A Modular Scheduling Approach for Grid Application Development Environments

Holly Dail Henri Casanova Fran Berman

San Diego Supercomputer Center
University of California at San Diego
{hdail, casanova, berman}@sdsc.edu

Version: June 4, 2002

In this paper we propose an adaptive scheduling approach designed to improve the performance of parallel applications in Computational Grid environments. A primary contribution of our work is that our design is modular and provides a separation of the scheduler itself from the application-specific components needed for the scheduling process. As part of the scheduler, we have also developed a search procedure which effectively and efficiently identifies desirable schedules.

As test cases for our approach, we selected two applications from the class of iterative, mesh-based applications. For each of the test applications, we developed data mappers and performance models. We used a prototype of our approach in conjunction with these application-specific components to perform validation experiments in production Grid environments. Our results show that our scheduler provides significantly better application performance than conventional scheduling strategies. We also show that our scheduler gracefully handles degraded levels of availability of application and Grid resource information. Finally, we demonstrate that the overheads introduced by our methodology are reasonable. This work evolved in the context of the Grid Application Development Software Project (GrADS). Our scheduling approach is designed to be easily integrated with other GrADS program development tools.

Key Words: scheduling, Grid computing, programming environments, parallel computing

1. INTRODUCTION

With vast improvements in wide-area network performance and the pervasiveness of commodity resources, distributed parallel computing can benefit from an increasingly rich computational platform. However, many focused development efforts have shown that taking advantage of these Computational Grid environments [19] for scientific computing requires extensive labor and support by distributed computing experts. Grid infrastructure projects [18, 21, 27] have provided many of the services needed for Grid computing; these middleware services help reduce programmer effort and can improve application performance on these platforms. However, such middleware generally does not account for the specific needs of applications. For example, each application has unique resource requirements which must be considered in scheduling the

This material is based upon work supported by the National Science Foundation under Grant #9975020.

application on Grid resources. More generally, if a programmer wants to take advantage of Computational Grids, they are responsible for all transactions that require knowledge of the application at hand; examples include discovering resources, selecting an application-appropriate subset of those resources, staging binaries on selected machines, and, for long-running applications, monitoring application progress. While many scientists could benefit from the extensive resources offered by Computational Grids, application development remains a daunting proposition.

One solution is to develop software that frees the user of these responsibilities. The Grid Application Development Software (GrADS) Project [7] seeks to provide such a solution by developing a comprehensive programming environment that explicitly incorporates application characteristics and requirements in application development decisions. The end goal of this project is to provide an integrated Grid application development solution that incorporates activities such as compilation, scheduling, staging of binaries and data, application launch, and monitoring of application progress during execution.

In this paper, we are interested specifically in the scheduling process required for such a system. The GrADS design [7, 25] assigns the scheduler the responsibility for discovery of available resources, the selection of an application-appropriate subset of those resources, and the mapping of data or tasks onto selected resources. Application schedulers have long been considered an important tool for usage of Computational Grids [6]. In fact, many projects have developed successful scheduling strategies for the Grid [1, 2, 3, 8, 9, 32, 33, 37, 40, 41]. Most of these schedulers incorporate the specific needs of applications in scheduling decisions, and would therefore seem to fulfill the design requirements of a scheduler in GrADS. However, if the GrADS solution is to be easy to apply in a variety of application-development scenarios, the scheduler must be easily applied to a variety of applications. Unfortunately, most of the schedulers mentioned previously have been developed for one application or for a specific class of applications, and the designs are generally not easily re-targeted for other applications or classes. The difficulty in re-targeting such designs arises from the fact that application-specific details or components are generally embedded in the scheduling software itself. Given such a design, it can be difficult to determine which components need to be replaced to incorporate the needs of the new application.

In this paper we propose a modular scheduling approach that explicitly separates general-purpose scheduling components from the application-specific components needed for the scheduling process. Specifically, application requirements and characteristics are encapsulated in a *performance model* (an analytical metric for the performance expected of the application on a given set of resources) and a *data mapper* (directives for mapping logical application data or tasks to physical resources). The core of the scheduler is a general-purpose schedule search procedure which effectively and efficiently identifies desirable schedules. Our scheduler provides a framework in which the schedule search procedure can be used in conjunction with an application-specific performance model and mapper to provide scheduling decisions that are appropriate to the needs of the application.

As test cases for our approach, we selected two applications from the class of iterative, mesh-based applications. For each of the test applications, we developed data mappers and performance models. We used a prototype of our approach in conjunction with these application-specific components to perform validation experiments in production Grid environments. Our results demonstrate that our scheduler provides significantly better application performance than conventional scheduling strategies. We also show that our scheduler gracefully handles degraded levels of availability of application and Grid information. Finally, we demonstrate that the overheads introduced by our methodology are reasonable.

We do not expect that general-purpose software will achieve the performance of a highly-tuned application-specific scheduler; instead, the goal is to provide consistently improved performance relative to conventional scheduling strategies. The primary contributions of our work are as follows.

i. Our approach is modular and can be easily instantiated for other applications.

ii. Our scheduler gracefully handles degraded levels of availability of application and Grid information

iii. Our scheduler simplifies usage of the Grid by automating the scheduling process. Furthermore, as compared to conventional approaches such as user-directed scheduling, our approach provides improved application performance and reduced failure rates in production Grid environments.

iv. Although our scheduler can function in a stand-alone fashion (and is validated as such in the experiments presented in this paper), it is in fact an integrated component of the GrADS system. The scheduler described in this paper was the first prototype scheduling component developed in GrADS for usage with multiple applications.

This paper is organized as follows. In Section 2 we describe the scheduler design itself. Section 3 details two test applications and presents performance model and mapper designs for each. In Section 4 we present the results we obtained when applying our methodology in Computational Grid environments. In Section 5 we describe related and future work, and then we conclude the paper.

2. SCHEDULING

This section describes our scheduler design. To provide context for the rest of this section, we describe the scheduling scenario we address.

Our scheduling scenario begins with a user who has an application and wishes to schedule that application on Computational Grid resources. The application is parallel and may involve significant inter-process communication. The target Computational Grid consists of heterogeneous workstations connected by local-area networks (LANs) and/or wide-area networks (WANs). The user may directly contact the scheduler to submit the scheduling request, or an intermediary, such as another component of the GrADS system, will contact the scheduler to submit the user's request. In either case, we assume that the goal of the scheduling process is to find a schedule which minimizes total turnaround time (scheduling time + application run-time). The selected schedule will be used without modification throughout application execution (we do not consider rescheduling).

2.1. Architecture

Figure 1 presents the primary components of our scheduler and the interactions among those components. We provide this figure as a reference for the rest of the paper; we do not expect that all components or interactions will be completely clear at this point.

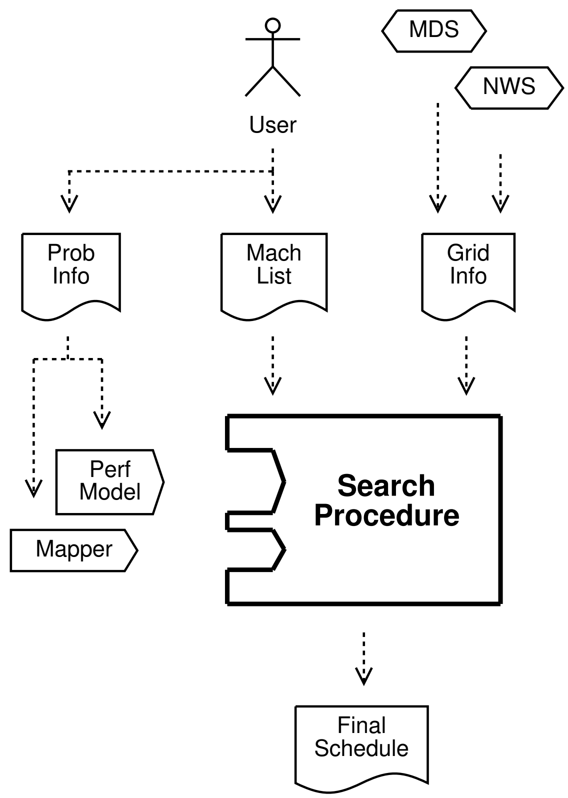

FIG. 1 Scheduler design.

The **Search Procedure** is the core of the scheduler. This procedure is responsible for searching for schedules which are appropriate to the target application. A **schedule** consists of an ordered list of machines and a mapping of data or tasks to those machines. The Search Procedure is responsible for finding the "best" schedule (**Final Schedule** in Figure 1).

Before submitting an application to the scheduler, the user must obtain or develop the following application-specific components.

- The **performance model** is a procedure call which provides a prediction of application performance on a given set of resources. There are a variety of performance metrics that might be used for scheduling;

in this section we assume that the performance model will predict application execution time.

- The **mapper** is a procedure call which maps logical application data or tasks to physical resources. For each machine in the schedule, the mapper must define which piece of the application data will be assigned to that processor.

These components are application-specific, but run-generic. In each application run, the user defines **problem parameters** such as problem size. The performance model and mapper are then instantiated with this information to make them appropriate to the current problem run. We describe several application-specific performance models and mappers in Section 3; in this section we simply assume that such components are available.

The user must also submit a **machine list** containing machine names that the user has access to; we forgo further discussion of this list until Section 2.4. For each machine in the machine list, we collect resource information such as CPU speed, available physical memory, and bandwidth between hosts. This information is retrieved from resource information providers such as the Network Weather System (NWS) and the Metacomputing Directory Service (MDS); we discuss these services in Section 2.4.

2.2. Search Procedure

The schedule search procedure is the core of the scheduling methodology. The goal of the searching process is to find groups of machines that could prove performance-efficient platforms for the application; we call these groups **candidate machine groups (CMGs)**. To find corresponding **candidate schedules**, the search procedure identifies CMGs and generates a data map for each one. A performance model is then used to select the best candidate schedule.

The most straightforward approach for the search process is an exhaustive search over all possible groups of machines (ignoring permutations since ordering is defined by the mapper). For larger resource set sizes or even moderately complex performance models and mappers, such a search is not feasible. A practical search procedure must therefore use extensive but careful pruning of the search space.

Pseudo-code for our schedule search procedure is given in Figure 2. In each *for* loop the list of target CMGs is refined based on a different resource set characteristic: connectivity in the outer-most loop, computational and memory capacity of individual machines in the second loop, and selection of an appropriate resource set size in the inner-most loop. The goal is to generate only a moderate number of CMGs while ensuring that we do not exclude performance-efficient CMGs.

The first step of our search procedure is to call the **FindSites** method; this method takes a list of machines and organizes them into disjoint subsets, or sites, such that the network delays within each subset are lower than the network delays between subsets. As a first implementation, we group machines into the same site if they share the same domain name; we plan to consider more sophisticated approaches [34, 28, 39] in future work. The **ComputeSiteCollections** method computes the power set of the set of sites (we exclude the

```
Algorithm : ScheduleSearch(machList, gridInfo, PerfModel, Mapper)

 sites ← FindSites(machList)
 siteCollections ← ComputeSiteCollections(sites)
 for each collection in siteCollections
   for each sortMetric in (computation, memory, dual)
     for targetSize ← 1 to size(collection)
       CMG ← FindBest(collection, sortMetric, targetSize)
       dataMap ← Mapper(CMG, gridInfo)
       if map == VALID
         currSched = GenerateSchedule(CMG, dataMap)
         if ScheduleCompare(currSched, bestSched, PerfModel) == FirstIsBetter
           bestSched ← currSched
 return (bestSched)
```

FIG. 2: Schedule search procedure.

null set). As an example, for the set of sites {A, B, C}, there are seven site collections: {A, B, C, A ∪ B, A ∪ C, B ∪ C, A ∪ B ∪ C}. Once all of the machine collections have been identified, the **outer-most loop** of the search procedure examines each one in turn.

