International Journal of Parallel Programming, Vol. 33, No. 2, June 2005 (© 2005)
DOI: 10.1007/s10766-005-3584-4

New Grid Scheduling and Rescheduling Methods in the GrADS Project

F. Berman,[1] H. Casanova,[1] A Chien,[1] K. Cooper,[2] H. Dail,[1] A. Dasgupta,[2] W. Deng,[3] J. Dongarra,[4] L. Johnsson,[5] K. Kennedy,[2,6] C. Koelbel,[2] B. Liu,[5] X. Liu,[1] A. Mandal,[2] G. Marin,[2] M. Mazina,[2] J. Mellor-Crummey,[2] C. Mendes,[3] A. Olugbile, [1] M. Patel,[5] D. Reed,[3] Z. Shi,[4] O. Sievert,[1] H. Xia,[1] and A. YarKhan[4]

Received ■; revised ■; accepted ■

The goal of the Grid Application Development Software (GrADS) Project is to provide programming tools and an execution environment to ease program development for the Grid. This paper presents recent extensions to the GrADS software framework: a new approach to scheduling workflow computations, applied to a 3-D image reconstruction application; a simple stop/migrate/restart approach to rescheduling Grid applications, applied to a

[1]Department of Computer Science and Engineering, University of California at San Diego, San Diego, CA 92093. E-mail: {berman, casanova, achien, hdail, lxin, aoo, osievert, huaxia}@ucsd.edu
[2]Computer Science Dept., Rice University, Houston, TX 77005. E-mail: {keith, anshuman, ken, chk, anirban, mgabi, mmzn, johnmc}@rice.edu
[3]Department Computer Science, University of Illinois, Urbana, IL 61801. E-mail: {weideng, cmendes, reed}@uiuc.edu
[4]Innovative Computing Lab, University of Tennessee, Knoxville, TN 37996. E-mail: {dongarra,yarkhan,shi}@utk.edu
[5]Department Computer Science, University of Houston, Houston, TX 77204. E-mail: {johnsson,bliu2,mpate}@uh.edu
[6]To whom correspondence should be addressed.

QR factorization benchmark; and a process-swapping approach to rescheduling, applied to an N-body simulation. Experiments validating these methods were carried out on both the GrADS MacroGrid (a small but functional Grid) and the MicroGrid (a controlled emulation of the Grid).

KEY WORDS: Grid computing; scheduling; rescheduling.

1. INTRODUCTION

Since 1999, the Grid Application Development (GrADS) Project has worked to enable an integrated computation and information resource based on advanced networking technologies and distributed information sources. In other words, we have been attacking the problems inherent in Grid computing.(1) In theory, the Grid connects computers, databases, instruments, and people into a seamless web of advanced capabilities. In practice, its lack of usability has limited its application to specialists.

Because the Grid is inherently more complex than stand-alone computer systems, Grid programs must reflect this complexity at some level. However, we believe that this complexity should *not* be embedded in the main algorithms of the application, as is often now the case. Instead, GrADS provides software tools that manage the Grid-specific details of execution with minimal effort by the scientists and engineers who write the programs. This increases usability and allows the system to perform substantial optimizations for Grid execution.

Figure 1 shows the program development framework that GrADS pioneered in response to this need.(2) Two key concepts are central to this approach. First, applications are encapsulated as *configurable object programs (COPs)*, which can be optimized rapidly for execution on a specific collection of Grid resources. A COP includes *code* for the application, a *mapper* that determines how to map an application's tasks to a set of resources, and a *performance model* that estimates the application's performance on a set of resources. Second, the system relies upon *performance contracts* that specify the expected performance of modules as a function of available resources.

The left side of Fig. 1 depicts tools used to construct COPs from either domain-specific components or low-level (e.g. MPI) programming. In either case, GrADS provides prototype tools that semi-automatically construct performance models and mappers. Although they are not the major focus of this paper, some of these tools are described in more detail in Section 3 below.

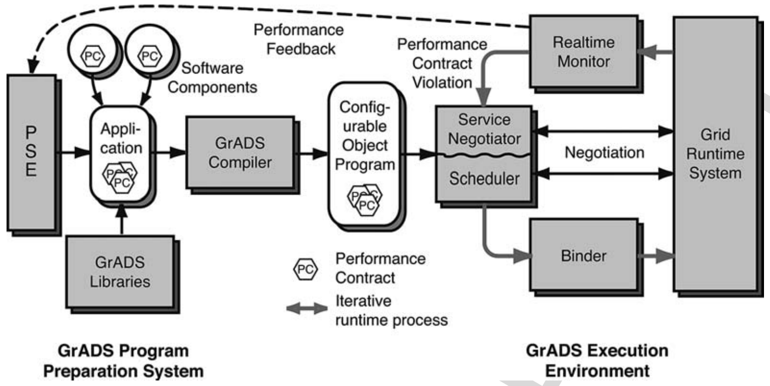

Fig. 1. GrADS Program Preparation and Execution Architecture.

The right side of Fig. 1 depicts actions when a COP is delivered to the execution environment. The GrADS infrastructure first determines which resources are available and, using the COP's mapper and performance model, schedules the application components onto an appropriate subset of these resources. Then the GrADS software invokes the *binder* to tailor the COP to the chosen resources and the *launcher* (not shown) to start the tailored COP on the Grid.

