New Grid Scheduling and Rescheduling Methods in the GrADS Project

K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey
Computer Science Dept., Rice University

F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia
Dept. of Computer Science, University of California at San Diego

L. Johnsson, B. Liu, M. Patel
Dept. of Computer Science, University of Houston

D. Reed, W. Deng, C. Mendes
Dept. of Computer Science, University of Illinois

Z. Shi, A. YarKhan, J. Dongarra
ICL, University of Tennessee

Abstract

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: (1) A new approach to scheduling workflow computations, applied to a 3-D image reconstruction application; (2) A simple stop/migrate/restart approach to rescheduling Grid applications, applied to a QR factorization benchmark; and (3) 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) and the results were demonstrated at the SC2003 conference.

1. Introduction

Since late 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, as set forth in *The Grid: Blueprint for a New Computing Infrastructure* [5]. The Grid connects computers, databases, instruments, and people in a seamless web, supporting computation-rich application concepts such as distributed supercomputing, smart instruments, and data-mining. However, its use has been limited to specialists, principally due to a lack of usability.

Because the Grid is inherently more complex than standalone 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, including a prototype execution environment and programming 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 [8]. 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 (e.g. an MPI program), a *mapper* that determines how to map an application's tasks to a set of resources, and an executable *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 Figure 1 depicts tools used to construct COPs. GrADS supports application development either by assembling domain-specific components from a high-level toolkit or by creating a module by relatively low-level (e.g., MPI) coding. 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.

The right side of Figure 1 depicts actions when a COP is delivered to the execution environment. The GrADS infrastructure first determines which resources are available and,

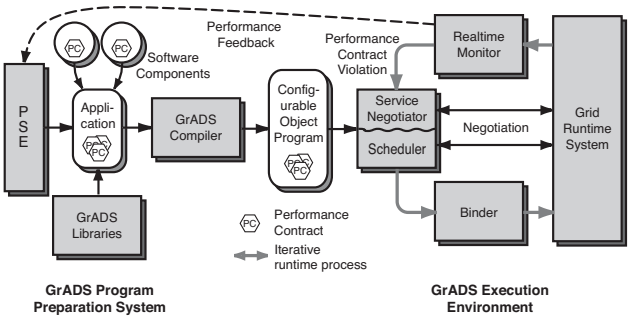

Figure 1. GrADS Program Preparation and Execution Architecture

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 [13], 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 [23], 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.

The GrADS facilities described thus far were, with the exception of the rescheduler, implemented in the first prototype GrADS system that was demonstrated at SC2002. In the year between SC2002 and SC2003, several new capabilities were added. First, we significantly enhanced the GrADS binder to support execution on heterogeneous computing resources. Second, we developed a new approach to the scheduling of workflow applications, which are extremely common in a Grid environment. Finally, we developed two different approaches to rescheduling and implemented them in the GrADS software system. These extensions, which were demonstrated at SC2003, are elaborated in the remainder of this paper.

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 one cluster at University of California at San Diego (UCSD, 10 machines), two clusters at University of Tennessee at Knoxville (UTK, 24 machines), two clusters at University of Illinois at Urbana-Champaign (UIUC, 24 machines), and one cluster at 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 [16]. 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 [14].)

Clearly, 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. (Early iterations of the GrADS testbed included only clusters of Pentiums.) It soon became clear that this approach would not suffice for a general system because most grids are homogeneous and because it was important to support linking GrADS programs against libraries of components that were preinstalled on resources across the Grid.

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 Grid 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 GrADS Information Service (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 and instruments the code. After the local code is instrumented with Autopilot sensors, it is configured and compiled. 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. This will be explored in future GrADS research.

3 Scheduling Workflow Graphs

Workflow applications are an important class of programs that can take advantage of the power of Grid computing. The LIGO [1] pulsar search and several image processing applications [7] are examples of workflow applications that harness the power of the Grid. 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 [17, 20, 24] and was not well suited to workflow applications. On the other hand, existing approaches to workflow scheduling, such as Condor DAGMan [18], 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. Examples of such objective functions include minimizing communication time and volume, minimizing overall job completion time, and 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 [4] and NWS [25]. 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:

$$rank(c_i, r_j) = w_1 \times eCost(c_i, r_j) + w_2 \times dCost(c_i, r_j)$$

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 [6]. 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 [3, 19].

