A Decoupled Scheduling Approach for the GrADS Program Development Environment

Holly Dail[*†], Henri Casanova[*†] and Fran Berman[*†]

[*]Department of Computer Science and Engineering
University of California at San Diego

[†]San Diego Supercomputer Center
University of California at San Diego

[hdail, casanova, berman]@sdsc.edu

Abstract—

Program development environments are instrumental in providing users with easy and efficient access to parallel computing platforms. While a number of such environments have been widely accepted and used for traditional HPC systems, there are currently no widely used environments for Grid programming. The goal of the Grid Application Development Software (GrADS) project is to develop a coordinated set of tools, libraries and run-time execution facilities for Grid program development.

In this paper, we describe a Grid scheduler component that is integrated as part of the GrADS software system. Traditionally, application-level schedulers (e.g. AppLeS) have been tightly integrated with the application itself and were not easily applied to other applications. Our design is generic: we decouple the scheduler core (the search procedure) from the application-specific (e.g. application performance models) and platform-specific (e.g. collection of resource information) components used by the search procedure. We provide experimental validation of our approach for two representative regular, iterative parallel programs in a variety of real-world Grid testbeds. Our scheduler consistently outperforms static and user-driven scheduling methods.

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

SC2002, Baltimore Maryland.
0-7695-1524-X/02 $17.00 (c) 2002 IEEE

I. Introduction

With recent improvements in wide-area network performance and the pervasiveness of commodity resources, distributed parallel computing can benefit from an increasingly rich computational platform. However, as shown by many focused development efforts, taking advantage of these Computational Grid environments [13] requires extensive labor and support by distributed computing experts. Grid infrastructure projects such as Globus [12], Legion [16], and Condor [21] have greatly simplified the processes of application development and deployment on Computational Grids. However, since such middleware generally does not account for the specific needs of applications, additional measures are usually necessary before acceptable performance can be achieved. Today, application developers typically perform all transactions that require specific knowledge of the application; such transactions may be performed by hand, or the developer may build special-purpose, application-specific software. Examples of such transactions include selecting an appropriate subset of available resources, staging data and binaries on selected machines, and, for long-running applications, monitoring application progress. Hence, while many scientists could benefit from the extensive resources offered by Computational Grids, application development remains a daunting proposition.

The Grid Application Development Software Project (GrADS) [4] seeks to address these issues with a comprehensive application development system. The end goal is to integrate Grid-enabled libraries, application compilation, scheduling, staging of binaries and data, application launch, and monitoring of application execution progress. A key feature of this system is that application characteristics are recorded and/or discovered by components such as a specialized compiler and Grid-enabled libraries. These application characteristics are communicated via well-defined interfaces to components that provide program execution services (e.g. the scheduler). Through this interaction, components such as the scheduler can be general-purpose, while still providing services that are appropriate to the application at hand.

In this paper, we propose a scheduling approach designed for the GrADS environment. We address the problems of discovering available resources, selecting an application-appropriate subset of those resources, and mapping of data and/or tasks onto selected resources. Our scheduling approach focuses on minimizing the execution time of a single application execution on a set of potentially shared resources. This approach has been termed application-level scheduling. In Section II we compare application-level scheduling and meta-scheduling, a related scheduling approach that considers the performance of many applications at once to improve overall system performance.

Our scheduler design seeks flexibility through modularity: our design explicitly *decouples* the scheduler core (the search procedure) from application-specific (e.g. performance models) and platform-specific (e.g. resource information collection) components used by the search procedure. We present a new schedule search procedure which is general-purpose and effective at identifying desirable groups of resources. We describe this search procedure in detail, and show it to be efficient due to its low complexity. To provide application-appropriate scheduling, our approach depends on the availability of two application-specific components: a *performance model* (an analytic metric for the performance expected of the application on a given set of resources) and *mapper* (directives for mapping logical application data or tasks to physical resources). Other members of the GrADS project are developing methods for automatic generation of such components and results to date are promising [18]. As this work matures, we hope to obtain such components automatically.

To validate our approach in the absence of such facilities, we hand-built performance models and mappers for two applications. As an initial validation, we tested our scheduler in a variety of scheduling scenarios with these applications, several testbeds including both local-area and wide-area networks, problem sizes spanning a wide range of application requirements, and different scenarios for resource information availability. In these experiments, our scheduler provided significantly improved performance relative to a user-directed approach. These results indicate that our approach is a feasible solution to generic scheduling within GrADS.

