Chapter 1

SCHEDULING IN THE GRID APPLICATION DEVELOPMENT SOFTWARE PROJECT *

Holly Dail[1], Otto Sievert[2], Fran Berman[1,2], Henri Casanova[1,2]
[1] *San Diego Supercomputer Center, University of California, San Diego*
[2] *Department of Computer Science and Engineering, University of California, San Diego*

Asim YarKhan, Sathish Vadhiyar, Jack Dongarra
Department of Computer Science, University of Tennessee

Chuang Liu[3], Lingyun Yang[3], Dave Angulo[3], Ian Foster[3,4]
[3] *Department of Computer Science, The University of Chicago*
[4] *Mathematics and Computer Science Division, Argonne National Laboratory*

Abstract

Developing Grid applications is a challenging endeavor, which at the moment requires both extensive labor and expertise. The Grid Application Development Software Project (GrADS) provides a system to simplify Grid application development. This system incorporates tools at all stages of the application development and execution cycle. In this chapter we focus on application scheduling, and present the three scheduling approaches developed in GrADS: development of an initial application schedule (launch-time scheduling), modification of the execution platform during execution (rescheduling), and negotiation between multiple applications in the system (metascheduling). These approaches have been developed and evaluated for platforms that consist of distributed networks of shared workstations, and applied to real-world parallel applications.

Keywords: scheduling rescheduling metascheduling Grid data-parallel

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

1. INTRODUCTION

Computational Grids can only succeed as attractive everyday computing platforms if users can run applications in a straightforward manner. In these collections of disparate and dispersed resources, the sum (the Grid) must be greater than the parts (the machines and networks). Since Grids are typically built on top of existing resources in an ad-hoc manner, software layers between the user and the operating system provide useful abstraction and aggregation.

In the past decade, many useful software products have been developed that simplify usage of the Grid; these are usually categorized as "middleware". Successful middleware efforts have included, for example, tools for collecting and disseminating information about Grid resources (e.g. Network Weather Service [36]), all-encompassing meta operating systems that provide a unified view of the Grid (e.g. Legion [17]), and collections of relatively independent Grid tools (e.g. the Globus Toolkit [16]). Middleware has simplified Grid access and usage, allowing the Grid to be useful to a wider range of scientists and engineers.

In order to build application-generic services, middleware developers have generally not incorporated application-specific considerations in decisions. Instead, application developers are expected to make any decisions that require application knowledge; examples include data staging, scheduling, launching the application, and monitoring application progress for failures. In this environment Grid application development remains a time-consuming process requiring significant expertise.

The Grid Application Development Software Project (GrADS) [9, 20] is developing an ambitious alternative: replace the discrete, user-controlled stages of preparing and executing a Grid application with an end-to-end software-controlled process. The project seeks to provide tools that enable the user to focus only on high-level application design without sacrificing application performance; this is achieved by incorporating application characteristics and requirements in decisions about the application's Grid execution. The GrADS architecture incorporates user problem solving environments, Grid compilers, schedulers, performance contracts, performance monitors, and reschedulers into a seamless tool for application development. GrADS builds on existing middleware and tools to provide a higher-level of service oriented to the needs of the application.

Scheduling, the matching of application requirements and available resources, is a key aspect of GrADS. This chapter focuses on scheduling needs identified for the GrADS framework. Specifically, we discuss the following distinct scheduling phases.

- **Launch-time scheduling** is the pre-execution determination of an initial matching of application requirements and available resources.

- **Rescheduling** involves making modifications to that initial matching in response to dynamic system or application changes.

- **Meta-scheduling** involves the coordination of schedules for multiple applications running on the same Grid at once.

Initial prototype development in GrADS revealed that existing application scheduling solutions were inadequate for direct application in the GrADS system. System-level schedulers (e.g. LoadLeveler [18] and Maui Scheduler [3]) focus on throughput and generally do not consider application requirements in scheduling decisions. Application-specific schedulers [8] have been very successful for individual applications, but are not easily applied to new applications. Other projects have focused on easy-to-characterize classes of applications, particularly master-worker applications [4, 11]; these solutions are often difficult to use with new classes of applications.

Consequently, we worked to develop scheduling strategies appropriate to the needs of GrADS. We began by studying specific applications or application classes and developing scheduling approaches appropriate to those applications or application classes [27, 6, 13, 31, 34, 32]. Concurrently, the GrADS team has been developing GrADSoft, a shared implementation of GrADS. Proven research results in all areas of the GrADS project are incorporated in GrADSoft with appropriate modifications to allow integration with other components. The resulting system provides an end-to-end validation of the GrADS research efforts. Many of the scheduling approaches discussed in this chapter have been incorporated in GrADSoft.

