specifies the name of the argument, if the argument is an input or output argument, the datatype of the argument, the number of elements of the argument if the argument is a vector, the number of rows and columns of the argument if the argument is a matrix etc. The number of elements in the vector arguments and the number of rows and columns of the matrix arguments can be constants or expressed in terms of the other input arguments.

An example of a IDL file written for a ScaLAPACK QR factorization problem is

$$
\begin{array}{rcl}
\langle \text{PROBLEMSTART} \rangle & \longrightarrow & \langle \text{PROBLEMDESC} \rangle \, \langle \text{FUNCTION} \rangle \\
\langle \text{PROBLEMDESC} \rangle & \longrightarrow & PROBLEM \, \langle \text{PROBLEMNAME} \rangle \\
\langle \text{FUNCTION} \rangle & \longrightarrow & \langle \text{LANGUAGE} \rangle \, FUNCTION \, \langle \text{FUNCDEFN} \rangle \, \langle \text{FUNCDESC} \rangle \\
 & & \langle \text{FUNCLIB} \rangle \, \langle \text{FUNCTYPE} \rangle \, \langle \text{CONTINUECAPACITY} \rangle \\
 & & \langle \text{RECONFIGCAPACITY} \rangle \\
\langle \text{LANGUAGE} \rangle & \longrightarrow & C \mid FORTRAN \\
\langle \text{FUNCDEFN} \rangle & \longrightarrow & \langle \text{FUNCNAME} \rangle \; (\langle \text{ARGLIST} \rangle) \\
\langle \text{FUNCDESC} \rangle & \longrightarrow & '' \, \langle \text{STRING} \rangle '' \\
\langle \text{FUNCLIB} \rangle & \longrightarrow & LIBS ='' \, \langle \text{STRING} \rangle '' \\
\langle \text{FUNCTYPE} \rangle & \longrightarrow & TYPE = \langle \text{TYPESTRING} \rangle \\
\langle \text{CONTINUECAPACITY} \rangle & \longrightarrow & CONTINUE = \langle \text{OPTIONSTRING} \rangle \\
\langle \text{RECONFIGCAPACITY} \rangle & \longrightarrow & RECONFIGURE = \langle \text{OPTIONSTRING} \rangle \\
\langle \text{ARGLIST} \rangle & \longrightarrow & \langle \text{ARGUMENT} \rangle \mid \langle \text{ARGLIST} \rangle \, , \langle \text{ARGUMENT} \rangle \\
\langle \text{ARGUMENT} \rangle & \longrightarrow & \langle \text{INOUTSTRING} \rangle \, \langle \text{DATATYPE} \rangle \, \langle \text{VARNAME} \rangle \\
 & & \mid \langle \text{INOUTSTRING} \rangle \, \langle \text{DATATYPE} \rangle \, \langle \text{VARNAME} \rangle \\
 & & \langle \text{VACTORATTR} \rangle \\
 & & \mid \langle \text{INOUTSTRING} \rangle \, \langle \text{DATATYPE} \rangle \, \langle \text{VARNAME} \rangle \\
 & & \langle \text{MATRIXATTR} \rangle \\
\langle \text{VECTORATTR} \rangle & \longrightarrow & [\langle \text{DIMENSIONEXPR} \rangle] \\
\langle \text{MATRIXATTR} \rangle & \longrightarrow & [\langle \text{DIMENSIONEXPR} \rangle] \; [\langle \text{DIMENSIONEXPR} \rangle] \\
\langle \text{DIMENSIONEXPR} \rangle & \longrightarrow & \langle \text{NUMBER} \rangle \mid \langle \text{VARNAME} \rangle \\
\langle \text{PROBLEMNAME} \rangle & \longrightarrow & \langle \text{IDENTIFIER} \rangle \\
\langle \text{FUNCNAME} \rangle & \longrightarrow & \langle \text{IDENTIFIER} \rangle \\
\langle \text{TYPESTRING} \rangle & \longrightarrow & sequential \mid parallel \\
\langle \text{OPTIONSTRING} \rangle & \longrightarrow & yes \mid no \\
\langle \text{INOUTSTRING} \rangle & \longrightarrow & IN \mid OUT \mid INOUT \\
\langle \text{DATATYPE} \rangle & \longrightarrow & INT \mid FLOAT \mid DOUBLE \mid CHAR \\
\langle \text{VARNAME} \rangle & \longrightarrow & \langle \text{IDENTIFIER} \rangle
\end{array}
$$

Figure 6.3: BNF of GrADSolve IDL

```
PROBLEM qrwrapper
C FUNCTION qrwrapper(IN int N, IN int NB, INOUT double A[N][N],
                    INOUT double B[N][1])
``a version of qr factorization that works with square matrices.''
LIBS = ``/home/grads23/GrADSolve/ScaLAPACK/pdgeqrf_instr.o \
        /home/grads23/GrADSolve/ScaLAPACK/pdscaex_instrQR.o \
        ...
        ...
        ...''
TYPE = parallel
CONTINUE_CAPABILITY = yes
RECONFIGURATION_CAPABILITY = yes
```

Figure 6.4: An example GrADSolve IDL for a ScaLAPACK QR problem

given in Figure 6.4.

After the library writer submits the IDL file to the GrADSolve system, GrADSolve translates the IDL file to a XML document. The XML document generated for the IDL file in Figure 6.4 is shown in Figure 6.5.

The GrADSolve translation system also generates a wrapper program. This wrapper program is a driver and acts as an entry point for remote execution of the actual function. The wrapper program when compiled and executed performs certain important functions. The wrapper performs the necessary initialization if the end application is a parallel application. The wrapper function also reads a configuration file that specifies the location of the Runtime Support System (RSS) and the Autopilot manager. The configuration file was generated and staged to the machines for remote execution when the end user submitted his problem to the GrADSolve system. The wrapper pro-

```
<?xml version="1.0"?>
  <function name="qrwrapper">
    <user>grads23</user>
    <description>a version of qr factorization that works
                  with square matrices.
    </description>
    <type>parallel</type>
    <language>C</language>
    <continue>1</continue>
    <reconfigure>1</reconfigure>
    <call>
      <argCount>4</argCount>
      <arg>
        <inout>IN</inout>
        <datatype>int</datatype>
        <objecttype>scalar</objecttype>
        <name>N</name>
      </arg>
      <arg>
        <inout>IN</inout>
        <datatype>int</datatype>
        <objecttype>scalar</objecttype>
        <name>NB</name>
      </arg>
      <arg>
        <inout>INOUT</inout>
        <datatype>double</datatype>
        <objecttype>matrix</objecttype>
        <name>A</name>
        <rowExpression>N</rowExpression>
        <colExpression>N</colExpression>
      </arg>
```


Figure 6.5: XML document generated for the IDL in Figure 6.4

```
    <arg>
      <inout>INOUT</inout>
      <datatype>double</datatype>
      <objecttype>matrix</objecttype>
      <name>B</name>
      <rowExpression>N</rowExpression>
      <colExpression>1</colExpression>
    </arg>
  </call>
</function>
```

Figure 6.5. Continued

gram then registers with the RSS using SRS_Init() and also registers with the Autopilot system. The wrapper program then retrieves the problem description from the XML database, initializes the input and output arguments, and reads the input data from the appropriate files into the input arguments. It then invokes the actual function specified in the IDL file with the input and output arguments. Once the actual problem is solved by the execution of the actual function, the wrapper program stores the output arguments to files. It finally performs finalization routines for deregistering from RSS, the Autopilot manager and the parallel execution environment.

The library writer has the option of instrumenting the end application with calls to SRS library and the Autopilot. If stopping and continuing capability is desired for the application, the application has to be instrumented with calls to the SRS library. When the library writer wants to add execution model for his application, he instruments the application with calls to the Autopilot library so that portions of the application for which execution model was written are timed and the actual execution times are reported for monitoring by the Contract Monitor.

After generating the wrapper program, the GrADSolve system compiles the wrapper program with the object files and the libraries specified in the IDL file and with the appropriate parallel libraries if the application is specified as a parallel application in the IDL file. The result of the compilation process is an executable file suitable for remote execution on the GrADSolve resources. The application is finally uploaded into the GrADSolve system. The uploading process consists of two phases. In the first phase, the executable file is staged to the remote machines in the GrADSolve system and stored in the user's accounts on the machines. The machines available in the system are determined by querying the *resources* table in the XML database and the user's home directories on the machines to which the executable file is stored are determined by querying the *users* table in the XML database. In the second phase of the uploading process, the XML document for the application generated from the IDL file is stored in the XML database keyed by the problem name. Also, stored in the XML database for the application is the information regarding the location of the executable files for the application on the remote resources.

If the library writer wants to add an execution model for his application, he executes the *getperfmodel_template* utility specifying the name of the application. The utility retrieves the problem description of the application from the XML database and generates a performance model template file. The template file contains the definitions of the execution model routines. The library writer fills in the execution model routines with the

```
int areResourcesSufficient(int N, int NB, double* A,
                           double* B, RESOURCEINFO* resourceInfo,
                           SCHEDULESTRUCT* schedule){

}

int getExecutionTimeCost(int N, int NB, double* A, double* B,
                         RESOURCEINFO* resourceInfo, SCHEDULESTRUCT*
                         schedule, double* cost{

}

int mapper(int N, int NB, double* A, double* B,
           RESOURCEINFO* resourceInfo, SCHEDULESTRUCT* inputSchedule,
           SCHEDULESTRUCT* mapperSchedule){


}
```

Figure 6.6: A Performance Model template generated by the GrADSolve system for the QR problem

appropriate code for predicting the execution cost of his application. The performance model template file generated by the *getperfmodel_template* for the ScaLAPACK QR problem is shown in Figure 6.6.

The performance model template file contains definitions for three functions. The first function, *areResourcesSufficient* takes as input the problem parameters, a given set of machines for problem execution, *schedule* and the resource capabilities of the machines, *resourceInfo*. The library writer fills the function such that the function will return 1 if the machines have adequate capacities for solving the problem and 0 otherwise. The second function *getExecutionTimeCost* takes as input the problem parameters, the given set of machines, the resource capabilities and returns as output, the predicted execution cost, *cost*, of the application if the application were to run on the given set of machines. The third function, *mapper* is an optional function. It is used for specifying the data distribution of the different data used by the application. The mapper can also change the order of the machines in the given set of machines represented by *inputSchedule* and return the new order of machines in the *mapperSchedule*. The execution model for the ScaLAPACK QR application filled with the code written by the library writer is shown in the Figure 6.7.