This paper is organized as follows: Section II describes related work, Section III describes the scheduler design itself, Section IV presents the results we obtained when applying our methodology in real-world Computational Grid environments, and Section V contains a final discussion of the work.

II. Related Work

The importance of application scheduling for the development and deployment of applications on Computational Grids has been recognized for some time [3], and many successful strategies have been developed; examples include the AppLeS Project [5], and application-specific schedulers developed within the GrADS project [22, 26]. These schedulers were generally developed as prototypes to support research into application-level scheduling algorithms for specific applications; they were not general-purpose, public-domain schedulers that one could download and use.

With these goals in mind, two successful and popular scheduler engineering strategies are to (1) embed scheduling logic in the application or (2) to embed application-specific information in the scheduler. As we have experienced with our previous AppLeS work [5, 9, 30, 31], in either case such schedulers are time-consuming to build and are not easily re-targeted for other applications or execution environments. In this paper, we explicitly decouple application-specific components from scheduling components; this decoupling provides a flexible scheduler which can be applied to diverse applications and environments.

Condor Matchmaker [27], Nimrod/G [1], and two previous AppLeS efforts [6, 29] have each developed flexible, retargetable scheduling approaches. However, the AppLeS and Nimrod/G schedulers targeted parallel master-slave applications and the Condor Matchmaker scheduler targets only single processor tasks which are scheduled independently. Our work does not restrict application type, but will likely provide the largest performance advantage for loosely-synchronous, parallel applications.

Prophet [36, 35] is another run-time scheduling effort which targeted heterogeneous systems and included parallel applications with inter-processor communications. The Prophet approach is similar to our work in that it exploits application structure and system resource information to promote application performance. However, Prophet requires the target application be written in the Mentat programming language and the approach has only been tested in local-area environments. If possible, we would like to compare the performance of our strategy to those of Prophet, though it may be difficult to find a suitable scenario for comparison that satisfies the requirements of both strategies.

A number of projects [32, 15, 17, 34] have focused on meta-scheduling: the optimization of some metric averaged over the performance of multiple applications. Examples of such metrics include average slow down, wait time, and system throughput. Meta-scheduler designs often employ a three phase approach: (1) application-level schedulers are used to determine schedules for each application, and (2) the meta-scheduler evaluates the performance of multiple applications in the system, and then (3) the meta-scheduler modifies the application-specific schedules to improve overall system performance. Our application-level scheduling approach could be paired with such a meta-scheduling approach; we plan to do so for a meta-scheduler under development for the GrADS framework [34].

III. Scheduling

This section describes our scheduling approach. To provide context for this description, we first detail the scheduling scenario we address. A user has an application and wishes to execute 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 LANs and/or WANs. When the user is ready to run the application, the scheduler is contacted. The scheduler retrieves resource, application, and user information; searches for a desirable subset of available resources; calculates a mapping of data and/or tasks to those resources; and then returns the “best” such schedule. “Best” is defined by some performance metric; in this paper we assume that metric is the lowest estimated application execution time. Finally, the application is launched on the selected resources and is allowed to run to completion. Rescheduling will be added through additional efforts in the GrADS collaboration.

A. Architecture

Figure 1 presents the primary components of our scheduler and the interactions among those components. The goal of our design is to *decouple* the core algorithms required for the scheduling process (the search procedure) and the components and information required by those algorithms (all other components in Figure 1).

To initiate an application-run, the user submits a **machine list** containing the names of all machines available to him or her. For each machine in this machine list, the scheduler collects 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 III-C. Our approach requires an application-specific performance model and mapper, which we expect ultimately to obtain from application development tools such as the GrADS compiler. The **performance model** is an analytic metric for predicting application execution times on a given set of