This chapter is organized as follows. Section 2 describes the GrADS project in greater detail and provides an overview of key Grid technologies we draw upon for our scheduling work. Section 3 describes the applications we have used to validate our designs. Section 4 presents our launch-time scheduling approach, Section 5 presents our rescheduling approach, and Section 6 presents our metascheduling approach. Section 7 concludes the chapter.

2. GRADS

Figure 1.1 provides a high-level view of the GrADS system architecture [20]. This architecture provides the framework for our scheduling approaches. The following is a conceptual view of the function of each

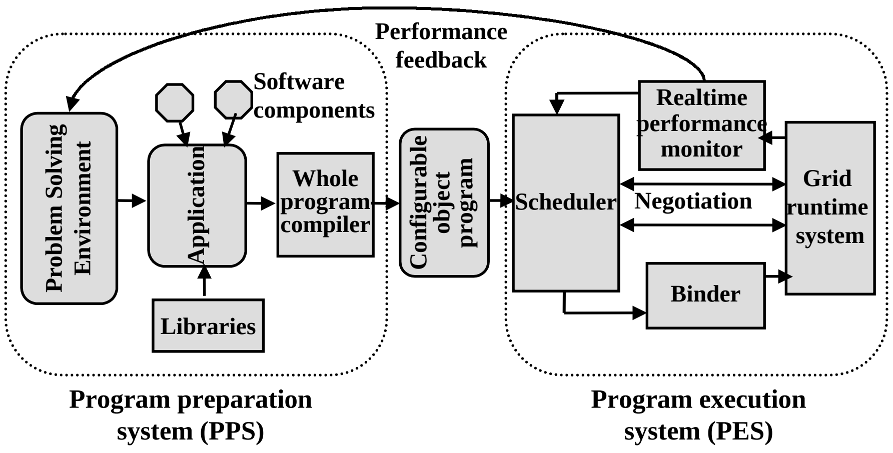
Performance feedback
Software components
Problem Solving Environment
Application
Whole program compiler
Libraries
Configurable object program
Scheduler
Realtime performance monitor
Negotiation
Grid runtime system
Binder
Program preparation system (PPS)
Program execution system (PES)

Figure 1.1. Grid Application Development Software Architecture.

component in the architecture. Note that the view given in Figure 1.1 and in the following discussion is an idealized view of how GrADS components should function and interact with one another. In reality, the system is under development and does not yet function exactly as described. This chapter will describe the current state of the scheduling components, but we will not have space to describe the current status of all GrADS components.

The *Program Preparation System* (PPS) handles application development, composition, and compilation. To begin the development process, the user interacts with a high-level interface called a problem solving environment (PSE) to assemble their Grid application source code and integrate it with software libraries. The resulting application is passed to a specialized GrADS compiler. The compiler performs application analysis and partial compilation into intermediate representation code and generates a configurable object program (COP). This work may be performed off-line, as the COP is a long-lived object that may be re-used for multiple runs of the application. The COP must encapsulate all results of the PPS phase for later usage; for example, application performance models and partially compiled code called intermediate representation code.

The *Program Execution System* (PES) provides on-line resource discovery, scheduling, binding, application performance monitoring, and rescheduling. To execute an application, the user submits parameters of the problem such as problem size to the GrADS system. The application COP is retrieved and the PES is invoked. At this stage the scheduler interacts with the Grid run-time system to determine which resources are available and what performance can be expected of those resources. The scheduler then uses application requirements specified in the COP to select an application-appropriate schedule (resource subset and a mapping of the problem data or tasks onto those resources).

The binder is then invoked to perform a final, resource-specific compilation of the intermediate representation code. Next, the executable is launched on the selected Grid resources and a real-time performance monitor is used to track program performance and detect violation of performance guarantees. Performance guarantees are formalized in a performance contract [35]. In the case of a performance contract violation, the rescheduler is invoked to evaluate alternative schedules.

2.1 Grid Technology

The previous section outlined the GrADS vision; as components of that vision are prototyped and validated, they are incorporated in a shared prototype implementation of GrADS called GrADSoft. Many of the scheduling approaches described in this chapter have been integrated into this prototype. We utilize the following Grid technologies to support our research efforts.

The *Monitoring and Discovery Service* (MDS) [12] and the *Network Weather Service* (NWS) [36] are two on-line services that collect and store Grid information that may be of interest to tools such as schedulers. 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, and number of CPUs available. The *NWS* is a distributed monitoring system designed to track and forecast dynamic resource conditions; Chapter **??** describes this system in detail. 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 approaches draw on both the MDS and NWS to discover properties of the Grid environment.

