Numerical Libraries and the Grid

Antoine Petitet, Susan Blackford, Jack Dongarra, Brett Ellis, Graham Fagg, Kenneth Roche and Sathish Vadhiyar

International Journal of High Performance Computing Applications 2001 15: 359
DOI: 10.1177/109434200101500403

NUMERICAL LIBRARIES AND THE GRID

Antoine Petitet

SUN FRANCE BENCHMARK CENTER, PARIS, FRANCE

Susan Blackford

MYRICOM, INC.

Jack Dongarra
Brett Ellis
Graham Fagg
Kenneth Roche
Sathish Vadhiyar

UNIVERSITY OF TENNESSEE, KNOXVILLE

Summary

This paper describes an overall framework for the design of numerical libraries on a computational grid of processors in which the processors may be geographically distributed and under the control of a grid-based scheduling system. Experiments are presented in the context of solving systems of linear equations using routines from the ScaLAPACK software collection along with various grid service components, such as Globus, NWS, and Autopilot.

Address reprint requests to Jack Dongarra, Department of Computer Science, University of Tennessee, 1122 Volunteer Boulevard, Suite 203, Knoxville, TN 37996-3450, U.S.A; dongarra@cs.utk.edu.

The International Journal of High Performance Computing Applications,
Volume 15, No. 4, Winter 2001, pp. 359-374

Motivation on the Grid

The goal of the Grid Application Development Software (GrADS) project (Berman et al., 2000) is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. The GrADS project is exploring the scientific and technical problems that must be solved to make grid applications development and performance tuning for real applications an everyday practice. This requires research in four key areas, each validated in a prototype infrastructure that will make programming on the Grid a routine task:

1. Grid software architectures that facilitate information flow and resource negotiation among applications, libraries, compilers, linkers, and runtime systems;
2. Base software technologies, such as scheduling, resource discovery, and communication, to support development and execution of performance-efficient grid applications;
3. Languages, compilers, environments, and tools to support creation of applications for the Grid and solution of problems on the Grid;
4. Mathematical and data structure libraries for grid applications, including numerical methods for control of accuracy and latency tolerance.

In this paper, we will describe the development of a prototype system designed specifically to be used with numerical libraries in the grid setting and the results of experiments with routines from the ScaLAPACK library (Blackford et al., 1997) on the Grid.

Motivation on Numerical Libraries

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

Some of the challenges in this arena have already been encountered. For example, to make effective use of current high-end machines, the software must manage both communication and the memory hierarchy. This problem has been approached with a combination of compile-time and runtime techniques. On the Grid, the increased scale of computation, depth of memory hierarchies, range of latencies, and increased variability in the runtime environment will make such problems more difficult.

To address these issues, we must rethink the way that we build libraries. The issues to consider include software architecture, programming languages and environments, compile-time versus runtime functionality, data structure support, and fundamental algorithm design. The challenges are as follows:

- The library software must support performance optimization and algorithm choice at runtime.
- The architecture of software libraries must facilitate efficient interfaces to a number of languages, as well as effective support for the relevant data structures in those languages.
- New algorithmic techniques will be a prerequisite for latency tolerant applications.
- New scalable algorithms that expose massive numbers of concurrent threads will be needed to keep parallel resources busy and to hide memory latencies.
- Library mechanisms that will interact with the Grid will be needed to dynamically deploy resources in solving the posed users' problems.

These considerations lead naturally to a number of important areas in which research in algorithm design and library architecture is needed.

". . . to make effective use of current high-end machines, the software must manage both communication and the memory hierarchy. This problem has been approached with a combination of compile-time and runtime techniques."

Grid-Aware Libraries. To enable the use of the Grid as a seamless computing environment, we are developing parameterizable algorithms and software annotated with performance contracts. These annotations will help a dynamic optimizer tune performance for a wide range of architectures. This tuning will, in many cases, be accomplished by having the dynamic optimizer and runtime system provide input parameters to the library routines that will enable them to make a resource-efficient algorithm selection. We are also developing new algorithms that use adaptive strategies by interacting with other GrADS components. For example, libraries will incorporate performance contracts for dynamic negotiation of resources, as well as runtime support for adaptive strategies to allow the compiler, scheduler, and runtime system to influence the course of execution.

We are using the grid information service and the grid event service to obtain the information needed to guide adaptation in the library routines. These services will be discussed in more detail later in this paper.

Latency-Tolerant Algorithm. Remote latency is one of the major obstacles in achieving high efficiency on today's high performance computers. The growing gap between the speed of the microprocessors and memory coupled with a deep memory hierarchy implies that the memory subsystem has become a large performance factor in modern computer systems such as the Department of Energy's Advanced Strategic Computing Initiative computers. In the unpredictable and dynamic world of the Grid, this problem will be even worse. Research into latency-tolerant algorithms that explore a wider portion of the latency and bandwidth/memory space is needed. Furthermore, tools are needed for measuring and managing latency. We are designing and constructing numerical libraries that are parameterized to allow their performance to be optimized over a range of current and future memory hierarchies, including those expected in computational grids.

Compiler-Ready Libraries. In the past, library development has typically focused on one architecture at a time, with the result that much of the work must be repeated to migrate the code to a new architecture and its memory hierarchy. We are exploring the design and development of parameterized libraries that permit performance tuning across a wide spectrum of memory hierarchies. Some developers of portable libraries rely on tools such as the HPC Compiler to analyze and parallelize their programs. These compilers are large and complex; they do not always discover parallelism when it is available. Automatic parallelization may be adequate for some simple types of scientific programming, but experts writing libraries find it frustrating because they must often perform tedious optimizations by hand—optimizations that could be handled by the compiler if it had a bit of extra information from the programmer. Programmers may find it both easier and more effective to annotate their code with information that will help the compiler generate the desired code with the desired behavior. We have begun to identify opportunities where information about algorithms contained in library functions can help the compiler and the runtime environment and work with the GrADS compiler group to develop a new system of annotation to provide information about semantics and performance that aids in compilation. At the library interface level, this would include memory-hierarchy tuning parameters and semantic information, such as dependency information to make it possible to block the LU factorization of a matrix, and floating-point semantic information (e.g., to indicate that it is acceptable to reorder certain floating-point operations or to handle exceptions in particular ways). The goal is to make it possible to "build in" to the compiler knowledge about these libraries far beyond what can be derived by a compile-time analysis of the source.