In the **middle loop** of the search procedure, we seek to identify machines that exhibit high local memory and computational capacities. Generally we will not know a priori which machine characteristics will have the greatest impact on application performance; we therefore define three metrics that are used to **sort** the machine list: the *computation metric* emphasizes the computational capacity of machines, the *memory metric* emphasizes the local memory capacity of machines, and the *dual metric* places equal weight on each factor.

The **inner-most loop** exhaustively searches for an appropriately-sized resource group. Resource set size selection is complex because it depends on problem parameters, application characteristics, and detailed resource characteristics. Rather than miss potentially good resource set sizes based on poor predictions, we include all resource set sizes in the search. As will be described momentarily, an application performance model can then be used to select amongst different schedules. Note that an exhaustive search at this level of the procedure is only feasible due to the extensive pruning performed in the first two loops. The **FindBest** method sorts the input machine list *collection* by the machine type *metric* and returns the best *targetSize* number of machines.

Next, the **mapper** is called to obtain a data map for the current CMG; since the mapping process is typically dependent on characteristics of the target resource group, gridInfo is included as a parameter to the mapper. If the mapper is unable to find a feasible mapping due to constraints such as local machine memory capacities, the current CMG is skipped and the search process continues. If the returned map is valid, then **GenerateSchedule** combines the map and the CMG to form a schedule. Finally, the **ScheduleCompare** method is called to compare the current schedule with the best schedule discovered by the search so far. The exact comparison mechanism depends on what type of performance model is available. By default, we assume that the performance model provides execution time predictions; the default ScheduleCompare therefore returns the schedule with the lowest predicted execution time.

2.3. Search complexity

The most straightforward schedule search method is an exhaustive search; such a search is guaranteed to identify the optimal CMG. However, for a resource pool of size p, the search must examine a number of CMGs equal to:

$$numCMGs = \sum_{k=1}^{p} \frac{p!}{k!(p-k)!} = 2^p. \tag{1}$$

For example, a schedule search for a 30 machine resource group would require evaluation of $2^{30} \approx 10^9$ CMGs. For a reasonably sized resource set and/or when the performance model evaluation or mapping process is time intensive, an exhaustive search is simply too costly.

In the vast majority of cases, our search procedure provides an impressive reduction in search space. To demonstrate this we develop a **loose upper bound on the number of CMGs considered by the search heuristic**. Assuming we have s sites in the resource set under consideration, we obtain 2^s site collections (in fact, we exclude the null-set leaving $2^s - 1$ such collections). We consider three resource orderings for each collection (*computation*, *memory*, and *dual*). Given these $3 * 2^s$ ordered collections, we exhaustively search all possible subset sizes for each. Since the number of resources in each site, and therefore in each topology-based collection, is dependent on the characteristics of each Grid environment, we can not predict a priori the number of resources in each of the $3 * 2^s$ ordered collections. As an upper bound, we assume each collection is of size p, the size of the entire resource pool. Then, in the third loop of our search procedure, p distinct subsets will be generated for each ordered collection. The upper bound on the total number of CMGs identified by the search procedure is therefore $3p2^s$. Recall our earlier example of a 30 machine set; the exhaustive search required evaluation of 10^9 CMGs in this case. Supposing this resource set included 3 sites, our search procedure would require evaluation of at most 720 CMGs. In fact, if we assume 10 machines in each of the 3 sites, our search procedure requires evaluation of only 360 CMGs.

The improvement gained from our methodology will be greatest when the number of sites under consideration is significantly less than the number of machines, which is the case in the vast majority of modern Grids.

2.4. Use of Grid information

Computational Grids are highly dynamic environments where compute and network resource availability varies and Grid information sources can be periodically unstable. We strive to provide best-effort service by supporting multiple information sources, when possible, for each type of information required by the scheduler.

We currently support information collection from the two most widely used Grid resource information systems, the Metacomputing Directory Service (MDS) [10] and the Network Weather Service (NWS) [43]. The **MDS** is a Grid information management system that is used to collect and publish system configuration, capability, and status information. Examples of the information that can typically be retrieved from an MDS server include operating system, processor type and speed, number of CPUs available, and software availability and installation locations. The **NWS** is a distributed monitoring system designed to track and forecast resource conditions. Examples of the information that can typically be retrieved from an NWS server include the fraction of CPU available to a newly started process, the amount of memory that is currently unused, and the bandwidth with which data can be sent to a remote host.

Our scheduling methodology can utilize several types of resource information: a list of machines available for the run, local computational and memory capacities for each machine, and network bandwidth and latency information. The list of machines is currently obtained directly from the user; once secure MDS publishing mechanisms are available, user account information can be published directly in the MDS and retrieved automatically by the scheduler. Local machine computational and memory capacity data are used to sort machines in our search procedure and will be needed as input to many performance model and mapper implementations. Network bandwidth and latency data will similarly be required as input to many performance model and mapper implementations.

An important characteristic of our approach is that the scheduler gracefully copes with degraded Grid information availability. Whenever possible we support more than one source for each type of resource information required by the scheduler. Furthermore, when a particular type of information is not available for specific machines in the machine list, but is required by the scheduler, the scheduler gracefully excludes those machines from the search process. In our experience, most application schedulers do not gracefully handle such situations, leading to many scenarios in which the scheduler fails.

3. APPLICATION CASE STUDIES

In this section we describe specific applications that we used to demonstrate our scheduling methodology in validation experiments. As required by our scheduling methodology, we develop performance models and mapping strategies for each application.

For each application, we develop a performance model that predicts both execution time and memory usage. We also present a strategy for comparing candidate schedules in the absence of an execution time model; this strategy demonstrates that our scheduling framework can be adjusted to accommodate alternative performance metrics or types of performance model. We also implement two mappers that can be applied to our test applications: a time balance mapper, which can be used when an execution time model is available, and an equal allocation mapper, which can be applied when application information is limited to a memory usage model.

3.1. Case study applications

We have chosen applications from the class of regular, iterative, mesh-based applications as they are important in many domains of science and engineering [14, 15, 20, 5]. Specifically, we focus on the **Game of Life** and **Jacobi**. We have selected these applications as our initial test cases because they are well-known, straightforward to describe, and share many performance characteristics with other applications.

Conway's **Game of Life** is a well-known binary cellular automaton whereby a fixed set of rules are used to determine a next generation of cells based on the state of the current generation [14]. A two-dimensional mesh of pixels is used to represent the environment, and each pixel of the mesh represents a cell. In each iteration, the state of every cell is updated with a 9-point stencil. We use a 1-D strip data partitioning strategy because this strategy typically exhibits lower communication costs than other partitioning schemes, an important consideration for Grid computing. Each processor manages a data strip and defines a 1-pixel wide set of ghost-cells along data grid edges. Each iteration consists of a computational phase in which processors update their portion of the data array, and a communication phase in which processors swap ghost cell data with their neighbors.

The **Jacobi method** is a simple algorithm that is often used in the context of Laplace's equation [5, 31]. Here we describe the general linear system solver version, which involves more communication (broadcasts). The method attempts to solve a square linear system $Ax = b$ with the following iteration formula:

$$x_j^{k+1} = \frac{1}{a_{jj}}(b_j - \sum_{i \neq j} a_{ji} x_i^k)$$

where x_j^k is the value of the j^{th} unknown at the k^{th} iteration. This method is guaranteed to converge only if matrix A is diagonally dominant.

A popular parallel data decomposition for the Jacobi method is to assign a portion of the unknown vector x to each processor where processors need only store rectangular sub-matrices of A. Each processor computes new results for its portion of x, and then *broadcasts* its updated portion of x to every other processor. The final phase in each iteration is a termination detection phase. The method is *stationary*, meaning that the matrix A is fixed throughout the application.

We implemented each test application as a SPMD-style computation using C and the Message Passing Interface (MPI) [30]. To allow load-balancing we implemented support for irregular data partitions in both applications. We used the Globus-enabled version of MPICH [22, 23], MPICH-G [16, 17], in order to run over a Computational Grid testbed.

3.2. Application performance modeling

Our scheduling framework is dependent on the availability of a performance model. The ScheduleCompare method described in the previous section assumes a performance model that predicts application execution time. Ultimately, such performance models may be automatically generated by the compiler in the GrADS framework. For the moment, we develop such a performance model for Jacobi and the Game of Life. We also develop a memory usage model, which will be required for the mapper discussion in Section 3.4.

Although both applications support rectangular data grids, we assume that the full data mesh is a 2-dimensional square. We use the following definitions in the rest of this section. We refer to the size of either dimension of the data mesh as N; note that the number of data elements, and therefore the amount of work, grows as N^2. We refer to the processors chosen for execution as $P_0, ..., P_{p-1}$ and the size of the data partitions allocated to each processor as $n_0, ..., n_{p-1}$.

3.2.1. Memory usage model

Given the magnitude of performance degradation due to paging of memory to disk, we must ensure that the application fits within the available memory of the processors selected for execution. We compute the amount of memory (in bytes) required for a data strip of size N x n_i as:

$$memReq_i = memUnit \times n_i \times N, \tag{2}$$

where $memUnit$ is the number of bytes of storage that will be allocated per element of the data domain. The Game of Life allocates 2 matrices of integers and Jacobi allocates 1 matrix of doubles. For the architectures we targeted in this paper, 4 bytes are allocated per integer and 8 bytes are allocated per double. Therefore, we use $memUnit = 8$ for both applications.

Recall that local processor available memory, mem_i, availability can be supplied by total physical memory values from the MDS [10] or free memory values from the NWS [43]. In practice, a close match of $memReq_i$ and mem_i provides an overly tight fit due to additional memory needed by the application and to memory contention with system or "small" user processes. Based on early experimental results and memory usage benchmarks, increasing $memReq_i$ by 20% provides a reasonable tradeoff for the GrADS Computational Grid environment [7].

3.2.2. Execution time model

Given that we target regular, synchronous iteration applications, the application execution time can be assumed proportional to the iteration time on the slowest processor. The iteration time on processor P_i is naturally modeled as the sum of a computation time and a communication time:

$$itTime_i = compTime_i + commTime_i. \tag{3}$$

The **computation phase** for our test applications primarily consists of the data update process in each iteration and may include a termination detection operation (e.g. for Jacobi). We model computation time on processor P_i as:

$$compTime_i = \frac{compUnit * n_i * N}{10^6 * comp_i}, \tag{4}$$

where $compUnit$ is the number of processor cycles performed by the application per element of the data domain. The computational capacity of processor P_i, $comp_i$, can be represented by raw CPU speed (from the MDS) and the currently available CPU (from the NWS), or a combination thereof. To fully instantiate this model, we need to determine an appropriate $compUnit$ value for each case study application. Rather than using methods such as source code or assembly code analysis, we opted for an empirical approach: we ran applications on dedicated resources of known CPU speed for 100 iterations and computed average $compUnit$ values.

