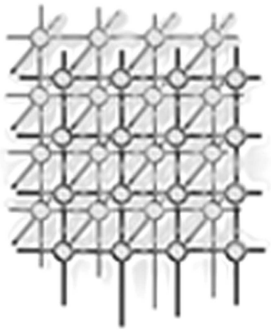
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **00**:1–26 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Self Adaptivity in Grid Computing



Sathish S. Vadhiyar[1,*] and Jack J. Dongarra[2,3]

[1] *Supercomputer Education and Research Centre*
Indian Institute of Science
Bangalore - 560012
India
[2] *Computer Science Department*
University of Tennessee
Knoxville, TN 37996-3450
USA
[2] *Computer Science and Mathematics*
Oak Ridge National Laboratory
USA

SUMMARY

Optimizing a given software system to exploit the features of the underlying system has been an area of research for many years. Recently, a number of self-adapting software systems have been designed and developed for various computing environments. In this paper, we discuss the design and implementation of a software system that dynamically adjusts the parallelism of applications executing on computational Grids in accordance with the changing load characteristics of the underlying resources. The migration framework implemented by our software system is aimed at performance-oriented Grid systems and implements tightly coupled policies for both suspension and migration of executing applications. The suspension and migration policies consider both the load changes on systems as well as the remaining execution times of the applications thereby taking into account both system load and application characteristics. The main goal of our migration framework is to improve the response times for individual applications. We also present some results that demonstrate the usefulness of our migration framework.

KEY WORDS: self adaptivity; migration; GrADS; rescheduling; redistribution; checkpointing

*Correspondence to: vss@serc.iisc.ernet.in
Contract/grant sponsor: This research was supported in part by the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC. and in part based on work supported by the National Science Foundation under Grant No. ACI 0103759.

Received 00
Revised 00
Accepted 00

1. Introduction

Optimization of software routines for achieving efficiency on a given computational environment has been an active area of research. Historically, the optimization was achieved by hand-tuning the software system to fit the needs of the computing environment. Although high optimization can be achieved, this process was found to be tedious and needs considerable scientific expertise. Also, the hand-tuning process was not portable across different computing environments. Finally, hand customization does not take into account the run-time load dynamics of the system and the input parameters of the application.

The solution to the above mentioned problems associated with hand-tuning software routines for the computing environment is to build *self-adaptive software system* that examines the characteristics of the computing environments and chooses the software parameters needed to achieve high efficiency on that environment. Recently, a number of self-adaptive software systems have been designed and implemented [14, 32, 29, 31, 8, 4]. Some of the software systems apply adaptivity to the computational processors [14, 32], some are tuned for communication networks [29], some are intended for workstation clusters [8] and some have been developed for computational Grids [4]. The various adaptive software systems also differ in the time when adaptivity is performed. Some perform adaptivity at installation time [32, 29, 31], while others perform adaptivity at run time [8, 4].

There are very few self-adaptive software systems that dynamically adapt to changes in the load characteristics of the resources on computational Grids. Computational Grids [12] involve large resource dynamics, so 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. In this paper, we focus on the migration of applications executing on distributed and Grid systems in order to adapt to the load dynamics of the resources.

There are at least two disadvantages in using the existing migration frameworks [20, 10, 18, 30, 35, 15] for adapting to load dynamics. First, due to separate policies employed by these migration systems for suspension of executing applications and migration of the applications to different systems, applications can incur lengthy waiting times between when they are suspended and when they are restarted on new systems. Second, due to the use of predefined 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 may lead to frequent satisfaction of the predefined conditions and hence could lead to frequent invocations of suspension and migration decisions.

In this paper, we describe a framework that defines and implements scheduling policies for migrating applications executing on distributed and Grid systems in response to varying resource load dynamics. In our framework, the migration of applications depends on

1. the amount of increase or decrease in loads on the resources,
2. the point during the application execution lifetime when load is introduced into the system,

3. the performance benefits that can be obtained for the application due to migration.

Thus, our migrating framework takes into account both the load and application characteristics. The policies are implemented in such a way that the executing applications are only suspended and migrated when better systems are found for application execution thereby invoking the migration decisions as infrequently as possible. Our migration framework is primarily intended for rescheduling long running applications. The migration of applications in our migration framework is dependent on the ability to predict the remaining execution times of the applications, which in turn is dependent on the presence of execution models that predict the total execution cost of the applications. The framework has been implemented and tested in the GrADS system [4]. Our test results indicate that our migration framework can help improve the performance of executing applications by more than 30%. In this paper, we present some of the descriptions and results from our earlier work [28] and also present new experiments regarding dynamic determination of rescheduling cost.

In Section 2, we present a general overview of self-adaptive software systems by describing some systems that perform adaptivity. In Section 3, we describe the GrADS system and the life cycle of GrADS applications. In Section 4, we introduce our migration framework by describing the different components in the framework. In Section 5, we describe the API of the checkpointing library used in our migration framework. In Section 6, the various policies regarding rescheduling are dealt with. In Section 7, other issues relevant to migration are described in brief. In Section 8, we describe our experiments and provide various results. In Section 9, we present related work in the field of migration. We give concluding remarks and explain our future plans in Section 10.

2. Self Adaptive Software Systems - An Overview

Recently, there have been a number of efforts in designing and developing self-adaptive software systems. These system differ in terms of the kind of computational environments, the kind of adaptive software system used and also the time when adaptivity is performed. The following subsections describe some illustrative examples.

2.1. ATLAS

ATLAS [32] stands for Automatically Tuned Linear Algebra Software. ATLAS exploits cache locality to provide highly efficient implementations of BLAS (Basic Linear Algebra Subroutine) and few LAPACK routines. During installation, ATLAS studies various characteristics of the hardware including the size of the cache, the number of floating point units in the machine and the pipeline length to determine the optimal or near-optimal block size for the dense matrices, the number of loop unrollings to perform, the kind of instruction sets to use etc. Thus, optimizations are performed for reducing the number of accesses to main memory and reduce loop overheads resulting in BLAS implementations that are competitive with the machine-specific versions of most known architectures.

2.2. ATCC

ATCC [29] (Automatically Tuned Collective Communications) is intended for optimizing MPI [26, 1] collective communications for a given set of machines connected by networks of specific configurations. The collective communication routines form integral parts of most of the MPI-based parallel applications. During installation, ATCC conducts experiments for different algorithms and segment sizes for different collective communications, number of processors and message sizes. ATCC then gathers the times for individual experiments in a look-up table. When the user invokes a collective communication routine with a given message size and a given number of processors, ATCC looks up the table and chooses the best collective communication algorithm and segment size for communication. Recent versions of ATCC include performance models for collective communication algorithms to reduce the time taken for conducting actual experiments.