Current Numerical Subroutine Libraries

Current numerical libraries for distributed memory machines are designed for heterogeneous computing and are based on MPI (Snir et al., 1998) for communication between processes. Two such widely used libraries are ScaLAPACK and PETSc (Balay et al., 1996), designed for dense and sparse matrix calculations, respectively. ScaLAPACK assumes a two-dimensional block cyclic data distribution among the processes, and PETSc provides a block-row or application-specific data distribution. The user must select the number of processes associated with an MPI communicator and the specific routine/algorithm to be invoked.

In the case of ScaLAPACK, the user has total control over the exact layout of the data, specifying the number of block rows and the number of block columns in each block to be distributed. These blocks are then distributed cyclically to maintain proper load balance of the application on the machine. It is the user's responsibility to distribute the data prior to invoking the ScaLAPACK routine. If the user makes a poor choice of data layout, it can significantly affect his application's performance. All data that have been locally distributed can be explicitly accessed in the user's program.

For PETSc, the library has a variety of data distribution routines from which to choose. The user can select the default block row distribution where the size of the block is determined by the PETSc system as a function of the size of the global matrix to be distributed. PETSc will choose the block size such that the matrix is evenly distributed among the processes. The user can also select an application-specific block-row distribution whereby the size of the block is a function of the application to be run on that process's data. In contrast to ScaLAPACK, the user does not have explicit access to individual elements in the data structure. The user must use specialized PETSc matrix manipulation routines to access the matrix data.

For both libraries, the user is responsible for making many decisions on how the data are decomposed, the number of processors used, and which software is to be chosen for the solution. Given the size of the problem to be solved, the number of processors available, and certain characteristics of the processors, such as CPU speed and the amount of available memory per processor, heuristics exist for selecting a good data distribution to achieve efficient performance. In addition, the *ScaLAPACK Users' Guide* (Blackford et al., 1997) provides a performance model for estimating the computation time given the speed of the floating-point operations, the problem size, and the bandwidth and latency associated with the specifics of the parallel computer. Equation (1) provides the model used by ScaLAPACK for solving a dense system of equations.

$$T(n, p) = C_f t_f + C_v t_v + C_m t_m, \tag{1}$$

where

$C_f = \dfrac{2n^3}{3p}$	=	total number of floating-point operations per processor
$C_v = \left(3 + \dfrac{1}{4}\log_2 p\right)\dfrac{n^2}{\sqrt{p}}$	=	total number of data items communicated per processor
$C_m = n(6 + \log_2 p)$	=	total number of messages
t_f	=	time per floating-point operation
t_v	=	time per data item communicated
t_m	=	time per message
n	=	matrix size
p	=	number of processors
$T(n, p)$	=	parallel execution time for a problem of size n run on p processors

The performance model assumes that the parallel computer is homogeneous with respect to both the processors and communication network. With the Grid, both of these assumptions are incorrect and a performance model becomes much more complex. With the dynamic nature of the grid environment, the grid scheduler must assume the task of deciding how many processors to use and the placement of data. This selection would be performed in a dynamic fashion using the state of the processors and the communication behavior of the network within the Grid in conjunction with a performance model for the application. The system would then determine the data layout, the number and location of the processors, and perhaps the algorithm selection for a given problem for the best time to solution on the Grid.

> ***"Our goal is to adapt the existing distributed memory software libraries to fit into the grid setting without too many changes to the basic numerical software."***

Adapting Current Libraries to the Grid Environment

Our goal is to adapt the existing distributed memory software libraries to fit into the grid setting without too many changes to the basic numerical software. We want to free the user from having to allocate the processors, make decisions on which processors to use to solve the problem, make decisions on how the data are to be decomposed to optimally solve the problem, allocate the resources, start the message-passing system, monitor the progress, migrate or restart the application if a problem is encountered, collect the results from the processors, and return the processors to the pool of resources.

Implementation Outline of the GrADS ScaLAPACK Demo

The ScaLAPACK experiment demonstrates the feasibility of solving large-scale numerical problems over the Grid and analyzes the added cost of performing the calculation over machines spanning geographically distributed sites. We solve a simple linear system of equations over the Grid using Gaussian elimination with partial pivoting via the ScaLAPACK routine, PDGESV. We illustrate how users, without much knowledge of numerical libraries, can seamlessly use numerical library routines over the Grid. We also outline the steps that are necessary for a library writer to integrate his library into the grid system. (The appendix contains a more detailed description of these parts.) Although ease of use is an important aspect of the Grid, performance improvement and the ability to perform tasks that are too large to be performed on a single tightly coupled cluster are important motivations behind the use of the Grid. Our experiments show that effective resources can be selected to solve the ScaLAPACK problem and, for our experiments, scalability (as the problem size and the number of processors increase) of the software is maintained.

Before the user can start his application, the grid system is assumed to have initialized three components: Globus MDS (Foster and Kesselman, 1997), Network Weather Service (NWS) (Wolski, Spring, and Hayes, 1999) sensors on all machines in the Globus MDS repository, and the Autopilot manager/contract monitor (Ribler et al., 1998). We assume that the user has already communicated with the grid system (Globus) and has been authenticate to use the grid environment. The Globus MDS maintains a repository of all available machines in the Grid, and the NWS sensors monitor a variety of system parameters for all of the machines contained in the Globus MDS repository. This information is necessary for modeling the performance of the application and for making scheduling decisions for the application on the Grid. Autopilot was designed and is maintained at the University of Illinois at Urbana-Champaign (UIUC). It is a system for monitoring the application execution and enabling corrective measures, if needed, to improve the performance while the application is executing. The Autopilot manager must be running on one of the machines in the Grid prior to the start of the experiment. The library routine (in our experiment, this is the ScaLAPACK code itself) must be instrumented with calls to Autopilot monitoring. It is hoped in the future that this instrumentation can be done by the compilation system. However, for our experiment, all instrumentation was inserted by hand. The contract monitor is a component that is started along with the application. Its main job is to monitor whether the application execution is meeting its performance guarantees. When the application starts executing, the sensors associated with the application register with the Autopilot manager. The contract monitor looks up the Autopilot manager to get information about the sensors and directly receives the application performance data from the sensors. Work is under way to implement an Autopilot service that maintains a pool of Autopilot managers and assigns an Autopilot manager from the pool to a particular application execution and contract monitoring. At present, however, the name of the machine running the Autopilot manager must be supplied by the user (see details below). There is also an ongoing effort in the compiler group of GrADS to produce a generic Configurable Object Program (Berman et al., 2000), and hence the user will not be required to maintain separate executables for each machine in the Grid. However, at present, this feature is not available. After the user has compiled the executable, he is responsible for copying this executable to every machine in the Grid.