The **Game of Life communication phase** consists of the swapping of ghost cells between neighboring machines. We use non-blocking sends and receives, and, in theory, all of the messages in each iteration could be overlapped. In practice, however, processors can not simultaneously participate in four message transfers at once without a reduction in performance for each message and, more importantly, processors do not reach the communication phase of each iteration at the same moment. As an initial approximation, we assume that messages with a particular neighbor can be overlapped, but that communication with different neighbors occurs in distinct phases which are serialized.

The **Jacobi communication phase** involves a series of p broadcasts per iteration; each machine in the computation is the root of one of these broadcasts. In the MPI implementation we used in this work, the broadcast is implemented as a binomial tree [4]. As a first approximation to modeling this communication structure, we calculate the average message time, $msgTime_{avg}$, and calculate the communication time on processor P_i as

$$commTime_i = p * log_2(p) * msgTime_{avg}. \tag{5}$$

Our Game of Life and Jacobi communication models each depend on a model for the **cost of sending a message** between two machines. We initially opted for the simple and popular latency/bandwidth model for a message sent from P_i to P_j:

$$msgTime_{i,j} = latency_{i,j} + msgSize/bandwidth_{i,j}, \tag{6}$$

where latency and bandwidth measurements and forecasts are provided by the NWS. However, we observed that this model significantly over-estimates the message transfer times of our applications. We found that a bandwidth-only model ($latency = 0$) led to much better predictions, possibly because of NWS measurement techniques, MPI implementation, and network topologies. In the rest of the paper we use the more accurate bandwidth-only model.

3.3. Alternative performance models and metrics

In Section 2 we presented our scheduling methodology with the assumption that an application-specific performance model would be available. What if one wanted to use a different performance metric or a different performance model? Due to the modularity of our scheduling approach, the only component that needs to be modified is the ScheduleCompare method implementation (see Figure 2).

As an example, suppose a memory usage model is available, but an execution time model is not. The memory usage information will be used by the mapper (see Section 3.4) to ensure that candidate schedules fulfill application memory requirements. An alternative to a performance model for the purpose of schedule comparisons is a series of heuristics that evaluate how well the candidate schedules satisfy a set of broad resource requirements such as bandwidth or computational capacity. Figure 3 presents a decision tree we've employed to implement such a series of simple heuristics. This decision tree is appropriate for Jacobi and the Game of Life and will be used in validation experiments in Section 4.

3.4. Application data mappers

The function of the mapper is to determine an appropriate mapping of work (i.e. strip widths $n_0, ..., n_{p-1}$) onto processors ($P_0, ..., P_{p-1}$). The mapping process involves two distinct subproblems. The first problem is to determine a **topological arrangement** of machines (e.g. which physical processor should be assigned to logical processor position P_0) such that application communication costs are minimized. The second problem is to find an **allocation of work** to processors (e.g. how many rows of the data mesh should be assigned to process P_0) such that application resource-requirements are met and, when possible, application execution time is minimized.

In the following sections we present two mappers: an equal allocation mapper and a time balance mapper.

3.4.1. Equal allocation mapper

In this mapper, work is simply allocated uniformly to all processors such that each is assigned an equally sized data strip of size N/p and the number of pixels assigned to each processor is $N * N/p$. Before returning this data map, the mapper verifies that each processor has sufficient local memory to support the application's memory requirements. When at least one processor does not have sufficient memory, the mapper returns an error. In the scheduling context presented in Section 2, the current machine group is removed from the list of CMGs and the search process continues.

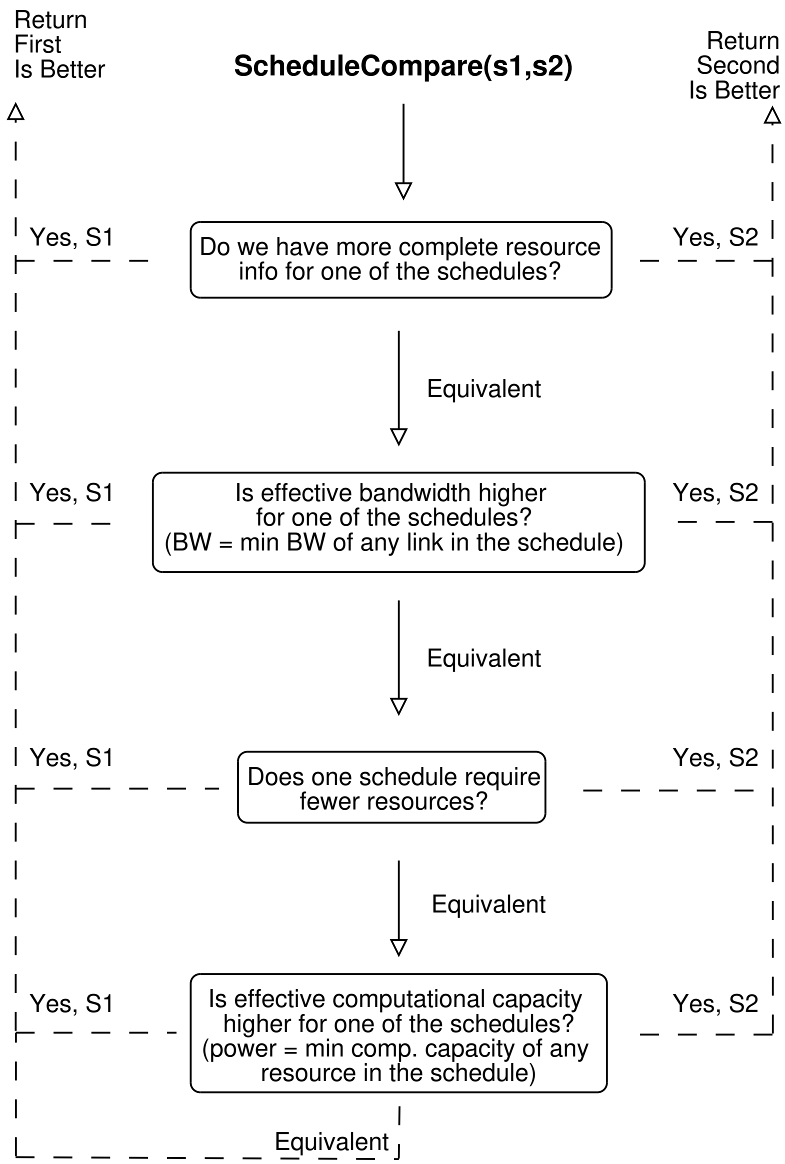

FIG. 3 An alternative schedule comparison design. This design employs a series of comparison heuristics in place of an execution time model.

This mapper requires only a memory usage model and not a full execution time model. In production Grid scheduling systems we expect there will be many applications for which a full performance model is not available. In such a case, our scheduling methodology can still be applied by pairing it with the equal allocation mapper and the alternative performance model presented in the preceding section. We explore such a scenario in Section 4.

3.4.2. Time balance mapper

For regular, synchronous iteration applications, application execution time is limited by the progress of the slowest processor. Total execution time can be minimized by finding a data map for which all processors complete their work in each iteration at the same time, thereby minimizing synchronization times. The goal of the *time balance mapper* is to find such a data map while ensuring that application memory requirements are met. Our approach is to formalize machine resource availabilities, application memory requirements, and execution time considerations as a series of constraints. Work-allocation can then be framed as a constrained optimization problem; the solution is a map of data strip widths to processors. We first describe the general operation of the mapper, and then describe our formalization of the problem.

When called, the time balance mapper first verifies that the aggregate memory of the CMG is sufficient for the aggregate requirements of the current application problem. If not, the mapper does not attempt to find a data map and returns an error. If the CMG has sufficient aggregate memory, the mapper searches for a perfectly time-balanced data map; if found, the map is returned. Sometimes, one or more machines do not have sufficient local memory to satisfy the memory requirements of a perfectly time balanced map. In this case, the mapper relaxes the time balance constraints to seek an alternative map which satisfies memory requirements. The mapper uses a binary search to find the map that provides the best time balance while satisfying application memory requirements. The parameters of the binary search are configurable; default values are provided as follows. The default maximum relax factor is 10, meaning that for an acceptable map, predicted iteration time on the slowest processor is no more than 10 times the predicted iteration time on the fastest processor. The default search tolerance is 0.01, meaning that the search refinement ends when the relax factor changes by less than 0.01 between search steps.

We now briefly describe our specification of this problem as a constrained optimization problem; see [11] for a thorough explanation and [13, 36] for previous work that applied a similar solution for the data mapping problem. The unknowns are the strip widths to be assigned to each processor: $n_0, ..., n_{p-1}$. Since strip widths are constrained to integer values, the problem can be framed as an integer programming problem [42]. Unfortunately, the integer programming problem is NP-complete, rendering the solution computationally expensive to compute and unacceptable for use in our scheduling methodology. We use the more efficient alternative of real-valued linear programming solvers [42] (specifically, we used the *lp_solve* package which is based on the simplex method). Although using a real-valued solver for an integer problem introduces some error, it provides sufficient accuracy for our needs.

The problem formulation begins with the specification of an **objective function**. Since it is impossible to express our true objective in a linear formulation, we instead minimize the computation time on P_0 (see Section 3.2.2 for our computation time model); later we specify constraints which ensure that the other processors are time-balanced with P_0. Next, we specify **bounds** on the unknown variables: each processor should be assigned a non-negative number of mesh rows not to exceed the total number of rows N: $\forall i \in \{0 : p-1\}, 0 \leq n_i \leq N$. The rest of the specification is in the form of **constraints**. First, the total number of data mesh rows allocated must be equal to N: $\sum_{i=0}^{p-1} n_i = N$. Next, the data allocated to each processor must fit within that processor's local memory: $\forall i \in \{0 : p-1\}, memUnit * N * n_i \leq mem_i$; refer to Section 3.2.1 for our memory usage model. Finally, we specify that each processor's predicted iteration time should equal P_0's predicted iteration time: $\forall i \in \{1 : p-1\}, |itTime_i - itTime_0| = 0$. We add support for relaxation of time balancing requirements with relax factor R and the constraint becomes: $\forall i \in \{1 : p-1\}, |itTime_i - itTime_0| \leq R * itTime_0$. After incorporating details from our execution time model (see Section 3.2.2), re-arranging, and using two inequalities to specify an absolute value, our last two sets of constraints are:

$$\begin{aligned} \forall i \in \{1 : p-1\}, -(1+R) * compTime_0 + compTime_i \\ \leq (1+R) * commTime_0 - commTime_i \end{aligned} \tag{7}$$

$$\begin{aligned} \forall i \in \{1 : p-1\}, (1-R) * compTime_0 - compTime_i \\ \leq (-1+R) * commTime_0 + commTime_i \end{aligned} \tag{8}$$

3.5. Validation results

As described in Section 2, our scheduler typically utilizes a performance model to compare candidate schedules. The ability of our scheduler to select the "best" schedule is therefore directly tied to the prediction accuracy of the performance model.

We performed a suite of validation experiments for the execution time model we described in Section 3.2. The goal of these experiments was to compare predicted application performance ($predTime$) with actual application performance ($actualTime$). We calculate the prediction error as:

$$predError = 100 * \frac{predTime - actualTime}{actualTime}. \tag{9}$$

We do not have space here to fully describe our experimental design; a full explanation is available in [11]. We tested model accuracy for both the Jacobi and the Game of Life applications on both a single site testbed and a three site testbed (see Section 4.1 for details). In total, we obtained 344 comparisons of actual and predicted times. A histogram of the prediction errors we measured in those experiments is shown in Figure 4.