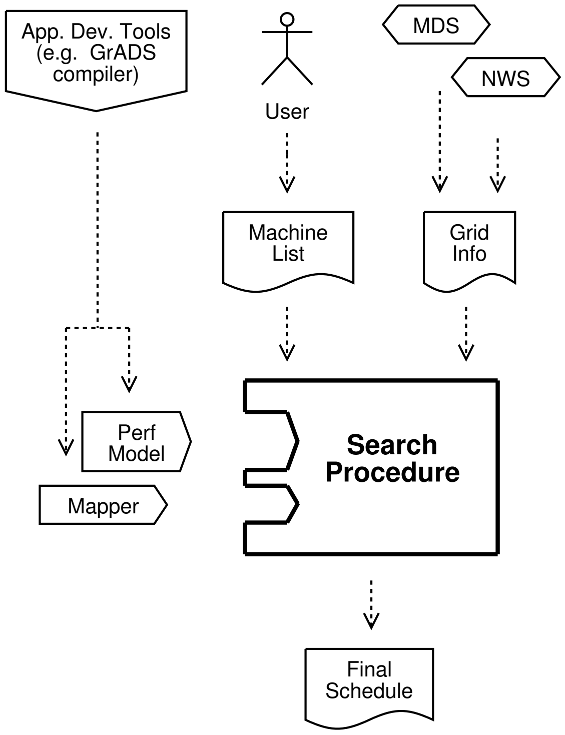

Fig. 1. Scheduler design.

resources with a given map. The **mapper** provides directives for mapping logical application data and/or tasks to physical resources; the goal of the mapper is to develop a map which minimizes application execution time. This mapping process may include workload balancing and / or arrangement of processors in a communication topology to reduce communication costs.

A **schedule** consists of an ordered list of machines and a mapping of data and / or tasks to those machines. The **search procedure** selects the "best" schedule and returns it as the **Final Schedule**. In the next section, we describe the schedule search process in detail.

B. Search procedure

To find good schedules, the search procedure first identifies groups of machines with both of the following qualities: (1) desirable individual machine characteristics and (2) desirable characteristics as an aggregate. For example, such groups would ideally be composed of computationally fast machines (an individual characteristic) and those machines would be connected by low-delay networks (an aggregate characteristic). We call such groups of machines **candidate machine groups (CMGs)**.

The most straightforward approach for the search for CMGs is an exhaustive search over all possible groups of machines. For a machine list with p machines, an exhaustive search will identify 2^p CMGs. For many testbed sizes of interest (dozens of machines), such a search will introduce unacceptable scheduling overheads and is therefore not a feasible solution. However, if the scheduler is going to provide reasonable application performance, it must identify CMGs that are reasonable for the application. Therefore, our search procedure uses 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 [28, 24] in future work. The **ComputeSiteCollections** method computes the power set of the set of sites (we exclude the null set). As an example, for the set of sites $\{A, B, C\}$, there are seven site collections: $\{A, B, C, A \cup B, A \cup C, B \cup C, A \cup B \cup C\}$. Once all 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 characteristic will have