After these preliminary steps have been completed, the user is now ready to execute his application on the Grid. The user interface to the ScaLAPACK experiment in the GrADS system is the routine Grads_Lib_Linear_Solve. This routine is the main numerical library driver of the GrADS system and calls other components in GrADS. It accepts the following inputs: the matrix size, the block size, and an input file. This input file contains information such as the machine on which the Autopilot manager is running, the path to the contract monitor, the subset of machines in the Grid on which the user could run his application, the path to the executable on each machine, and so on. In the future, this file will not be necessary, since most of the parameters in the file will be maintained by the grid system. For purposes of this experiment, the input matrices are either read from a URL or randomly generated. A more flexible user interface for the generation and distribution of matrices is being developed. Future development will also encompass the automatic determination of the value of the block size that will yield the best performance of the application on the Grid. This block size determines the data distribution of the matrices to the processors in the Grid, and likewise the size of the computational kernel to be used in the block-partitioned algorithm.

The Grads_Lib_Linear_Solve routine performs the following operations:

1. Sets up the problem data;
2. Creates the "coarse grid" of processors and their NWS statistics by calling the resource selector;
3. Refines the coarse grid into a "fine grid" by calling the performance modeler;
4. Invokes the contract developer to commit the resources in the fine grid for the problem (repeat steps 2-4 until the fine grid is committed for the problem);
5. Launches the application to execute on the committed fine grid.

The appendix contains additional details about these steps, as well as pseudocode for Grads_Lib_Linear_Solve() and APIs for the GrADS components.

SETTING UP THE PROBLEM

This step supplies the GrADS system data structures with the user-passed parameters. In the future, this step will involve contacting the GrADS name server to get information about the Autopilot managers. This step will also involve the building of the Configurable Object Program, which will be an extension to the normal object code, encapsulating annotations about the runtime system and user preferences. These annotations will later be used for rescheduling the applications when the grid parameters change. Finally, future enhancements will entail the automatic determination of the best block size for this DGEMM-based application on each of the machines in the Grid.

RESOURCE SELECTOR

The resource selector contacts the MDS server, maintained by the Information Sciences Institute as part of the Globus system, to check the status of the machines needed by the user for the application. If the MDS server does not detect failures with the machines, the resource selector then contacts the NWS, maintained at the University of Tennessee (UT), to obtain machine-specific information pertaining to available CPU, available memory, and latency and bandwidth between machines. At the end of the resource selection step, a coarse grid is formed. This coarse grid is essentially all of the machines available along with the statistics returned by NWS.

PERFORMANCE MODELER

The performance modeler calculates the speed of the machine as a percentage of the peak Mflop/s on the machine. The user currently supplies the peak Mflop/s rate of the machine. In the future, the GrADS system will be able to determine the peak Mflop/s rate of a machine. The percentage used in the calculation of speed is heuristically chosen by observing previous ScaLAPACK performance, in this case routine PDGESV, on the given architecture. Typically, PDGESV achieves approximately 75% of the local DGEMM (matrix multiply) performance per processor, and because the percentage of peak performance attained by DGEMM is approximately 75%, we use a heuristic measure of 50% of the theoretical peak performance for routine PDGESV from ScaLAPACK.

The performance modeler then performs the following steps in determining the collection of machines to use for the problem:

- i. Determine the amount of physical memory that is needed for the current problem.
- ii. If possible, find the fastest machine in the coarse grid that has the memory needed to solve the problem.
- iii. Find a machine in the coarse grid that has the maximum average bandwidth relative to the other machines in the Grid. Add this to the fine grid.
- iv. Do the following:
  - a. Find the next machine in the coarse grid that has maximum average bandwidth relative to the machines that are already in the fine grid.
  - b. Calculate the new time estimate for the application using the machines in the fine grid. This time estimate is calculated by executing a performance model for the application. We are assuming that the library writer has provided a performance model for the library routine. This performance model takes into account the speed of the machines as well as the latency and bandwidth as returned by the NWS.
  - c. Repeat step iv until the time estimate for the application runtime increases.
- v. If a single machine is found in step ii, the time estimate for the problem using the single machine is compared with the time estimate for the problem using the machines found in steps iii and iv. If the

time estimate for the machine found in step ii is less than the time estimate for the machines found in steps iii and iv, then use the single machine in step ii for the fine grid. Else, use the machines found in steps iii and iv for the fine grid. If step ii was not able to find a single machine that is able to solve the problem, then the machines found in steps iii and iv are used for the fine grid.

At the end of performance modeling, the fine grid, which consists of a subset of machines that can solve the problem in the fastest possible time, given the grid parameters, is committed to the run.

The performance model, for the current case of solving a system of linear equations on a heterogeneous computational grid, relies specifically on modeling the time to solution for the PDGESV kernel. This is the routine that is responsible for solving the linear system. A full description of how PDGESV solves the system may be found in the *ScaLAPACK Users' Guide* (Blackford et al., 1997). It should be recalled that the process is dominated by an LU factorization of the coefficient matrix A; that is, solving the equations is a Level 3 BLAS process. As such, the time to perform this factorization is arguably the major time constraint in the time to solution for solving a system of linear equations, in particular, as the problem size grows larger.

Figure 1 provides a measure of how precisely the current performance model is making predictions. The plots are a statistical study of grid-based runs on the TORC cluster at UT. The runs represent a homogeneous, nondedicated cluster case. This is the simplest realistic case possible because the communication lines, available memory, and CPU architectures are the same for each computing node. The way the model makes decisions is based on the grid conditions returned at the time of request as described above. More specifically, in the current model the problem is distributed over the fine grid in a one-dimensional block-cyclic fashion with N/NB-sized panels (which are determined from the size of A, the size of N, and the chosen block size, NB). In the LU factorization there are three major steps: a factorization phase, a broadcast phase, and an update phase. The current model predicts times for each of these phases as would be reflected on the root-computing node of the fine grid. The dominant phase is the update, which is a call to the Level 3 PBLAS (Blackford et al., 1997) routine PDGEMM.