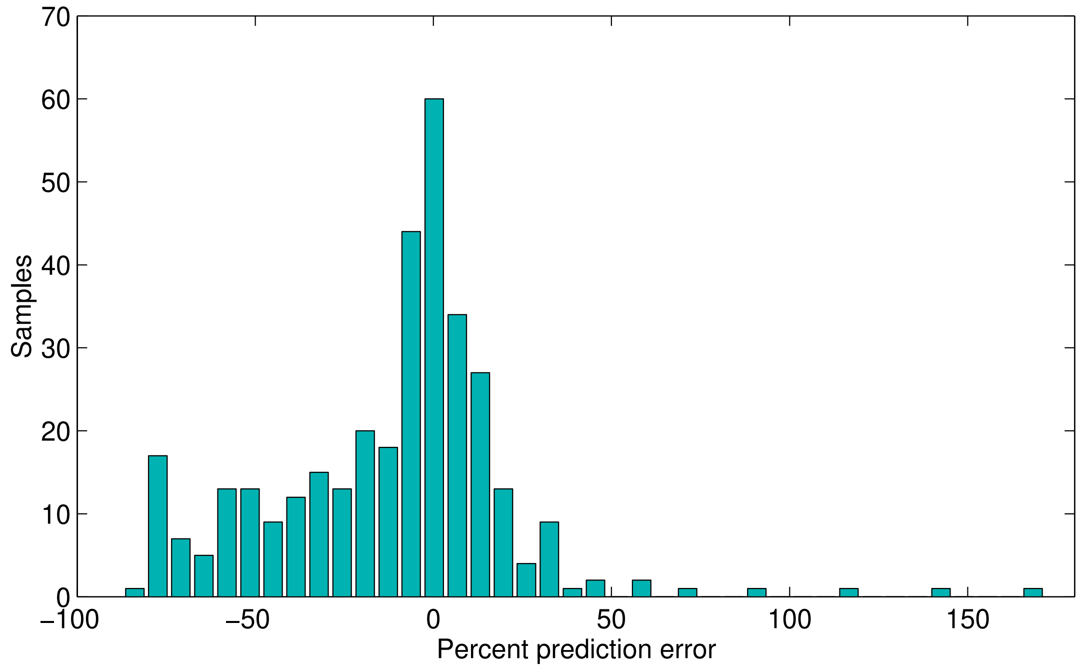

FIG. 4 Histogram of prediction errors measured in a total of 344 experiments. Results are aggregated from experiments conducted with Jacobi and the Game of Life on both a single site testbed and a three site testbed.

The prediction accuracy of our execution time model is moderate. The objective of this work is not to provide performance models for applications, but rather to demonstrate how such models can be utilized as part of our scheduling strategy. More sophisticated and precise models could be developed and used. Our evaluation results will show how our approach behaves with reasonably accurate models. Such are the models we hope to obtain automatically from the GrADS compiler when it becomes available.

4. RESULTS

In this section we describe experimental results obtained for the Jacobi and Game of Life applications in realistic Grid usage scenarios. We designed these experiments to investigate the following questions.

i. *Does the scheduler provide reduced application execution times relative to conventional scheduling approaches?*

ii. *Can the scheduler effectively utilize dynamic resource information to improve application performance? Can reasonable schedules still be developed in the absence of dynamic resource information?*

iii. *How is scheduler behavior affected by degraded application information? Specifically, can reasonable schedules still be developed in the absence of an application execution time model?*

4.1. Experimental design

4.1.1. Testbeds

Our experiments were performed on a subset of the GrADS testbed composed of workstations at the University of Tennessee, Knoxville (UTK), the University of Illinois, Urbana-Champaign (UIUC), and the University of California, San Diego (UCSD). Figure 5 depicts our testbed and provides a snapshot of available bandwidths on networks links. Table 1 summarizes testbed resource characteristics. This collection of resources is typical of Computational Grids: it is used by many users for an array of purposes on an everyday basis, the resources fall under a variety of administrative domains, and the testbed is both distributed and heterogeneous.

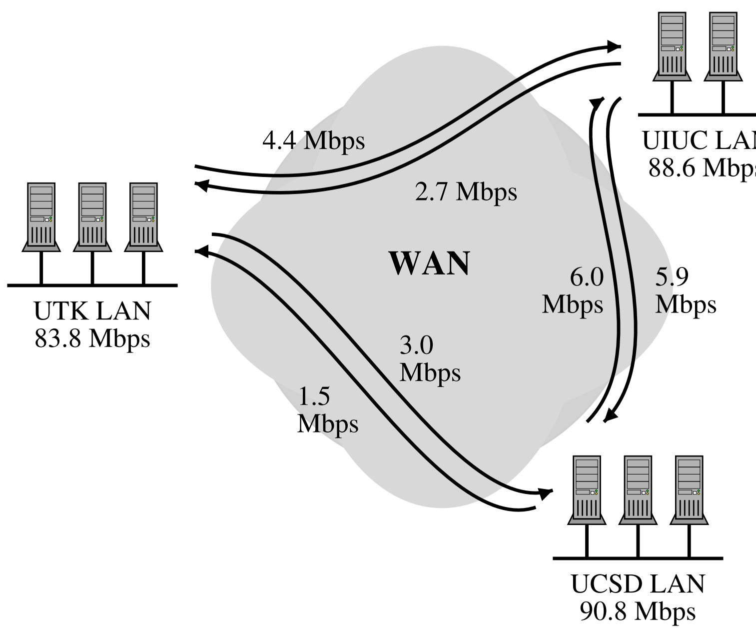

FIG. 5 A heterogeneous, distributed network of workstations. Network links are labeled with available bandwidth in megabits per second; these values were collected by Network Weather Service network monitoring sensors on November 1, 2001 at around 5:30 PM.

Experiments were performed on the full **three-site tested** and on a **one-site testbed** with only UCSD resources. We ran experiments on the one-site testbed with **problem sizes** of {600, 1200, 2400, 4800, 7200, 9600} and on the three-site testbed with problem sizes of {600, 4800, 9600, 14400, 16800, 19200}.

	Circus machines	Torc machines	Opus machines	Major machines
Domain	ucsd.edu	cs.utk.edu	cs.uiuc.edu	cs.uiuc.edu
Nodes	6	8	4	6
Names	dralion, mystere soleil, quidam saltimbanco nouba	torc1, torc2 torc3, torc4 torc5, torc6 torc7, torc8	opus13-m opus14-m opus15-m opus16-m	amajor, bmajor cmajor, fmajor gmajor, hmajor
Processor	450 MHz PIII dralion, nouba 400 MHz PII others	550 MHz PIII	450 MHz PII	266 PII
CPUs/Node	1	2	1	1
Memory/Node	256 MB	512 MB	256 MB	128 MB
OS	Debian Linux	Red Hat Linux	Red Hat Linux	Red Hat Linux
Kernel	2.2.19	2.2.15 SMP	2.2.16	2.2.19
Network	100 Mbps shared ethernet	100 Mbps switched ethernet	100 Mbps switched ethernet	100 Mbps shared ethernet

TABLE 1
Summary of testbed resource characteristics.

4.1.2. Scheduling strategies

To help answer questions i-iii, we developed four scheduling strategies, *dynamic*, *static*, *basic*, and *user*, each based on a realistic Grid scheduling scenario. The *dynamic* strategy uses our scheduler design and represents the case when full information is available about the application and testbed. For this strategy, the scheduler is coupled with the full execution time performance model (see Section 3.2.2) and time balance mapper (see Section 3.4.2). Scheduling decisions are made at run-time and are based on near real-time CPU availability, free memory, and available bandwidth information from the NWS as well as CPU speed data from the MDS.

The *static* strategy models our scheduler's behavior when full application information is available, but resource information is degraded. Specifically, this strategy uses the same performance model and mapper as the dynamic strategy, but assumes dynamic resource information is not available at run-time (i.e. the NWS server is unavailable). Scheduling decisions are made off-line, and are based on static information from the MDS such as available physical memory and total CPU speed. Estimates of available bandwidth from the NWS are used, but are not collected at run-time.

The *basic* strategy models our scheduler's behavior when resource information is fully available, but a complete execution time model is not. This strategy uses the memory usage model described in Section 3.2.1 and the alternative ScheduleCompare method defined in Section 3.3. Mapping is done with the equal allocation mapper defined in Section 3.4.1. This mapper does not require execution time information.

The *user* strategy is designed to emulate the scheduling process that a typical Grid user might employ and provides a comparison with a conventional approach. We assume that users will generally only invest time in scheduling once per application configuration; in this scenario static resource information is sufficient since scheduling occurs off-line. We also assume that a user has a preferred ordering of resources; for example, most users will utilize their "home" resources before resources on which they are a "guest". For the three-site testbed, we assume a resource ordering of {UCSD, UTK, UIUC}. We assume a typical user will not have a detailed performance model, but may be able to estimate application memory usage. The strategy therefore uses our memory usage model and selects the minimum number of resources that will satisfy application memory requirements. Figure 6 summarizes the application and Grid information usage by each of the four scheduling strategies.

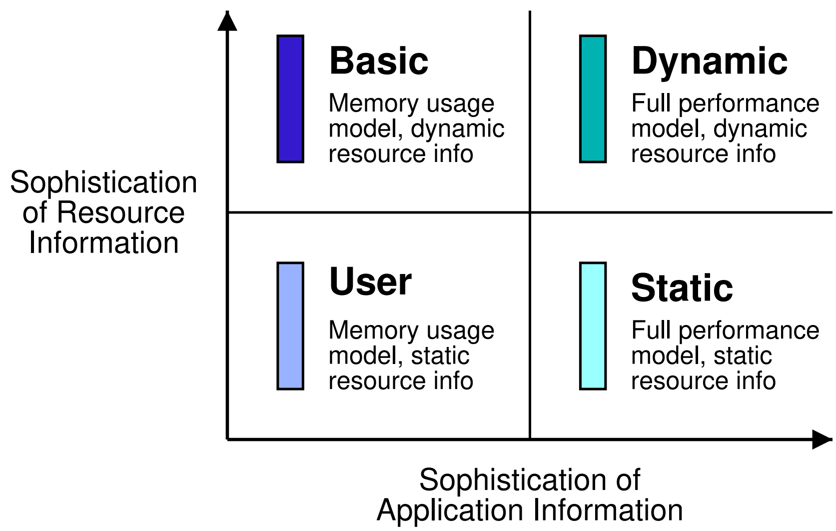

FIG. 6 Summary of user, basic, static, and dynamic scheduling strategies. For each strategy we note the availability of sophisticated application and resource information. Bars correspond to the colors used for each strategy in our scheduling results graphs.

Comparison of our scheduler against the performance achieved by an automated, run-time scheduler would clearly be a desirable addition to the four strategy comparison we have defined. Unfortunately, there is currently no comparable Grid scheduler that is effective for the applications and environments that we target. We listed other Grid scheduler efforts in Section 1; we plan to investigate other applications and environments for which a reasonable scheduler comparison could be made.

4.1.3. Experimental procedure

A scheduling strategy comparison experiment consists of back-to-back runs of the application as configured by each scheduling strategy. In each application execution, 104 iterations were performed; an average and standard deviation of the iterations times was then taken of all but the first 4 "warmup" iterations. Based on the characteristics of iterative, mesh-based applications, we compare application performance based on the worst average iteration time reported by any processor in the computation.