Once launched, execution is tracked by the *contract monitor*, which detects anomalies and invokes, when necessary, the *rescheduler* to take corrective action. Performance monitoring in GrADS is based on Autopilot,[3] a toolkit for real-time application and resource monitoring and closed-loop control. Autopilot provides sensors for performance data acquisition, actuators for implementing optimization commands and a decision-making mechanism based on fuzzy logic. Part of the tailoring done by the binder is to insert the sensors needed for monitoring a particular application. Autopilot then assesses the application's progress using performance contracts,[4] which specify an agreement between application demands and resource capabilities. The contract monitor takes periodic data from the sensors and uses Autopilot's decision mechanism to verify that the contract is being met. If a contract violation occurs, the monitor takes corrective action, such as contacting a GrADS rescheduler. GrADS incorporates a variety of utilities associated with contract monitoring, including a Java-based Contract Viewer GUI to visualize the performance contract validation activity in real-time.

To support research into and evaluation of GrADS capabilities, GrADS has constructed two research testbeds. The *MacroGrid* consists of Linux clusters with GrADS software installed at several participating GrADS sites, including clusters at University of California at San

Diego (UCSD, 10 machines), University of Tennessee at Knoxville (UTK, 24 machines), University of Illinois at Urbana-Champaign (UIUC, 24 machines), and University of Houston (UH, 24 machines). The experiments in Section 3 and Section 4.1 run on this testbed. The *MicroGrid* is a Grid emulation environment that runs on clusters and permits experimentation with extreme variations in network traffic and loads on compute nodes.(5) Section 4.2 describes experiments run on this platform. (We earlier ran very similar experiments on the MacroGrid, validating both the MicroGrid's emulation and the rescheduling method's practicality.(6))

The experiments we describe exercise many parts of the GrADS environment. This paper closes with a brief discussion of what we learned from these experiences, and an outline of future work.

2. LAUNCHING COMPONENTS ON THE GRID

Once an application schedule has been chosen, the GrADS *application manager* must prepare the configurable object program and map it onto the selected resource configuration. In turn, the application manager invokes the binder, which is responsible for creating and configuring the application executable, instrumenting it, and then launching it on the Grid. The original GrADS binder did most of its work by editing the entire application binary, which limited its applicability to homogeneous collections of processors (such as our original testbed). It soon became clear that this approach would not suffice for a general system because most grids (including later generations of our own testbed) are heterogeneous and because many grid programs require linking against libraries of components preinstalled on Grid resources.

To address these issues, we developed a new distributed GrADS binder that executes on all Grid resources specified in the schedule. The new binder receives three sets of inputs: resource specific information (such as hardware and software capabilities) via the GrADS Information Service (GIS), characteristics of the target architecture that can be used for machine-specific optimizations, and a *compilation package* that consists of the application's source code in an intermediate representation, a list of required libraries, and a script to configure the application for compilation.

A binder process executes on each machine chosen by the scheduler. For this to be possible, the global binder must know the locations of all software resources, including application-specific libraries, general libraries, and the binder itself. To that end, the global binder queries the GIS to locate necessary software on the scheduled node, starting with the local binder code. The global binder then launches the local binder process, which further queries GIS for the locations of application-specific libraries, instruments the code with Autopilot sensors, configures, compiles, and links the application. Finally, the global binder enables the launch of the application. If the application is an MPI application, then a global synchronization must be carried out as part of the MPI protocol at the beginning of the execution. In this case, the binder returns control to the application manager which launches the application after synchronization. In non-MPI applications, the binder launches the application and notifies the application manager when the program terminates.

Note that by using a high-level representation of the program and configuring and compiling it only at the target machine, the binder naturally deals with heterogeneous resources. This is important in any Grid context. Moreover, preserving high-level program information until the target machine is known also provides opportunities for architecture-specific optimizations.

3. SCHEDULING WORKFLOW GRAPHS

Workflow applications are an important class of programs that can take advantage of the power of Grid computing, such as the LIGO(7) pulsar search image processing applications.(8) As the name suggests, a workflow application consists of a collection of components that need to be executed in a partial order determined by control and data dependences.

The previous version of the GrADS scheduler was designed to support tightly-coupled MPI applications(9–11) and was not well suited to workflow applications. On the other hand, existing approaches to workflow scheduling, such as Condor DAGMan,(12) are not able to effectively exploit the performance modeling available within GrADS to produce better schedules. To address these shortcomings, we developed a new GrADS workflow scheduler that resolves the application dependences and schedules the components, including parallel components, onto available resources using GrADS performance models as a guide.

3.1. Workflow Scheduling

A Grid scheduler for a workflow application must be guided by an objective function that it tries to optimize, such as minimizing communication time or maximizing throughput. For the GrADS Project, we have chosen to minimize the overall job completion time, also known as the *makespan*, of the application. The GrADS scheduler builds up a model of Grid resources using services such as MDS(13) and NWS.(14) The scheduler also obtains performance models of the application using a scalable technique developed for GrADS. Using these models, the scheduler then provides a mapping from the workflow components to the Grid resources.

A stricter definition of the problem can be formulated with the help of two sets: the set $C = \{c_1, c_2, \ldots, c_m\}$ of available application components from the application DAG, and the set $G = \{r_1, r_2, \ldots, r_n\}$ of available Grid resources. The goal of the scheduler is to construct a mapping from elements of C onto elements of G.