The first plot of Figure 1 is the total wall clock time measured for performing the linear solution for problem sizes ranging from $N = 600$ to $N = 10{,}000$. The corresponding predicted values are also shown. Five runs were made for each data point on this linear-log plot. The second plot gives a precise comparison of the ratios of these numbers. Naturally, in the second plot, we see the outlying data points for the smaller problem sizes. If one disregards those points, the current model is better than 95% accurate on average for this simplistic, homogeneous cluster study.

CONTRACT DEVELOPER

Currently, the contract developer commits all of the machines that were returned by the performance modeler for the application. In the future, the contract developer will be more robust and able to build contracts for the given user problem and the grid parameters (Vraalsen et al., 2001; Vetter and Reed, 2000).

APPLICATION LAUNCHER

The application launcher spawns the parallel application over the machines in the fine grid and starts the contract monitor. The sensors associated with the application register themselves with the Autopilot manager. The contract monitor, through the Autopilot manager, contacts the sensors, obtains instrumentation data from the sensors, determines whether the application behaves as predicted by the application model, and prints the output. In the future, the contract monitor will send its output to a scheduler, which may then reschedule, perhaps migrate, the application if the contracts are violated. As soon as the parallel application has been spawned, the input matrices are randomly generated (or read from a URL) and block cyclically distributed among the processors in the fine grid. After the input matrices are distributed, the ScaLAPACK routine PDGESV is invoked to solve the system of linear equations, the validity of the solution is checked, and the results are returned to the user's application.

Experiments

Two sets of experiments were conducted. The first set of experiments compares the grid version of ScaLAPACK (using MPICH-G) and the native ScaLAPACK (using MPICH-P4) as implemented on a cluster. These measurements give an estimate of the overhead cost associated with performing numerical computations using the GrADS system. The second set of experiments illustrate the functionality of GrADS by running the application on a dynamically chosen number of processors that exist and are available on the Grid.

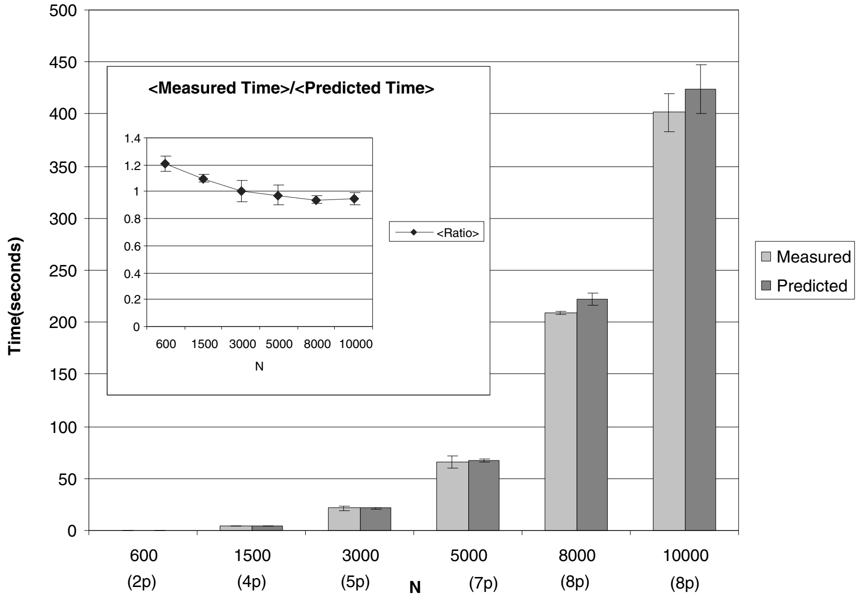

Fig. 1 **Performance model versus measured performance**

In viewing the following graphs, it is important to note that the GrADS strategy for building the fine grid attempts to optimize the time to solution. This model does not optimize for best efficiency. However, the model could be changed to alternatively optimize for best efficiency. There are many possible grid resources (dedicated, shared, etc.), and the needs of the user changes depending on the types of resources available to him. Similarly, his definition of best performance could be measured in terms of Mflop/s to most effectively utilize the CPUs in his machine, instead of minimum time to solution. Keeping these issues in mind, in Figure 2, the number of processors selected by the GrADS strategy for the fine grid is close to the best choice for raw ScaLAPACK with respect to the time-to-solution criterion and the one-dimensional mapping of processors constraint.

COMPARISON OF GRID AND RAW SCALAPACK

In the following experiments, ScaLAPACK runs using the GrADS system were compared with the native ScaLAPACK runs without the GrADS system on a local Linux cluster (TORC) (http://icl.cs.utk.edu/projects/torc/) at UT. Each machine is a dual-processor 550-MHz Pentium III running Linux, and only one process was spawned per node. The comparison is between the (ScaLAPACK+MPICH-G) (Foster and Karonis, 1998) performance over the Grid and the (ScaLAPACK+ MPICH-P4) (Gropp et al., 1996; Gropp and Lusk, 1996) performance without the Grid. Although the goal of the GrADS system is to solve large problems across multiple wide-area clusters, the following experiments reveal the costs of grid-related overhead in the most ideal setting, a local cluster. The results yield an approximate 30% overhead in running MPICH-G versus MPICH-P4. In a best-case scenario, this is a lower bound on the overhead to be incurred when running across geographically distributed clusters. Experiments were run using the TORC cluster in dedicated and nondedicated modes.

Figure 2 depicts a sample case from the nondedicated runs on TORC. It is a statistical set of measurements (10 runs for each point on the graph) for the total turnaround time involved in solving a linear system of equations with 5000 unknowns. In other words, the interval from the time the user submits a request to the time the system has been solved and the application has been cleanly removed from the computing resource is compared.