To avoid undesirable interactions between each application execution and the dynamic information used by the next scheduler test, we included a three-minute sleep phase between tests. To obtain a broad survey of relative strategy performance, we ran scheduling strategy comparison experiments for all combinations of the two applications, the two testbeds, and six problem sizes (2*2*6 = 24 testing scenarios). We performed 10 repetitions of each testing scenario for a total of 240 schedule comparison experiments and 720 scheduler / application executions.

4.1.4. Strategy comparison metric

We use a common comparison metric, *percent degradation from best* [26]. For each experiment we find the lowest average iteration time achieved by any of the strategies, $itTime_{best}$, and compute

$$degFromBest = 100 * \frac{itTime - itTime_{best}}{itTime_{best}}, \tag{10}$$

for each strategy. The strategy that achieved the minimum iteration time is thus assigned $degFromBest = 0$. Note that an optimal scheduler would consistently achieve a 0% degradation from best.

4.2. Aggregate results

Figure 7 presents an average of the percent degradation from best achieved by each scheduling strategy across all scheduling strategy comparison experiments. Each bar in the graph represents an average of approximately 70 values. Table 2 presents additional statistics for the same data set. In all application-testbed combinations, the user strategy is outperformed, on average, by the three other strategies. Since all but the user strategy are variations of our scheduling methodology these results provide sufficient evidence to answer *question i* in the affirmative: our approach does provide reduced application execution times relative to conventional approaches. The improvement in average performance from the user to the static strategy partially answers *question ii*: reasonable schedules can still be developed in the absence of dynamic resource information. Additionally, recall that the primary difference between the user and basic strategy pair and the static and dynamic strategy pair is the usage of dynamic information. Since the basic strategy outperforms the user strategy and the dynamic strategy outperforms the static strategy we can answer the rest of *question ii* in the affirmative: the scheduler does effectively utilize dynamic resource information to improve application performance. Finally, in *question iii* we posed the query of how scheduler behavior is affected by the availability of an accurate performance model. As expected, the scheduling strategies which utilize an accurate application performance model (i.e. static and dynamic) outperform those that do not (i.e. user and basic).

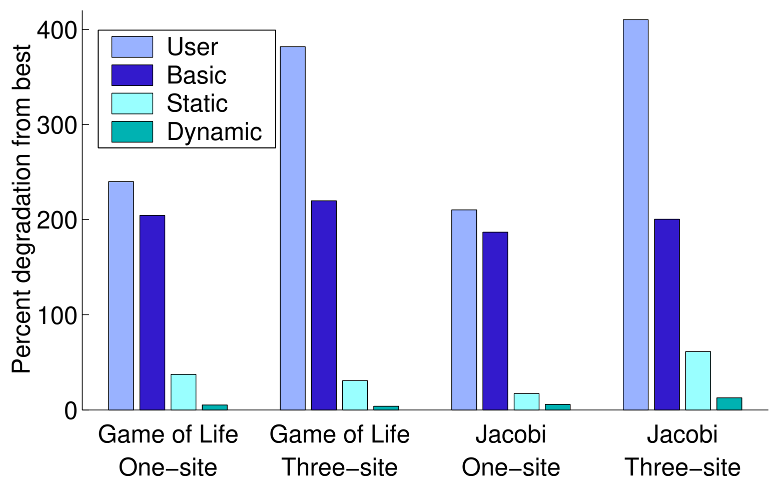

FIG. 7 Average percent degradation from best for each scheduling strategy and each application-testbed combination.

App & Testbed	Statistic	User	Basic	Static	Dynamic
Game of Life 1-site	Avg ± std [Min, Max]	240.0 ± 152.0 [7.7, 507.7]	204.4 ± 135.6 [15.2, 433.5]	37.3 ± 40.4 [0, 156.9]	5.1 ± 12.9 [0, 69.3]
Game of Life 3-site	Avg ± std [Min, Max]	381.9 ± 466.6 [45.3, 2748.0]	219.8 ± 268.2 [6.6, 1109.2]	30.8 ± 63.3 [0, 421.8]	3.8 ± 10.7 [0, 68.5]
Jacobi 1-site	Avg ± std [Min, Max]	210.3 ± 130.6 [16.4, 466.4]	186.9 ± 139.8 [7.9, 487.7]	17.2 ± 28.2 [0, 90.5]	5.7 ± 12.6 [0, 69.7]
Jacobi 3-site	Avg ± std [Min, Max]	410.3 ± 212.7 [0, 862.9]	200.4 ± 203.4 [0, 629.6]	61.3 ± 145.8 [0, 739.2]	12.7 ± 40.6 [0, 215.1]

TABLE 2
Summary statistics for percent degradation from best for each scheduling strategy over all application-testbed scenarios.

While the scheduling strategies show a clear ordering in average performance, examination of individual experimental results shows that relative scheduler performance can be heavily influenced by run-time conditions, application characteristics, and other factors. In the following sections, we present a detailed analysis of a small subset of our experiment results. We use these case studies to provide insight as to the behavior of each scheduling strategy, and to highlight conditions where a specific strategy was particularly effective or ineffective.

4.3. Case study I: Variability with time

We first detail experimental results for the Jacobi application on the three-site testbed with a problem size of 9600. We performed 10 experiment repetitions over the period of Oct 16 - Nov 10, 2001. Specifically, we collected repetitions 1-3 on October 16-17, 4-6 on November 6-7, and 7-10 on November 9-10. In this section, we present the resource selection decisions made by each scheduling strategy, and then describe application performance results obtained for these schedules.

Figure 8 reports the schedules selected by each scheduling strategy in each experiment repetition. The number of machines selected is shown with grayed rectangles. Machine selection is reported for each site in the testbed with UIUC machines further differentiated into the Opus cluster (labeled UIUC-O) and the Major cluster (labeled UIUC-M). In 38 of the 40 schedules shown in Figure 8 the schedule includes machines from only a single site. While the user strategy is constrained to select machines in a particular order, the other strategies evaluated performance tradeoffs and automatically selected a subset of the total resource pool. Figure 8 also shows that the user and static strategies each used the same schedule for all repetitions, while the basic and dynamic strategies each employed different schedules from repetition to repetition. The user and static strategies perform scheduling off-line with static information, while the basic and dynamic strategies utilize dynamic information to make run-time scheduling decisions. Notice also that the static and dynamic strategies typically select more machines than do either the user or basic strategies. The dynamic and static strategies select a resource set size that minimizes predicted application execution time; since the user and basic strategies model situations where an execution time model is not available, these strategies try to reduce communication costs by selecting the minimum number of resources that satisfy application memory requirements.

An interesting characteristic of Figure 8 is that UTK resources are so frequently chosen, particularly by the static and dynamic strategies. In this testbed, the UTK site includes a substantial number of machines (8), each of which is more powerful (in terms of memory and computation) than any machine provided by the other sites (see Table 1). The UTK machines are clearly a good choice when 8 machines is sufficient for the current problem run. Note, however, that when the dynamic strategy selected more than 8 machines (repetition 1 and 2), it did not include UTK machines in the schedule. In fact, throughout the time we were running the experiments for this paper we found that WAN performance between UTK and either UCSD or UIUC was significantly worse than WAN performance between UCSD and UIUC. For example, in repetition

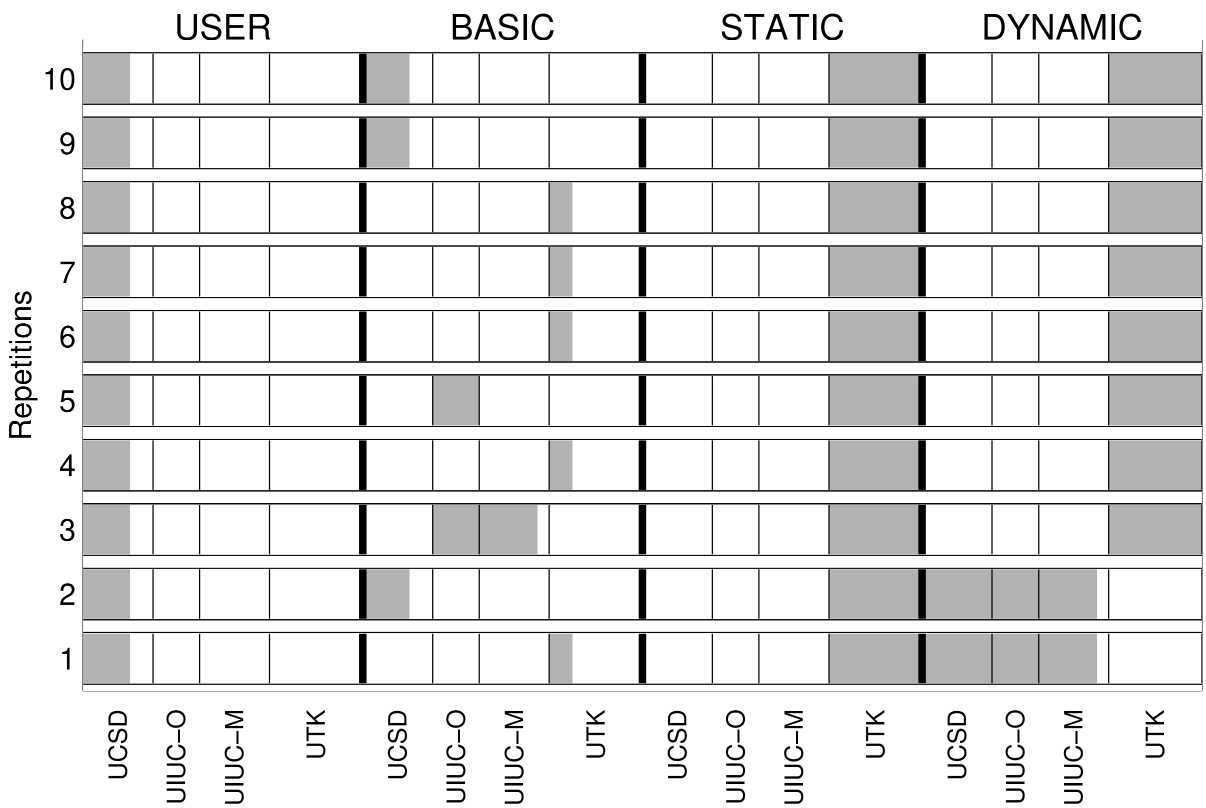

FIG. 8: Processor selections for each scheduling strategy for the Jacobi application on the three-site testbed, problem size of 9600. For each experiment repetition, the processors selected by each strategy are highlighted with gray boxes.