For each application component, the GrADS workflow scheduler ranks each eligible resource, reflecting the fit between the component and the resource. Lower rank values, in our convention, indicate a better match for the component. After ranking the components, the scheduler collates this information into a performance matrix. Finally, it runs heuristics on the performance matrix to schedule components onto resources.

Computing rank values The scheduler ensures that resources meet certain minimum requirements for a component. Resources that do not qualify under these criteria are given a rank value of infinity. For all other resources, the rank of the resource r_j is calculated by using a weighted sum of the expected execution time on the resource and the expected cost of data movement for the component c_i:

$$rank(c_i, r_j) = w_1 \times eCost(c_i, r_j) + w_2 \times dCost(c_i, r_j) \tag{1}$$

The expected execution time *eCost* is calculated using a performance modeling technique that will be described in the next section. The cost of data movement *dCost* is estimated by a product of the total volume of data required by the component and the expected time to transfer data given current network conditions. For this measurement, NWS is used to obtain an estimate of the current network latency and bandwidth. The weights w_1 and w_2 can be customized to vary the relative importance of the two costs.

Scheduling application components Once ranks have been calculated, a performance matrix is constructed. Each element of the matrix p_{ij} denotes the rank value of executing the ith component on the jth resource. This matrix is used by the scheduling heuristics to obtain a mapping of components onto resources. Such a heuristic approach is necessary since the mapping problem is NP-complete.(15) We apply three heuristics to obtain three mappings and then select the schedule with the minimum makespan. The heuristics that we apply are the min-min, the max-min, and the sufferage heuristics.(16,17)

3.2. Component Performance Modeling

As described in the previous section, estimating the performance of a workflow component on a single node is crucial to constructing a good overall workflow schedule. We model performance by building up an architecture-independent model of the workflow component from individual component models. To obtain the component models, we consider both the number of floating point operations executed and the memory access pattern. We do not aim to predict an exact execution time, but rather provide an estimated resource usage that can be converted to a rough time estimate based on architectural parameters. Because the resources are architecture-independent, our models can be used on widely varying node types.

To understand the floating point computations performed by an application, we use hardware performance counters to collect operation counts from several executions of the program with different, small-size input problems. We then apply least squares curve-fitting on the collected data.

To understand an application's memory access pattern, we collect histograms of memory reuse distance (MRD)—the number of unique memory blocks accessed between a pair of references to the same block—observed by each load and store instruction.(18) Using MRD data collected on several small-size input problems to the application, we model the behavior of each memory instruction, and predict the fraction of hits and misses for a given problem size and cache configuration. To determine the cache miss count for a different problem size and cache configuration, we evaluate the MRD models for each reference at the specified problem size, and count the number of accesses with predicted reuse distance greater than the target cache size.

3.3. Workflow Scheduling Test Case

In this section, we apply some of the strategies described in the previous sections to the problem of adapting EMAN,(19) a bio-imaging application developed at Baylor College of Medicine, for execution on the Grid using the GrADS infrastructure. EMAN automates a portion producing 3-D reconstructions of single particles from electron micro-graphs. Human intervention and expertise is needed to define a preliminary 3-D model from the electron micro-graphs, but the refinement from a preliminary model to the final model is fully automated. This refinement process is the most computationally intensive step and benefits the most from harnessing the power of the Grid. Figure 2 shows the components in the EMAN refinement workflow, which forms a linear graph in which some components can be parallelized.

We have conducted experiments on workflow scheduling with two EMAN data-sets-GrOEL, a small data-set with 200 MB input data and rdv, a medium data-set with 2 GB input data. For these experiments, we used 6 nodes from the Itanium IA-64 cluster [i2-53 to i2-58] at UH and 7

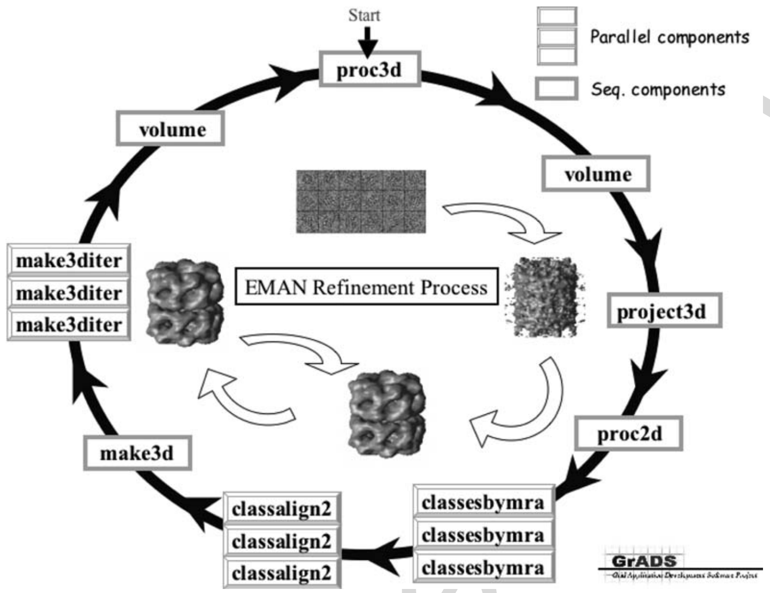

Fig. 2. EMAN refinement workflow.

nodes from the IA-32 cluster [torc1 to torc7] at UTK. Note that the testbed is heterogeneous in terms of architecture, CPU-speeds, memory and storage. Also, note that "classesbymra" is the most computationally intensive step in the EMAN refinement and is a parameter sweep that can be distributed across multiple clusters. "classalign2" on the other hand cannot be distributed across multiple clusters.