2.3. BeBOP

The BeBOP project from Berkeley attempts to optimize sparse matrix kernels, namely, matrix-vector multiplication, triangular solve and matrix triple product for a given architecture. For each of the sparse matrix kernels, the BeBOP project considers a set of implementations and chooses the optimal or near-optimal implementation for a given architecture. Given a sparse matrix, machine, and kernel, the BeBOP approach in choosing an implementation consists of two steps. First, the possible implementations are benchmarked off-line in a matrix independent, machine dependent way. When the matrix structure is known during runtime, the matrix is sampled to extract relevant aspects of its structure, and performance models that combine the benchmark data and the estimated matrix properties are evaluated to obtain the near-optimal implementation. The BeBOP [31] approach has been successfully applied to optimize register blocking for sparse matrix-vector multiplication.

2.4. LFC

The LFC (LAPACK for Clusters) project [8] aims to simplify the use of parallel linear algebra software on computational clusters. Benchmark results are obtained for sequential kernels that are invoked by the parallel software. During runtime, adaptivity is performed by taking into account the resource characteristics of the computational machines and an optimal or near-optimal choice of a subset of resources for the execution of the parallel application is made by the employment of scheduling algorithms. LFC also optimizes the parameters of the problem, namely the block size of the matrix. LFC is intended for the remote invocation of parallel software from a sequential environment and hence employs data movement strategies. The LFC approach has been successfully used for solving ScaLAPACK LU, QR and Cholesky factorization routines.

3. The GrADS System

GrADS (Grid Application Development Software) [4] is an ongoing research project involving a number of institutions and its goal is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. GrADS approach is similar to the LFC approach, but more suited to Grid computing due to the employment of Grid computing tools. The University of Tennessee investigates issues regarding integration of numerical libraries in the GrADS system. In our previous work [22], we demonstrated the ease with which numerical libraries like ScaLAPACK can be integrated into the Grid system and the ease with which the libraries can be used over the Grid. We also showed some results to prove the usefulness of a Grid in solving large numerical problems.

In the architecture of GrADS, the user wanting to solve a numerical application over the Grid invokes the GrADS application manager. The life cycle of the GrADS application manager is shown in Figure 1.

As a first step, the application manager invokes a component called Resource Selector. The Resource Selector accesses the Globus Monitoring and Discovery Service(MDS) [11] to obtain a list of machines in the GrADS testbed that are available and then contacts the Network Weather Service(NWS) [34] to retrieve system information for the machines. The application manager then invokes a component called Performance Modeler with problem parameters, machines and machine information. The Performance Modeler, using an execution model built specifically for the application, determines the final list of machines for application execution. By employing an application specific execution model, GrADS follows the AppLeS [5] approach to scheduling. The problem parameters and the final list of machines are passed as a contract to a component called Contract Developer. The Contract Developer may either approve or reject the contract. If the contract is rejected, the application manager develops a new contract by starting from the resource selection phase again. If the contract is approved, the application manager 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 Contract Monitor through an Autopilot mechanism [24] monitors the times taken for different parts of applications. The GrADS architecture also has a GrADS Information Repository(GIR) that maintains the different states of the application manager and the states of the numerical application. 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 external intervention. These application states are passed to the application manager through the GIR. 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 the state of the end application to change to "RESUME" and then collects new machine information by starting from the resource selection phase again.