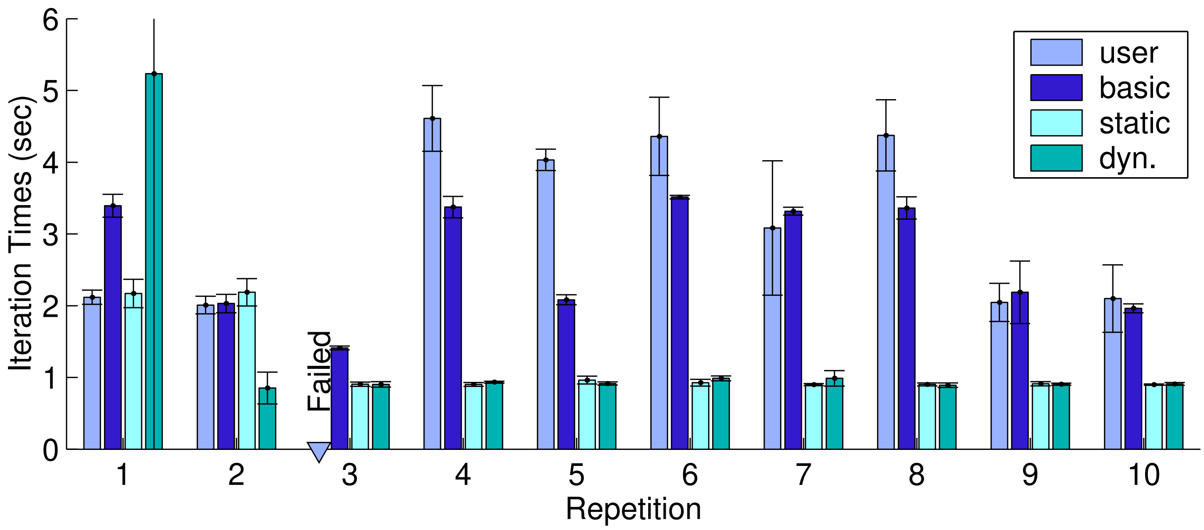

FIG. 9: Average and standard deviation in iteration times for the Jacobi application on the three-site testbed, problem size of 9600.

4 of the current series (Jacobi application on the 3-site testbed with a problem size of 9600) the dynamic scheduler obtained the following bandwidth predictions from the NWS: UTK to UIUC 0.21 Mbps, UTK to UCSD 0.14 Mbps and UIUC to UCSD 5.92 Mbps. Accordingly, our scheduler (as represented by the basic, static, and dynamic strategies) automatically avoids schedules that use UTK machines in a multi-site schedule.

Figure 9 reports results obtained when the application was executed with each of these schedules. In this figure, bar height indicates average iteration times, and error bars indicate standard deviation of iteration times. In this figure, the relative performance of the schedulers does not match the results shown in Figure 7.

No times are reported for the user strategy in the third repetition because the application failed to complete. Upon closer examination, we found that the size of the data allocated to one of the machines exceeded its available physical memory, leading to serious interference with other users' jobs. The application was killed to allow normal progress for the other jobs. This experiment highlights the importance of run-time scheduling with dynamic resource information.

The results shown for the first repetition in Figure 9 are striking. In this repetition, the dynamic strategy performed particularly poorly on average, and the standard deviation in iteration times was surprisingly high (41.2 sec). Figure 10 shows the time measured for each iteration of the application in each of the four scheduler / application runs. **The iteration times for each scheduler are plotted on the same graph for comparison purposes only; the application runs were actually performed at different times and, possibly, under different conditions**. We have selected this case study for its usefulness in demonstrating a few points; the behavior of the scheduling strategies seen here is in fact anomalous (refer to Figure 7).

While the dynamic strategy was the worst performer on average in this repetition, Figure 10 shows that the dynamic strategy was actually the best performer for the majority of iterations and that a few dramatic iteration time jumps were responsible for poor average performance. We investigated system behavior during the most dramatic jump (413 seconds), and found that NWS CPU availability measurements for bmajor.cs.uiuc.edu, one of the machines in this schedule, were almost completely missing during 320 seconds of the 413 second iteration. We believe that during the long iteration period bmajor.cs.uiuc.edu was completely off-line or so disrupted that even lightweight NWS sensors, and therefore our application, could not run. We also identified a correlation between a dramatic increase in bmajor.cs.uiuc.edu computation times during the last 40 or so iterations, and the broad shift upward in application iteration times in the last 40 or so iterations. This case demonstrates how sensitive the overall performance of even loosely synchronous applications can be to the performance of individual machines. This case also reveals the limitations of reporting only average iteration times; however, average iteration times are representative of total execution time, which is the application metric experienced by users.

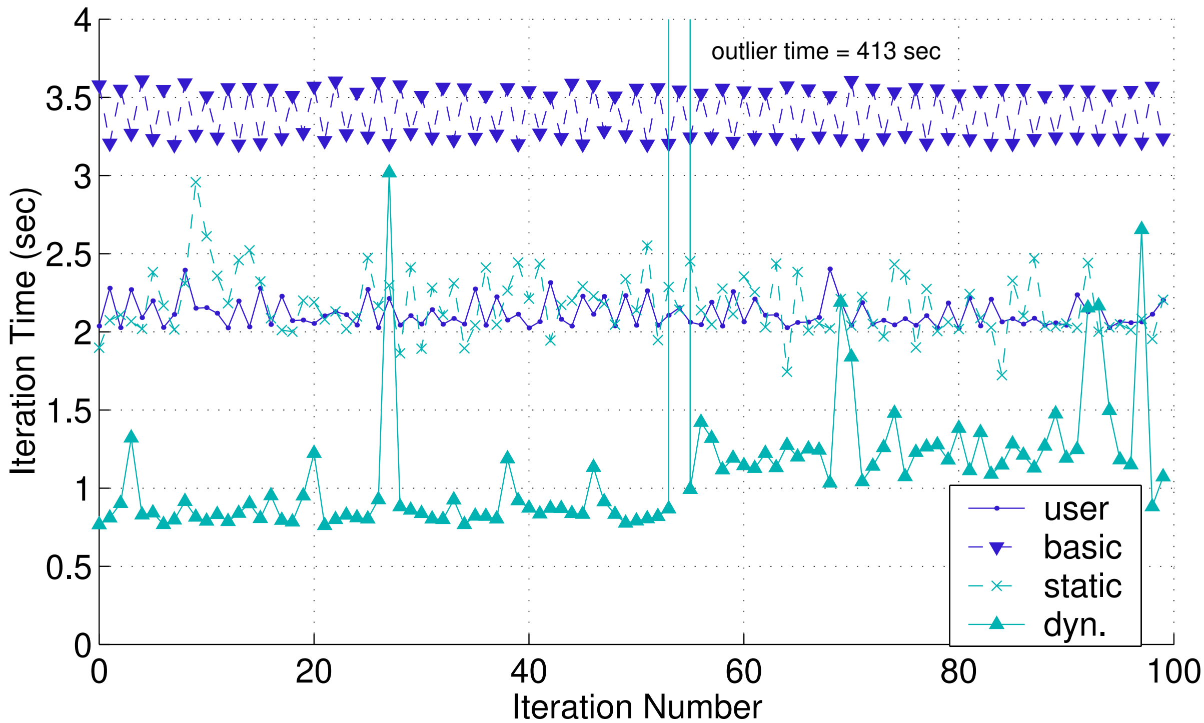

FIG. 10 Time for each application iteration as a function of iteration number. Each run was performed at a different time and possibly under different conditions; they are plotted together for comparison purposes only. Results are for the Jacobi application on the 3-site testbed with $N = 9600$ and repetition = 1. The dramatic drop is performance of the dynamic strategy at iteration 53 is responsible for that strategy's poor average performance in repetition 1 of Figure 9.

4.4. Case study II: Variability with problem size

In the preceding case study, we examined variations in scheduler behavior and performance across different repetitions of the same experiment. In this section, we again focus on experiments for the Jacobi application on the 3-site testbed, but we detail experiments performed during a shorter time period (5 hours on November 4, 2001) and across a variety of problem sizes. Figure 11 reports the machine selections used by each scheduling strategy in these experiments, and Figure 12 reports the measured application performance for each schedule. The iteration times reported in Figure 12 extend from 0.01 to 20.68 seconds per iteration, a range of 3 orders of magnitude.

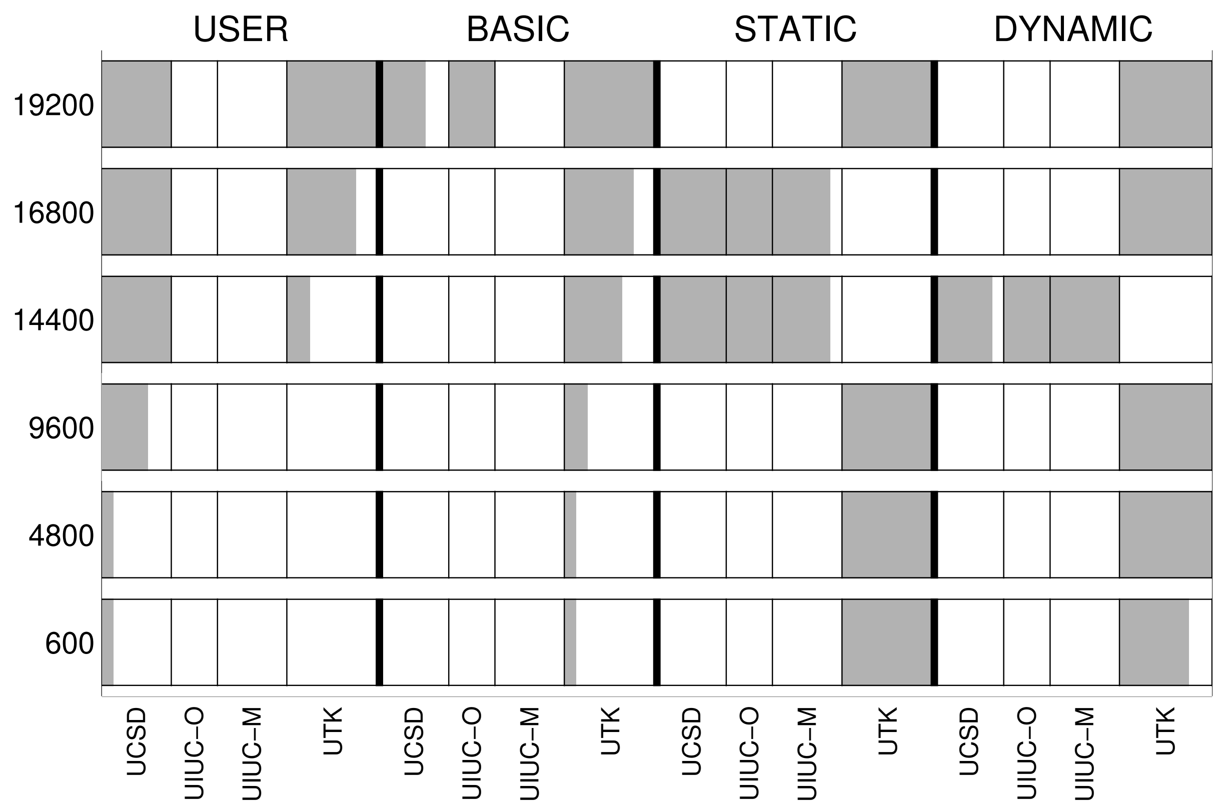

FIG. 11: Processor selections for each scheduling strategy for the Jacobi application on the three-site testbed for all six problem sizes. For each experiment repetition, the processors selected by each strategy are highlighted with gray boxes.