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

 $sites \leftarrow FindSites(machList)$
 $siteCollections \leftarrow ComputeSiteCollections(sites)$
 for each $collection\ \ in\ \ siteCollections$
    for each $machineMetric\ \ in\ \ (computation, memory, dual)$
       for $targetSize \leftarrow 1$ to $size(collection)$
          $list \leftarrow SortDescending(collection, machineMetric)$
          $CMG \leftarrow GetFirstN(list, targetSize)$
          $currSched \leftarrow GenerateSchedule(CMG, Mapper, PerfModel)$
          if $currSched.predTime < bestSched.predTime$
            $bestSched \leftarrow currSched$
 return $(bestSched)$
```

Fig. 2. Schedule search procedure.

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. 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 **SortDescending** method sorts the input machine list *collection* by the machine characteristic *machineMetric* in descending order (the most desirable machines will be first). The **GetFirstN** method call simply returns the first *targetSize* machines from the sorted *list*.

A key aspect of our approach is that *no application-specific characteristics or components have been involved in the search procedure* to this point. We have simply generated a large number of CMGs that could be of interest to applications in general.

Next, to evaluate each CMG, the **GenerateSchedule** method (1) uses the *Mapper* to develop a data mapping for the input *CMG*, (2) uses the *Performance model* to predict the execution time for the given schedule (*predtime*), and (3) returns a schedule structure which contains the CMG, the map, and the predicted time. Finally, schedules are compared to find the schedule with the minimum predicted execution time; this schedule is returned as the *bestSched*.

Given a base machine list of size p with s distinct sites, the upper bound on the number of CMGs that must be evaluated by our search procedure is $3p2^s$; see [8] for details of this bound development. An exhaustive search requires evaluation of 2^p CMGs. As long as the number of sites is significantly smaller than the number of resources (universally true in production Computational Grids today), then our search procedure greatly reduces search space as compared to an exhaustive search.

C. Use of Grid information

Computational Grids are highly dynamic environments where compute and network resource availability varies and Grid information sources can be periodically unavailable. 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) [7] and the Network Weather Service (NWS) [37]. 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 utilizes 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; versions of the MDS which support secure publishing mechanisms have been released recently and we plan to experiment with obtaining the list of machines from the MDS. Local machine computational and memory capacity data are used to sort machines in our search procedure and will be needed as input to typical performance model and mapper implementations. Network bandwidth and latency data will similarly be required as input to typical performance model and mapper implementations.

Our use of static and dynamic Grid information is based upon the successes of previous application-level schedulers in using such information [9, 31, 6]. An important distinction between this work and many previous efforts 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 excludes those machines from the search process. In our experience, most application schedulers do not gracefully handle such situations, leading to frequent scheduler failures.

IV. Validation

In this section, we present validation results which investigate the following questions about our scheduler.

i. *Does our scheduling approach provide flexibility and ease of use?* As shown by a large base of software engineering research, these are difficult qualities to demonstrate; while we do not expect to prove such qualities, we hope to provide evidence that they are true. We incorporate a large variety of scheduling scenarios in our experiments including a span of applications, testbeds, resource information availabilities, and problem sizes. Each of these scenarios was easily handled by our scheduler with only minor modifications to a configuration file.

ii. *Does the scheduler provide reduced application execution times relative to user-directed scheduling approaches?* Our previous AppLeS efforts [5] have shown that special-purpose application-level schedulers consistently outperform user-directed approaches. We compare our scheduling strategy against a user-directed strategy to test whether this important characteristic holds for our decoupled approach as well.

A. Validation scenarios

In this section we describe the variety of scheduling scenarios in which we test our scheduling strategy.

