A Performance Oriented Migration Framework For The Grid *

Sathish S. Vadhiyar and Jack J. Dongarra
Computer Science Department
University of Tennessee
{vss, dongarra}@cs.utk.edu

Abstract

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. In this paper we describe 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 our migration framework is to improve the response times for individual applications. We also present some results that demonstrate the usefulness of our migrating system.

1. Introduction

Computational Grids [9] 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.

In this paper, we focus on migration of applications executing on the distributed and Grid systems when the loads on the system resources change. There are at least two disadvantages in using the existing migration systems [13, 7, 12, 18, 22, 10, 12] 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 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 will to lead to frequent satisfaction of the predefined conditions and hence will lead to frequent invocation 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 system load changes. In our 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, 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 suspended and migrated only when better systems are found for application execution thereby invoking the migration decisions as infrequently as possible. The framework has been implemented and tested on top of the GrADS system [2]. Our test results indicate that our migrating system is useful for applications on the Grid.

In Section 2, we describe the GrADS system and the life cycle of GrADS applications. In Section 3, we introduce our migration framework by describing the different components for migration. In Section 4, we describe our experiments and provide various results. In Section 5, we present

*This work is supported in part by the National Science Foundation contract GRANT #EIA-9975020, SC #R36505-29200099 and GRANT #EIA-9975015

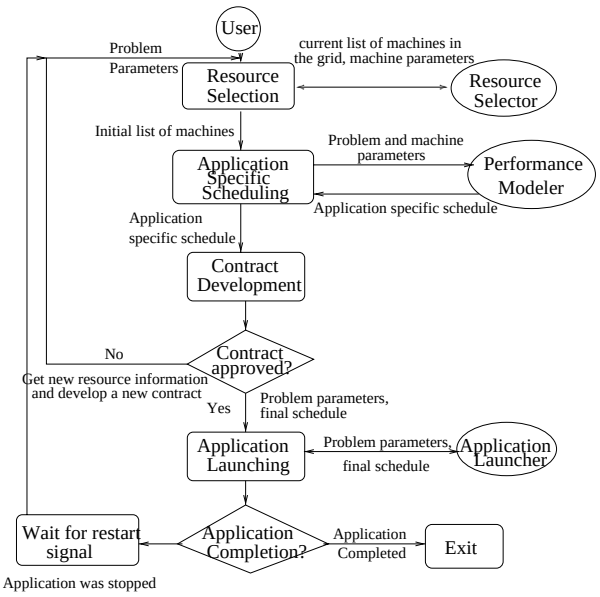

Figure 1. GrADS application manager

related work in the field of migration. We give concluding remarks and explain our future plans in Section 6.

2. The GrADS System

GrADS [2] 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. The University of Tennessee investigates issues regarding integration of numerical libraries in the GrADS system. In our previous work [14], 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) [8] to get a list of machines in the GrADS testbed that are alive and then contacts the Network Weather Service(NWS) [20] to get 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 the application specific execution model, GrADS follows the AppLeS [3] 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 can 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 [16] 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 a resume signal and then collects new machine information by starting from the resource selection phase again.

3. The Migration Framework

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 between the different components involved in the migration framework is illustrated in Figure 2. These components are described in detail in the following subsections.

3.1. 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

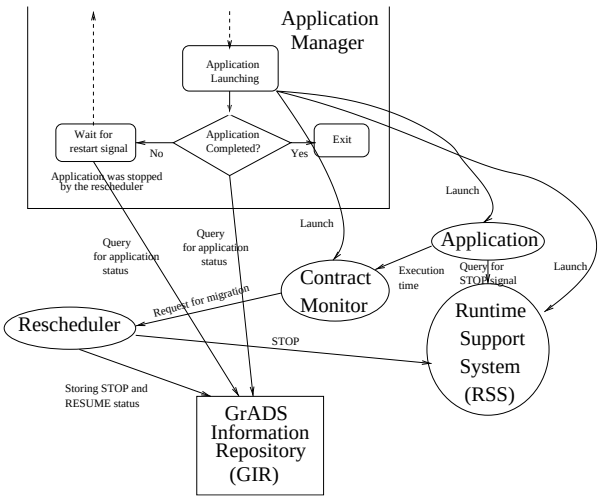

Figure 2. Interactions in Migration framework

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 [11], 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 - 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 can perform application-specific stop actions.