The *ClassAds / Matchmaking* approach to scheduling [28] was pioneered in the Condor high-throughput computing system [22]. Refer to Chapter **??** for a discussion of Condor and Chapter **??** for a discussion of ClassAds. ClassAds (i.e., *Classified Advertisements*) are records specified in text files that allow resource owners to describe the resources they own and resource consumers to describe the resources they need. The ClassAds/Matchmaking formalism includes three parts: The *ClassAds language* defines the syntax for participants to create ClassAds. The *advertising protocol* defines basic conventions regarding how ClassAds and matches must be communicated among participants. The *matchmaker* finds matching requests and resource descriptions, and notifies the advertiser of the result. Unfortunately, at the time of our research efforts the ClassAds/Matchmaker framework only considered a single resource request at a time. As will be described in Section 4.3.1, our work has extended the ClassAds/Matchmaker framework to allow scheduling of parallel applications on multiple resources.

To provide focus and congruency among different GrADS research efforts we restricted our initial efforts to particular execution scenarios. Specifically, we focus on applications parallelized in the Message Passing Interface (MPI) [15]. This choice was based on both (1) the popularity and ease of use of MPI for the implementation of parallel scientific applications and (2) the recent availability of MPICH-G2 [19] for execution of MPI applications in the wide-area. We use the Globus Toolkit [16] for authentication and job launching and GridFTP [5] for data transfers.

3. FOCUS APPLICATIONS

The GrADS vision to build a general Grid application development system is an ambitious one; to provide context for our initial research efforts, we selected a number of real applications as initial focus points. Our three primary selections were ScaLAPACK, Cactus, and FASTA; each of these applications (1) is of significant interest to a scientific research community and (2) has non-trivial characteristics from a scheduling perspective. Additionally, we selected three iterative applications (Jacobi, Game of Life, and Fish) for use in rapid development and testing of new scheduling techniques. These applications were incorporated into GrADS by the following efforts: ScaLAPACK [27], Cactus [6], Jacobi [13], Game of Life [13], and Fish [31]. The FASTA effort has not been described in a publication, but was led by Asim YarKhan and Jack Dongarra. Our focus applications include implementations in Fortran, C, and C++.

3.1 ScaLAPACK

ScaLAPACK [10] is a popular software package for parallel linear algebra, including the solution of linear systems based on LU and QR factorizations and the determination of eigenvalues. It is written in a single program multiple data (SPMD) style and is portable to any computer that supports MPI or PVM. Linear solvers such as those provided in ScaLAPACK are ubiquitous components of scientific applications across diverse disciplines.

As a focus application for GrADS, we chose the ScaLAPACK right-looking LU factorization code based on 1-D block cyclic data distribution; the application is implemented in Fortran with a C wrapper. Performance prediction for this code is challenging because of data-dependent and iteration-dependent computational requirements.

3.2 Cactus

Cactus [7] was originally developed as a framework for finding numerical solutions to Einstein's equations and has since evolved into a general-purpose, open source, problem solving environment that provides a unified, modular, and parallel computational framework for scientists and engineers. The name Cactus comes from the design of central core (or "flesh") which connects to application modules (or "thorns") through an extensible interface. Thorns can implement custom applications, as well as computational capabilities (e.g. parallel I/O or checkpointing). See Chapter **??** for an extended discussion of Cactus.

We focus on a Cactus code for the simulation of the 3D scalar field produced by two orbiting sources; this application is implemented in C. The solution is found by finite differencing a hyperbolic partial differential equation for the scalar field. This application decomposes the 3D scalar field over processors and places an overlap region on each processor. In each iteration, each processor updates its local grid points and then shares boundary values with neighbors. Scheduling for Cactus presents challenging workload decomposition issues to accommodate heterogeneous networks.

3.3 FASTA

In bio-informatics, the search for similarity between protein or nucleic acid sequences is a basic and important operation. The most exacting search methods are based on dynamic programming techniques and tend to be very computationally expensive. FASTA [26] is a sequence alignment technique that uses heuristics for fast searches. Despite these optimizations, due to the current size of the sequence databases and the rate at which they are growing (e.g. human genome database), searching remains a time consuming process. Given the size of the sequence databases, it is often undesirable to transport and replicate all databases at all compute sites in a distributed Grid.