Applications – As initial test cases for our scheduler design, we selected two iterative, mesh-based applications [14]: **Jacobi** [2] and **Game of Life** [10]. We plan to test more complex applications in future work; we chose these applications as our initial test cases because they are well-known and representative of many important iterative, data-parallel science and engineering codes. Additionally, these applications includes significant communication and synchronization and are therefore significant test cases for capabilities of our scheduling approach. We implemented each application as a SPMD-style computation using

C and MPI. For execution over the wide-area, we used MPICH-G [11]. Jacobi and the Game of Life are each dominated by an iterative application phase involving repeated application of a set of operations over a 2-dimensional array of data. We consider only square data sets and refer to problem size as N, the size of each dimension.

As described in Section III, our approach utilizes an application-specific performance model and mapper to assist the schedule search procedure. To test our approach, we hand-built a performance model and mapper for each test application. Conceptually similar models and mappers have been integrated within our previous AppLeS schedulers [9, 5]. Our **performance models** are procedural and use simple models of computation and communication to predict application iteration times. Our **mappers** search for a data map whereby the machines in the computation are as time-balanced as possible; in an ideal data map, no machine is ever idle during application execution. We frame work-allocation constraints as a constrained optimization problem and we use the freely available lp_solve package [23] to solve it. Our mappers also re-arrange machine ordering to limit wide-area communication costs. For details on these designs the interested reader is referred to [8].

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). At UTK and at UCSD the resources we targeted were on a single LAN. At UIUC, we targeted two distinct resource groups: the Opus cluster and the Major cluster. Figure 3 depicts a snapshot of our testbed and 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.

Experiments were performed on the full **three-site testbed** with all 24 machines available to the scheduler, and on a **one-site testbed** with only 6 UCSD machines available to the scheduler.

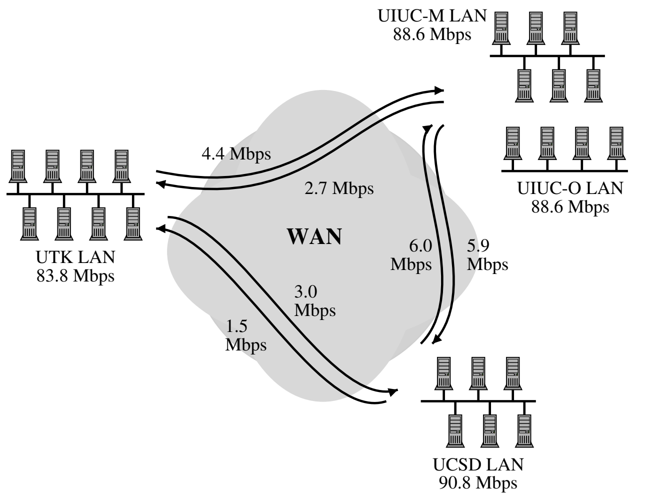

Fig. 3. A snapshot of testbed resources. Network links are labeled with available bandwidth in megabits per second; these values were collected on November 1, 2001 at 5:30 PM by Network Weather Service sensors.

Problem sizes – We ran experiments on the one-site testbed with **problem sizes** of N = {600, 1200, 2400, 4800, 7200, 9600} and on the three-site testbed with N = {600, 4800, 9600, 14400, 16800, 19200}. We selected these sizes to exercise the scheduler for a wide range of application behavior and resource requirements. For example, aggregate application memory requirements for these sizes range from 3 MB for N = 600 to 3375 MB for N = 19200.

Availability of resource information – We also tested the capacity of the scheduler to function with degraded resource information availability. Specifically, we perform experiments where dynamic resource information is available to the scheduler at run-time (e.g., NWS is "up") and where dynamic information is not available at run-time (e.g., NWS is "down").

B. Experimental procedure

The basis of our experiments is a comparison of the performance achieved by **three scheduling strategies**. **i.** The **dynamic** strategy uses our scheduler, scheduling decisions are made at run-time, and the scheduler uses both dynamic resource information from the NWS (CPU availability, free memory, and available bandwidth) and static information from the MDS (CPU speed).

	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.

ii. The **static** strategy also uses our scheduler, but scheduling decisions are made off-line and employ primarily static resource information.
iii. The **user** strategy is designed to emulate the scheduling process that a typical Grid user might employ. We assume that users will generally only invest time in scheduling once per application configuration; static resource information is therefore sufficient since scheduling occurs off-line. We also assume that users have 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 selects the minimum number of resources that will satisfy application memory requirements.

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

A **scheduling strategy comparison experiment** consists of back-to-back runs of the three schedulers. In each application execution, 104 iterations were performed; an average and standard deviation of the iteration times was then taken of all but the first 4 "warm up" 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.

We selected experiments to run with the goals of obtaining (1) a broad survey of the performance of the scheduling strategies in many different scenarios and (2) statistically significant results. We ran scheduling strategy comparison experiments for all combinations of the two applications, the two testbeds, and six problem sizes (for a total of 2*2*6 = 24 testing scenarios). We performed 10 repetitions of each testing scenario for a total of 240 comparison experiments involving

720 scheduling strategy tests.