The library writer uploads his execution model by executing the *add_perfmodel* utility. After performing certain error checking mechanisms, the *add_perfmodel* utility generates a wrapper program for the execution model. This wrapper program contains functions that act as entry points for the execution model. The functions in the wrapper program initialize certain parameters with default values and invoke the functions in the execution model. The *add_perfmodel* utility finally uploads the execution model for the application by storing the location of the wrapper program and the execution model to the XML database corresponding to the entry for the application.

6.4.3 End Users

The end users solve problems over remote GrADSolve resources by writing a client program. This client program can be written in C or Fortran. The client program includes an invocation of a routine called *gradsolve()* passing to the function, the name of the end application and the input and output parameters needed by the end application.

```
int areResourcesSufficient(int N, int NB, double* A,
                           double* B, RESOURCEINFO* resourceInfo,
                           SCHEDULESTRUCT* schedule){
  memAvailable = 0.0;
  for(i=0; i<schedule->count; i++){
    memAvailable += resourceInfo->meminfo[schedule->indices[i]];
  }
  memNeeded = (double)N *( (double)N + (double)NB + 1.0) * sizeof(double);
  if(memNeeded > memAvailable){
    return 0; /*resources not sufficient */
  }
  return 1; /*resources sufficient */
}
int getExecutionTimeCost(int N, int NB, double* A, double* B,
                         RESOURCEINFO* resourceInfo, SCHEDULESTRUCT*
                         schedule, double* cost{
  for(j=0; j<N; j+=NB){
    trun += t1+t2; /* t1 and t2 are times for different phases
                      in the iteration */
  }
  *cost = trun;
  return 0;
}
int mapper(int N, int NB, double* A, double* B,
           RESOURCEINFO* resourceInfo, SCHEDULESTRUCT* inputSchedule,
           SCHEDULESTRUCT* mapperSchedule){
  setBlockCyclicDistribution("A", mapperSchedule, N*NB);
  B_distribution = (int*)malloc(sizeof(int)*mapperSchedule->count);
  B_distribution[0] = N;
  for(i=1; i<mapperSchedule->count; i++){
    B_distribution[i] = 0;
  }
  setDistribution("B", mapperSchedule, B_distribution);
  free(B_distribution);
  getExecutionTimeCost(N, NB, A, B, resourceInfo, mapperSchedule,
                       &(mapperSchedule->cost));
  return 0;
}
```

Figure 6.7: A QR Performance Model filled with library writer code

An example of a GrADSolve client program written in C is shown in Figure 6.8.

The invocation of the *gradSolve()* routine triggers the execution of the GrADSolve Application Manager. As a first step, the Application Manager verifies if the user has credentials to execute applications on the GrADSolve system. GradSolve uses Globus Grid Security Infrastructure (GSI) [32] for the authentication of users. The Application Manager then queries the XML Database to verify if the application had been previously uploaded by the library writer. If the application had not been uploaded, the Application Manager displays an error message to the user and aborts operation. If the application exists in the GrADSolve system, the Application Manager registers the problem run with the PostgreSQL database component of the metascheduler. The Application Manager then retrieves the problem description from the XML database and matches the user's data with the input and output parameters required by the end application.

If an execution model exists for the end application, the Application Manager downloads the execution model from the remote location where the library writer had previously stored the execution model. The Application Manager compiles the execution model programs with other wrapper programs, starts the application-specific Performance Modeler service and stores the location of the service in the Database Manager of the metascheduler. The Application Manager then retrieves the list of machines in the GrADSolve system from the *resources* table in the XML database, and retrieves resource characteristics of the machines from the NWS. Similar to the ad-hoc infras-

```
#include "gradsolve.h"

int main(int argc, char** argv){
int N, NB;
double* A, * B;
int i;

  N = 1000;
  NB = 40;

  A = (double*)malloc(sizeof(double)*N*N);
  B = (double*)malloc(sizeof(double)*N);

  srand48(time(0));

  for(i=0; i<N*N; i++){
    A[i] = drand48();
    if(i < N){
      B[i] = drand48();
    }
  }

  gradsolve("qrwrapper", N, NB, A, B);

  free(A);
  free(B);

  return 1;


}
```

Figure 6.8: GrADSolve C client code for the QR problem

tructure, the Application Manager passes the list of machines, along with the resource characteristics to the Permission Service to receive permission to proceed to the next stages of the Application Manager. If the permission is granted by the Permission Service, the Application Manager proceeds to the Schedule Generation phase.

In the Schedule Generation phase, the Application Manager first determines if the end application has an execution model. If an execution model exists, the Application Manager contacts the Performance Modeler service, passes the problem parameters and the list of machines with the machine capabilities and receives the final list of machines for the end application execution from the Permission Service. In this mode of operation, the Schedule Generation phase of the GrADSolve system is equivalent to the Performance Modeling phase of the ad-hoc infrastructure. Along with the final list of machines and the predicted execution cost for the final schedule, the Performance Modeling service also returns information about the data distribution for the different data in the end application. The Performance Modeler service also generates a *contract_file* that contains a list of predicted execution costs for the different phases of the end application. If an execution model does not exist for the end application, the Schedule Generation phase adopts default scheduling strategies to generate the final schedule for end application execution.

If the end application is a sequential application, the default scheduling procedure first determines the total size of the input data. For each machine in the GrADSolve system, the scheduling procedure determines the computation time and the time for movement of input and output data between the machine where the Application Manager executes and the target machine. For the computation time, the scheduling procedure assumes the total number of operations in the application to be equal to the input data size. For the data movement times, the network capacity between the machines is taken into account. The machine with the least value for the sum of the computation and the data movement times is chosen for the end application execution. If the end application is a parallel application, the default scheduling procedure assumes the parallel application to be a sequential broadcast operation where a single machine sends the input data to all the other machines. The time for sending the input data is determined as the broadcast time of the machine. The total time for each machine is calculated as the sum of the broadcast, computation and data movement times. After sorting the list of machines in the order of the total times, the scheduling procedure traverses the list and continues to select the machines for the final schedule till the ratios of the total times for the successive machines in the list increases beyond a tolerance limit.

At the end of the Schedule Generation phase, the GrADSolve Application Manager receives a list of machines for final application execution. Similar to the ad-hoc infrastructure, the GrADSolve Application Manager then passes the final list of machines and the expected execution cost as a contract to the Contract Negotiator of the metascheduler and receives either approval or rejection of the contract. If the contract is rejected, the Application Manager restarts from the Resource Selection phase again. If the contract is accepted, the Application Manager stores the status of the problem run and the final schedule in the PostgreSQL Database Manager of the metascheduler. The GrADSolve Application Manager then starts an Autopilot Manager [93] if an Autopilot Manager is not executing on the machine where the Application Manager was started.

The Application Manager then creates working directories on the remote machines of the final schedule for end application execution and enters the Application Launching phase. The Application Launching phase consists of several important functions. As a first step, the Application Launcher starts the Runtime Support System (RSS) needed for the SRS library. It then creates configuration files needed for Contract Monitoring (*AP.config*) and the SRS library (*srs.config*) and stages these files to the remote machine chosen for executing the first process of the end application. The Application Launcher then stores the input data to files and stages these files to the corresponding remote machines chosen for application execution. If data distribution information for an input data does not exists, the Application Launcher stages the entire input data to all the machines involved in end application execution. If the information regarding data distribution for an input data exists, the Application Launcher stages only the appropriate portions of the data to the corresponding machines. This kind of selective data staging significantly reduces the time needed for the staging for entire data especially if large amount of data is involved. Apart from staging the input data, the Application Launcher also stages the information regarding data distribution to the remote machines.

After the staging of input data, the Application Launcher launches the end application on the remote machines chosen for the final schedule using the Globus MPICH-G [56] mechanism. If an execution model exists for the problem, the Application Launcher also starts the Contact Monitor for monitoring the progress of the end application. The end application, after registering with the RSS and the Autopilot, reads the input data that were previously staged by the Application Launcher and solves the problem. The end application then stores the output data to the corresponding files on the machines in the final schedule and deregisters from the RSS and the SRS library. The SRS library, during the deregistration of the application, stores the completion status of the problem in the PostgreSQL Database Manager.

If the end application was stopped by a metascheduler component, the Application Launcher waits for a RESUME signal from the component and restarts from the Resource Selection Phase. If the end application finished execution, the Application Launcher copies the output data from the remote machines to the user's memory space. The staging in of the output data from the remote locations is a reverse operation of the staging out of the input data to the remote locations. The GrADSolve Application Manager finally returns success state to the user's client program.

6.5 Execution Traces in GrADSolve - Storage, Management and Usage

One of the unique features in the GrADSolve system is the ability provided to the users to store and use execution traces of problem runs. There are many applications in which the outputs of the problem depend on the exact number and configuration of the machines used for problem solving. For example, considering the problem of adding large number of double precision numbers, one of the parallel implementations of the problem is to partition the list of double precision numbers among all processes of the parallel application, compute local sums of the numbers in each process and compute the global sum of the local sums computed on each process. The final sum obtained for the same set of double precision numbers may vary from one problem run to another depending on the number of elements in each partition, the number of processes used in the parallel application and the actual processes used in the computation. This is due to the impact of the round off errors caused by the addition of double precision numbers. In general ill-conditioned problems or unstable algorithms can give rise to vast changes in output results due to small changes in input conditions. For these kinds of applications, the user may desire to use the same input environment for all problem runs. Also, during testing of new numerical algorithms over the Grid, different groups working on the algorithm may want to ensure that same results are obtained when the algorithms are executed with same input data on the same configuration of resources.