Table-I shows the results of the run of the rdv data on unloaded resources on the testbed. The first column represents the name of the component in the linear DAG. The second column denotes the resources chosen by the Workflow scheduler for the particular component. The third column denotes the number of instances mapped by the Workflow scheduler to the selected resources. The last column denotes the time it took for that component to run on the selected set of resources.

For the sequential and single-cluster components, the scheduler chose the best node or cluster for execution. The interesting case is the case of the parameter sweep step called "classesbymra". From the execution time of the "classesbymra" step, the following can be inferred:

— The makespan of the "classesbymra" step was 84 h 30 min [the time the instances finished on the UH cluster]. Since the instances at the UTK machines finished in 81 h 41 min, it can be inferred

Table I. Results of EMAN Workflow Execution with rdv Data

Component	Resources Chosen	Num Instances	Component Exec Time
Proc3d	i2-58	1	<1 min
Project3d	i2-58	1	1 h 48 min
Proc2d	i2-58	1	<1 min
Classesbymra	*i2-53 to i2-58*	*68 [i2-*]*	*84 h. 30 min*
	torc1 to torc7	*42 [torc*]*	*81 h. 41 min*
Classalign2	i2-53 to i2-58	379	45 min
Make3d	i2-58	1	47 min
Proc3d	i2-58	1	<1 min
Proc3d	i2-58	1	<1 min

that the load was optimally balanced across the two clusters since the granularity of a single instance is greater than 7 h.

— The optimal load balance is primarily due to accurate performance models and efficient Work-flow scheduling. Rank of a "classesbymra" instance on a node in UH cluster was 5077.76 and on a node in UTK cluster was 8844.91.

For the GrOEL data-set, the makespan for the classesbymra step for heuristic scheduling was compared with that obtained from random scheduling. Random scheduling picks a node randomly for the next available instance. The results in Table-II use 2 nodes from the UH cluster and 7 nodes from the UTK cluster and all the resources are unloaded. The number in the braces after execution times indicate the average number of classesbymra instances mapped to the site. From these results, it can be inferred that accurate relative performance models on heterogeneous platforms combined with heuristic scheduling result in good load balance of the classesbymra instances when the grid resources are unloaded. Heuristic scheduling is better than random scheduling by 25 percent in terms of makespan length.

The second set of results shows the effect of loaded machines on the quality of schedule. Five loaded nodes from the UH cluster and 7 unloaded nodes from UTK cluster were used for these experiments. From the results in Table-III, it is observed that there is uneven load balance due to loading of the UH nodes. Random scheduling does better because the random distribution maps more instances to the unloaded UTK cluster which had more nodes in the universe of resources. So, it can be inferred

Table II. Results for GrOEL Data with Unloaded Resources

	Heuristic Run Average	Random Run Average
Exectime(uh)	12 min 42 sec [38]	6 min 3 sec [17]
Exectime(utk)	11 min 47 sec [60]	15 min 48 sec [81]
Makespan	12 min 42 sec	15 min 48 sec

Table III. Results for GrOEL Data with Loaded Resources

	Heuristic Run Average	Random Run Average
Exectime(uh)	16 min 41 sec [60]	6 min 38 sec [44]
Exectime(utk)	7 min 51 sec [38]	10 min 28 sec [54]
Makespan	16 min 41 sec	10 min 28 sec

that for performance model based scheduling to work, either the underlying set of resources should be reliable [implying advanced reservation] or the variability of resource performance can be predicted and taken into account during scheduling.

The third set of results show the effect of inaccurate performance models on the quality of schedule. A rank value of 4.57 instead of 7.60 was used for a classesbymra instance on a UH node. Rank value for the UTK nodes was kept correct. Six nodes from the UH cluster and 7 nodes from the UTK cluster were used. From the results in Table-IV it can be inferred that, inaccurate relative performance models on different heterogeneous platforms result in poor load balance of the classesbymra instances.

Table IV. Results for GrOEL Data with Inaccurate Performance Models

	Heuristic Run Average	Random Run Average
Exectime(uh)	21 min 37 sec [77]	5 min 24 sec [45]
Exectime(utk)	3 min 57 sec [21]	10 min 30 sec [53]
Makespan	21 min 37 sec	10 min 30 sec

4. RESCHEDULING

Normally, a contract violation activates the GrADS *rescheduler*. The retcheduling process must determine whether rescheduling is profitable, based on the sensor data, estimates of the remaining work in the application, and the cost of moving to new resources. If rescheduling appears profitable, the rescheduler computes a new schedule (using the COP's mapper) and contacts *rescheduling actuators* located on each processor. These actuators use some mechanism to initiate the actual migration or load balancing. Sections 4.1 and 4.2 describe two rescheduling mechanisms that we have explored. Both rely on application-level migration, although we designed both so that the required additional programming is minimal. Whether a migration is done or not, the rescheduler may contact the contract monitor to update the terms of the contract.