Strategy comparisons. We use a common comparison metric for comparisons of scheduling strategies, *percent degradation from best* [20]. For each experiment we find the lowest average iteration time achieved by any of the strategies, $itTime_{best}$, and compute

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

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.

C. Aggregate results

Figure 4 presents the 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 other strategies. This result verifies that our scheduler does provide consistently improved application performance relative to a user-directed strategy, thus answering *question ii* in the affirmative. In addition to good performance on average, our approach performed well across the variety of scenarios tested in these results. For example, notice the *Avg* and *Std* lines in Table 2; the dynamic strategy shows a low standard deviation in performance for all four application-testbed combinations. In three of the application-testbed scenarios, the user strategy is never the best strategy (i.e. the *Min* statistic is greater than 0). These results provide evidence for answering *question i*: our approach seems flexible and applicable to a range of scenarios. These experiments also demonstrate that our strategy benefits greatly from the availability of dynamic resource information (i.e. the dynamic strategy consistently outperforms the static strategy), but, when such information is not available, our approach still provides a performance advantage relative to a user-directed approach (i.e. the static strategy consistently outperforms the user strategy).

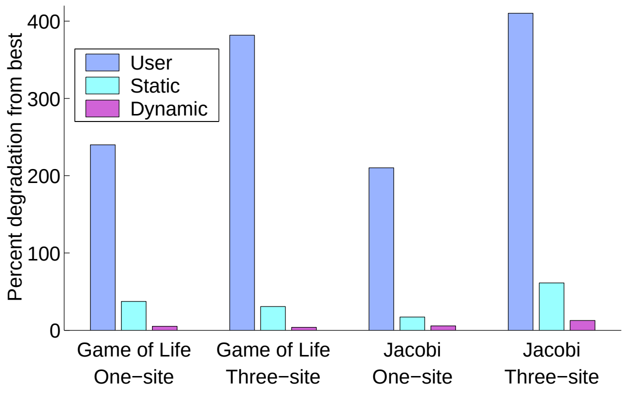

Fig. 4. Average percent degradation from best for each scheduling strategy and each application-testbed combination.

D. Case study

We detail experimental results for the Jacobi application on the three-site testbed with a problem size of 16800. We performed 10 repetitions of this experiment over a period from 10/16/2001 - 11/10/2001. Specifically, we collected reps 1-3 on October 16-17, reps 4-6 on November 6-7, and reps 7-10 on November 9-10.

Figure 5 presents average iteration times for each scheduling strategy for each of the 10 repetitions; errorbars represent the standard deviation of iteration times. No times are reported for the user strategy in the ninth repetition because the application failed to complete in this run. The dynamic strategy yields more consistent iteration times (smaller errorbars) and leads to more consistent performance across experiment repetitions (between 2 and 3 seconds in each of the 10 repetitions).

Figure 6 shows the number of processors selected from each site for each of the scheduling strategies. Since the user and static strategies are run only once per scheduling scenario, only one resource group is displayed. For the dynamic strategy, processor selection occurs at run-time so we display each of the 10 resource groups selected. We did not have space here to display the exact processor selections, but the static and dynamic strategies do differentiate among

	Game of Life - 1 site			Game of Life - 3 site			Jacobi - 1 site			Jacobi - 3 site		
	User	Stat	Dyn	User	Stat	Dyn	User	Stat	Dyn	User	Stat	Dyn
Avg	240.0	37.3	5.1	381.9	30.8	3.8	210.3	17.2	5.7	410.3	61.3	12.7
Std	152.0	40.4	12.9	466.6	63.3	10.7	130.6	28.2	12.6	212.7	145.8	40.6
Min	7.7	0	0	45.3	0	0	16.4	0	0	0	0	0
Max	507.7	156.9	69.3	2748.0	421.8	68.5	466.4	90.5	69.7	862.9	739.2	215.1

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

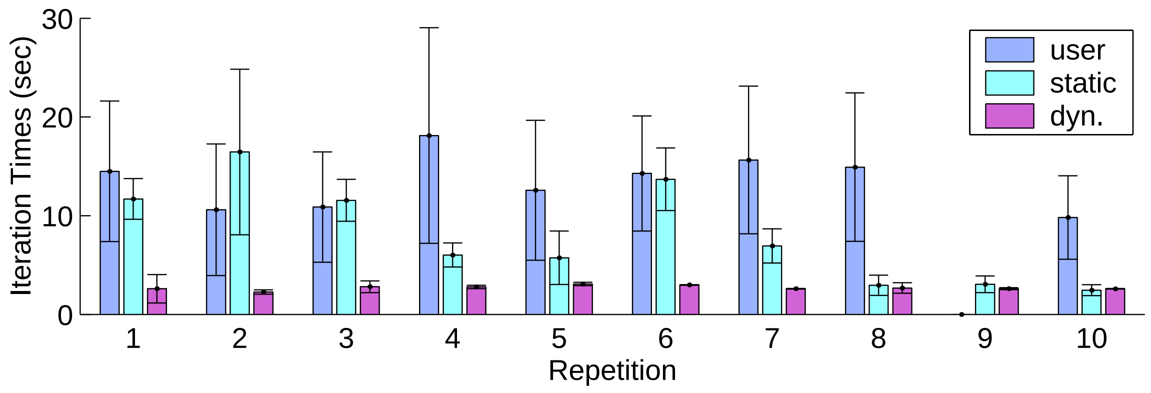

Fig. 5. Average and standard deviation in iteration times for the Jacobi application on the three-site testbed, problem size of 16800.

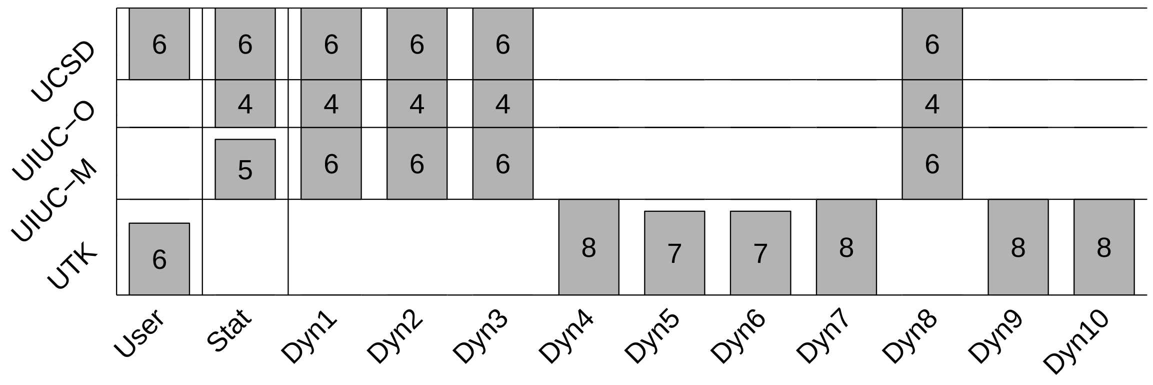

Fig. 6. Processor selections for each scheduling strategy for the Jacobi application on the three-site testbed, problem size of 16800.

resources within each site based on computational and memory capacities.

For the **user** strategy we see that processor selection followed the preference ordering of {UCSD, UTK, UIUC} and that, given this list of resources, 12 processors were required to fulfill application memory requirements for this problem size. Note that the number of resources required will vary with the memory capacities of the resources. The **static** strategy selected 15 processors, taken from the UCSD and UIUC resources. Given that UTK offers the most powerful individual resources (refer to Table 1), it may seem surprising that UTK resources were not included by this strategy. However, the Jacobi application involves significant communication costs and the bandwidth between UTK and either of the other sites tends to be significantly lower than that between UCSD and UIUC. Given these conditions, the static strategy avoided these high-delay links and determined that the most performance efficient resource set included resources from UCSD and UIUC. For the **dynamic** strategy, resource selection decisions are made at run-time and involve dynamic bandwidth, CPU, and memory availability estimates. In repetitions 1, 2, 3, and 8 this strategy predicted that a UCSD+UIUC resource set would prove most performance efficient, in agreement with the choice of the static strategy. In the other repetitions, the dynamic strategy predicted that the smaller, more tightly coupled resource group of only UTK resources would be more performance efficient. In all cases the strategy avoided schedules spanning the UTK-UIUC and UTK-UCSD links. Although the dynamic strategy utilized a varying resource group across the 10 repetitions, it was the strategy with the most consistent iteration times. Run-time scheduling allowed this strategy to avoid loaded networks and systems, thus providing the user of the scheduler with more consistent performance.