Notice that while all four strategies were successful in finding a schedule for N = 19200 (see Figure 11), both the user and dynamic strategies failed during the launch or execution of the application itself (see Figure 12). The dynamic strategy experiment failed during application launch due to an unidentified problem on the launching machine. The user strategy experiment failed because the application heavily interfered with a user's work, who then killed the application. When the application interferes with the work of other users to this extent it suggests that (1) best estimates of available memory should be used at run-time

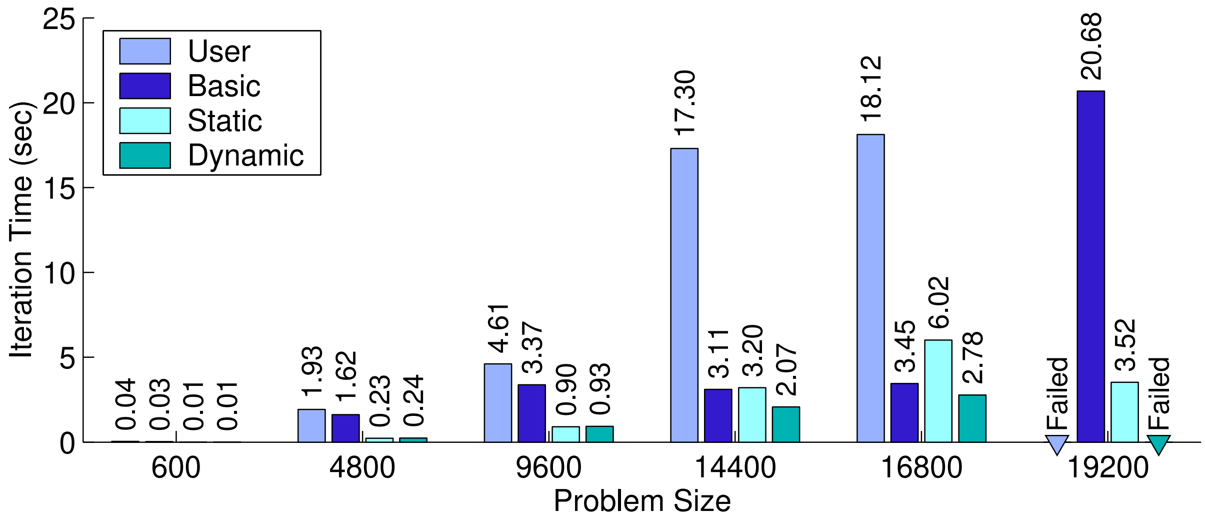

FIG. 12: Average and standard deviation in iteration times for the Jacobi application on the three-site testbed for six problem sizes.

to ensure machine memory availability and avoid thrashing and (2) our memory usage model may be too conservative in its estimate of how much memory should be available for the application on each machine. One must always consider the benefits of additional memory for the application with the communication costs associated with utilizing more machines.

There are 4 schedules shown in Figure 11 that include both UTK machines and machines at another site: the user strategy for N = 14400, 16800, and 19200 and the basic strategy for N = 19200. These cases correspond exactly to the worst iteration time results shown in Figure 12; these schedules performed poorly because of the poor network performance between UTK and the other sites (refer to the preceding section for details). Since the ordering of machine selection is predefined for the user strategy {UCSD, UTK, UIUC}, it is not surprising that the user strategy selected these schedules. It is surprising, however, that the basic strategy selected such a schedule.

Let us investigate this behavior in some detail. A problem size of 19200 is demanding for this testbed; in aggregate, at least 3375 MB of RAM are required. Given an equal work allocation, this translates into per machine available memory requirements, for example, of 140.6 MB / machine for 24 machines, 210.9 MB / machine for 16 machines, or 421.9 MB / machine for 8 machines. Given unloaded conditions and an equal work allocation, a schedule for N = 19200 can not utilize all 24 machines because the Major machines at UIUC have only 127 MB each, 16 machines can be utilized only if the Major machines are excluded, and 8 machine can be utilized only if all 8 machines are from UTK.

Notice that the basic strategy selected 16 machines. Since the basic strategy selects the smallest resource set that satisfies application memory requirements, it may seem surprising that the strategy selected more than just the 8 Torc machines. Recall however that the basic strategy collects and uses dynamic resource information at schedule-time (i.e. run-time). In this experiment, the basic strategy found that some of the UTK machines were partially loaded and could no longer provide the minimum 421.9 MB / machine needed to run the application on just the 8 UTK machines. The strategy therefore selected more machines, thereby reducing the minimum memory required per machine to levels which could be supported by the UTK machines.

Looking again at Figure 11, it may now seem strange that both the static and dynamic strategies determined that an 8 UTK machine was an acceptable choice. The static strategy uses only static information and therefore assumes all resources are unloaded; under unloaded conditions the 8 UTK machines are an appropriate choice. Looking at the results shown in Figure 12, it appears that this choice was also reasonable in the conditions experienced by the static strategy (i.e. the strategy performed reasonably). In other cases, blindly assuming unloaded conditions can have drastic affects.

When the dynamic strategy ran, it retrieved run-time resource information and found that the UTK machines were partially loaded. However, usage of the time balance mapper provided this strategy with the added flexibility of unequal work allocations. The time balance mapper found a map which allowed usage of the 8 UTK machines by shifting some work from the partially loaded machines to the unloaded machines. In general, we found that the time balance mapper was not only generally successful in reducing application iteration times, but was also very useful in increasing the number of scheduling choices available.

4.5. Scheduler and application failures

During the experiments reported in this paper we encountered a number of application and scheduler failures. A detailed analysis of the types of failures that occur and with what frequency can be found in [12]; we summarize those results here for completeness.

A scheduler failure occurs when the scheduler can not find a feasible schedule for the application. Our scheduling methodology reduces the frequency of scheduling failures by (1) using an effective search heuristic to ensure that feasible machine groups are identified, should they exist, and (2) using a time-balancing mapper that adapts data allocations to match the available memory capacities of individual machines. For these measures to succeed, computation and memory capacity information must be available for targeted machines; our methodology occasionally fails due to lack of resource information.

We identified a variety of failures which we label "application failures": the application launch process occasionally failed due to unexplained script failures; memory allocation by the application can fail if there is not sufficient available memory; application communication processes failed occasionally due to either Globus or socket communication bugs; one of the machines involved in the computation can fail to participate in communication (either the machine went down or off-line); and, finally, application resource usage can interfere with other users' work at which time they will sometimes kill the application. The frequency of launch and run-time failures provides useful insight into the stability of program execution on the Grid.

4.6. Scheduling overhead

A scheduler design is practical only if the overhead of the scheduling process is reasonable when compared to application execution times. The results presented in the preceding sections of this paper have not included the overheads introduced by the scheduler itself. In this section we quantify these overheads.

We consider the total overhead for scheduling, as well as the overhead for each of the two activities performed by our scheduling methodology: (i) the collection of resource information from the MDS and NWS and (ii) the search for candidate schedules. We measured scheduling overheads for the Jacobi application on the 3-site testbed with the same scheduler configuration as was used for the dynamic scheduling strategy. While this case study provides a reasonable overview of the overheads of our methodology, note that the cost of scheduling is dependent on problem run configuration, the selected testbed, the target application, the complexity of the chosen performance model and mapper, and variable load on the GrADS NWS nameserver and MDS server.

It is important to differentiate the costs of data retrieval from the MDS and NWS servers from the cost of transferring the request for data and the response over WANs. We set up an NWS nameserver and an MDS cache at UCSD and we performed the tests from a machine at UCSD; these sources of information will be referred to as the *local NWS* and the *local MDS cache*. We include test scenarios in which NWS information is retrieved from either the local NWS or from the GrADS NWS, which was located at UTK in Knoxville, Tennessee. We also include scenarios in which MDS information is retrieved from the local MDS cache or from the GrADS MDS, which was located at ISI in Los Angeles, California. Specifically, we test the following retrieval modes.

- Mode A used the GrADS NWS nameserver and the GrADS MDS server.
- Mode B used the GrADS NWS nameserver and a local MDS cache. For these experiments, the local MDS cache contained all needed information (i.e. it was *fully warmed*).
- Mode C used the local NWS nameserver and a fully warmed local MDS cache.

We ran the scheduler with each of the three retrieval modes in a back-to-back manner; we completed 10 such triplets. For each run, we measured the time required for the entire scheduling execution ($TotalTime$) and the time required for Grid information collection ($CollectTime$); we consider the cost for the schedule search ($SearchTime$) to be all scheduling time that is not spent in information collection: $SearchTime = TotalTime - CollectTime$.

Table 3 presents summary results over all 10 repetitions for the mean and standard deviation of the $CollectTime$, $SearchTime$, and the $TotalTime$. For reference, when we ran scheduling experiments with a similar experimental configuration, the four scheduling strategies typically achieved application iteration times between 1.8 and 11 seconds. Since we ran roughly 100 iterations in these experiments, the application's iterative phase occupied 180 to 1100 seconds.

	Mode A	Mode B	Mode C
Collect Time, Avg ± std	1087.5 ± 303.3	59.6 ± 3.9	2.0 ± 0.7
Search Time, Avg ± std	0.8 ± 0.3	2.4 ± 0.4	2.5 ± 0.3
Total Time, Avg ± std	1088.4 ± 303.3	62.1 ± 3.9	4.5 ± 0.9

TABLE 3
Scheduling overheads in seconds to schedule Jacobi on the three-site testbed, $N = 14400$, dynamic scheduling strategy.

The cost of **Grid information collection** dominates scheduling overhead in all modes except C. In mode C only 2 seconds, on average, were required to collect information on all 24 machines in this testbed. This overhead is very reasonable when compared with application run-times; we conclude that our procedure for collecting resource information is efficient enough and that the cost of data retrieval from an NWS server is reasonable. In mode B, NWS data were retrieved from a remote NWS server, which increased information collection times to approximately 60 seconds. We conclude that collection of information from a remote NWS server is efficient enough for most uses of our scheduler. The overhead could become problematic for a larger testbed; in this case, our scheduler can be used without run-time resource information as was done for the static strategy in this paper. Finally, mode C utilized both the remote NWS nameserver and the remote MDS server, thus increasing collection times to 1087.5 seconds, or approximately 18 minutes. This overhead is prohibitive, and, in practice, would prevent usage of our scheduling approach. We conclude that until retrieval times are reduced for the MDS, local caching of MDS information will be necessary. The MDS information that is retrieved and used by our methodology changes on the order of weeks or months; local caching is therefore an acceptable solution for this work. Note that ongoing development work in both MDS and NWS is seeking to reduce information retrieval latencies.

The cost of the **schedule search process** is less than 2.5 seconds for all three collection modes and is therefore an acceptable overhead for our scheduling scenarios. This low search time overhead is due to (1) the low computational complexity of our execution time model and mapping strategy and (2) the extensive search pruning performed during the search process. Notice that the average search time in mode A is only about 33% of the search time for modes B and C. Mode A retrieves some resource information from the GrADS MDS and during these experiments that server was unable to provide much of the required information. Our scheduling methodology does not consider machines for which no data is available, thus leading to pruning of the schedule search space and a reduction in search time.

5. DISCUSSION

In this section we describe related work (Section 5.1), describe possible extensions to our work (Section 5.2), and conclude the paper (Section 5.3).

5.1. Related work

The **Application-Level Scheduling Project (AppLeS)** has developed many successful scheduling strategies for the Grid including [8, 9, 13, 37, 38]. These efforts provided important foundations for the core components of our scheduling approach: the schedule search procedure, the mappers, and the performance models.

Two of these efforts are particularly related to this work [8, 13]. The first focused on the scheduling of a Jacobi solver for the finite-difference approximation to Poisson's equation [8]. The second effort focused on scheduling of a parallel magnetohydrodynamics simulation (PMHD3D), which is also classified as an iterative, mesh-based application [13]. Each of these efforts demonstrated significant improvements in application performance as compared to conventional scheduling efforts. The performance models and mappers presented in Section 3 are based in part on the models and mapping strategies used in the previous Jacobi and PMHD3D work. We also used these efforts as examples for the development of our search procedure; we believe that the search procedure we have developed is more thorough and therefore more likely to discover desirable machine groups. Our search procedure has also been developed to work with both LANs and WANs and has been thoroughly tested in both environments; the Jacobi and PMHD3D schedulers were tested only in LAN environments.