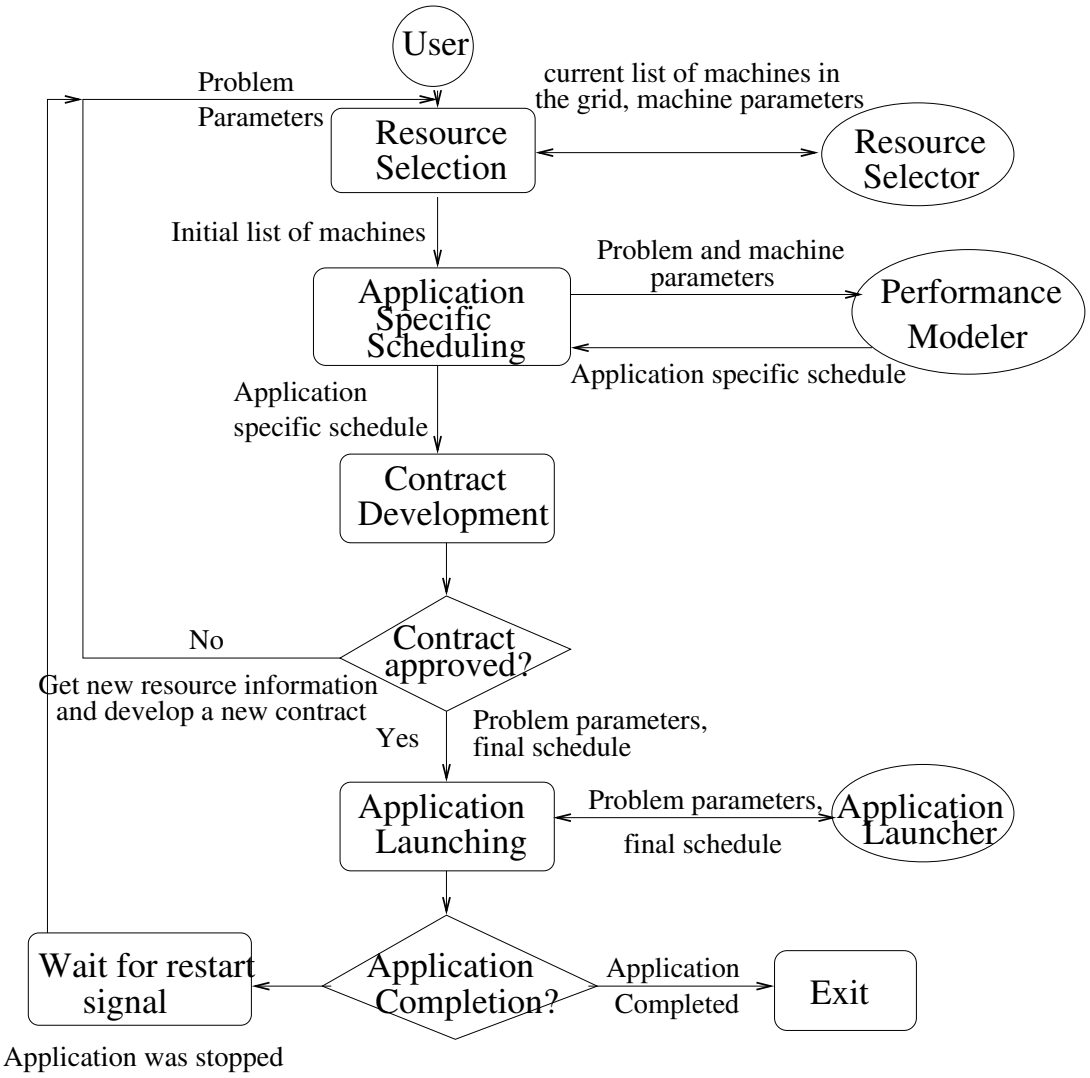


Figure 1. GrADS application manager

4. The Migration Framework and Self Adaptivity

Though the GrADS architecture explained in the previous section has provisions for continuing an end application after the application was stopped, it lacks components that perform the actual stopping of the executing end application and informing the application manager of the various states of the end application. Hence, the GrADS architecture as described in the previous section does not adapt the executing application to the changing resource characteristics once the application is committed to a set of resources. It is highly desirable to adapt and migrate the application to a different set of resources if the resources on which the application is executing do not meet the performance criteria. The ability to migrate applications in the GrADS system is implemented by adding a 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 among the different components involved in the migration framework is illustrated in Figure 2. These components are described in detail in the following subsections.

4.1. Migrator

A user-level checkpointing library called SRS (**S**top **R**estart **S**oftware) is used to provide migration capability to the end application. The application, by making calls to the SRS API, achieves 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 on a new configuration of resources. Although the method of rescheduling in SRS, by stopping and restarting executing applications, incurs more overhead than process migration techniques [6, 7, 27] where a single process or a set of processes of the application is either migrated to another processor or replaced by a set of processes, the approach followed by SRS allows reconfiguration of executing applications and achieves portability across different MPI implementations, particularly MPICH-G [13], a popular MPI implementation for Grid computing. The SRS library uses Internet Backplane Protocol(IBP) [23] for storage of the checkpoint data. IBP storage depots are started on all the machines in the GrADS testbed.

The application launcher, apart from launching the end application and the contract monitor, also launches a component called RSS (Runtime Support System). RSS is included as part of the SRS checkpointing package. An external component (e.g., the rescheduler) wanting to stop an executing end application interacts with the RSS daemon. 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 daemon 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 GIR and interacts with RSS to perform various functions. These functions include initialization of certain data structures in

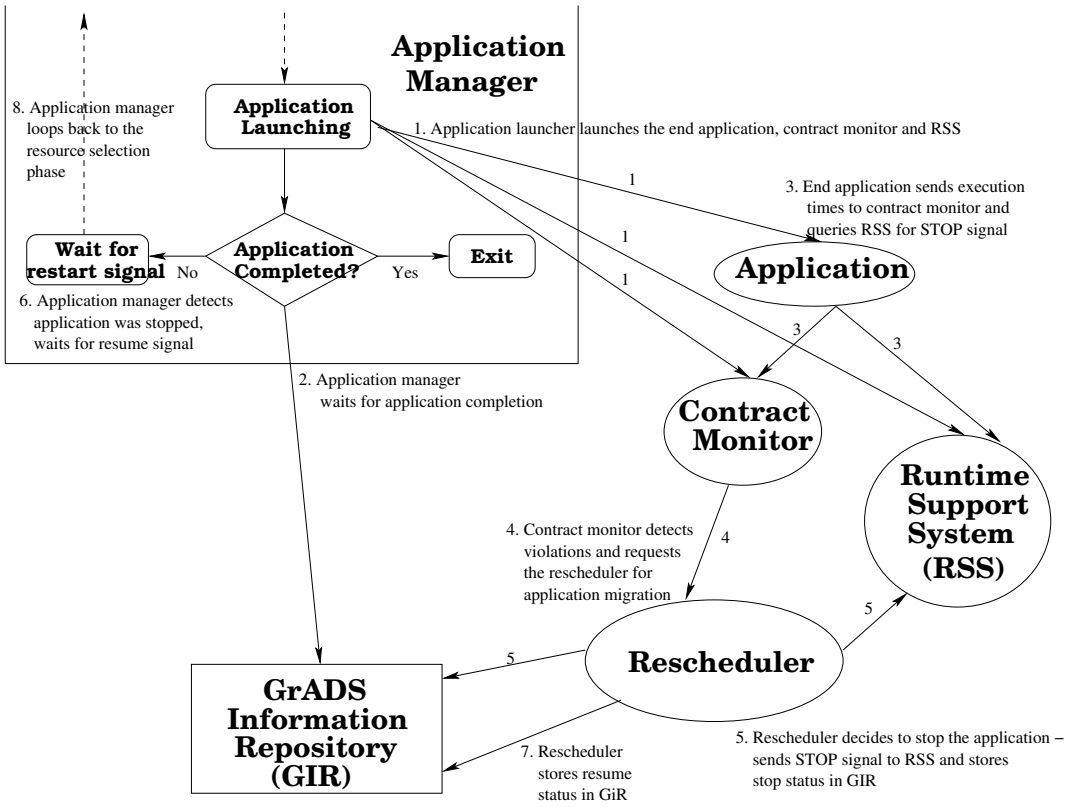


Figure 2. Interactions in Migration framework

the library, checking if the application needs to be stopped and storing and retrieving various information including pointers to the checkpointed data, processor configuration and data distribution used by the application. RSS is implemented as a threaded service that receives asynchronous requests from external components and the application.

4.2. Contract Monitor

The Contract Monitor is a component that uses the Autopilot infrastructure to monitor the progress of applications in GrADS. Autopilot[24] 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 send the execution times taken for the different phases of the application to the contract monitor. The contract monitor compares the actual execution

times with the predicted execution times. When the contract monitor detects large differences between the actual and the predicted performance of the end application, it contacts the rescheduler and requests it to migrate the application.

4.3. Rescheduler

Rescheduler is the 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 any contract monitor has not contacted the rescheduler for migration, the rescheduler periodically queries the GrADS Information Repository(GIR) for recently completed applications. If a GrADS application was recently completed, the rescheduler determines if performance benefits can be obtained for an currently executing application by migrating it to use the resources that were freed by the completed application. This is called opportunistic rescheduling.