To guarantee reproducibility of numerical results in the above situations, GrADSolve provides capability to the users to store *execution traces* of problem runs and use the execution traces during subsequent executions of the same problem with the same input data. For storing the execution trace of the current problem run, the user executes his GrADSolve program with a configuration file called *input.config* in the working directory containing the following line:

TRACE_FLAG = 1

During the registration of the problem run with the PostgreSQL Database Manager of the metascheduler, the value of the TRACE_FLAG variable is stored. The GrADSolve Application Manager proceeds to other stages of its execution. After the end application completes it execution and the output data are copied from the remote machines to the user's memory, the Application Manager, under default mode of operation, removes the remote working directories used for storing the files containing the input data for the end application. But when the user wants to store the execution trace of the problem run, i.e. when the *input.config* file contains "TRACE_FLAG = 1" line, the Application Manager retains the input data used for the problem run in the remote machines. At the end of the problem run, the Application Manager generates an output configuration file called *output.config* containing the following line

TRACE_KEY = <key>

The value *key* in the *output.config* is a pointer to the execution trace stored for the problem run.

When the user wants to execute the problem with the execution trace previously stored, he executes his client program specifying the line, TRACE_KEY = <key>

in the *input.config* file. The value *key* in the input.config, is the same value previously generated by the GrADSolve Application Manager when the execution trace was stored. The Application Manager first checks if the TRACE_KEY exists in the Database Manager. If the TRACE_KEY does not exist, the Application Manager displays an error message to the user and aborts operation. If the TRACE_KEY exists for an execution trace of a previous problem run, the Application Manager registers the current problem run with the Database Manager and proceeds to the other stages of its execution. During the Schedule Generation phase, the Application Manager, instead of generating a schedule for the execution of the end application, retrieves the schedule used for the previous problem run corresponding to the TRACE_KEY, from the Database Manager. The Application Manager then checks if the capacities of the resources in the schedule at the time of trace generation are comparable to the current capacities of the resources. If the capacities are not comparable, the Application Manager displays an error message to the user and aborts the operation. If the capacities are comparable, the Application Manager proceeds to the rest of the phases of its execution. During the Application Launching phase, the Application Manager, instead of staging the input data to remote working directories, copies the input data and the data distribution information, used in the previous problem run corresponding to the TRACE_KEY, to the remote working directories. The use of the same number of machines and the same input data used in the previous schedule also guarantees the use of the same data distribution for the current problem run. Thus GrADSolve guarantees the use of the same execution environment used in the previous problem run for the current problem run, and hence guarantees reproducibility of numerical results.

To support the storage and use of execution traces in the GrADSolve system, two PostgreSQL trigger functions are used. One trigger function called *trace_usage_trigger* updates the last usage time of an execution trace when the execution trace is used for a problem run. Another trigger function called *cleanup_trigger* is used for periodically deleting entries in the Database Manager thereby maintaining the size of the *problems* table in the Database. The *cleanup_trigger* is invoked whenever a new entry corresponding to a problem run is added to the *problems* table. The *cleanup_trigger* first deletes those entries for which the execution traces were not stored if the entries existed in the Database for more than 10 minutes. The *cleanup_trigger* then deletes those entries for which the execution traces were stored, if the time of last usage of the execution trace is greater than 30 days. Thus by using longer duration for those problem runs for which execution traces were stored, the *cleanup_trigger* function provides greater opportunity for the usage of the execution traces for the subsequent problem runs. If no entries meet the above criteria for deletions and the number of entries for which the execution traces were stored is greater than 100, the *cleanup_trigger* function orders the entries based on completion times and deletes the first few entries in the list till the number of entries in the DataBase decreases to less than 100.

6.6 Metascheduler in GrADSolve

The metascheduler in GrADSolve is similar to the metascheduler in the ad-hoc infrastructure with few minor modifications. While the metascheduler in the ad-hoc infrastructure assumes the existence of preemptible parallel applications for which execution models exist, mechanisms have been plugged into the metascheduler in the GrADSolve system to support different kinds of applications with different capabilities.

For the Database Manager, the popular and more robust PostgreSQL was used. In addition to the fields for the *problems* table used in the Database Manager for ad-hoc infrastructure, the *problems* table in the PostgreSQL Database also contains fields for storing different capabilities of the applications including if an execution model exists for the end application, if the execution model possesses a mapper function, if the end application can continue from its previous point in execution once stopped, if the end application can be reconfigured etc. The PostgreSQL Database also contains fields for storing and managing execution traces for the problem runs. These fields include execution trace flag that indicates if the execution trace for the problem run has to be stored, the execution trace key that a problem run uses, the time of last usage of an execution trace etc. The PostgreSQL Database Manager is also supported by trigger functions that update certain fields on the occurrence of certain events. The use of the robust PostgreSQL Database enabled to increase the stability of the Database Manager of the metascheduler.

The Permission Service in the GrADSolve metascheduler accesses the PostgreSQL

Database Manager to retrieve the different capabilities of the end applications. If the end application for which the Permission decision is being made does not possess an execution model, the Permission Service assumes that the resources have adequate capacities to execute the end application and simply grants permission for the corresponding GrADSolve Application Manager to continue to the next stages of its execution. Also, during the decision to stop an executing application to accommodate a new application, the Permission Service utilizes the remaining execution time of the executing end application if the application can continue from a particular point in execution after stopping while the Permission Service utilizes the entire predicted execution time of the application if the end application cannot continue after stopping execution.

The Contract Negotiator in the GrADSolve metascheduler is similar to its ad-hoc infrastructure counterpart. It first ensures that the GrADSolve application for which contract decision is being made retrieved resource information from NWS for its application-level schedule before any executing applications started. If the GrADSolve application received its resource information before the start of any executing applications, the Contract Negotiator simply sends CONTRACT_OK for those applications that do not possess execution models. For those applications for which execution models exist, the Contract Negotiator tries to improve the performance contract. While evaluating the potential performance benefits for the new application due to stopping an executing application, the Contract Negotiator takes into account only those executing applications for which execution models exist and for which execution traces are not stored or used. If such executing applications can continue after stopping, the Contract Negotiator utilizes the remaining execution times of the applications in its calculations while it uses the entire predicted execution times of the applications, if the applications cannot continue after stopping execution. Similarly, while trying to reduce the impact on executing applications due to the addition of the new application, the Contract Negotiator considers only those new applications for which executions models exists and which do not utilize a previous execution trace. It also tries to evaluate impacts on only those executing applications for which execution models exist.

The Rescheduler of the metascheduler in the GrADSolve system considers rescheduling only those executing applications that possess execution models and for which execution traces are not stored or used. While deciding to migrate an executing application to a new set of resources, the rescheduler retrieves a new schedule consisting of the same number of machines in the current schedule from the Performance Model service, if the end application cannot reconfigure after stopping execution. Similarly, while making migrating decisions for an executing application, the Rescheduler considers either the remaining execution time of the application with the new schedule of resources or the entire predicted cost of the application with the new schedule depending on the continuation capability of the application after stopping execution.

6.7 Experiments and Results

For the experiments in this section, 4 machines from the GrADS testbed were used - a *msc* machine from UT, a *opus* machine from UIUC and two *circus* machines from UCSD. In the experiments, GrADSolve was used to remotely invoke ScaLAPACK driver for solving the linear system of equation, $AX = B$. The driver invokes ScaLAPACK QR factorization for the factorization of matrix, A. Block cyclic distribution was used for the matrix, A and the right-hand side vector, B. A GrADSolve IDL was written for the driver routine and an execution model that predicts the execution cost of the QR problem was uploaded into the GrADSolve system. The GrADSolve user invokes the remote parallel application by passing the size of the matrix, the matrix, A and the right-hand side vector, B to the *gradsolve()* call.

GrADSolve was operated in 3 modes. In the first mode, the execution model did not contain information about the data distribution used in the ScaLAPACK program. In this case, GrADSolve transported the entire data to each of the locations used for the execution of the end application. This mode of operation is practiced in RPC systems that do not support the information regarding data distribution. In the second mode, the execution model contained information about the data distribution used in the end application. In this case, GrADSolve transported only the appropriate portions of the data to the locations used for the execution of end application. In the third mode, GrADSolve was used with an execution trace corresponding to a previous run of the same problem. In this case, data is not staged from the user's address space to the

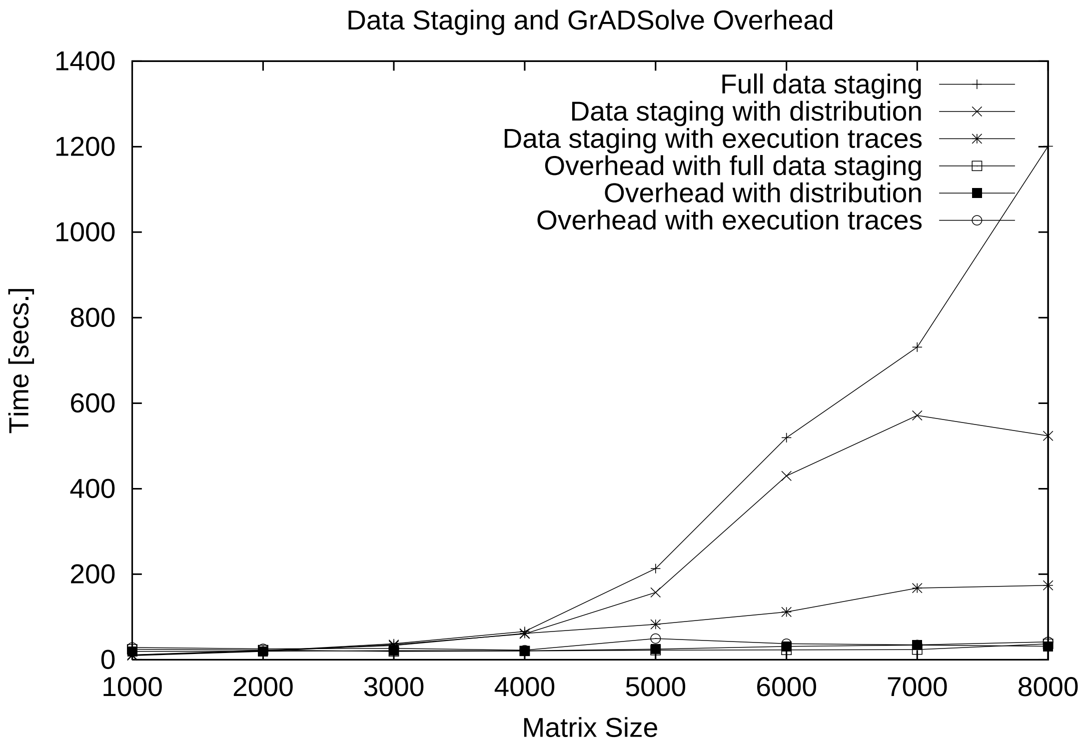

Figure 6.9: Data staging and other GrADSolve overhead

remote machines, but temporary copies of the input data used in the previous run are made for the current problem run.