4.1. Rescheduling by Stop and Restart

Our first approach to rescheduling relied on application migration based on a stop/restart approach. The application is suspended and migrated only when better resources are found for application execution. When a running application is signaled to migrate, all application processes checkpoint user specified data and terminate. The rescheduled execution is then launched by restarting the application on the new set of resources, which then read the checkpointed data and continue the execution.

4.1.1. Implementation

We implemented a user-level checkpointing library called SRS (Stop Restart Software)$^{(20)}$ to provide application migration support. Via calls to SRS, the application can checkpoint data, be stopped at a particular execution point, be restarted later on a different processor configuration and be continued from the previous point of execution. SRS can transparently handle the redistribution of certain data distributions (e.g., block cyclic) between different numbers of processors (i.e., N to M processors). The SRS library is implemented atop MPI and is hence limited to MPI-based parallel programs. Because checkpointing in SRS is implemented at the application rather than the MPI layer, migration is achieved by exiting of the application and restarting it on a new system configuration.

The SRS library uses the Internet Backplane Protocol (IBP)$^{(21)}$ for checkpoint data storage. An external component (e.g., the rescheduler) interacts with a daemon called Runtime Support System (RSS). RSS exists for the duration of the application execution and can span multiple migrations: Before the application is started, the launcher initiates the RSS daemon on the machine where the user invokes the GrADS application manager. The actual application, through the SRS, interacts with RSS to perform some inifialization, to check if the application needs to be checkpointed and stopped, and to store and retrieve checkpointed data.

The contract monitor retrieves the application's registration through the Autopilot(3) infrastructure. The applications are instrumented with sensors that report the times taken for the different phases of the execution to the contract monitor.

The contract monitor compares the actual execution times with predicted ones and calculates the ratio. 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 that the application be migrated. If the rescheduler chooses not 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 lowers the tolerance limits, if necessary.

The rescheduler component evaluates the performance benefits that might accrue by migrating an application and initiates the migration. The rescheduler daemon operates in two modes: *migration on request* and *opportunistic migration*. When the contract monitor detects unacceptable performance loss for an application, it contacts the rescheduler to request application migration. This is called migration on request. Additionally, the rescheduler periodically checks for a GrADS application that has recently completed. If it finds one, the rescheduler determines if another application can obtain performance benefits if it is migrated to the newly freed resources. This is called opportunistic rescheduling. In both cases, the rescheduler contacts the Network Weather Service (NWS) for updated Grid resource information. The rescheduler uses the COP's performance model to predict remaining execution time on the new resources, remaining execution time on the current resources, and the overhead for migration and determines if migration is desirable.

4.1.2. Evaluation

We have evaluated stop/restart rescheduling based on application migration for a ScaLAPACK(22) QR factorization application. The application was instrumented with calls to the SRS library that checkpointed application data including the matrix A and the right-hand side vector B.

In the experiments, 4 UTK machines and 8 UIUC machines were used. The UTK cluster consists of 933 MHz dual-processor Pentium III machines running Linux and connected to each other by 100 Mb switched Ethernet. The UIUC cluster consists of 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 via the Internet.

A given matrix size for the QR factorization problem was input to the application manager. Initially, the scheduler used the more powerful UTK cluster. However, five minutes after the start of the application, an artificial load was introduced on a UTK node, which could make it more efficient to execute the application the UIUC cluster.

The contract monitor requested the rescheduler to migrate the application due to the loss in predicted performance caused by the artificial load. The rescheduler evaluated the potential performance benefits due to migration and either migrated the application or allowed the application to continue on the original machines.

The rescheduler was operated in two modes — default and forced. In normal operation, the rescheduler works under default mode, while the forced mode allows the rescheduler to require the application to either migrate or continue on the same set of resources. Thus, if the default mode is to migrate the application, the forced mode will continue the application on the same set of resources and vice versa. For the experiments, results were obtained for both modes, allowing comparison of the scenarios and verification that the rescheduler made the right decision.

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 left bar represents the running time when the application was not migrated and the right bar represents the time when the application was migrated.

Several observations can be made from Fig. 3. First, the time for reading checkpoints dominated the rescheduling cost, as it involves moving data across the Internet and redistributing data to more processors. On the other hand, the time for writing checkpoints is insignificant since the checkpoints are written to IBP storage on local disks.

In addition, the rescheduling benefits are greater for large problem sizes because the remaining lifetime of the application is larger. For matrix sizes of 7000 and below, the migration cost overshadows the performance benefit due to rescheduling, while for larger sizes the opposite is true. Our rescheduler actually kept the computation on the original processors for