As mentioned, the GrADS linear system solver is built with ScaLAPACK embedded in grid-based software. Thus, it is important to understand what overhead is introduced when going from the raw ScaLAPACK runs to the full grid-based application. A fair question to ask for the grid-based runs for a given problem size is: “How can one guarantee that the fine grid will consist of exactly X processors, where X is varying for a set problem size?” Actually, if you allow the entire set of grid-based computing resources to be considered when solving the $N = 5000$ problem, the answer is you cannot. However, one can constrain the study to the TORC cluster and to a preset number of processors through a well-defined configuration file used at runtime. Furthermore, although restriction to a cluster is easy to impose, the number of nodes requested versus the number that is actually selected for the fine grid cannot be imposed with certainty. This is due to the performance model. The point is, once the maximum number of compute nodes requested becomes larger than the optimal number predicted by the model, the request is ignored and the model chooses the number it thinks would best solve the problem in a timely fashion. Naturally, because the grid resources are dynamic, this optimal number of machines in the fine grid will vary. Thus, the number of nodes for the grid-based runs is seen to terminate at six processors in the plot.

The plot in Figure 2 compares three sets of measurements:

1. The MPICH1.2.0-P4 + ScaLAPACK1.6 + MPIBLACS1.1 numbers—which are referred to as the raw ScaLAPACK runs;
2. The MPICH-G1.1.2 (Globus-based MPI) + Autopilot2.3 + ScaLAPACK1.6 + MPIBLACS1.1 runs;
3. The MPICH-G1.1.2 (Globus-based MPI) + NWS2.0pre2 + MDS + Autopilot2.3 + ScaLAPACK1.6 + MPIBLACS1.1 runs—the actual grid runs.

In this example, the raw ScaLAPACK had a minimal runtime, on average, when run on five processors (as did the type 2 runs). The grid-based runs were solved the fastest on four of the TORC computing nodes.

Clearly, going from the type 1 to type 3 runs incurs additional overhead in time associated with the gathering of information for the grid run. A detailed breakdown of where this extra time comes from in the grid application is provided in Figure 3.

With the exception of process spawning, the grid overhead remains more or less constant as the size of the problem and the number of processors chosen increase. The time it takes to spawn processes in the Grid is noticeably more expensive than in the non-Grid case, and increases with the number of processes. Because the complexity of the problem being solved is $O(n^3)$, for very large problems this overhead will become negligible.

ScaLAPACK ACROSS CLUSTERS

In this experiment, we used two separate clusters from UT called TORC and CYPHER (http://www.cs.utk.edu/sinrg/) and a cluster from UIUC called OPUS. For this experiment, four TORC machines, six CYPHER machines, and eight OPUS machines were available. Table 1 shows some of the system parameters of the machines.

Figure 4 shows the total time taken for GrADS for each experiment as the matrix size is increased.

In the above experiments, GrADS chose only OPUS machines at UIUC for matrix sizes up to 8000. In fact, for this problem size, the system can be solved on one cluster. The OPUS and the CYPHER clusters are comparable in terms of their network parameters. At the time the experiments were conducted, the OPUS cluster was found to have better network bandwidth than the CYPHER cluster due to the network load on CYPHER. Hence, the OPUS cluster was the best choice among the pool of resources. For a matrix size of 8000, the amount of memory needed per node is on the order of 512 MB. Because none of the UIUC machines has this much memory, GrADS used

> *". . . although restriction to a cluster is easy to impose, the number of nodes requested versus the number that is actually selected for the fine grid cannot be imposed with certainty."*

both the UIUC and CYPHER machines at UT for matrix sizes larger than 8000. The GrADS framework gave preference to the CYPHER machines over the TORC machines because of the superior network speed of the CYPHER network. Hence, we find a steep increase in execution time from matrix size 7000 to 8000.

We also found that the number of machines chosen for a matrix size of 10,000 is smaller than the number of ma-

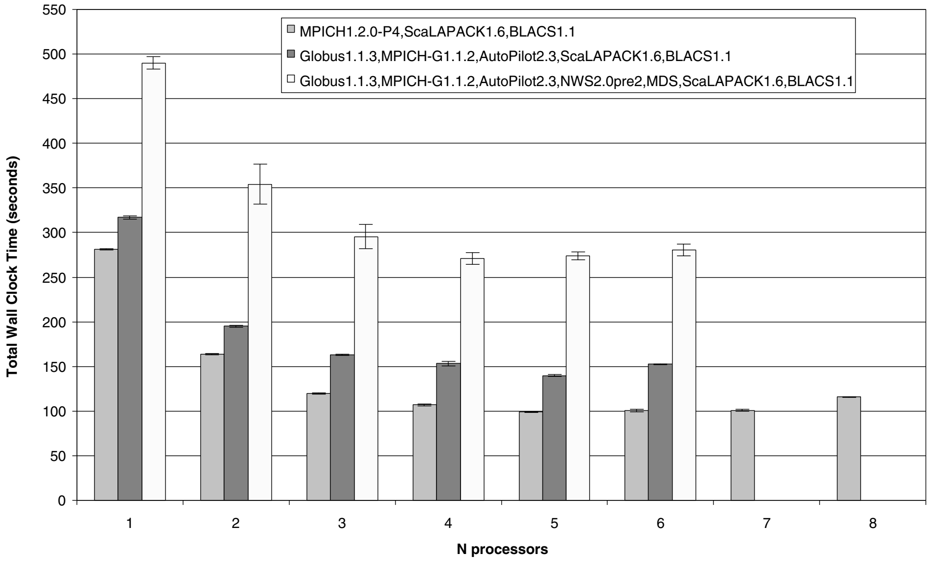

Fig. 2 *Ax* = *b*, *N* = 5000, multiprocessor runs

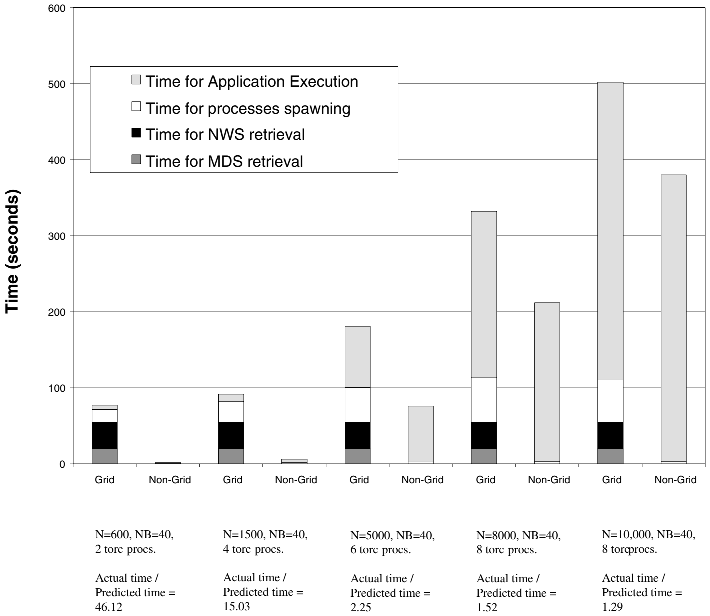

Fig. 3 Overhead in grid runs (NWS = Network Weather Service, MDS = Metacomputing Directory Service