Figure 6.9 shows the times taken for data staging and other GrADSolve overhead for different matrix sizes and for the three modes of GrADSolve operation. Since the times taken for the execution of the end application are same in all the three modes, we focus only on the times taken for data staging and possible Grid overheads. The machines that were chosen by the GrADSolve application-level scheduler for the execution of end application for different matrix sizes are shown in Table 6.1.

Comparing the first two modes in Figure 6.9, we find that for smaller problem sizes,

Table 6.1: Machines chosen for application execution

Matrix size	*Machines*
1000	1 UT machine
2000	1 UT machine
3000	1 UT machine
4000	1 UT machine
5000	1 UT, 1 UIUC machines
6000	1 UIUC, 1 UCSD machines
7000	1 UIUC, 1 UCSD machines
8000	1 UT, 1 UIUC, 2 UCSD machines

the times taken for data staging in both the modes are the same. This is because only one machine was used for problem execution and the same amount of data are staged in both the modes when only one machine is involved for problem execution. For larger problem sizes, the times for data staging with distribution information is less than 20-55% of the times taken for staging the entire data to remote resources. Thus the use of data distribution information in GrADSolve can give significant performance benefits when compared to staging the entire data that is practiced in some of the RPC systems. Data staging in the third mode is basically the time taken for creating temporary copies of data used in the previous problem runs in remote resources. We find this time to be negligible when compared to the first two modes. Thus execution traces can be used as caching mechanisms to use the previously staged data for problem solving. The GrADSolve overheads for all the three modes are found to be the same. This is because of the small number of machines used in the experiments. For experiments when large number of machines are used, we predict that the overheads will be higher in the first two modes than in the third mode. This is because in the first two modes, the application-level scheduling will explore large number of candidate schedules to determine the machines used for end application while in the third mode, a previous application-level schedule will be retrieved from the database and used.

Chapter 7

Conclusions and Future Work

7.1 Contributions of the Research

In this research, we discussed the need of a preemptive-based metascheduler for distributed computing by identifying the problems faced with other scheduling strategies, namely, application-level scheduling. Specifically, the research intended to meet three objectives of scheduling that are critical for distributed computing - to provide high performance to individual applications within the constraints of the system loads, to accommodate maximum number of applications into the system without overwhelming the system resources and to provide high throughput of the overall system.

Towards meeting the objectives, the research designed and developed a metascheduler for the Grid that takes into account both the application level and system level considerations. The different components of the metascheduler, viz., the Database Manager, Permission Service, Contract Negotiator and the rescheduler were explained in detail. These components provided valuable scheduling services for meeting our scheduling objectives. The Permission Services tries to accommodate new applications into the system by preempting executing large applications. The Contract Negotiator acts as an arbitrator between different application-level schedulers and balances the interests of different applications. It acts as a queue manager ensuring that the application-level schedulers made their scheduling with recent resource information. This guarantees that the application-level schedulers avoid conflicting claims on the same set of resources and also develop correct expectations for their performance. The Contract Negotiator preempts executing applications for improving the performance contract values of new applications and also reduces the impact on executing applications by the new applications. These decisions of the Contract Negotiator ensure high performance for individual applications and high throughput of the system. The Rescheduler, by employing a robust migration framework reschedules applications if performance guarantees are not met or to make use of free resources. The policies of the Rescheduler again help in improving the performance of individual applications and providing high throughput of the overall system. The policies of the metascheduling components were validated with the help of demonstrative experiments and were found to meet the scheduling objectives for distributed computing. Practical experiments were also conducted to study the behavior of the metascheduler and to compare with situations when the metascheduler was not used. These practical experiments proved that the metascheduler was helpful in guaranteeing the performance contracts of the applications and maintaining the mean performance contract ratios to be as small as 2.5 and the standard deviation of the ratios to be as low as 1.5. The use of the metascheduler also helped in accommodating atleast 3 more large applications than when the metascheduler was not used. Finally, the use of the metascheduler also helped in increasing the throughput of the system by atleast 15%.

The migration framework employed by the Rescheduler contains robust and unique mechanisms for rescheduling executing applications. Many existing migrating systems that migrate applications under loading conditions implement simple policies that cannot be applied to Grid systems. The migration framework utilized in the metascheduler takes into account both the system load and application characteristics. The migrating decisions are based on factors like the amount of load, the time of the application when the load is introduced and the size of the applications. The research also implemented a framework that migrates executing applications to make use of additional free resources. Experiments were conducted and results were presented to demonstrate the capabilities of the migration framework. Based on the load conditions of the resources, the migration framework can yield upto 70% improvement in performance for executing applications. The performance benefits that can be obtained due to rescheduling depends on the time taken to redistribute data to the new set of processors. This time for redistribution depends on the resource and network characteristics of the resources involved in the current and the new schedule at the time of rescheduling and the amount of data movement involved. An initial design for retrieving these parameters at the time of rescheduling and predicting the redistribution cost dynamically was integrated into the migration framework. Our experiments and results showed that the predicted redistribution cost correspond with the actual redistribution costs, thereby increasing the accuracy of the rescheduling decisions made by the migration framework.

The migration framework and other components of the metascheduler rely on the existence of premptible applications. A checkpointing infrastructure called SRS for developing and executing malleable and migratable parallel applications across heterogeneous sites was explained. The SRS API has limited number of functions for seamlessly enabling parallel applications malleable. The uniqueness of the SRS system is achieved by the use of IBP distributed storage infrastructure. Results were shown to evaluate the overhead incurred to the applications and the times for storing, reading and redistributing checkpoints. Our experiment results indicate that parallel applications, with instrumentation to SRS library, were able to achieve reconfigurability incurring only about 15-35% overhead.

Finally, to achieve practical utility of the metascheduler, an actual metascheduler-based Grid RPC system called GrADSolve was developed. The Grid system is different from many other Grid computing systems in that it is able to incorporate applications with different capabilities. These capabilities include the presence of execution models to predict the execution cost of the application, the presence of data distribution information, the ability for the applications to be preempted during execution, the ability to reconfigure etc. GrADSolve is an RPC system intended for efficient execution of remote parallel software. The efficiency is achieved by dynamically choosing the machines used for parallel execution and staging the data to remote machines based on data distribution information. The GrADSolve RPC system also supports maintaining and utilizing execution traces for problem solving. Our experiments showed that the GrADSolve system is able to adapt to the problem sizes and the resource characteristics and yielded significant performance benefits with its data staging and execution trace mechanisms. The data staging mechanisms in GrADSolve helps reduce the data staging times in RPC systems by 20-50%.

7.2 Future Directions of the Research

Though the metascheduler has been found to achieve its objectives with the help of demonstrative experiments, there is a need of formal set of mathematical formulations to determine the optimal values that can be achieved by metascheduling techniques in general and to evaluate the efficacy of the implementation of our metascheduler in achieving the scheduling objectives. The evaluation of the metascheduler can also be achieved by means of simulation techniques. Current simulation techniques are able to evaluate only the low-level schedulers that determine the final set of resources for application execution. In a complex metascheduling system where there are many interactions between the low-level schedulers and the metascheduler and between the metascheduling components themselves, robust and innovative simulation techniques have to be developed. Also, our metascheduler consists of a set of ad-hoc techniques that implement scheduling policies. Designing and implementing the metascheduler based on format set of specifications and formulations will be the subject of future research.

Our metascheduler uses various thresholds for implementing various policies regarding preempting executing applications to accommodate new applications and to improve the performance contracts of new applications, determining the relative problem sizes of different applications, determining the impact of the executing applications on the performance contract of new applications, waiting for an executing application to complete before accommodating new applications, allowing a new application to execute in the presence of executing applications, specifying acceptable limits of impact on executing applications by new applications and evaluating the benefits that can be obtained by rescheduling executing applications. These threshold values were obtained by conducting trial-and-error experiments with problems of different sizes and different resource combinations available in the GrADS testbed. These thresholds depend on various factors like the accuracy of resource information, the type of applications, the predictions of various resource parameters etc. and will have to be determined dynamically. Also, in our metascheduling system, the applications are preempted at fixed intervals. The intervals for preemption can also be determined dynamically based on the kind of job mix and the history of workloads. Thus dynamically determining various parameters for metascheduling will be an interesting avenue for future research.

Currently our metascheduler has been tested with only regular applications. Regular applications offer ample opportunities to predict execution costs, to integrate with the contract developer and monitor and to predict the remaining execution time of the executing applications. We plan to explore the challenges involved in integrating non-regular and multi-component applications into our metascheduler. We also plan to test our metascheduling framework in environments involving separate domains where number of local scheduling policies are involved.

There are also few issues regarding implementation of the current metascheduler. Though the architecture of our metascheduler is decentralized with number of metascheduling components and application-level schedulers, the architecture of the individual metascheduling components are centralized accepting requests from all applications in the system. The centralized approach of the metascheduling components can lead to issues in scalability, especially when large number of applications are involved. We plan to implement a distributed metascheduler to improve the scalability of the architecture. Also, the current metascheduler implements loosely coupled locks so that the policies implemented by a metascheduler component do not override the policies implemented by other metascheduler component. We plan to implement robust distributed locks for the purpose. The current implementation of the metascheduler also has to be enhanced to improve the mean response times of rejected applications.