E. Scheduling overheads

The performance metrics we reported in this paper do not incorporate the cost of the scheduling process itself. The scheduling process is dominated by two phases: resource information retrieval and the schedule search process itself. In each test, we measured the time required for collection of resource information and the time required for the search procedure itself. We performed 10 repetitions of each test and calculated an average. All tests were performed with the full 24-machine three-site testbed; we expect that overhead times will be higher for larger sets of machines and lower for smaller sets.

All tests were performed from a 450 MHz Pentium III shared workstation located in San Diego, California. Since our approach can be used in a variety of scenarios, we measured scheduling overheads for several different information service configurations. We utilized the following servers for these configurations.

i. A remote MDS server - the GrADS MDS server located in Los Angeles, California.
ii. A remote NWS server - the GrADS NWS server, which was located in Knoxville, Tennessee during these experiments.
iii. A local MDS cache - a file-based cache of MDS data located on the experiment machine in San Diego.
iv. A local NWS server - an NWS server located on the experiment machine in San Diego.

For all experiments reported in earlier sections of this paper, we utilized an NWS nameserver and an MDS cache which were local to the scheduler. In these conditions, collection of resource information required an average of 2.0 seconds and the search procedure required an average of 2.5 seconds. In total, a scheduling time of 4.5 seconds is reasonable relative to execution times for applications of even moderate problem size. When the same experiments were performed with a remote NWS nameserver, resource information retrieval required an average of 59.6 seconds (62.4 seconds overall for scheduling). When both a remote NWS nameserver and a remote MDS server were used, information retrieval required an average of 1087.5 seconds (1088.4 seconds overall for scheduling). We assume that a Grid user will probably be willing to wait 60 seconds for scheduling, but will probably not be willing to wait 1000 seconds. These results indicate that, given the technologies available at the time of these experiments, our scheduling approach may be used with either a local or a remote NWS server, but is only feasible when used with a local MDS cache. Newer versions of the NWS and MDS have been released since these experiments were conducted; reports indicate the new versions are more efficient than those reported on here. We plan to repeat these experiments with the newer technologies.

V. Discussion

A. Scope of the work

Our scheduling approach can be expected to provide the largest performance advantage under the following conditions.
i. Dynamic and/or static resource information is available for the target testbed. While we attempt to function despite poor resource information, our approach will be most effective when resource information is available.
ii. An application performance model and mapping strategies are already available or can be easily created. These components need not be sophisticated or precise, but the advantage provided by the scheduler will vary with the quality of application information available to it.
iii. We have focused on shared Computational Grid environments that do not include dedicated machines or batch schedulers (e.g. the GrADS testbed). Hence, in this work we do not focus on issues such a co-scheduling and advanced reservation.
iv. Our strategy is most effective for applications that have ”moderate” computation to communication ratios (e.g. loosely synchronous applications). By comparison, synchronous applications perform well only on clusters or supercomputers with high-performance networks, and embarrassingly parallel applications can be easily scheduled via other schedulers [1, 6, 27].

B. Future Work

We plan to extend our validation to other applications from a range of application classes. As described in Section II, work has begun on a GrADS meta-scheduler [34]; we plan to investigate ways in which application-level and system-level schedulers can coordinate to balance application and system performance. As part of this effort, we also plan to test our approach for performance metrics other than execution time and extend our approach, if necessary. Finally, other members of the GrADS research community are investigating the feasibility of compiler generation of application information and performance models [18], as well as the inclusion of such models in Grid-enabled libraries [18, 25]. As this work matures, we expect to use such models for application scheduling.