SRS library uses Internet Backplane Protocol(IBP)[15] 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 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 applications like ScaLAPACK and PETSC.

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 GIR 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.

3.2. Contract Monitor

Contract Monitor is a component that uses the Autopilot infrastructure to monitor the progress of the applications in GrADS. Autopilot [16] 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 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.

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.

3.3. 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 no contract monitor has 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 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

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

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 is given in Table 1. The times shown in Table 1 were obtained by conducting a number of experiments with 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.

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

Table 1. Times for rescheduling phases

If the rescheduling gain is greater than 30%, the rescheduler sends STOP signal to the application, and stores the stop status in GIR. The application manager then waits for the RESUME signal. The rescheduler stores the RESUME value 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 [19] 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.

4. Experiments and Results

The GrADS experimental testbed consists of about 40 machines that reside in institutions across United States including University of Tennessee, University of Illinois, University of California at San Diego, Rice University etc. For the sake of clarity, our experimental testbed consists of two clusters, one in University of Tennessee and another in University of Illinois, Urbana-Champaign. The Tennessee cluster consists of 8 933 MHz dual- processor Pentium III machines running Linux and connected to each other by 100 Mb switched Ethernet. The Illinois cluster consists of 16 450 MHz single-processor Pentium II machines running Linux and connected to each other by 1.28 Gbit/second full duplex myrinet. The two clusters are connected by means of Internet.

In our experiments, ScaLAPACK QR factorization was used as the end application. 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 were checkpointed by the SRS library for the application included the matrix, A and the right-hand side vector, B.

4.1. Migration on Request

In all the experiments in this section, 4 Tennessee machines and 8 Illinois machines were used. A given matrix size for the QR factorization problem was input to the application manager. Since the Tennessee machines were faster 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 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 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 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

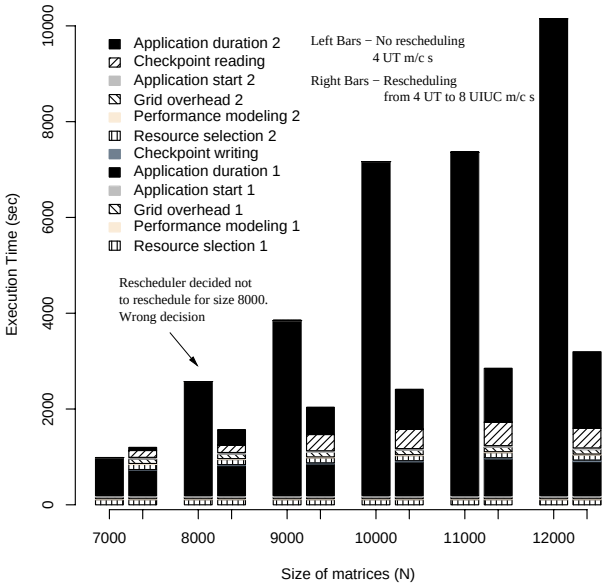

Figure 3. Problem Sizes and Migration

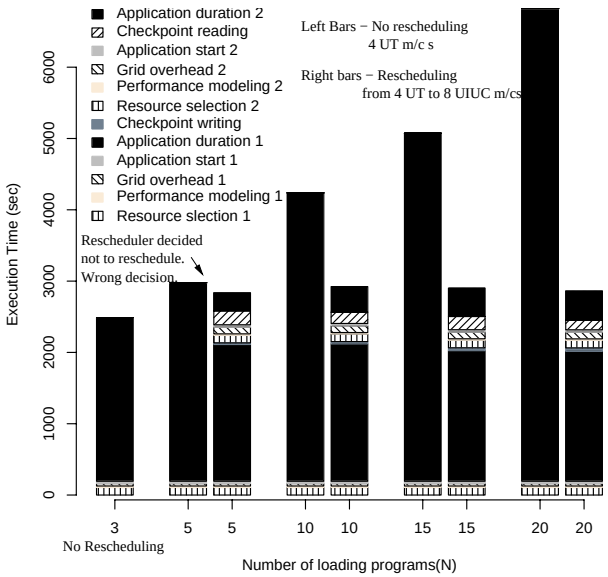

Figure 4. Load Amount and Migration

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 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 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 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 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, 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, 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