5. The SRS API

The application interfaces for SRS look similar to CUMULVS [16], but unlike CUMULVS, SRS does not require a PVM virtual machine to be setup on the hosts. The SRS library consists of 6 main functions - SRS_Init(), SRS_Finish(), SRS_Restart_Value(), SRS_Check_Stop(), SRS_Register() and 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 should perform application-specific stop actions. There is no relationship between the locations of the SRS_Check_Stop() calls and the calls to extract the execution times of the different phases of application.

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. 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 are beyond the scope of the current discussion. For the current discussion, it is suffice to notice that the SRS library is generic and has been tested with numerical libraries like ScaLAPACK and PETSC.

6. Rescheduling Policies

6.1. Policies for Contacting the Rescheduler

The contract monitor calculates the ratios between the actual execution times and the predicted execution times of the application. The tolerance limits of the ratios 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 that the application be migrated. 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 temporarily on the machine and hence caused 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.

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 not only reduces the amount of communication between the contract monitor and the rescheduler but also hides the deficiencies in the application-specific execution time model.

6.2. Policies for Migration

For both *migration on request* and *opportunistic migration* modes, the rescheduler first contacts the Network Weather Service (NWS) to obtain 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 current total percentage completion time for the application and the predicted total 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*,

Table I. Times for rescheduling phases

Rescheduling Phase	*Time (secs.)*
Writing checkpoints	40
Waiting for NWS to update 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

the remaining execution time of the application if it were to continue executing on the original set of machines. The rescheduler then calculates the rescheduling gain as

$$rescheduling\_gain = \frac{(ret\_current - (ret\_new + 900))}{ret\_current}$$

The number 900 in the numerator of the fraction is the worst case time in seconds needed to reschedule the application. The various times involved in rescheduling are given in Table I. The times shown in Table I were obtained by conducting several experiments with different problem sizes and obtaining the maximum times for each phases of rescheduling. Thus, the rescheduling strategy adopts a pessimistic approach for rescheduling, with the result that migration of applications will be avoided in certain cases where the migration could yield performance benefits.

If the rescheduling gain is greater than 30%, the rescheduler sends stop signal to the RSS and hence to the executing application, and stores the "STOP" status in GIR. The application manager then waits for the state of the end application to change to "RESUME". After the application has stopped, the rescheduler stores "RESUME" as the state of the application in the GIR 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 rescheduler contacts the contract monitor prompting the contract monitor to adjust its tolerance limits.

The rescheduling threshold [33] 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 trial-and-error experiments using a range of different problem sizes for the different applications that were considered and for different configurations of the available resources, 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.

7. Other Migration Issues

The calculation of remaining execution and percentage completion times of the application forms the backbone of our rescheduling architecture and lends uniqueness to our approach when compared to other migration research efforts. The contract monitor, based on the actual and predicted execution times of the different phases of the executing application and the predicted execution time from the execution model, calculates the refined expected execution time of the application. Based on the current elapsed time and the refined expected time of the executing application, the total percentage completion time and the remaining execution time of the application is calculated by the rescheduler. When calculating the remaining execution time of the application on a new set of resources, the total predicted execution time from the execution model for the new set of resources is also taken into account. Though our approach of calculating the remaining execution and percentage completion times is most suitable for iterative applications, it can also be applied to other kinds of applications.

Also, in order to prevent possible conflicts between different applications due to rescheduling, the rescheduler is implemented as a single GrADS service that is contacted by the contract monitors of different applications. The rescheduler implements a queuing system and at any point in time services the request for a single application by contacting the corresponding application manager of that application. The stopping of the application by the rescheduler occurs in two steps. First, the external component contacts the RSS and sends a signal to stop the application. This stop signal occurs concurrently with application execution. When the application executes the next SRS_Check_Stop() call, it contacts the RSS, obtains the stop information from RSS and proceeds to stop.

8. Experiments and Results

The GrADS experimental testbed consists of about 40 machines that reside in institutions across United States including the University of Tennessee, the University of Illinois, the University of California at San Diego, Rice University etc. For the sake of clarity, our experimental testbed consists of two clusters, one in the University of Tennessee and another in the University of Illinois, Urbana-Champaign. The characteristics of the machines are given in Table II. The two clusters are connected by means of Internet. Although the Tennessee machines are dual-processor machines, the applications in the GrADS experiments use only one processor per machine.

About 5 applications, namely, ScaLAPACK LU and QR factorizations, ScaLAPACK eigenvalue problems, PETSC CG solver, and heat equation solver have been integrated into the migration framework by instrumenting the applications with SRS calls and developing performance models for the applications. In general, our migration framework is suitable for iterative MPI-based parallel applications for which performance models predicting the

Table II. Resource Characteristics

Cluster name	*Location*	*Nodes*	*Processor type*	*Speed (MHz)*	*Memory (MByte)*	*Network*	*Operating System*	*Globus Version*
msc	University of Tennessee	8	Pentium III	933	512	100 Mb switched Ethernet	Redhat Linux 7.3	2.2
opus	University of Illinois, Urbana-Champaign	16	Pentium II	450	256	1.28 Gbit/sec full duplex myrinet	Redhat Linux 7.2 (2.4.18 kernel)	2.2

execution costs can be written. In the experiments shown in this paper, ScaLAPACK QR factorization was used as the end application. Similar encouraging results were also obtained for other applications.

The performance model of ScaLAPACK QR factorization was derived by simulating the routine PDGEQRF. The simulation was based on benchmark performance of matrix multiplication and other basic linear algebra routines on the resource testbed and a prediction of communication costs for a given set of network links and for given message sizes. A more detailed description of the QR performance model is beyond the scope of this paper. For a general idea of the methodology, the reader is referred to earlier work [22]. The application was instrumented with calls to SRS library such that the application can be stopped by the rescheduler at any point of time and can be continued on a different configuration of machines. The data that was checkpointed by the SRS library for the application included the matrix, A and the right-hand side vector, B. Only the PDGEQRF routine and the driver routine for PDGEQRF were modified for instrumentation with SRS calls. The percentage increase in size of the code due to the modifications was less than 4%. Lower tolerance limit of 0.7 and upper tolerance limit of 2 were used as thresholds for the contract monitor. These thresholds were derived by conducting preliminary performance model validation tests on the testbed.