Many AppLeS efforts [8, 13, 37, 38] have targeted specific applications; the most important contribution of our work is that we have separated the application-specific components from the application-generic. Due to this formal separation, we believe that our scheduling approach is more easily re-targeted to new applications than most previous AppLeS efforts. Two AppLeS efforts have successfully targeted classes of applications [9, 35]. However, both of these efforts target master-slave applications, whereas we target applications which may involve significant inter-processor communications.

There are a number of **other scheduling projects** that are notable for targeting a variety of applications or an entire application class; examples include the Condor Matchmaker [33], Prophet [40], and Nimrod/G [1]. The Nimrod/G effort focuses on embarrassingly parallel applications and so is more comparable with other AppLeS efforts [9, 35] than with the current effort.

Prophet is a run-time scheduling system designed for parallel applications written in the Mentat programming language [41, 40]. Another related effort is Prophet, a run-time scheduling system designed for parallel applications written in the Mentat programming language [41, 40]. This scheduling system is similar to our work in that it exploits application structure and system resource information to promote application performance. Prophet was demonstrated for both SPMD applications and applications based on task-parallel pipelines; the scheduler design was tested in heterogeneous, local-area environments. If possible, we would like to compare the performance of our strategies to those of Prophet, though it may be difficult to find a suitable scenario for comparison that satisfies the requirements of each scheduling strategy. For example, Prophet requires the target application be written in Mentat and we have not used Mentat in our efforts.

Another project of interest is the Condor Matchmaker [33]. In the Matchmaking system, users specify the resource requirements of their application to the system, resource providers similarly specify the capabilities of their resources, and a centralized Matchmaker is used to *match* application resource requirements with appropriate resources. This design is quite general and can therefore be applied to many different types of applications. The Matchmaking strategy, while more general that the scheduler presented in this paper, differs in that it is primarily a resource discovery mechanism and is not able to provide detailed schedule development.

5.2. Future work

We have an initial prototype of our scheduler which we used to obtain the experiments presented in this paper. We are currently refining this prototype and integrating our software into the main GrADS software base [25]. An interesting extension to our work would incorporate in our search procedure distinct searches for different types of machines; for example, given a master-slave application, one might want to first find the best machine for the master process, and then search for machines for the slave processes (excluding the machine selected for the master). We also plan to extend the validation of our scheduler by testing it with other applications and application classes as well as other testbeds.

Several schedulers have been developed in the GrADS project for use with specific applications [3, 32]. We plan to test our approach with those applications and then compare the performance achieved by each scheduler. We are also collaborating with the developers of these schedulers to define the fundamental characteristics of a successful scheduling approach in the GrADS environment.

For the purposes of this work, we designed and built the application performance models and mapping strategies. However, if Grid application development is to be accessible to a larger number of users, then we cannot expect users to provide detailed performance models and mapping strategies. Recognizing this, other members of the GrADS research community are investigating the feasibility of compiler generation of application information and performance models [24] as well as the inclusion of such models in Grid-enabled libraries [24, 29]. As this work matures, we plan to experiment with the usage of such models for application scheduling.

5.3. Conclusions

In this paper we proposed an adaptive scheduling approach designed to improve the performance of parallel applications in Computational Grid environments. In Section 2 we presented the architecture of our scheduler and we detailed our search procedure, which lies at the heart of the scheduler. In Section 3 we described Jacobi and the Game of Life, two iterative, mesh-based applications which we selected as test cases for our scheduler. For each application, we presented data mappers and performance models appropriate for use by a scheduler. For validation of our approach, we used a prototype of our scheduler in conjunction with the mappers and performance models developed in Section 3.

In Section 4 we presented the results of experiments where we applied our scheduling approach in realistic usage scenarios in production Grid environments. These experiments demonstrate that our scheduler provides significantly better application performance than conventional scheduling strategies. Our experiments included scheduling strategies in which application and/or resource information was limited; with these experiments we demonstrated that our scheduler gracefully handles degraded levels of availability of application and Grid resource information. Finally, we showed that the overheads introduced by our approach are reasonable.

ACKNOWLEDGMENTS

This work benefited greatly from collaborations with the GrADS team. We especially thank the Innovative Computing Laboratory at UTK and the Pablo group at UIUC for allowing us to run experiments on their resources. We also thank Adam Birnbaum for his insightful review of the manuscript.

REFERENCES

[1] Abramson, D., Giddy, J., and Kotler, L. High performance parametric modeling with Nimrod/G: Killer application for the global Grid? In *International Parallel and Distributed Processing Symposium* (May 2000).

[2] Alhusaini, A. H., Prasanna, V. K., and Raghavendra, C. A unified resource scheduling framework for heterogeneous computing environments. In *Proceedings of the 8th Heterogeneous Computing Workshop* (April 1999).

[3] Allen, G., Angulo, D., Foster, I., Lanfermann, G., Liu, C., Radke, T., Seidel, E., and Shalf, J. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a grid environment. *International Journal of High Performance Computing Applications 15*, 4 (2001), 345–358. To appear.

[4] Banikazemi, M., Sampathkumar, J., Prabhu, S., Panda, D. K., and Sadayappan, P. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *Proceedings of the 8th Heterogeneous Computing Workshop* (April 1999).

[5] Barrett, R., Berry, M. W., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[6] Berman, F. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999, ch. 12: High-Performance Schedulers, pp. 279–309.

[7] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed, D., Torczon, L., and Wolski, R. The GrADS Project: Software support for high-level Grid application development. *International Journal of Supercomputer Applications 15*, 4 (2001), 327–344. To appear.

[8] Berman, F., Wolski, R., Figueira, S., Schopf, J., and Shao, G. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing* (November 1996).

[9] Casanova, H., Obertelli, G., Berman, F., and Wolski, R. The AppLeS Parameter Sweep Template: User-level middleware for the Grid. In *Proceedings of Supercomputing* (November 2000).

[10] Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing* (August 2001).

[11] Dail, H. A modular framework for adaptive scheduling in grid application development environments. Master's thesis, University of California at San Diego, March 2002. Available as UCSD Tech. Report CS2002-0698.

[12] Dail, H., Casanova, H., and Berman, F. A modular scheduling framework for GrADS. Submitted to 11th IEEE Symposium on High Performance Distributed Computing.

[13] Dail, H., Obertelli, G., Berman, F., Wolski, R., and Grimshaw, A. Application-aware scheduling of a magnetohydrodynamics application in the Legion Metasystem. In *Proceedings of the 9th Heterogenous Computing Workshop* (May 2000).

[14] Flake, G. W. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation.* MIT Press, Cambridge, MA, 1998.

[15] Foster, I. *Designing and Building Parallel Programs.* Addison-Wesley, 1995, ch. 2. Available at `http://www-unix.mcs.anl.gov/dbpp`.

[16] Foster, I., Geisler, J., Gropp, W., Karonis, N., Lusk, E., Thiruvathukal, G., and Tuecke, S. Wide-area implementation of the Message Passing Interface. *Parallel Computing 24*, 12 (1998), 1735–1749.

[17] Foster, I., and Karonis, N. T. A Grid-enabled MPI: Message passing in heterogeneous disributed computing systems. In *Proceedings of Supercomputing Conference* (November 1998).

[18] Foster, I., and Kesselman, C. The Globus Project: A status report. In *Proceedings of the 7th Heterogeneous Computing Workshop* (1998).

[19] Foster, I., and Kesselman, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1999.

[20] Fox, G. C., Williams, R. D., and Messina, P. C. *Parallel Computing Works!* Morgan Kaufmann, San Francisco, CA, 1994. Available at `http://www.npac.syr.edu/pcw`.

[21] Grimshaw, A., Ferrari, A., Knabe, F., and Humphrey, M. Wide-Area Computing: Resource sharing on a large scale. *IEEE Computer 32*, 5 (May 1999), 29–37.

[22] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*, 6 (1996), 789–828.

[23] Gropp, W. D., and Lusk, E. *User's guide for `MPICH`, a portable implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[24] Kennedy, K., Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnsson, L., Mellor-Crummey, J., and Torczon, L. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing 61*, 12 (2001), 1803–1826.

[25] Kennedy, K., Mazina, M., Aydt, R., Mendes, C., Dail, H., and Sievert, O. GrADSoft and its Application Manager: An execution mechanism for Grid applications. GrADS Project Working Document V, available at `http://hipersoft.cs.rice.edu/grads/publications_reports.htm`, Oct 2001.

[26] Kwok, Y.-K., and Ahmad, I. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing 59*, 3 (1999), 381–422.

[27] LITZKOW, M. J., LIVNY, M., AND MUTKA, M. W. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (June 1988).

[28] MILLER, N., AND STEENKISTE, P. Collecting network status information for network-aware applications. In *INFOCOM'00* (March 2000).

[29] MIRKOVIC, D., MAHASOOM, R., AND JOHNSSON, L. An adaptive software library for fast fourier transforms. In *Proceedings of the 2000 International Conference on Supercomputing* (2000).

[30] MPI Forum webpage at `http://www.mpi-forum.org`.

[31] PACHECO, P. S. *Parallel Programming With MPI*, second ed. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997, ch. 10, pp. 218–225.

[32] PETITET, A., BLACKFORD, S., DONGARRA, J., ELLIS, B., FAGG, G., ROCHE, K., AND VADHIYAR, S. Numerical libraries and the Grid. *International Journal of High Performance Computing Applications 15*, 4 (2001), 359–374. To appear.

[33] RAMAN, R., LIVNY, M., AND SOLOMON, M. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing* (July 1998).

[34] SHAO, G., BERMAN, F., AND WOLSKI, R. Using Effective Network Views to promote distributed application performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications* (1999).

[35] SHAO, G., WOLSKI, R., AND BERMAN, F. Master/slave computing on the Grid. In *Proceedings of the 9th Heterogenous Computing Workshop* (May 2000).

[36] SMALLEN, S., CASANOVA, H., AND BERMAN, F. Applying scheduling and tuning to on-line parallel tomography. In *Proceedings of Supercomputing Conference* (November 2001).

[37] SMALLEN, S., CIRNE, W., FREY, J., BERMAN, F., WOLSKI, R., SU, M.-H., KESSELMAN, C., YOUNG, S., AND ELLISMAN, M. Combining workstations and supercomputers to support Grid applications: The parallel tomography experience. In *Proceedings of the 9th Heterogenous Computing Workshop* (May 2000).

[38] SU, A., BERMAN, F., WOLSKI, R., AND STROUT, M. M. Using AppLeS to schedule simple SARA on the Computational Grid. *International Journal of High Performance Computing Applications 13*, 3 (1999), 253–262.

[39] SWANY, M., AND WOLSKI, R. Building performance topologies for computational grids. In *Proc. 11th IEEE Symp. on High Performance Distributed Computing* (2002). Submitted.

[40] WEISSMAN, J. Prophet: Automated scheduling of SPMD programs in workstation networks. *Concurrency: Practice and Experience 11*, 6 (1999).

[41] WEISSMAN, J., AND ZHAO, X. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing 1*, 1 (1998), 109–118.

[42] WILLIAMS, H. *Model Building in Mathematical Programming*, second ed. Wiley, Chichester, New York, 1995.

[43] WOLSKI, R., SPRING, N. T., AND HAYES, J. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *The Journal of Future Generation Computing Systems* (1999).