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 [11]. 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 [10], 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 micrographs. Human intervention and expertise is needed to define a preliminary 3-D model from the electron micrographs, 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

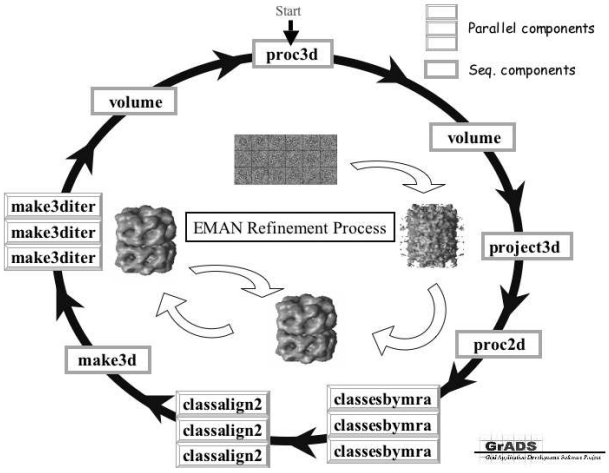

Figure 2. EMAN refinement workflow

forms a linear graph in which some components can be parallelized.

We have scheduled the EMAN refinement workflow components using the GrADS workflow scheduler (incorporating the constructed performance models) and executed the workflow on a grid of heterogeneous platforms using the GrADS binder, which enabled us to use both IA-32 and IA-64 machines, validating our approach to heterogeneity. This configuration was demonstrated live at the SC2003 conference.

4. Rescheduling

Normally, a contract violation activates the GrADS *rescheduler*. The rescheduling 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) [22] 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) [12] 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 initialization, 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 [13] 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 [2] 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

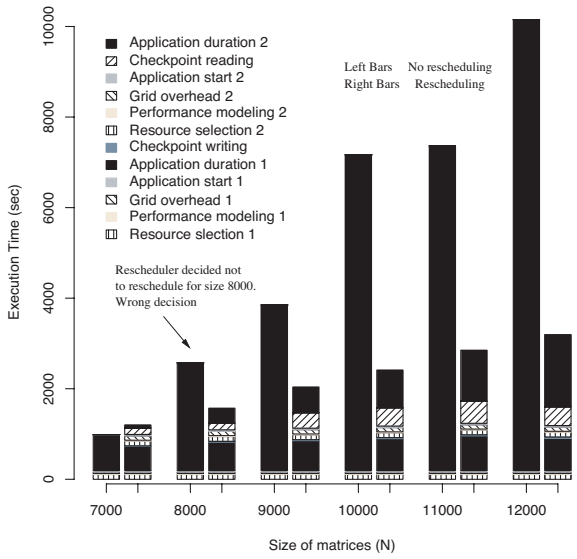

Figure 3. Problem size and migration

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 Figure 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 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 seconds while the actual rescheduling cost was about 420 seconds. 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 [21], 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 [15], 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 [15]) 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 [14] and we have experimentally evaluated our process swapping implementation using an N-body solver [14, 15].

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 [9, 16, 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 550Mhz Pentium II nodes. The UIUC cluster consists of three 450Mhz 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 30ms, and between UTK and UIUC the latency is 11ms. 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 2.4Ghz dual-processor Xeon machines for network simulation, and seven 450Mhz 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 seconds, we added two competitive processes to consume CPU time on one UTK machine.

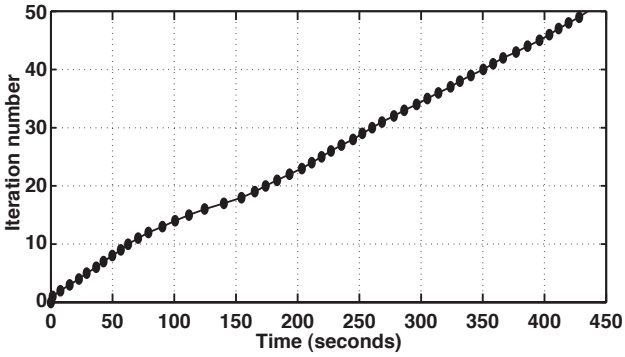

Figure 4. Emulated application progress during N-body demonstration run

The rescheduling infrastructure detected poor performance and migrated all three working application processes to the UIUC cluster by time 150 seconds. Figure 4 shows the resulting application progress, first slowed by the competitive load, then increased by the migration to free resources.