8.1. Migration on Request

In all the experiments in this section, 4 Tennessee and 8 Illinois machines were used. A given matrix size for the QR factorization problem was input to the application manager. For large problem sizes, the computation time dominates the communication time for the ScaLAPACK application. Since the Tennessee machines have higher computing power than the Illinois machines, the application manager by means of the performance modeler chose the 4 Tennessee machines for the end application run. A few minutes after the start of the end application, artificial load is introduced into the 4 Tennessee machines. This artificial load is achieved by executing a certain number of loading programs on each of the Tennessee

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 Illinois machines and either migrated the application or allowed the application to continue on the 4 Tennessee 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 is to force the opposite decision of whether or not to migrate. 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 decisions.

Three parameters were involved in each set of experiments - the size of the matrices, the amount of load on the resources 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 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 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 written to local storage. The rescheduling benefits are more for large problem sizes since the remaining lifetime of the end application when load is introduced is larger. 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 correct decisions 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 by using a worst-case rescheduling cost in the rescheduler will lead to underestimating the performance benefits due to rescheduling in some cases. We also observe from the figure that the times for reading checkpoints and data distribution do not necessarily increase linearly with increasing matrix sizes. For e.g., the time for data distribution is more

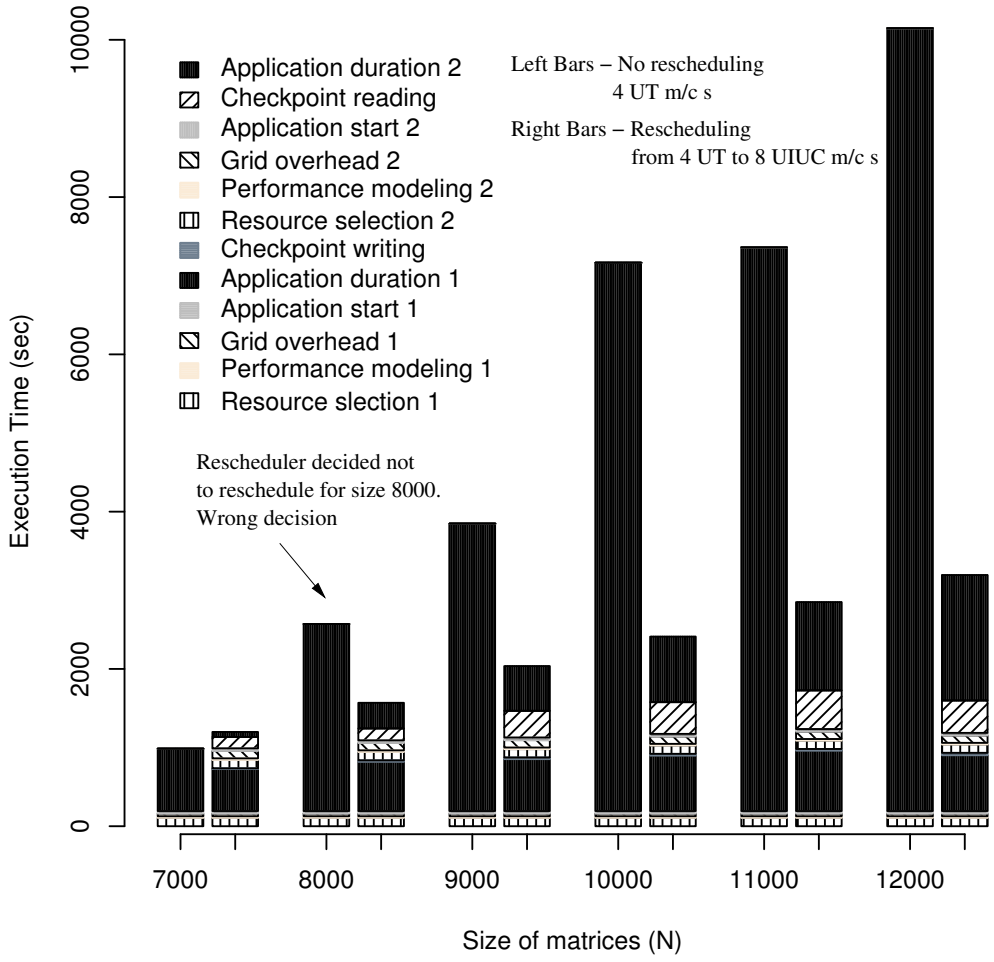


Figure 3. Problem Sizes and Migration

for matrix size 11000 than for matrix size 12000. This is due to the transient loads associated with the Internet between Tennessee and Illinois.

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 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 Tennessee machines and the bar on the right represents the case when the application was migrated to 8 Illinois machines.

Similar to the first set of experiments, we find only one case when the rescheduler made an incorrect decision for rescheduling. This case, when the number of loading programs 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 because the application completed during the time for rescheduling decision. Also, the greater the load, the higher 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 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. As with Figure 3, we see that the times for checkpoint reading show variance for the same matrix size in Figure 4. Again, this is due to the variance in network loads on the Internet connection between Tennessee and Illinois.

In the third set of experiments, shown in Figure 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 execution time in minutes of the 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, 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 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 Illinois machines. The most startling result in Figure 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 correctly 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.