Table 1
Grid of Computers Used

	TORC	CYPHER	OPUS
Type	Dual Pentium III	Dual Pentium III	Pentium II
OS	Red Hat Linux 2.2.15 SMP	Debian Linux 2.2.17 SMP	Red Hat Linux 2.2.16 SMP
Memory	512 MB	512 MB	128 or 256 MB
CPU speed	550 MHz	500 MHz	265-448 MHz
Network	Fast Ethernet (100 Mbit/s) (3Com 3C905B) and switch (BayStack 350T) with 16 ports	Gigabit Ethernet (SK-9843) and switch (Foundry FastIron II) with 24 ports	IP over Myrinet (LANai 4.3) + Fast Ethernet (3Com 3C905B) and switch (M2M-SW16 & Cisco Catalyst 2924 XL) with 16 ports each

chines chosen for matrix size of 8000. This is because certain UIUC machines have small memory size, and they were found to be not optimal by the GrADS system for a matrix size of 10,000. Also, because the experiment was conducted on nondedicated systems, the GrADS scheduling system did not choose some machines from the collection when the system and network loads corresponding to those machines significantly increased.

For matrix sizes larger than 10,000, machines from all three of the clusters were chosen. We find that the transition from 10,000 to 11,000 is not as steep as the transition from 7000 to 8000. This is because the transition from 7000 to 8000 involved Internet connections between UIUC and UT machines and the transition from 10,000 to 11,000 involved UT campus interconnections between the CYPHER and the TORC machines.

As can be seen from these experiments, the GrADS infrastructure is making intelligent decisions based on the application and the dynamics of the system parameters.

We also ran an experiment using all the machines in the system including 6 TORC machines, 12 CYPHER machines, and 11 UIUC machines. Due to memory limitations (and CPU load on the machines), the maximum problem size that was solvable by the collection of machines was a matrix of size 30,000. In this case, GrADS chose 17 processors to solve the problem. These 17 machines consisted of 8 TORC machines and 9 CYPHER machines. The total time taken for the problem was 81.7 minutes. It took 55 seconds to retrieve information from the MDS and NWS; the remaining time was spent launching and executing the application. Thus, for the problem size of 30,000, GrADS was able to achieve 213.4 Mflop/s. The theoretical peak performance achievable is 500 Mflop/s on CYPHER

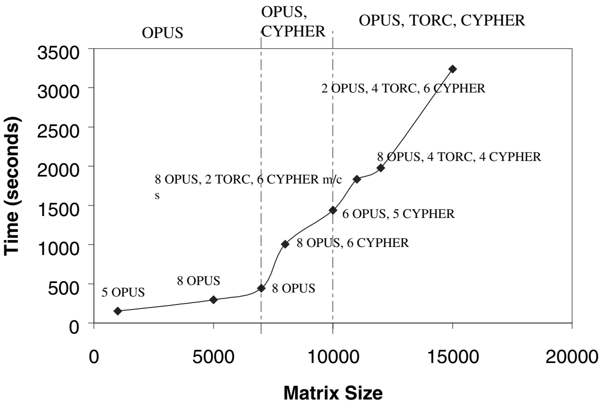

Fig. 4 Performance on the Grid

and 550 Mflop/s on TORC. Thus, GrADS was able to achieve 42.6% of the peak performance whereas the raw ScaLAPACK can achieve about 50% of the peak performance. Thus, we find that the performance of GrADS over the Grid is not far from the performance of the native numerical application over a local cluster. The GrADS system as configured was not able to solve problem sizes larger than 30,000 due to the memory limitations (and CPU load) of the available machines.

The timings for grid ScaLAPACK need to more precisely reflect the total overhead cost for performing ScaLAPACK on the Grid. Grid ScaLAPACK timings were performed with the NWS clique leader not included as one of the computational nodes, so the complete overhead associated with NWS is not totally reflected in the timings. Also, the TORC nodes are dual processors; the timings were performed on one processor of each node, but the kernel was configured as a dual processor. Because the communication is done over IP, the amount of overhead communication cost that is not being captured by the timings is unknown because this cost is being offset on the second processor per node. For future work, to more accurately reflect the total overhead associated with performing ScaLAPACK over the Grid, timings will be performed with the TORC nodes configured as uniprocessors and the clique leader included as one of the compute nodes.

The timings reported for grid ScaLAPACK and raw ScaLAPACK include the time to spawn the MPI processes, generate the matrices, solve the system, and perform an error check to validate the computed solution. The cost of the error check is negligible. In terms of performance, on the TORC cluster, for example, each machine's theoretical peak performance is 550 Mflop/s (each processor is 550 MHz and one flop per cycle), and ATLAS's (Whaley, Petitet, and Dongarra, 2001) DGEMM achieves 400 Mflop/s, 73% of the theoretical peak performance. As a general rule of thumb, when optimizing for best efficiency per processor, ScaLAPACK achieves 75% of DGEMM performance, approximately 300 Mflop/s per processor. This measure depends greatly on the network and the mapping of the processors (one-dimensional vs. two-dimensional). Thus, in its best case, raw ScaLAPACK performs at 55% of the theoretical peak performance of the machine. As can be seen from the grid ScaLAPACK timings and the $N = 30{,}000$ case, the code achieves approximately 210 Mflop/s per processor, which is 40% of the theoretical peak performance. This performance is quite good considering the fact that we have a heterogeneous group of machines connected across the Internet, most of which are slower than 550 MHz.