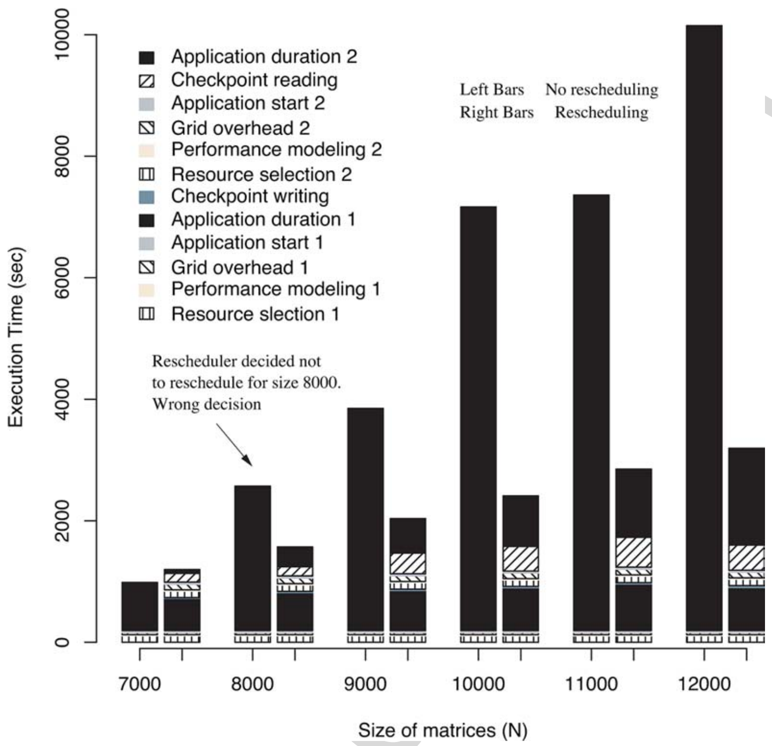

Fig. 3. Problem size and migration.

matrix sizes up to 8000. So, except for matrix size 8000, the rescheduler made the correct decision.

For matrix size 8000, the rescheduler assumed an experimentally-determined worst-case rescheduling cost of 900 s while the actual rescheduling cost was about 420 s. Thus, the rescheduler evaluated the performance benefit to be negligible. Hence, in some cases, the pessimistic approach of assuming a worst-case rescheduling cost will lead to underestimating the performance benefits due to rescheduling.

In another paper,$^{(23)}$ we examine the effects of other parameters (e.g., the load and the time after the start of the application when the load was introduced) and the use of opportunistic rescheduling.

4.2. Rescheduling by Processor Swapping

Although very flexible, the natural stop, migrate and restart approach to rescheduling can be expensive: each migration event can involve large data transfers. Moreover, restarting the application can incur expensive startup costs, and significant application modifications may be required for specialized restart code. Our process swapping approach, which was initially described in,[24] provides an alternative that is lightweight and easy to use, but less flexible than our migration approach.

4.2.1. Basic Approach

To enable swapping, the MPI application is launched with more machines than will actually be used for the computation; some of these machines become part of the computation (the *active* set) while some do nothing initially (the *inactive* set). The user's application sees only the active processes in the main communicator (MPI_Comm_World); communication calls are hijacked, and user communication calls to the active set are converted to communication calls to a subset of the full process set.

During execution, the contract monitor periodically checks the performance of the machines and swaps slower machines in the active set with faster machines in the inactive set. This approach requires little application modification (as described in[24]) and provides an inexpensive fix for many performance problems. On the other hand, the approach is less flexible than migration—the processor pool is limited to the original set of machines, and the data allocation can not be modified.

MPI Swapping was implemented in the GrADS rescheduling architecture in which performance contract violations trigger rescheduling. The swapping rescheduler gathers information from sensors, analyzes performance information and determines whether and where to swap processes. We have designed and evaluated several policies[6] and we have experimentally evaluated our process swapping implementation using an N-body solver.[6,24]

4.2.2. Evaluation

This section describes how we used the MicroGrid to evaluate the GrADS rescheduling implementation.

The MicroGrid Understanding the dynamic behavior of rescheduling approaches for Grids requires experiments under a wide range of resource network configurations and dynamic conditions. Historically, this has been difficult, and simplistic experiments with either a few resource configurations or simple models of applications have been used. We use a general tool, the MicroGrid, which supports systematic, repeatable, scalable, and observable study of dynamic Grid behavior, to study the behavior of the process swapping rescheduling system on a range of network topologies. We show data from a run of an N-body simulation, under the N–N rescheduling system, running on the MicroGrid emulation of a distributed Grid resource infrastructure.

The MicroGrid allows complete Grid applications to execute on a set of virtual Grid resources. It exploits scalable parallel machines as compute platforms for the study of applications, network, compute, and storage resources with high fidelity. For more information on the MicroGrid see.(5,25,26)

Experiments with process-swapping rescheduling The first step in using the MicroGrid is to define the virtual resource and network infrastructure to be emulated. For our demonstration, we created a virtual Grid which is a subset of the GrADS testbed, consisting of machines at UCSD, UIUC, and UTK. The virtual Grid includes two clusters at UTK and UIUC and a single compute node at UCSD. The UTK cluster includes three 550 MHz Pentium II nodes. The UIUC cluster consists of three 450 MHz Pentium II machines. Both clusters are internally connected by Gigabit Ethernet. The single UCSD machine is a 1.7 GHz Athlon node. The latency between UCSD and the other two sites is 30 ms and between UTK and UIUC the latency is 11 ms. These configurations are described for MicroGrid in standard Domain Modeling Language (DML) and a simple resource description for the processor nodes.

The MicroGrid uses a Linux cluster at UCSD to implement its Grid emulation. We allocated two 2.4 GHz dual-processor Xeon machines for network simulation, and seven 450 MHz dual-processor Pentium II machines to model the compute nodes in the above virtual Grid.