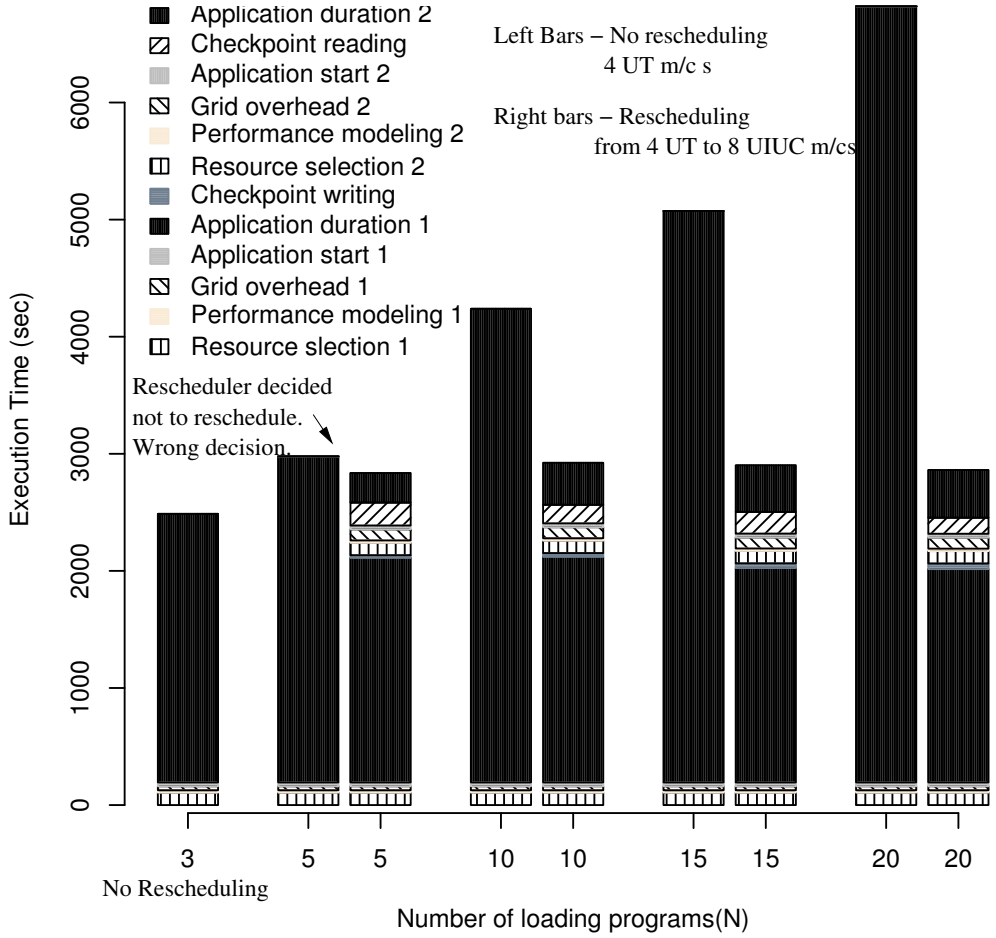


Figure 4. Load Amount and Migration

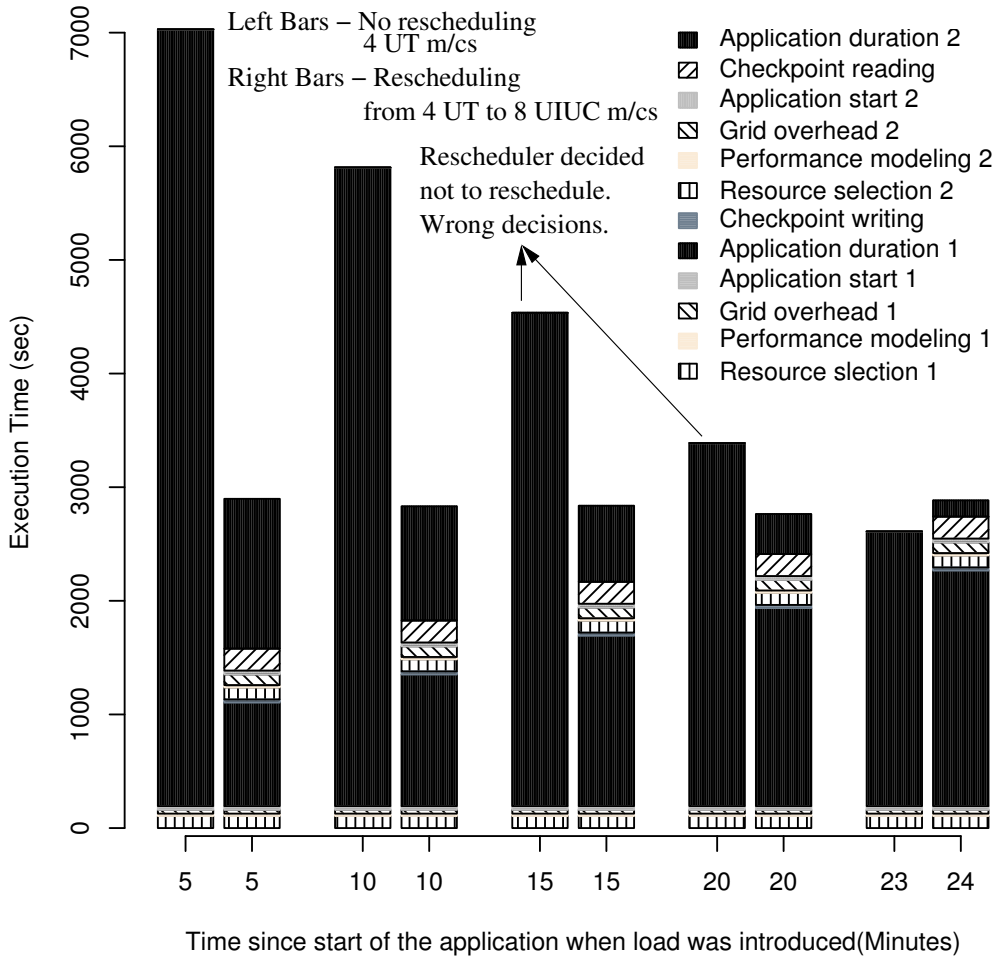


Figure 5. Load Introduction Time and Migration

8.2. 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 these experiments, two problems were involved. For the first problem, matrix size of 14000 was used and 6 Tennessee 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 Tennessee machines on which the first problem was executing and 2 Illinois machines were made available. Due to the presence of the first problem, the 6 Tennessee machines alone were insufficient to accommodate the second problem. Hence the performance model chose the 6 Tennessee machines and 2 Illinois 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 Tennessee machines. Although this involved constricting the number of processors for 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 6 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 Tennessee machines. The middle bar represents the execution time of the second application when the entire application was executed on 6 Tennessee and 2 Illinois machines. The bar on the right represents the execution time of the second application, when the application was initially executed on 6 Tennessee and 2 Illinois machines and later migrated to execute on only 6 Tennessee machines when the first application completed.

For the second problem, for both 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 after the completion of the first application.

8.3. Predicting Redistribution Cost

As observed in Figures 3 - 5, the Rescheduler can make wrong decisions for rescheduling in certain cases. In cases where the Rescheduler made the wrong decision, 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 I for different phases of rescheduling while the actual rescheduling cost was less than the worst-case rescheduling cost.