There are various opportunities of interesting research in the area of rescheduling executing applications to improve the performance of the applications. Currently, the migration framework takes into account the the current load on the machines and the remaining execution time of the applications to evaluate the performance benefits due to rescheduling. In our future effort, the rescheduler will also take into account the predictions of the rate of change of loads on the system resources, the history of workload on the resources, the rate of availability of the machines and the load caused by the executing applications themselves to determine more accurate values of performance benefits due to rescheduling. Though we have obtained encouraging initial results for predicting the cost for redistribution of data, predicting the redistribution cost for any application with any data distribution is a hard problem. We plan to provide robust interfaces for the library writers to communicate information about the data distribution used in the application to the rescheduler framework. Also, our opportunistic migration currently migrates executing applications when certain resources become lightly loaded. Our future work will involve opportunistic migration when new resources are added to the Grid system. Another major avenue for future research direction is to extend our SRS checkpointing library. One of the main goals will be to use precompiler technologies to restore the execution context and to relieve the user from having to make major modifications in his program to provide malleability of his applications. The precompilation strategies will be similar to the approaches taken by Ferrari [50], Dome [24], Zandy [119] and Sun et. al. [102]. Other future investigations include support for checkpointing files, complex pointers and structures and to provide support for different kinds of applications. Although the design of the checkpointing framework supports migration in heterogeneous environments, the current implementation stores the checkpoint data as raw bytes. This approach will lead to misinterpretation of the data by the application if, for example, the data is stored on a Solaris system and read by a Linux machine. This is due to the different byte orderings and floating point representations followed on different systems. We plan to use the External Data Representation (XDR) or Porch Universal Checkpointing Format (UCF) [100, 89] for representing the checkpoints. We also plan to separate the storage nodes for checkpoints from the computational nodes for application execution by employing the eXNode [22] architecture. This will provide robust fault tolerant mechanism for withstanding the processor failures in SRS. We also intend to collaborate with the CUMULVS project [63] to provide a generic visualization architecture that will be used to monitor the execution of malleable applications. There are also plans to extend the RSS daemon to make it fault-tolerant by periodically checkpointing its state so that the RSS service can be migrated across sites. Presently, all the processes of the parallel application communicate with a single RSS deamon. This may pose a problem for the scalability of the checkpointing system, especially when large number of machines are involved. Our future plan is to implement a distributed RSS system to provide scalability.

Finally, there are also future plans to extend our GrADSolve RPC system. Pluggable interfaces to the library writers for expressing new capabilities of the end application to the GrADSolve system will be developed. Remote execution of non-MPI parallel programs and applications with different modes of parallelism are also being considered. Support for remote invocation in different programming languages including MATLAB are also part of our future efforts. Lastly, automatic usage of execution traces based on the supplied parameters will alleviate the need for the GrADSolve users to supply trace keys for usage.

Bibliography

Bibliography

[1] Apache xindice. *http://xml.apache.org/xindice.*

[2] Corba. *http://www.corba.org.*

[3] GrADS Web Site. *http://hipersoft.cs.rice.edu/grads/index.htm.*

[4] Grand Challenge Problems. *http://www.nhse.org/grand_challenge.htm.*

[5] Java Remote Method Invocation (Java RMI). *java.sun.com/products/jdk/rmi.*

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

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

[8] NWS Interactive Query. *http://nws.cs.ucsb.edu/CGI/graphIt.cgi.*

[9] Portable Batch System. *http://www.openpbs.org.*

[10] Postgresql. *http://www3.us.postgresql.org.*

[11] PVM. *http://www.csm.ornl.gov/pvm.*

[12] Xml-rpc. *http://www.xmlrpc.com.*

[13] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. *The 4th IEEE Symposium on High Performance Distributed Computing, Virginia*, August 1995.

[14] A.Chowdhury. Dynamic Reconfiguration: Checkpointing Code Generation. In *In Proceedings of IEEE 5th International Symposium on Assessment of Software Tools and Technologies (SAST97)*, 1997.

[15] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *In the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, August 1999.

[16] J.N.C. Arabe, A.B.B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. *Supercomputing*, 1995.

[17] P. Arbenz, W. Gander, and M. Oettli. The remote computation system. *Parallel Computing*, 23:1421–1428, 1997.

[18] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[19] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.

[20] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[21] A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 122–129, August 1995.

[22] M. Beck, T. Moore, and J. Plank. An End-to-End Approach to Globally Scalable Network Storage. In *ACM SIGCOMM 2002 Conference*, Pittsburgh, PA, USA, August 2002.

[23] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and Vaidy Sunderam. Tools for Heterogeneous Network Computing. In R. Sincovec et al., editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 854–861, Philadelphia, USA, 1993. SIAM Publications.

[24] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. Technical Report CMU-CS-96-157, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, August 1996.

[25] F. Berman. High-performance schedulers. In *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–203. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

[26] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Applications and Supercomputing*, 15(4):327–344, Winter 2001.

[27] F. Berman and R. Wolski. The AppLeS Project: A Status Report. *Proceedings of the 8th NEC Research Symposium*, May 1997.

[28] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[29] B. Bieker, G. Deconinck, E. Maehle, and J. Vounckx. Reconfiguration and Checkpointing in Massively Parallel Systems. In *Proceedings of 1st European Dependable Computing Conference (EDCC-1)*, volume Lecture Notes in Computer Science Vol. 852, pages 353–370. Springer-Verlag, October 1994.

[30] M. Bishop, M. Valence, and L. F. Wisniewski. Process Migration for Heterogeneous Distributed Systems. Technical Report TR95-264, 1995.

[31] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[32] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

[33] R. Buyya, D. Abramson, and J. Giddy. Nimrod-G Resource Broker for Service-Oriented Grid Computing. *IEEE Distributed Systems Online*, 2(7), November 2001.

[34] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[35] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing, 1995.

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

[37] T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, SE-14(2):141–154, February 1988.

[38] C-C. Chang, G. Czajkowski, and T. von Eicken. MRPC: A High Performance RPC System for MPMD Parallel Computing. 29(1):43–66, 1999.

[39] Y. Chen, K. Li, and J. S. Plank. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. In *SC97: High Performance Networking and Computing*, San Jose, November 1997.

[40] G. Deconinck and R. Lauwereins. User-Triggered Checkpointing: System-Independent and Scalable Application Recovery. In *Proceedings of 2nd IEEE Symposium on Computers and Communications (ISCC97)*, pages 418–423, Alexandria, Egypt, July 1997.

[41] G. Deconinck, J. Vounckx, R. Lauwereins, and J.A. Peperstraete. User-triggered Checkpointing Library for Computation-intensive Applications. In *Proceedings of 7th IASTED-ISMM International Conference On Parallel and Distributed Computing and Systems (IASTED, Anaheim-Calgary-Zurich) (ISCC97)*, pages 321–324, Washington, DC, October 1995.

[42] Deng, Gu, Brecht, and Lu. Preemptive Scheduling of Parallel Jobs on Multiprocessors. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1996.

[43] A. Denis, C. Pérez, and T. Priol. Achieving Portable and Efficient Parallel CORBA Objects. *Concurrency and Computation: Practice and Experience*, 2002.

[44] A. Denis, C. Prez, and T. Priol. Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing. In *Proc. of the 7th International Euro-Par'01 Conference (EuroPar'01)*, pages 835–844. Springer, August 2001.

[45] A. Denis, C. Prez, and T. Priol. Towards High Performance CORBA and MPI Middlewares for Grid Computing. In Craig A. Lee, editor, *Proc. of the 2nd International Workshop on Grid Computing*, number 2242 in LNCS, pages 14–25. Springer-Verlag, November 2001.

[46] L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems. In Wolfgang Gentzsch and Uwe Harms, editors, *Lecture notes in computer science 797, High Performance Computing and Networking*, volume Proceedings Volume II, Networking and Tools, pages 273–277, Munich, Germany, April 1994. Springer Verlag.

[47] F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[48] Xing Du and Xiaodong Zhang. Coordinating Parallel Processes on Networks of Workstations. *Journal of Parallel and Distributed Computing*, 46(2):125–135, 1997.

[49] M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[50] A.J. Ferrari, S.J. Chapin, and A.S. Grimshaw. Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification. Technical Report Technical Report CS-97-05, Department of Computer Science, University of Virginia, March 1997.

[51] S.M. Figueira and F. Berman. Modeling the Effects of Contention on the Performance of Heterogeneous Applications. *The 5th International Symposium on High Performance Distributed Computing (HPDC '96)*, pages 392–, August 1996.

[52] S.M. Figueira and F. Berman. Predicting Slowdown for Networked Workstations. *The 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, pages 92–101, August 1997.

[53] S.M. Figueira and F. Berman. Modeling the Slowdown of Data-Parallel Applications in Homogeneous and Heterogeneous Clusters of Workstations. *Seventh Heterogeneous Computing Workshop*, pages 90–101, March 1998.

[54] S.M. Figueira and F. Berman. A Slowdown Model for Applications Executing on Time-Shared Clusters of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):653–669, June 2001.

[55] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. volume Proc. 6th IEEE Symp. on High-Performance Distributed Computing, pages 365–375, 1997.

[56] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. *In Proceedings of SuperComputing 98 (SC98)*, 1998.

[57] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[58] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

[59] J. Gehring and A. Reinefeld. MARS - A Framework for Minimizing the Job Execution Time in a Metacomputing Environment. *Future Generation Computer Systems*, 12(1):87–99, 1996.

[60] Jörn Gehring and Thomas Preiss. Scheduling a Metacomputer with Uncooperative Sub-schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 179–201. Springer Verlag, 1999.

[61] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. A Users' Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[62] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[63] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of High Performance Computing Applications*, 11(3):224–236, August 1997.

[64] E. Godard, S. Setia, and E. White. DyRecT: Software Support for Adaptive Parallelism on NOWs. In *in IPDPS Workshop on Runtime Systems for Parallel Programming*, Cancun, Mexico, May 2000.

[65] A. Grimshaw, W. Wulf, J. French, A. Weaver, and Jr. P. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, 1994.

[66] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[67] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[68] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of Job-Scheduling Strategies for Grid Computing. In *GRID 2000*, pages 191–202, November 2000.

[69] C. Hofmeister and J. M. Purtilo. Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for Replacement. In *Proceedings of the 13 th International Conference on Distributed Computing Systems*, Pittsburgh, USA, May 1993.

[70] A. Jeong and D. Shasha. PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 96–105. IEEE, 1994.

[71] L.V. Kalé, S. Kumar, and J. DeSouza. A Malleable-Job System for Timeshared Parallel Machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[72] R. Kettimuthu, V. Subramani, S. Srinivasan, T.B. Gopalsamy, D. K. Panda, and P. Sadayappan. Selective Preemption Strategies for Parallel Job Scheduling. In *Proceedings of 2002 International Conference on Parallel Processing (ICPP 2002)*, August 2002.