C. Conclusions

In this paper we presented a decoupled scheduling approach for parallel applications in Computational Grid environments. Our approach explicitly decouples a general schedule search procedure from components and information specific to the application and execution environment. We tested our software prototype for a wide range of scheduling scenarios on real-world Computational Grids. These experiments showed that

i. our approach reduces application execution times as compared to user-directed scheduling approaches; and
ii. our scheduler successfully exploits dynamic resource information when it is available, and can gracefully tolerate lack of such information.

In recent collaborative work with GrADS researchers at Rice and UTK, we have developed APIs for component interactions within the GrADS system, including interactions with the scheduler described herein. We have also integrated our software prototype into the main GrADS software base [19]. In conjunction with this effort, a researcher from UTK applied our scheduler prototype to the ScaLAPACK application in less than 2 days, and found that the scheduler worked well for that application [33]; as our prototype improves, we expect this time to decrease. In our previous AppLeS work, schedulers were fundamentally tied to the application itself, and therefore could not be re-used for other applications without significant adaptation time.

These results support and extend the results of previous application-level scheduling efforts within the AppLeS project [5]. In the context of application-level scheduling in general, our results are particularly notable because they were not obtained with a special-purpose, application-specific scheduler. Instead, these results were obtained with a decoupled scheduler design that is easy to apply in a variety of scheduling scenarios.

VI. 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 thank Otto Sievert and Renata Teixeira for their insightful comments on previous versions of this manuscript. Finally, we thank the reviewers whose suggestions have greatly improved the final paper.

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] 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.

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

[4] 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.

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

[6] 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).

[7] 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).

[8] 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.

[9] 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).

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

[11] 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.

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

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

[14] 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`.

[15] Gehring, J., and Preiss, T. Scheduling a metacomputer with uncooperative sub-schedulers. In *Proceedings of JSSPP'99* (1999), pp. 179–201.

[16] 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.

[17] Hamscher, V., Schwiegelshohn, U., Striet, A., and Yahyapour, R. Evaluation of job-scheduling strategies for grid computing. In *Proceedings of Grid'00* (2000), pp. 191–202.

[18] 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.

[19] 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.

[20] 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.

[21] 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).

[22] LIU, C., YANG, L., FOSTER, I., AND ANGULO, D. Design and evaluation of a resource selection framework for Grid applications. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing* (July 2002). To appear.

[23] lp_solve FTP site at `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

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

[25] 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).

[26] 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.

[27] 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).

[28] 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).

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

[30] 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).

[31] 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.

[32] SUBRAMANI, V., KETTIMUTHU, R., SRINIVASAN, S., AND SADAYAPPAN, P. Distributed job scheduling on computational grids using multiple simultaneous requests. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing* (July 2002). To appear.

[33] VADHIYAR, S., May 2002. Personal Communication.

[34] VADHIYAR, S. S., AND DONGARRA, J. J. A metascheduler for the Grid. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing* (July 2002). To appear.

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

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

[37] 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).