As shown in Table I, the cost for reading and redistribution of checkpoint data is the highest of the various costs involved in rescheduling. The checkpoint reading and redistribution are

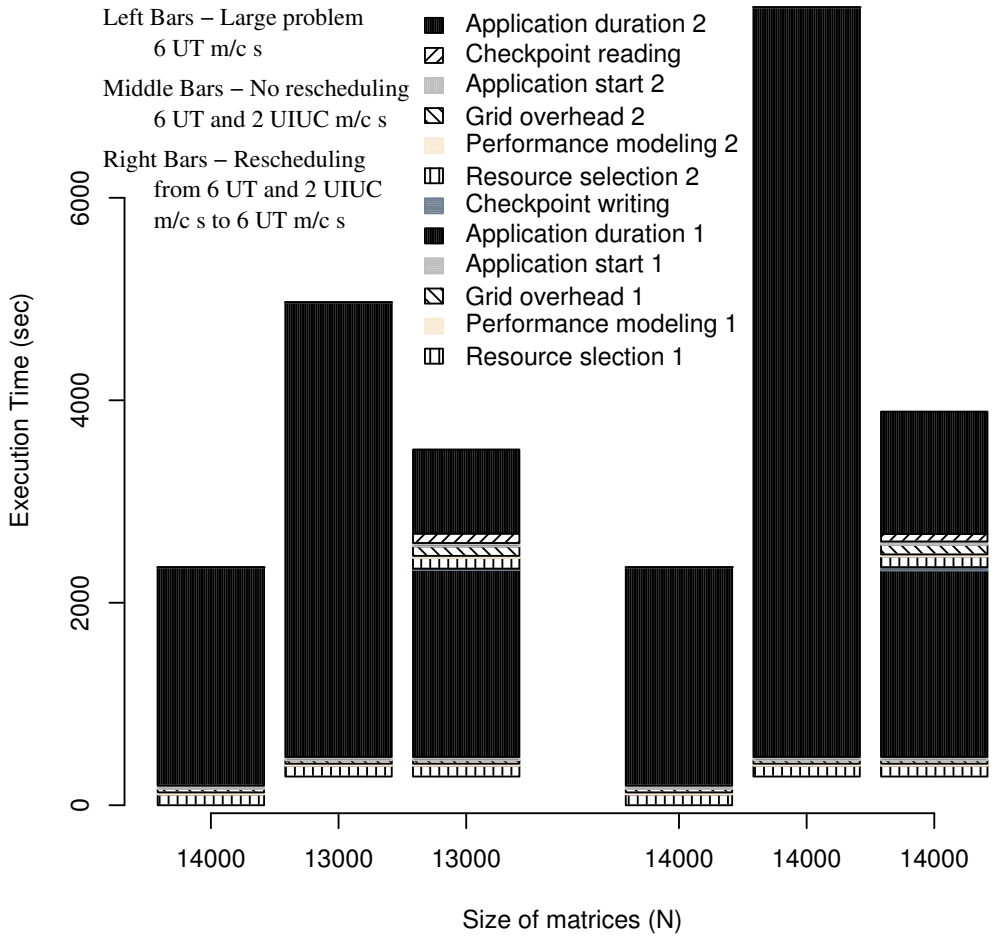


Figure 6. Opportunistic Migration

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 [23] 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, 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, the checkpoint writing time is insignificant because the processes write checkpoint data to the local storage. 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 distributions such as block, cyclic and block-cyclic distributions. When the end application calls SRS_Register to register data to be checkpointed, it also specifies the data distribution used for that 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 example, if a 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 indicates 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 of 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 Tennessee and 8 Illinois machines were used. A ScaLAPACK QR factorization problem was submitted to the GrADS Application Manager. Since the Tennessee machines were faster than the Illinois machines, the 4 Tennessee 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 were introduced in the Tennessee machines by the execution of 10 loading programs on each of the Tennessee machines.

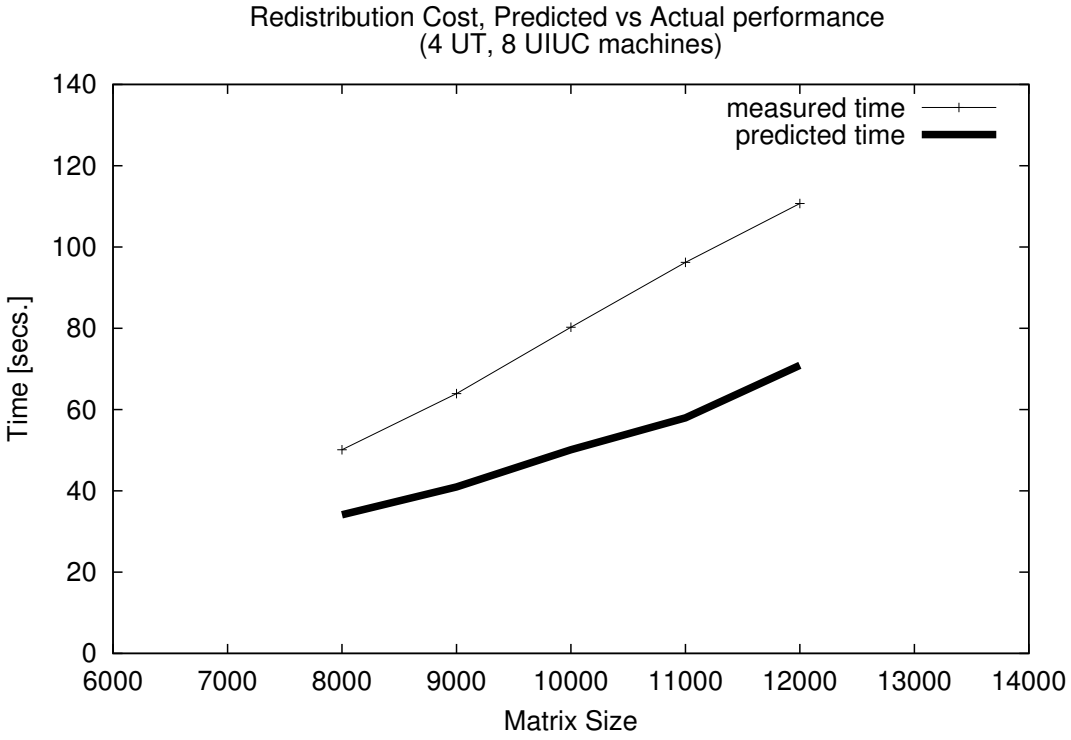


Figure 7. Redistribution Cost Prediction

When the Contract Monitor contacted the Rescheduler requesting that the application be rescheduled, the Rescheduler dynamically predicted the redistribution cost involved in rescheduling the application. Figure 7 compares the predicted and the actual cost for redistribution of the data 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 7, we find that the Rescheduler was able to perform a reasonably accurate 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. By employing the predicted redistribution cost, the Rescheduler was able to make the right decisions for rescheduling for cases in Figures 3, 4 and 5 when it previously made wrong decisions.