[73] R. Koo and S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[74] K. Li, J.F. Naughton, and J.S. Plank. Real-time Concurrent Checkpoint for Parallel Programs. In *In Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.

[75] M. Litzkow, M. Livney, and M. Mutka. Condor - a Hunter for Idle Workstations. *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[76] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.

[77] D. McLaughlin, S. Sardesai, and P. Dasgupta. Preemptive Scheduling for Distributed Systems. In *11th International Conference on Parallel and Distributed Computing Systems*, September 1998.

[78] R. Mirchandaney, D. Towsley, and J. A. Stankovic. Adaptive Load Sharing in Heterogeneous Distributed Systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.

[79] V. K. Naik, S. P. Midkiff, and J. E. Moreira. A checkpointing strategy for scalable recovery on distributed parallel systems. In *SuperComputing (SC) '97*, San Jose, November 1997.

[80] E. W. Parsons and K. C. Sevcik. Implementing Multiprocessor Scheduling Disciplines. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 166–192. Springer Verlag, 1997.

[81] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical Libraries and the Grid: The GrADS Experiments with Scalapack. *Journal of High Performance Applications and Supercomputing*, 15(4):359–374, Winter 2001.

[82] J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. *NetStore99: The Network Storage Symposium*, 1999.

[83] James S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, 1997.

[84] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. Technical Report UT-CS-94-242, 1994.

[85] C. Prez, T. Priol, and A. Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. In Craig A. Lee, editor, *Proc. of the 3nd International Workshop on Grid Computing*, LNCS. Springer-Verlag, November 2002.

[86] P. Pruitt. An Asynchronous Checkpoint and Rollback Facility for Distributed Computations. Technical report, Honors Thesis, The College of William and Mary, Computer Science, Williamsburg, VA, 1998.

[87] R. Rabenseifner. The dfn remote procedure call tool for parallel and distributed applications. In *In Kommunikation in Verteilten Systemen - KiVS 95. K. Franke,*

U. Huebner, W. Kalfa (Editors), Proceedings, Chemnitz-Zwickau, pages 415–419, February 1995.

[88] A. Radulescu and Arjan J. C. van Gemund. Preemptive Task Scheduling for Distributed Systems (Research Note). *Lecture Notes in Computer Science*, 1900:272–276, 2001.

[89] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogenous architectures. In *27th International Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[90] D. A. Reed. Grids, the TeraGrid and Beyond. *IEEE Computer*, pages 62–68, January 2003.

[91] C. René and T. Priol. MPI Code Encapsulation using Parallel CORBA Object. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 3–10. IEEE, August 1999.

[92] C. René and T. Priol. MPI Code Encapsulating using Parallel CORBA Object. *Cluster Computing*, 3(4):255–263, 2000.

[93] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: Adaptive Control of Distributed Applications. *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.

[94] S. H. Russ, B. K. Flachs, J. Robinson, and B. Heckel. Hector: Automated Task Allocation for MPI. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium*, pages 344–348, Honolulu, Hawaii, April 1996.

[95] K.A. Saqabi, S.W. Otto, and J. Walpole. Gang Scheduling in Heterogeneous Distributed Systems. Technical report, OGI, 1994.

[96] H. Nakada M. Sato and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. In *Future Generation Computing Systems, Metascomputing Issue*, volume 15, pages 649–658, 1999.

[97] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. In *In Workshop on OpenMP Applications and Tools (WOMPAT2001)*, July 2001.

[98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference - The MPI Core*, volume 1. Boston MIT Press, 2nd edition, September 1998.

[99] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pages 526–531, Honolulu, Hawaii, 1996.

[100] V. Strumpen and B. Ramkumar. Portable Checkpointing and Recovery in Heterogeneous Environments. Technical Report Technical Report 96-6-1, Department of Electrical and Computer Engineering, University of Iowa, June 1996.

[101] V. Subramani, R. Kettimuthu, S. Srinivasan, and P Sadayappan. Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 359–366, July 2002.

[102] X. H. Sun, V. K. Naik, and K. Chanchio. Portable hijacking. In *SIAM Parallel Processing Conference*, March 1999.

[103] T. Suzumura, T. Nakagawa, S. Matsuoka, H. Nakada, and S. Sekiguchi. Are Global Computing Systems Useful? - Comparison of Client-Server Global Computing Systems Ninf, Netsolve versus CORBA. In *In Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS '00*, pages 547–559, May 2000.

[104] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobb's Journal*, pages 40–48, February 1995.

[105] Herwig Unger and Thomas Boehme. A Fuzzy Based Load Sharing Mechanism for Distributed Systems. Technical Report TR-98-026, International Computer Science Institute, Berkeley, CA, 1998.

[106] S. Vadhiyar and J. Dongarra. A Metascheduler for the Grid. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 343–351, July 2002.

[107] G.D. van Albada, J. Clinckemaillie, A.H.L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B.J. Overeinder, A. Reinefeld, and P.M.A. Sloot. Dynamite - Blasting Obstacles to Parallel Cluster Computing, April 1995.

[108] J.S. Vetter and D.A. Reed. Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids. *The International Journal of High Performance Computing Applications*, 14(4):357–366, 2000.

[109] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[110] F. Vraalsen, R.A. Aydt, C.L. Mendes, and D.A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior. In *Proceedings of the 2nd International Workshop on Grid Computing/LNCS (GRID 2001)*. Springer Verlag, November 2001.

[111] C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. *In First Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association*, pages 1–11, 1995.

[112] J. Weissman. The Interference Paradigm for Network Job Scheduling. *Proceedings of the Heterogeneous Computing Workshop*, pages 38–45, April 1996.

[113] J.B. Weissman. Prophet: Automated Scheduling of SPMD Programs in Workstation Networks. *Concurrency: Practice and Experience*, 11(6), November 1999.

[114] R. Wolski, G. Shao, and F. Berman. Predicting the Cost of Redistribution in Scheduling. *Proceedings of 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[115] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, October 1999.

[116] D. Wright. Cheap Cycles from the Desktop to the Dedicated Cluster: Combining Opportunistic and Dedicated Scheduling with Condor. *Proceedings of the Linux Clusters: The HPC Revolution conference, Champaign - Urbana, IL*, June 2001.

[117] A. Yarkhan and J. Dongarra. Experiments with Scheduling Using Simulated Annealing in a Grid Environment. In M. Parashar, editor, *Lecture notes in computer science 2536 Grid Computing - GRID 2002*, volume Third International Workshop, pages 232–242, Baltimore, MD, USA, November 2002. Springer Verlag.

[118] Z. You-Hui and P. Dan. A Task Migration Mechanism for Mpi Applications. In *In Proceedings of 3rd Workshop on Advanced Parallel Processing Technologies (APPT'99)*, pages 74–78, Changsha, China, October 1999.

[119] V.C. Zandy, B.P. Miller, and M. Livny. Portable hijacking. In *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 177–184, August 1999.

[120] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. The impact of migration on parallel job scheduling for distributed systems. In *Lecture Notes in Computer Science 1900*, volume 6th International Euro-Par Conference, pages 242–251, Aug/Sep 2000.

[121] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneo us Distributed Computer Systems. *Software – Practice and Experience*, 23(12):1305–1336, December 1993.

Appendices

Appendix A

Algorithms

A.1 Ad-hoc Search Procedure

1. Determine the amount of physical memory that is needed for the current problem.
2. If possible, find a fastest machine in the coarse grid that has the memory needed to solve the problem.
3. For each cluster in the coarse grid:
   (a) Find a machine in the cluster that has the maximum average bandwidth relative to the other machines in the grid. Add this to the fine grid.
   (b) Do the following:
      i. Find the next machine in the coarse grid that has maximum average bandwidth relative to the machines that are already in the fine grid.

ii. Calculate the new time estimate for the application by passing the machines in the fine grid to the execution model.

iii. Repeat Step 3.b.i until the time estimate for the application run time increases.

4. Compare the different fine grids obtained in Step 3 and choose the fine grid with the lowest time estimate.

5. If a single machine was found in Step 2, the time estimate for the problem using the single machine is compared with the time estimate for the problem using the machines found in Step 4. If the time estimate for the machine found in Step 2 is less than the time estimate for the machines found in Step 4, then use the single machine in Step 2 for the fine grid. Else use the machines found in Step 4 for the fine grid. If Step 2 was not able to find a single machine that is able to solve the problem, then the machines found in Step 4 are used for the fine grid.

A.2 Calculation of Approximate Remaining Execution Time of an Executing Application

BEGIN COMMENTS

total_pct - total percentage completion time

pct - percentage completion time retrieved from the Database Manager

time_diff - difference between current global time and the start time of the application pred_time - predicted execution cost for the application

avg_ratio - average of all ratios between the actual and the predicted cost for the application obtained from the Contract Monitor

ret - remaining execution time of the application

last_5_avg - average of the last 5 ratios between the actual and predicted cost of the application

END COMMENTS

$$total\_pct = pct + \frac{time\_diff \times 100.0}{pred\_time \times avg\_ratio} \tag{A.1}$$

$$ret = \frac{(100.0 - total\_pct) \times (pred\_time * avg\_ratio)}{100.0} \tag{A.2}$$

BEGIN COMMENTS

The above is the remaining execution time obtained assuming the ratio between the actual and predicted execution cost to be avg_ratio. A further approximation is assuming the ratio to be average of last 5 ratios of actual and predicted costs.