Conclusion

The experiments reported in this paper were more challenging than originally anticipated. We had to coordinate a number of machines across different administrative domains, and there were varying degrees of maturity in the software and the sheer amount of software involved in getting the experiments in place and maintaining a workable configuration over a long period of time. It is hoped that this situation will improve as the software matures, more sites engage in grid-based computing, and the software infrastructure is more widely used.

Part of the point of conducting these experiments was to show that using geographically distributed resources under a grid framework through the control of the library routine can lead to an improved time to solution for a user. As such, the results, for this modest number of experiments, show that performing a grid-based computation can be a reasonable undertaking. In the case of solving dense matrix problems, we have the situation in which there are $O(n_2)$ data to move and $O(n_3)$ operations to perform. So the fact that we are dealing with geographically distributed systems is not a major factor in performance when the data have to be moved across slow networks. If the problem characteristics are different the situation might not be the same in terms of grid feasibility.

Future work will involve the development of a system that implements a migration system if the time to solution violates the performance contract and a mechanism to deal with fault tolerance.

APPENDIX: GrADS NUMERICAL LIBRARY INTERFACE

The following example is for a user running an application on a sequential machine that is connected to the network. The user will make a grid-enabled library call, and the computation will be done on a set of processors provided by the system. In this example, the user wants to solve a system of linear equations using Gaussian elimination with partial pivoting. The framework provided here can be expanded to include other mathematical software.

We assume the user has already communicated with the system (Globus) and has been authenticated (we assume using grid-proxy-init).

The user must include “grads.h” in his program and invokes Grads_Lib_Linear_Solve() as follows:

```
ierr = Grads_Lib_Linear_Solve( USER_ARGS );
```

The USER_ARGS data structure contains all the parameters passed by the user including the matrix size, the block size, the list of machines on which the user wants to run his application, the machine that is running the Autopilot manager and so on.

Below is the pseudocode for Grads_Lib_Linear_Solve, as well as the APIs for the GrADS components detailed in this paper.

```
int Grads_Lib_Linear_Solve( demo_args_T *USER_ARGS )
{
        Grads_Resource_Coarse_Grid_Handle_T                 coarse_grid;
        Grads_Lib_Fine_Grid_Handle_T                        fine_grid;
        Grads_Lib_Problem_Handle_T                          problem;
        int                                                 n, match;
                                        /* Create a problem of type “Linear_Solve” */
        Grads_Lib_Problem_Create( Linear_Solve, &problem );
                                        /* Set problem attributes */
        Grads_Lib_Problem_Set_Attr( Problem_Matrix_Size, &Matrix_Size, problem );
        Grads_Lib_Problem_Set_Attr( Problem_Block_Size, &Block_Size, problem );
        do
        {
                                            /* For a given problem, retrieve a grid */
                ierr = Grads_Resource_Selector( problem, &coarse_grid );
                                            /* Extract sub-grid to work with */
                Grads_Lib_Performance_Modeler( problem, coarse_grid, USER_ARGS,
                                             &fine_grid );
                /* we are done with coarse_grid; free memory coarse_grid points to. */
                Grads_Resource_Coarse_Grid_Remove( problem, coarse_grid );
                                             /* Try to commit the fine grid for the problem */
                match = Grads_Contract_Developer( problem, fine_grid );
                                            /* If this list of machines is not good - release it */
                if( match != Grads_SUCCESS ) Grads_Lib_Grid_Free( problem, fine_grid );
        } while( match != Grads_SUCCESS );

        ierr = Grads_Application_Launcher( problem, fine_grid, USER_ARGS );

         /* Release the Resources */
        Grads_Lib_Fine_Grid_Remove( fine_grid );
        Grads_Lib_Problem_Remove ( problem );
        return( ierr );
}
```

The APIs for the GrADS components detailed in this paper are listed below. For complete information, refer to the “grads.h” include file.

RESOURCE SELECTOR

Grads_Resource_Selector(problem, coarse_grid)

IN	problem	the problem handle
OUT	coarse_grid	handle to a structure with the following information
	int no_coarse_grid	number of processors potentially available
	int array name(no_coarse_grid)	
		names of the available processors (perhaps ip addresses)
	int array memory(no_coarse_grid)	
		amount of memory available on each of the processors
	int array communication(no_coarse_grid^2)	
		a 2-d array containing bandwidth and latency information on the link between available processors
	int array speed(no_coarse_grid)	
		peak speed for each processor according to some metric.
	int array load(no_coarse_grid)	
		load on each processor at the time the call was made.
OUT	ierr	error flag from the Resource_Selector

```
int Grads_Resource_Selector( Grads_Lib_Problem_Handle_T problem,
                             Grads_Resource_Coarse_Grid_Handle_T *coarse_grid )
```

PERFORMANCE MODELER

Grads_Lib_Performance_Modeler(problem, coarse_grid, USER_ARGS, fine_grid)

IN	problem	a unique problem identifier for this library call
IN	coarse_grid	struct (see above call)
IN	USER_ARGS	struct
OUT	fine_grid	handle to a structure specifying the machine configuration to use
OUT	ierr	error code returned by the function

```
int Grads_Lib_Performance_Modeler( Grads_Lib_Problem_Handle_T problem,
                                   Grads_Resource_Coarse_Grid_Handle_T coarse_grid,
                                   demo_args_T *USER_ARGS,
                                   Grads_Lib_Fine_Grid_Handle_T *fine_grid )
```

CONTRACT DEVELOPER

match = Grads_Contract_Developer(problem, fine_grid)

IN	problem	a unique problem identifier for this library call
OUT	fine_grid	handle to machine configuration
OUT	match	will be 0 if the processors are available for this run.

```
int Grads_Contract_Developer( Grads_Lib_Problem_Handle_T problem,
                              Grads_Lib_Fine_Grid_Handle_T fine grid )
```

APPLICATION LAUNCHER

ierr = Grads_Application_Launcher(problem, fine_grid, USER_ARGS)

IN	problem	a unique problem identifier for this library call
IN	fine_grid	handle to machine configuration
IN	USER_ARGS	user arguments
OUT	ierr	error flag from the Application_Launcher

```
int Grads_Application_Launcher( Grads_Lib_Problem_Handle_T problem,
                                Grads_Lib_Fine_Grid_Handle_T fine_grid,
                                demo_args_T *USER_ARGS )
```

ACKNOWLEDGMENTS

The work is supported in part by the National Science Foundation contract GRANT #E81-9975020, SC R36505-29200099, R01-1030-09. We would like to thank the following individuals who contributed code and support to the ScaLAPACK experiment: Ruth Aydt, Dan Reed, Frederik Vraalsen, and Celso Mendes of UIUC; Fran Berman, Holly Dail, Otto Sievert, and Graziano Obertelli of UCSD; Sridhar Gullapalli at USC/ISI; Rich Wolski, Martin Swany, and Clay England at UT.