9. Related Work

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 [25, 7, 6, 30, 9], or to provide fault resilience [3], or to reduce the obtrusiveness to workstation owner [3, 18]. The particular projects that are closely related to our work are Dynamite [30], MARS [15], LSF [35], Condor [18] and Cactus [2].

The Dynamite system [30] based on Dynamic PVM [9] migrates applications when certain machines in the system get 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.

In LSF [35], 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 enough remaining execution time so that performance benefits can be obtained by 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 because of the tight coupling of suspension and migration decisions.

Of the Grid computing systems, only Condor [18] 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 account the performance benefits of migrating applications. Unlike our rescheduler framework, the Condor system does not possess 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.

The Cactus [2] migration framework was also developed in the context of the GrADS project and hence follows most of the design principles of our migration framework. Their migration thorn is similar to our migrator and their performance detection thorn also performs contract monitoring and detects contract violation similar to our contract monitor. Their migration logic manager is similar in principle to our rescheduler. The differences lie in the decisions made to contact the rescheduler service for migration, and decisions made in the rescheduler regarding

when to migrate. While our migration framework makes decisions using a threshold for the average performance ratio, the Cactus framework uses a maximum number of consecutive contract violations as the threshold for migration. Although Cactus allows the thresholds to be changed dynamically by the user using a HTTP interface, often the user does not possess adequate expertise in determining the threshold and hence a more automatic mechanism like the one followed in our approach is desirable for Grid systems. Also, the Cactus migration framework uses only the resource characteristics to discover better systems for migrating, whereas our system uses predicted application performance on the new systems. Also, similar to other approaches, Cactus does not take into account the remaining execution time of the application.

The GridWay framework [17] has a number of similarities with the GrADS framework both in terms of concepts and the design of the architecture. Hence GridWay's job migration framework by Montero et. al [21] performs most of the functionalities of our migration framework. Their job migration framework takes into account the proximity of the execution hosts to the checkpoint and restart files. Their job migration framework also performs opportunistic migration and migration under performance degradation. However, their work does not mention about the migration of parallel MPI jobs and the possible reconfiguration of hosts and the redistribution of data. By considering dynamic redistribution costs based on network bandwidths , our migration framework indirectly takes into account the proximity of the new hosts to the checkpoint files. Lastly, the execution models used by our migration framework simulate the actual application and hence are more robust than their mathematical models.

10. Conclusions and Future Work

Many existing migration systems that migrate applications under resource load conditions implement simple policies that cannot be applied to Grid systems. We have implemented a migration framework that takes into account both the system load and application characteristics. The migrating decisions are based on factors including the amount of resource load, the point during the application lifetime when the load is introduced, and the size of the applications. We have also implemented a framework that opportunistically 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.

We intend to provide more robust frameworks in the SRS system and in the Rescheduler to efficiently predict the cost for the redistribution of data. Also, instead of fixing the rescheduler threshold at 30%, our future work will involve determining the rescheduling threshold dynamically based on the dynamic observation of load behavior on the system resources. We propose to investigate the usefulness of our approach for complex applications involving multiple components and/or written in multi-programming languages similar to the efforts of Mayes et. al. [19]. Currently, the average of performance ratios is used to determine when a contract monitor will contact the rescheduler for migration. In the future, we plan to investigate more robust policies for contacting the rescheduler. Mechanisms for quantifying the deficiencies of the execution model detected during contract monitoring and communicating the information to the application developer also need to be investigated.

ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their very helpful comments toward improving the quality of the paper.

REFERENCES

1. MPI. *http://www-unix.mcs.anl.gov/mpi*.
2. G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, Ed Seidel, and J. Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *The International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.
3. 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.
4. 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.
5. F. Berman and R. Wolski. The AppLeS Project: A Status Report. *Proceedings of the 8th NEC Research Symposium*, May 1997.
6. J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing, 1995.
7. 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.
8. Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters. Submitted to Parallel Computing, 2003.
9. 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. Proceedings Volume II, Networking and Tools*, pages 273–277, Munich, Germany, April 1994. Springer Verlag.
10. F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.
11. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–375, 1997.
12. I. Foster and C. Kesselman eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.
13. I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. *In Proceedings of SuperComputing 98 (SC98)*, 1998.
14. M. Frigo. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the ICASSP Conference*, volume 3, page 1381, 1998.
15. 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.
16. 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.
17. E. Huedo, R. S. Montero, and I. M. Llorente. An Experimental Framework for Executing Applications in Dynamic Grid Environments. Technical Report 2002-43, NASA/CR-2002-211960. ICASE, November 2002.
18. 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.
19. K. Mayes, G.D. Riley, R.W. Ford, M. Lujn, L. Freeman, and C. Addison. The Design of a Performance Steering System for Component-based Grid Applications. In V. Getov, M. Gerndt, A. Hoisie, A. Maloney, and B. Miller, editors, *Performance Analysis and Grid Computing*, pages 111 – 127. Kluwer Academic Publishers, 2003.

20. 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.
21. R. S. Montero, E. Huedo, and I. M. Llorente. Grid Resource Selection for Opportunistic Job Migration. In *Proceedings of the 9th International Euro-Par Conference. Lecture Notes in Computer Science (LNCS) series*, volume 2790, pages 366–373, Klagenfurt, Austria, August 2003. Springer-Verlag.
22. 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.
23. 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.
24. 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.
25. K.A. Saqabi, S.W. Otto, and J. Walpole. Gang Scheduling in Heterogeneous Distributed Systems. Technical report, OGI, 1994.
26. 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.
27. 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.
28. S. Vadhiyar and J. Dongarra. Performance Oriented Migration Framework for the Grid. In *Proceedings of The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 130–137, Tokyo, Japan, May 2003.
29. S. Vadhiyar, G. Fagg, and J. Dongarra. Automatically Tuned Collective Communications. In *Proceedings of SuperComputing2000*, November 2000.
30. 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. In *P.M.A. Sloot and M. Bubak and A.G. Hoekstra and L.O. Hertzberger, editors, High-Performance Computing and Networking (HPCN Europe '99), Amsterdam, The Netherlands, in series Lecture Notes in Computer Science, nr 1593, Springer-Verlag, Berlin, ISBN 3-540-65821-1.*, pages 300–310. April 1995.
31. R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.
32. R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC98: High Performance Networking and Computing*, 1998.
33. 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.
34. 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.
35. S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software – Practice and Experience*, 23(12):1305–1336, December 1993.