END COMMENTS

$$ret = \frac{ret \times last\_5\_avg}{avg\_ratio} \tag{A.3}$$

A.3 Algorithm for Permission Service

```
Permission Service:


Input: problemParameters, resourceRequirements, resourceInformation


  resourceCapacity = getCapacity( resourceInformation )

  if (resourceCapacity > resourceRequirements)

    send (PERMISSION)

  else


    if (no currently running resource consuming applications)

      send (NO_PERMISSION)

    else

      shortRemainingTimeApplications =

         resource consuming applications that are going to end in

         5 minutes


      if (such applications exist) then

        waitForCompletion (shortApplication)

        send(PERMISSION)
```

```
else
  ratio = remaining execution time of big application /
          predicted execution time of new application in the
          absence of big application

  maxRatio = maximum value of ratio for all the big
             applications

  if (maxRatio >20 and bigApplication == checkpointable)
    stop bigApplication
    send permission to new application
    wait for new application to complete
    resume big application
```

A.4 Algorithm for Contract Negotiator

```
Contract Negotiator:


Input: problemParameters, finalListOfMachines, predictedTime


  rsTime = time of resource selection of newApplication

  executingList = getListOfExecutingApplications()

  if (rsTime < minimum starting time of applications in executingList)

    send (CONTRACT_NOT_OK)

  else

    for each application i in executingList


      timeAbs = predicted time of new application in the absence of i

      timePre = predicted time of new application in the presence of i


      if (timePre/timeAbs > 1.5)

        bigApplication = i

        break out of loop


    remainingExecAbs = remaining execution time of bigApplication in

                         the absence of new application
```

```
remainingExecPre = remaining execution time of bigApplication in
                     the presence of new application
impactTime = impact of current application on bigApplication



if (bigApplication is checkpointable and
    2*timeAbs < 0.5*min(remainingExecTime+timeAbs,
                        impactTime + timePre))
  stop bigApplication;
  send (CONTRACT_NOT_OK) to new application
  wait for the new application to complete
  resume bigApplication

else if ((impactTime + timePre) > 1.2*(remainingExecTime+timeAbs))

  Continue bigApplication
  Wait for bigApplication to complete
  Send (CONTRACT_NOT_OK) to new application
```

```
forever repeat /* never ending loop */

  for each application i in executingList

    percentIncreaseInImpact_i = increase in impact due to addition

                                of current application

  end for


  maxPercent = max(percentIncreaseInImpact_i)

  if(maxPercent > 30)

    new_resources = original_resources for current application -

                    a single resource

    current_predicted_time = predicted execution cost of current

                             application with new_resources

    pred_cost_increase = current_predicted_time /

                         original_predicted_time


    if (pred_cost_increase > 2*maxPercent)

      break out of the never ending loop


send CONTRACT_OK to the current application
```

A.5 Algorithm for Rescheduler

```
if (a Contract Monitor has requested for rescheduling the
   end application)

  resources_new =  new resource characteristics from NWS.
  new_schedule = application-level schedule with
                 resources_new
  ret_current = remaining execution when the application
                continues execution on the original set
                of resources.
  ret_new = remaining execution when the application
            continues on new_schedule

  rescheduling_gain = (ret_current -
                      (ret_new+rescheduling_cost)) /
                      ret_current

  if (rescheduling_gain > gain_limit)
    stop application
    continue application on new_schedule
```

```
else
  Query the data base manager for recently completed
  applications.
  if (some applications have completed)
    executingList = list of executing applications
    for each application i in executingList
      reschedulingGain = (remaining execution time of i
                          without rescheduling -
                (remaining execution time of i with
                          rescheduling + rescheduling time)) /
                          current remaining exec time
      maxReschedulingGain = maximum of rescheduling gains
      maxApplication = application that has maximum
                       rescheduling gain
    if (maxReschedulingGain > 0.5)
      stop maxApplication
      get new candidate schedule for maxApplication
      continue maxApplication on new set of resources
```

Appendix B

SRS Library Reference

B.1 SRS_Init

C - void SRS_Init()

Fortran - SRS_INIT()

This function is called at the beginning of user's application code. Should be called after MPI_Init().

B.2 SRS_Finish

C - void SRS_Finish()

Fortran - SRS_FINISH()

This function is called at the end of the user's application code. Should be called before MPI_Finalize().

B.3 SRS_Restart_Value

C - int SRS_Restart_Value()

Fortran - SRS_RESTART_VALUE(RESTART_VALUE)

INTEGER RESTART_VALUE

This function is used to indicate if the program is executed under the start or restart mode.

Return values:

0 - Program started for the first time

1 - Program restarted

The user uses these values for conditional execution of certain statements of his application. For e.g., if the application uses a matrix and the matrix will be checkpointed when the application is stopped in the middle of its execution, then the matrix needs to be initialized with initial values only when the application is executed for the first time.

```
int* matrix;
int restart_value;
matrix = (int*)malloc(sizeof(int)*10);
restart_value = SRS_Restart_Value();
if(restart_value == 0){
  for(i=0; i<10; i++){
    matrix[i] = i;
  }
}
```

B.4 SRS_Check_Stop

C - int SRS_Check_Stop()

Fortran - SRS_CHECK_STOP(STOP_VALUE)

INTEGER STOP_VALUE

This function returns 1 when the application has to be stopped and 0 otherwise. The application periodically calls this function to check if it has to stop. When the value returned by SRS_Check_Stop() is 1, the application executes any application specific exit statements and stops. This function performs the actual checkpointing of data when the return value is 1.

Example Usage

```
int main(){

int stop_value;

int *a;

  stop_value = SRS_Check_Stop();

  if(stop_value == 1){

    /* perform exit */

    /* do data cleanup */

    free(a);

    MPI_Finalize();

    exit(0);

  }

}
```

B.5 SRS_Register

C - int SRS_Register(char* name, void* data, int data_type, int size,

int distribute_handle, void* distribute_data)

Fortran - SRS_REGISTER(NAME, DATA, DATATYPE, SIZE,

DISTRIBUTION_HANDLE, DISTRIBUTION_DATA)

CHARACTER* NAME

<type> **DATA, DISTRIBUTION_DATA**

INTERGER DATATYPE, SIZE, DISTRIBUTION_HANDLE

This function allows the user to "mark" the data that will be checkpointed when the application is stopped. Only those data that are marked by SRS_register() will be checkpointed when the application is stopped. The function just adds the data to the list of data to be checkpointed. It does not do the actual checkpointing.

name **- a string containing less than 30 characters used to represent the data. The user can use any arbitrary string name**

data **- pointer to data containing** *size* **data elements of data type** *data_type*

data_type **- data type of the elements of** *data*. **The data_type can be one of GRADS_INT for integers, GRADS_FLOAT for single precision reals, GRADS_DOUBLE for double precision reals, GRADS_CHAR for characters and GRADS_BYTE for bytes.**

distribute_handle **- can be one of BLOCK, BLOCKCYCLIC, CYCLIC or 0. The handles returned by SRS_DistributeFunc_Create and SRS_DistributeMap_Create can also be used.**

distribute_data **- any specific information needed by the distribution function. For e.g., for BLOCKCYCLIC, a pointer to the block size is passed as distribute_data. NULL is used if the distribution function does not need any specific information.**

Return values:

0 - on success

-1 - on failure

Example Usage

```
int main(){
double x[10];
int i;
  SRS_Register("X", x, GRADS_DOUBLE, 10, BLOCK, NULL);
  SRS_Register("iterator", &i, GRADS_INT, 1, 0, NULL);
}
```

B.6 SRS_Read

C - int SRS_Read(char* name, void* data, int distribute_handle, void* distribute_data)

Fortran - SRS_READ(NAME, DATA, DISTRIBUTION_HANDLE, DISTRIBUTION_DATA)

CHARACTER* NAME

<type> DATA, DISTRIBUTION_DATA

INTERGER DISTRIBUTION_HANDLE

SRS_Read() is called to read the data that was previously checkpointed.

Hence this function should be called only in the restart mode. The first argument is a character string of less than 30 characters used to identify the data. The *name* **is the same string that that was used in the SRS_Register() function in the previous application run for checkpointing the data.** *data* **is a pointer to the application's address space into which the checkpointed data will be read.**

distribute_handle **can be one of BLOCK, BLOCKCYCLIC, CYCLIC or 0. The handles returned by SRS_DistributeFunc_Create and SRS_DistributeMap_Create can also be used. Following are the meanings of the various handles.**

0 - When 0 is used for *new_distribute_handle*, **the data distributed over the set of processes in the previous application run is copied one-one over the corresponding set of processes in the current application run.**

SAME - When SAME is used, the data stored by process 0 in the old application run is copied to all the processes in the new application run. This is useful for storing and retrieving iterator values.

BLOCK - This specifies 1-d block distribution.

CYCLIC - This specifies 1-d cyclic distribution.

BLOCKCYCLIC - This specifies 1-d block cyclic distribution.

distribute_data **- any specific information needed by the distribution function. For e.g., for BLOCKCYCLIC, a pointer to the block size is passed as**

distribute_data. NULL is used if the distribution function does not need any specific information.

Example Usages

In the following examples, only partial code statements are shown to demonstrate SRS_Read() call.

Example 1: This is a simple example in which an array of integers are copied from the set of processes in the old application to the corresponding set of processes in the new application run.

```
int main(){
int A[10];
int i;
  SRS_Init();
  restart_value = SRS_Restart_Value();
  if(restart_value == 1){
    SRS_Read("A", A, 0, NULL);
  }
  SRS_Register("A", A, GRADS_INT, 10, 0, NULL);
}
```

Example 2: In this example, block-cyclic data distribution is used for both old and new application runs. Thus the same data is distributed in

a block cyclic fashion over a new set of processes when the application is restarted. Unlike Example 1, this example can be stopped and restarted on a different set of processes.

```
int main(){
int A[10];
int i;
  SRS_Init();
  restart_value = SRS_Restart_Value();
  if(restart_value == 1){
    SRS_Read("A", A, BLOCKCYCLIC, NULL);
  }
  SRS_Register("A", A, GRADS_INT, 10, BLOCKCYCLIC, NULL);
}
```

Example 3: This example demonstrates the use of SAME value for new distribution in SRS_Read(). In this example, SAME is used for propagating the checkpointed iterator to all the processes so that all the processes in the current application run can start from the same iteration.

```
int main(){
int i, iter_start;
  SRS_Init();
  restart_value = SRS_Restart_Value();
  if(restart_value == 1){
    SRS_Read("iterator", &iter_start, SAME, NULL);
  }
  SRS_Register("iterator", &i, GRADS_INT, 1, 0, NULL);
  for(i=iter_start; i<10; i++){
  }
}
```

B.7 SRS_StoreMap

C - void SRS_StoreMap()

Fortran - SRS_STOREMAP()

SRS_StoreMap is called by the user to store the data maps corresponding to the data distributions of the data passed previously in SRS_Register. This is useful if an external component wants to know the data distributions used by the executing application.