To perform the process swapping rescheduling experiment on the virtual Grid, we first launched the MicroGrid daemons (instantiating the virtual Grid). From this point on, all processes launched on UCSD, UTK, or UIUC machines ran on the virtual Grid nodes. Second, we launched the contract monitor infrastructure (the Autopilot manager and contract monitor processes) and rescheduler process on the UCSD node. Third, we launched the N-body simulation application to the UTK and UIUC clusters which then connected to the contract monitor and rescheduler. All three of the initial active application processes started on the UTK nodes. At (virtual) time 80 s, we added two competitive processes to consume CPU time on one UTK machine. The rescheduling infrastructure detected poor performance and migrated all three working application processes to the UIUC cluster by time 150 s. Figure 4 shows the resulting application progress, first slowed by the competitive load, then increased by the migration to free resources.

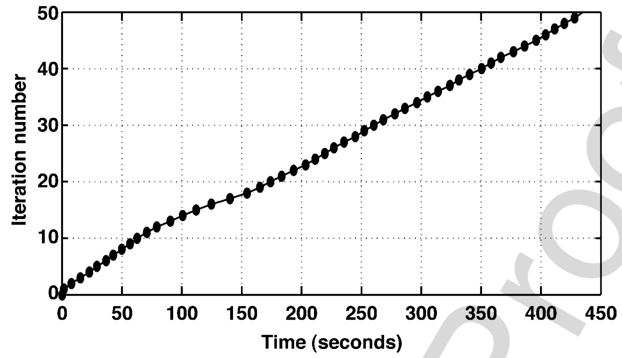

Fig. 4. Emulated application progress during N-body demonstration run.

5. FUTURE DIRECTIONS: VIRTUAL GRIDS

GrADS provided a foundation for an evolving compilation and execution infrastructure, *GrAD-Soft*, which we and others have used to conduct a range of application experiments(27–31) such as those described in this paper. These application experiments have not only validated the basic GrADS approach, but have also informed our focus on the most critical remaining challeges. These efforts are the focus of our new *Virtual Grid Application Development Software (VGrADS)* project.

One of the key lessons of the GrADS project is that the complexity of grid resource environments induces complexity in both application development and execution. First, execution on a shared grid of heterogeneous resources such as the TeraGrid forces an application developer to explicitly consider resource heterogeneity, dynamically fluctuating loads, and the interaction between local users and resource policies. There is little question that this complicates grid application programs, increasing programming difficulty and discouraging grid applications. Second, a rigid view by applications that prescribe a "perfect" set of resources, complicates resource management requiring search of a great expanse of resources and rapid, detailed matching of applications to resources. This too is a major technological challenge. Finally, it is our observation from working with many leading grid application teams that when faced with complex application performance structure and complex resource environments compounded with poor predictive information, expert programmers are reduced to use of *ad hoc* heuristics (albeit sophisticated ones) that require much tuning and debugging to achieve acceptable resource utilization and application performance.

Building on the knowledge and infrastructure of the GrADS project, our new approach adopts the concept of a *Virtual Grid (VGrid)* as a fundamental element of the software architecture which supports a separation of concerns for VGrADS.

Vgrids cleanly separate high-level programming tools, applications, and services from the complexity of dynamic grid scheduling and resource management. This approach is analogous to one that has proven effective in sequential and parallel computing contexts, where optimizations target abstract uniprocessors and multiprocessors rather than the physical resources themselves. The same concept will form the basis of our approach to simplifying the task of Grid application development.

Virtual grids support simpler high-level program preparation tools by providing simplified resource management and simple monitored performance guarantees. This supports the development and use of more powerful programming abstractions. We believe that virtual grids will enable the execution system to quickly and scalably identifying appropriate resources for applications, simplifying both application and system-level resource management. Finally, the virtual grid approach simplifies performance monitoring and resource adaptation by making explicit (and application neutral) the performance expectations and guarantees. In short, virtual grids provide a cleaner separation of responsibilities across the program preparation, execution system, and monitoring and adaptation systems, allowing each to be simplified and as a result more effective.

VGrADS research focuses on two major areas: execution environments and programming tools. *Execution environment* research explores the synthesis, coordination, and measurement of grid resources. The goals of this work are to explore (1) aggregation and virtualization of resource and Grid service aggregates; (2) intelligent, rapid resource selection and management in complex, heterogeneous environments; (3) performance measurement and tuning to achieve high individual application performance; and (4) fault-resilience through replication and intermediate, program state management. The resulting system will enable the nimble adaptation of applications to changing Grid conditions.

Programming tools research explores the mapping of two distinct, high-level programming models to VGrids. The *abstract parallel machine* model treats a computation as a collection of parallel tasks without concern for mapping that computation to the actual hardware. The *abstract component machine* model, on the other hand, represents a computation as a (possibly dynamic) graph of component invocations with specific data dependencies. In this model, applications and services might be high-level scripts that invoke operations from a component integration frame-work. The VGrADS execution system, working on behalf of the application, will use VGrids to instantiate both of these programming models.

ACKNOWLEDGMENTS

The research reported here was supported under National Science Foundation awards 9975020, 0103759, and 0331645.