5. Conclusions and Future Work

Although the results presented here are preliminary, they establish several promising outcomes of the GrADS Project:

- Rescheduling by stop/migration/restart and by single-processor swapping are both feasible and require little additional programming. Of the two approaches, stop/migration/restart is very flexible, whereas the overhead for processor swapping is quite low.
- Advanced scheduling of workflow applications can be done successfully given the necessary infrastructure. In the GrADS framework, this includes good node performance estimation, a distributed launch process and suitable scheduling heuristics.
- Grid computations can be successfully emulated by a controllable testbed (i.e., the MicroGrid). This allows a much wider range of experimentation and development than traditional full-scale testing on the Grid, as well as more accurate analysis of the results.
- Using simple computation and communication performance metrics, captured via PAPI and the MPI profiling interface with automatically-inserted sensors, allows the detection of performance variations. Moreover, these techniques enable GrADS applications to reschedule themselves for improved performance.

We have recently started to apply these insights in our new Virtual Grid Application Development (VGrADS)

project. This project adds an abstraction layer called "virtual Grids" (vgrids) to the current Grid infrastructure. A vgrid will incorporate many of the GrADS techniques discussed here, notably the workflow scheduler and the rescheduling mechanisms, as well as new capabilities, such as fault tolerance and "Grid economies" for allocating resources.

Acknowledgement

The research reported here is based on work supported by the National Science Foundation Next Generation Software Program under NSF awards EIA-9975020 and ACI-0103759.

References

[1] B. Barish and R. Weiss. Ligo and detection of gravitational waves. *Physics Today*, 52(10), 1999.

[2] 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.

[3] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing workshop (HCW'2000)*, 2000.

[4] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, August 1997.

[5] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition, 2003.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of Np-Completeness*. MIT Press, 1979.

[7] 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. In *3rd International Symposium on Cluster Computing and the Grid*, 2003.

[8] 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. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium)*, Fort Lauderdale, Florida, April 2002.

[9] X. Liu and A. Chien. Traffic-based load balance for scalable network emulation. In *Proceedings of SC2003*, November 2003.

[10] S. Ludtke, P. Baldwin, and W. Chiu. EMAN: Semiautomated software for high-resolution single-particle reconstructions. *J. Struct. Biol.*, 128:82–97, 1999.

[11] G. Marin. Semi-automatic synthesis of parameterized performance models for scientific programs. Master's thesis, Dept. of Computer Science, Rice University, April 2003.

[12] J. S. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The internet backplane protocol: Storage in the network. In *NetStore99: The Network Storage Symposium*, 1999.

[13] R. L. Ribler, H. Simitci, and D. A. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, 2001.

[14] O. Sievert and H. Casanova. Policies for swapping MPI processes. In *Proceedings of HPDC-12, the Symposium on High Performance and Distributed Computing*, June 2003.

[15] 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.

[16] H. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: A scientific tool for modeling computational grids. In *Proceedings of SC2000*, November 2000.

[17] K. Taura and A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, May 2000.

[18] D. Thain and M. Livny. Building reliable clients and services. In *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter 19. Morgan Kaufmann, second edition, 2003.

[19] 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.

[20] S. Vadhiyar and J. Dongarra. A metascheduler for the grid. In *Proceedings of the High Performance Distributed Computing Conference*, July 2002.

[21] S. Vadhiyar and J. Dongarra. A performance oriented migration framework for the grid. In *IEEE Computing Clusters and the Grid (CCGrid)*, May 12-15 2003.

[22] 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.

[23] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed. Performance contracts: Predicting and monitoring grid application behavior. In *Lecture Notes in Computer Science*, volume 2242, pages 154–165. Springer Verlag, November 2001.

[24] R. Wolski, J. Plank, J. Brevik, , and T. Bryan. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium (IPDPS)*, March 2001.

[25] 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.

[26] H. Xia, H. Dail, H. Casanova, F. Berman, and A. Chien. Evaluating the GrADS scheduler in diverse grid environments using the microgrid. submitted for publication, May 2003.