B.8 SRS_DistributeFunc_Create

C - int SRS_DistributeFunc_Create(DataMapInfo* (*distribute_func)(int data_size, int proc_count, void* other_info, char* input_arg), int* handle)

SRS_Read() and SRS_Register require handles to data distributions. When the data distribution is specified by means of a function, the handles can be created by SRS_DistributeFunc_Create(). The first argument *distribute_func* **is a pointer to a function that constructs a data map. The function accepts 4 arguments.**

data_size **- total number of elements of data of data type GRADS_INT, GRADS_FLOAT, GRADS_DOUBLE, GRADS_CHAR, GRADS_BYTE**

proc_count **- total number of processes**

other_info **- any other information needed for the data distribution.**

input_arg **- encoding of** *other_info* **returned by the function.**

The function returns a pointer to a structure called *DataMapInfo* **whose specification is given below.**

```
typedef struct{
    int info_count;
    int* offset;
    int* size;
    int* proc;
    } DataMapInfo;
```

This structure is used for specifying the data map used for the data. offset, size and proc are arrays. Each <*offset, size, proc*> triple contains information about a particular data panel.

info_count **- number of elements in offset, size and proc arrays.**

offset **- offset[i] contains the global offset in terms of the data types GRADS_INT, GRADS_DOUBLE, GRADS_FLOAT, GRADS_CHAR, GRADS_BYTE of the data panel i. The elements in the offset array should be sorted by ascending order.**

size **- size[i] contains the number of elements of data type of either GRADS_INT, GRADS_DOUBLE, GRADS_FLOAT, GRADS_CHAR or GRADS_BYTE of data panel i.**

proc **- proc[i] contains the process number that holds the data panel i.**

The 2nd argument of SRS_DistributeFunc_Create() is the handle that is set by the SRS_DistributeFunc_Create(). This handle will be used by SRS_Read() and SRS_Register().

Example Usage

In this example, SRS_DistributeFunc_Create() is used to create a handle to a block data distribution. This handle is used in the subsequent SRS_Register().

```
DataMapInfo* block_distribution(int data_size, int proc_count, void*
                              other_data, char* input_arg){
int i, total_offset;
DataMapInfo* data_map;
  data_map = (DataMapInfo*)malloc(sizeof(DataMapInfo));
  data_map->info_count = proc_count;
  data_map->offset = (int*)malloc(sizeof(int)*proc_count);
  data_map->size = (int*)malloc(sizeof(int)*proc_count);
  data_map->proc = (int*)malloc(sizeof(int)*proc_count);
  total_offset = 0;
  for(i=0; i<proc_count; i++){
    data_map->offset[i] = total_offset;
    data_map->size[i] = data_size/proc_count +
                        ((data_size % proc_count) > i);
    data_map->proc[i] = i;
    total_offset += data_map->size[i];
  }
  input_arg = NULL;
  return data_map;
}
```

```
int main(){

int A[10];

int restart_value;

int distributefunc_handle;

DataMapInfo* (*distribute_func)(int, int , void*, char*);

  MPI_Init();

  SRS_Init();

  restart_value = SRS_Restart_Value();

  distribute_func = block_distribution;

  SRS_DistributeFunc_Create(distribute_func, &distributefunc_handle);

  SRS_Register(‘‘A’’, A, GRADS_INT, 10, distributefunc_handle, NULL);

  SRS_Finish();

  MPI_Finalize();

}
```

B.9 SRS_DistributeMap_Create

C - int SRS_DistributeMap_Create(DataMapInfo* dataMap, int* handle)

SRS_DistributeMap_Create() is one method of creating handle to a data map needed by SRS_Register() and SRS_Read(). The first argument *dataMap*

is a pointer to a structure called DataMapInfo whose specification is given below.

```
typedef struct{
    int info_count;
    int* offset;
    int* size;
    int* proc;
    } DataMapInfo;
```

This structure is used for specifying the data map used for the data. offset, size and proc are arrays. Each *<offset, size, proc>* **triple contains information about a particular data panel.**

info_count **- number of elements in offset, size and proc arrays.**

offset **- offset[i] contains the global offset in terms of the data types GRADS_INT, GRADS_DOUBLE, GRADS_FLOAT, GRADS_CHAR, GRADS_BYTE of the data panel i. The elements in the offset array should be sorted by ascending order.**

size **- size[i] contains the number of elements of data type of either GRADS_INT, GRADS_DOUBLE, GRADS_FLOAT, GRADS_CHAR or GRADS_BYTE of data panel i.**

proc **- proc[i] contains the process number that holds the data panel i.**

The 2nd argument of SRS_DistributeMap_Create() is the handle that

is set by the SRS_DistributeMap_Create(). This handle will be used by SRS_Read() and SRS_Register().

Example

In this example, the block cyclic data distribution is constructed using the data map structure and a handle is created using SRS_DistributeMap_Create(). This handle is used in the subsequent SRS_Register() call.

```
int main(){

int A[10];

int handle;

int restart_value;

  MPI_Init();

  SRS_Init();

  restart_value = SRS_Restart_Value();

  dataMap = (DataMapInfo*)malloc(sizeof(DataMapInfo));

  dataMap->info_count = 5;

  dataMap->offset = (int*)malloc(sizeof(int)*5);

  dataMap->size = (int*)malloc(sizeof(int)*5);

  dataMap->proc = (int*)malloc(sizeof(int)*5);

  dataMap->offset[0] = 0;

  dataMap->size[0] = 2;

  dataMap->proc[0] = 0;

  dataMap->offset[1] = 2;

  dataMap->size[1] = 2;

  dataMap->proc[1] = 1;

  dataMap->offset[2] = 4;

  dataMap->size[2] = 2;

  dataMap->proc[2] = 2;

  dataMap->offset[3] = 6;
```

```
  dataMap->size[3] = 2;

  dataMap->proc[3] = 0;

  dataMap->offset[4] = 8;

  dataMap->size[4] = 2;

  dataMap->proc[4] = 1;

  SRS_DistributeMap_Create(dataMap, &handle);

  SRS_Register(``A'', A, GRADS_INT, 10, handle, NULL);

  SRS_Finish();

  MPI_Finalize();

}
```

B.10 The big picture - A working example

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include "mpi.h"

#include "srs.h"

#include "datatype.h"

int main(int argc, char** argv){

int* global_A;

int* local_A;

int rank, size;

int global_size, local_size;

int proc_number, local_index;

int i, j, iter_start, restart_value, stop_value;

MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);

  SRS_Init();

  MPI_Comm_rank(comm, &rank);

  MPI_Comm_size(comm, &size);

  global_size = atoi(argv[1]);
```

```
local_size = global_size/size;
restart_value = SRS_Restart_Value();
global_A = (int*)malloc(sizeof(int)*global_size);
local_A = (int*)malloc(sizeof(int)*local_size);
if(restart_value == 0){
  if(rank == 0){
    for(i=0; i<global_size; i++){
      global_A[i] = i;
    }
  }
  MPI_Scatter (global_A, local_size, MPI_INT, local_A, local_size,
               MPI_INT, 0, comm );
  iter_start = 0;
}
else{
  SRS_Read("A", local_A, BLOCK, NULL);
  SRS_Read("iterator", &iter_start, SAME, NULL);
}
SRS_Register("A", local_A, GRADS_INT, local_size, BLOCK, NULL);
SRS_Register("iterator", &i, GRADS_INT, 1, 0, NULL);
```

```
printf("Proc. %d initial: ", rank);
for(j=0; j<local_size; j++){
  printf("%d ", local_A[j]);
}
printf("\n");
for(i=iter_start; i<global_size; i++){
  stop_value = SRS_Check_Stop();
  if(stop_value == 1){
    free(global_A);
    free(local_A);
    MPI_Finalize();
    exit(0);
  }
  proc_number = i/local_size;
  local_index = i%local_size;
  if(rank == proc_number){
    local_A[local_index] += 10;
  }
  printf("Proc. %d Iter. %d: ", rank, i);
  for(j=0; j<local_size; j++){
```

```
(60)        printf("%d ", local_A[j]);
(61)      }
(62)      printf("\n");
(63)      sleep(1);
(64)    }
(65)    free(global_A);
(66)    free(local_A);
(67)    SRS_Finish();
(68)    MPI_Finalize();
(69)    exit(0);
(70)  }
```

In this example, an array A whose size is divisible by the number of processors is evenly distributed across all the processors. When the application is started for the first time (line 24), the root process initializes the array (lines 25-29) and distributes sub arrays to all the processors using MPI_Scatter (lines 30-31). Each process executes a loop whose number of iterations is equal to the size of the array (lines 45-64). In each iteration of the loop, a single element of the array whose array index is given by the iteration number, is incremented by 10. This increment is carried by the processor that owns the element (lines 55-57).

Each process registers its subarray and the iteration number for check-

pointing (lines 38-39). At the start of each iteration of the loop, each process calls SRS_Check_Stop() to check if the application has to stop (line 46). If the application has received a stop signal, each process frees the allocated arrays and calls MPI_Finalize() and exit() (lines 47-52).

When the application is restarted, each process reads its portion of the array and the array is once again distributed in a block fashion (line 35). Each process also reads the iteration number from which it has to continue. Since all the processes have to continue from the same iteration, SAME is used for SRS_Read() (line 36). Thus this example can be started on m number of processors, stopped and can be restarted on n number of processors where n can be different from m. The only requirement for this example is that the size of the array should be divisible by m and n. The program can be stopped and restarted on different sets of processors any number of times. At the end of program completion, the unique correct values of the array, which are 10 - (size of the array-1)+10, are displayed.

Vita

Sathish Vadhiyar was born in Madras, India on November 16, 1975. He completed his high school in 1991 after which he joined Thiagarajar College of Engineering, Madurai, India for pursuing his undergraduate degree. He obtained his Bachelor of Engineering in Computer Science and Engineering in 1997. After then, he traveled to United States to pursue graduate studies. He obtained his Master's degree in Computer Science from Clemson, University, South Carolina in 1999. After graduation, he applied and joined the Doctorate program in Computer Science in University of Tennessee in May, 1999.

During the course of his Doctorate program, he worked as a Graduate Research Assistant in the Innovative Computing Laboratory (ICL) under the guidance of Dr. Jack Dongarra. During this period, he traveled to many high profile conferences and presented various research papers. He was also involved in GrADS, a multi-institutional research project. His current research interests include parallel, distributed and Grid computing. Sathish Vadhiyar is expected to receive his Doctoral degree in May 2003.