BIOGRAPHIES

Antoine Petitet is a benchmark engineer at Sun Microsystems in Paris, France. Until February 2001, he was a research scientist in the Computer Science Department at the University of Tennessee, Knoxville. His research interests primarily focused on parallel computing, numerical linear algebra, and the design of scientific parallel numerical software libraries for distributed-memory concurrent computers. He was involved in the design and implementation of the software packages ScaLAPACK and ATLAS.

Susan Blackford is a member of the technical staff at Myricom Inc. Before joining the software development team at Myricom, she was a senior research associate at the University of Tennessee's Innovative Computing Laboratory (1990-2001). She received a B.S. in mathematics from the University of Tennessee in 1988 and an M.S. in computer science from the University of Tennessee in 1990. Her interests lie in high performance computing, and she specializes in the development, testing, and documentation of high-quality mathematical software. She was involved in the design, implementation, and support of the software packages LAPACK and ScaLAPACK, and was recently involved in the use of numerical libraries in NetSolve.

Jack Dongarra holds an appointment as university distinguished professor of computer science in the Computer Science Department at the University of Tennessee and is an adjunct R&D participant in the Computer Science and Mathematics Division at Oak Ridge National Laboratory and an adjunct professor in computer science at Rice University. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing, and documentation of high-quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports, and technical memoranda, and he is coauthor of several books. He is a fellow of the American Association for the Advancement of Science, the Association for Computing Machinery, and the Institute of Electrical and Electronics Engineers, and a member of the National Academy of Engineering.

Brett Ellis is a senior computer systems specialist in the Innovative Computing Laboratory (ICL), a research group within the Computer Science Department at the University of Tennessee. He has received B.S. degrees in mathematics 1996 and in computer science in 2000. At ICL, he has been involved with work on the research done in support of the computing infrastructure and installation/troubleshooting of the software created within.

Graham Fagg received a B.Sc. in computer science and cybernetics from the University of Reading (U.K.) 1991 and a Ph.D. in computer science in 1998. From 1991 to 1993, he worked on CASE tools for interconnecting array processors and Inmos T-800 transputer systems as part of the ESPRIT Alpha project. From 1994 to the end of 1995, he was a research assistant in the Cluster Computing Laboratory at the University of Reading working on code generation tools for group communications. From 1991 to 1996, he has worked as a senior research associate at the University of Tennessee. Since 1999, he has been a research assistant professor. His current research interests include distributed scheduling, resource management, performance prediction, benchmarking, cluster management tools, and high-speed networking. He is currently involved in the development of a number of different metacomputing and grid middleware systems including SNIPE, MPI_Connect(), HARNESS, and a fault-tolerant MPI implementation (FT-MPI).

Kenneth Roche works in the Linear Algebra and Distributed Computing groups at the Innovative Computing Laboratory (ICL) in Knoxville, Tennessee. His professional activities in the past 5 years include studies in scientific computing and theoretical physics. In physics, quantum mechanical many body theory, nuclear astrophysics, and some mesoscopic phenomena have been his interests. In computer science, his interests are in numerical mathematics (libraries) and distributed computing environments. He is a member of the American Physical Society and the Society for Industrial and Applied Mathematics.

Sathish Vadhiyar is a Ph.D. student in the Department of Computer Science at the University of Tennessee. He received an M.S. in computer science from Clemson University. In his master's work, he concentrated on parallel compilers involving threads. He was also involved in a research group on graphics. He is currently a graduate research assistant in the Innovative Computing Laboratory in Knoxville, Tennessee. His main research interests are in the fields of parallel, distributed, and grid computing and architecture-specific tuning. He works on three different projects: a metacomputing system called HARNESS, a client-server based grid computing system called NetSolve, and Grid Application Development Software. His work includes architecture-specific tuning of MPI collective communications. He is currently involved in finding solutions for efficient scheduling in grid environments. He is a student member of the Institute of Electrical and Electronics Engineers.

REFERENCES

Balay, S., et al. 1996. *PETSc 2.0 Users' Manual*. Argonne, IL: Argonne National Laboratory.

Berman, F., et al. 2000. *The GrADS Project: Software Support for High-Level Grid Application Development*. Houston, TX: Rice University.

Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. *ScaLAPACK Users' Guide*. Philadelphia: SIAM.

Foster, I., and Karonis, N. 1998. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SuperComputing 98 (SC98)*, Orlando, FL.

Foster, I., and Kesselman, C. 1997. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications* 11:115-128.

Gropp, W., Lusk, E., Doss, N. and Skjellum, A. 1996. A high performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22:789-828.

Gropp, W., and Lusk, W. 1996. *Users' Guide for MPICH, a Portable Implementation of MPI*. Argonne, IL: Mathematics and Computer Science Division, Argonne National Laboratory.

Ribler, R. L., et al. 1998. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, Chicago.

Snir, M., et al. 1998. *MPI: The Complete Reference, Volume 1, the MPI Core*. 2nd ed. Boston: MIT Press.

Vetter, J. S., and Reed, D. A. 2000. Real-time performance monitoring, adaptive control, and interactive steering of computational grids. *International Journal of High Performance Computing Applications* 14:357-366.

Vraalsen, F., et al. 2001. Performance contracts: Predicting and monitoring grid application behavior. In *2nd International Workshop on Grid Computing*, Denver, CO.

Whaley, R., Petitet, A., and Dongarra, J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Computing* 27 (1-2): 3-25.

Wolski, R., Spring, N., and Hayes, J. 1999. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 15:757-768.