REFERENCES

1. I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, 2nd Ed., Morgan Kaufmann (2003).
2. K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, S. L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski, Towards a Framework for Preparing and Executing Adaptive Grid Programs, *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium)*, Fort Lauderdale, Florida (April 2002).
3. R.L. Ribler, H. Simitci, and D.A. Reed, The Autopilot Performance-directed Adaptive Control System, *Future Generation Computer Systems*, **18**(1):175–187 (September 2001).
4. F. Vraalsen, R.A. Aydt, C.L. Mendes, and D.A. Reed, Performance Contracts: Predicting and Monitoring Grid Application Behavior, *Lecture Notes in Computer Science*, Vol. 2242, pp. 154–165, Springer Verlag (November 2001).
5. H. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, The MicroGrid: A Scientific Tool for Modeling Computational Grids, *Proceedings of SC2000* (November 2000).
6. O. Sievert and H. Casanova, Policies for Swapping MPI Processes, *Proceedings of HPDC-12, the Symposium on High Performance and Distributed Computing* (June 2003).
7. B. Barish and R. Weiss, Ligo and detection of gravitational waves, *Physics Today*, **52**(10) (1999).
8. S. Hastings, T. Kurc, S. Langella, U. Catalyurek, T. Pan, and J. Saltz, Image Processing on the Grid: A Toolkit or Building Grid-enabled Image Processing Applications, *3rd International Symposium on Cluster Computing and the Grid* (2003).
9. K. Taura and A. Chien, A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources, *Heterogeneous Computing Workshop* (May 2000).
10. S. Vadhiyar and J. Dongarra, A Metascheduler for the Grid, *Proceedings of the High Performance Distributed Computing Conference* (July 2002).
11. R. Wolski, J. Plank, J. Brevik, and T. Bryan, G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid, *Proceedings of 2001 International Parallel and Distributed Processing Symposium (1PDPS)* (March 2001).
12. Condor Team, Condor Version 6.4.7 Manual, //www.cs.wisc.edu/condor/manual/v6.4/.
13. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, A Directory Service for Configuring High-Performance Distributed Computations, *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pp. 365–375 (August 1997), URL papers/fitzgerald–hpdc97-mds.pdf.

14. R. Wolski, N.T. Spring, and J. Hayes, The network weather service: a distributed resource performance forecasting service for metacomputing, *Future Generation Computer Systems*, **15**(5–6):757–768 (1999), URL citeseer.nj.nec.com/wolski98network.html.
15. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of Np-Completeness*, MIT Press (1979).
16. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, Heuristics for Scheduling Parameter Sweep applications in Grid environments, *9th Heterogeneous Computing workshop (HCW'2000)* (2000).
17. Tracy D. Braun *et al.* A Comparision of eleven Static Heuristics for Maping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems, *Journal of Parallel and Distributed Computing*, **61**:810–837 (2001).
18. G. Marin, *Semi-Automatic Synthesis of Parameterized Performance Models for Scientific Programs*, Master's thesis, Department of Computer Science, Rice University (April 2003).
19. S. Ludtke, P. Baldwin, and W. Chiu, EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions, *J. Struct. Biol.*, **128**:82–97 (1999), URL http: //ncmi.bcm.tmc.edu/homes/stevel/EMAN/doc.
20. S. Vadhiyar and J. Dongarra, SRS A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems, *Parallel Processing Letters*, **13**(2):291–312 (June 2003), iSSN 0129-6264.
21. J.S. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski, The Internet Backplane Protocol: Storage in the Network, *NetStore99: The Network Storage Symposium* (1999).
22. L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammerling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, *ScaLAPACK User's Guide* (1997).
23. S. Vadhiyar and J. Dongarra, A Performance Oriented Migration Framework for the Grid, *IEEE Computing Clusters and the Grid (CCGrid, http://www.ccgrid.org)* (May 12–15 2003).
24. O. Sievert and H. Casanova, A Simple MPI Process Swapping Architecture for Iterative Applications, *The International Journal of High Performance Computing Applications* (2004), to appear.
25. X. Liu and A. Chien, Traffic-based Load Balance for Scalable Network Emulation, *Proceedings of SC2003* (November 2003).
26. H. Xia, H. Dail, H. Casanova, F. Berman, and A. Chien, Evaluating the GrADS Scheduler in Diverse Grid Environments Using the MicroGrid (May 2003), submitted for publication.
27. A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar, Numerical Libraries and the Grid, *Proceedings of SC'01* (November 2001).
28. M. Ripeanu, A. Iamnitchi, and I. Foster, Cactus Application: Performance Predictions in a Grid Environment, *Proceedings of European Conference on Parallel Computing (EuroPar)2001* (August 2001).
29. W. Chrabakh and R. Wolski, *GrADSAT: A Parallel SAT Solver for the Grid*, Technical Report CS-2003-05, University of California, Santa Barbara (2003), available from http://www.cs.ucsb.edu/research/trcs/index.shtml.
30. H. Dail, *A Modular Framework for Adaptive Scheduling in Grid Application Development Environments*, Master's thesis, University of California, San Diego, Department of Computer Science and Engineering (Mardh 2002), available as UCSD Technical Report CS2002-0698.

31. A. Mandal, *Mapping HPF onto the Grid*, Technical report TR03-417, Department of Computer Science, Rice University, Houston (November 2002), URL http://www.cs. rice. edu/~anirban/MSthesis.ps.gz.