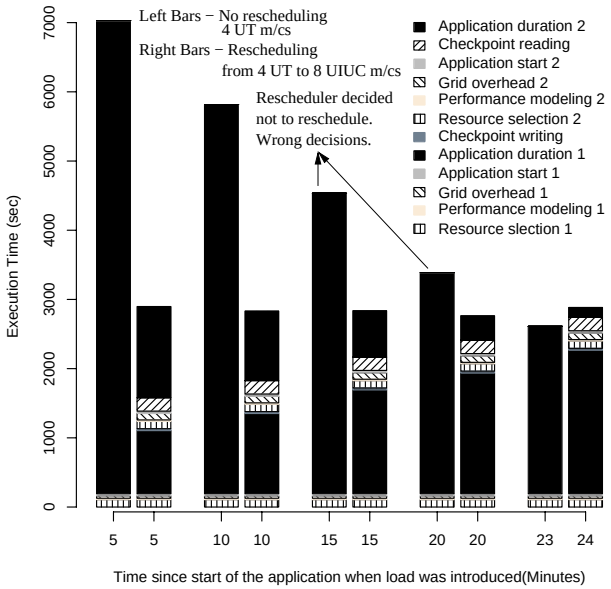

Figure 5. Load Introduction Time and Migration

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 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.

4.2. 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 input to the application manager 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 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 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.

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 after the completion of the first application.

5. Related Work

Different systems have been implemented to migrate executing applications onto different sets of resources. These systems migrate applications either to efficiently use underutilized resources [13, 17, 5, 4, 21, 18, 6], to provide fault resilience [1] or to reduce the obtrusiveness to workstation owner [1, 12]. The particular projects that are closely related to our work are Dynamite [18], MARS [10], LSF [22] and Condor [12].

The Dynamite system [18] based on Dynamic PVM [6] 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

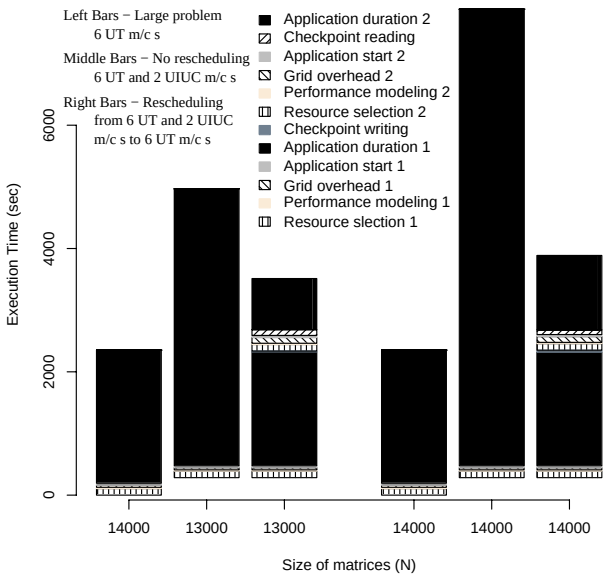

Figure 6. Opportunistic Migration

does not necessarily evaluate the remaining execution time of the application and the resulting performance benefits due to migration. MARS [10] 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 [22], 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 predefined 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 [12] 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.

6. Conclusions and Future Work

Many existing migrating systems that migrate applications under loading 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 like the amount of load, the time of the application when the load is introduced and the size of the applications. We have 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.

Of the various costs involved in rescheduling, the cost for data redistribution is the only significant cost that depends on the number and amount of checkpointed data, the data distributions used for the data and the current and future processors sets for the application. We are planning to modify the SRS library to store these information in the Runtime Support System (RSS). The rescheduler, instead of associating a fixed worst-case redistribution cost for the application, will then use relevant information from RSS and also the NWS information for bandwidth and latency between the machines to calculate the redistribution cost dynamically. 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.

References

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

[2] 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.

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

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

[5] 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.

[6] L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems. In W. Gentzsch and U. 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] G. van Albada, J. Clinckemaillie, A. Emmen, J. Gehring, O. Heinz, F. van der Linden, B. Overeinder, A. Reinefeld, and P. 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.