The GrADS project has adapted a FASTA sequence alignment implementation [1] to use remote, distributed databases that are partially replicated on some of the Grid nodes. When databases are located at more than one worker, workers may be assigned only a portion of a database. The application is structured as a master-worker and is implemented in C. FASTA provides an interesting scheduling challenge due to the database locality requirements and large computational requirements.

3.4 Iterative Applications

We selected three simple iterative applications to support rapid development and testing of new scheduling approaches. We chose these applications because (i) they are representative of many important science and engineering codes and (ii) they are significant test cases for a scheduler because they include various and significant computation, communication, and memory usages. All three applications are implemented in C and support non-uniform data distribution.

The *Jacobi method* is a simple linear system solver. A portion of the unknown vector x is assigned to each processor; during each iteration, every processor computes new results for its portion of x and then broadcasts its updated portion of x to every other processor.

Conway's *Game of Life* is a well-known binary cellular automaton [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 and then processors send data from their edges (ghost cells) to their neighbors in the mesh.

The *Fish* application models the behavior and interactions of fish and is indicative of many particle physics applications. calculates Van der Waals forces between particles in a two-dimensional field. Each computing process is responsible for a number of particles, which move about the field. Because the amount of computation depends on the location and proximity of particles, this application exhibits a dynamic amount of work per processor.

4. LAUNCH-TIME SCHEDULING

The launch-time scheduler is called just before application launch to determine how the current application execution should be mapped to available Grid resources. The resulting *schedule* specifies the list of target machines, the mapping of virtual application processes to those machines, and the mapping of application data to processes. Launch-time scheduling is a central component in GrADSoft and is key to achieving

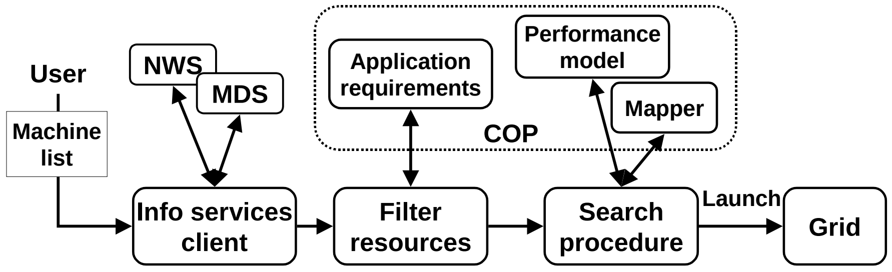

Figure 1.2. GrADS launch-time scheduling architecture.

application performance; for this reason, we have experimented with a variety of launch-time scheduling approaches including those described in [24, 13, 27]. Through this evolution we have identified an application-generic launch-time scheduling architecture; in this section we describe that architecture and key contributions we have made to each component of the design. Many of the approaches described here have been incorporated in GrADSoft; the resulting unified system has been tested with all of the applications described in Section 3.

4.1 Architecture

Figure 1.2 shows our launch-time scheduling architecture. This architecture is designed to work in concert with many other GrADS components as part of the general GrADS architecture described in Section 2. Launch-time scheduling proceeds as follows; we describe each component in detail in the next section. A user submits a list of machines to be considered for this application execution (the machine list). The list is passed to the GrADS information services client; this component contacts the MDS and the NWS to retrieve information about Grid resources. The resulting Grid information is passed to the filter resources component. This component retrieves the application requirements and filters out resources that can not be used for the application. The remaining Grid information is then passed to the search procedure. This procedure searches through available resources to find application-appropriate subsets; this process is aided by the application performance model and mapper. The search procedure eventually selects a final schedule, which is then used to launch the application on the selected Grid resources.

4.2 GrADS Information Services Client

To enable adaptation to the dynamic behavior of Grid environments, Grid resource information must be collected in real time and automatically incorporated in scheduling decisions. The MDS and NWS, described in Section 2.1, collect and store a variety of Grid resource information. The GrADS information services client retrieves resource information of interest to the scheduler and stores that information in a useful format for the scheduler.

4.3 Configurable Object Program

As described in Section 2, the output of the GrADS program preparation system is a configurable object program, or COP. The scheduler obtains the application requirements specification, the performance model, and the mapper from the COP. As the GrADS program preparation system matures, the COP will be automatically generated via specialized compilers. In the meantime, we used various approaches, described below, to develop hand-built COP components for our focus applications. These hand-built components have been useful in studying scheduling techniques and have provided guidance as to what sorts of models and information should be automatically generated by the PPS in the future.

4.3.1 Application requirements specification. A reasonable schedule must satisfy application resource requirements. For example, schedules typically must include sufficient local memory to store the entire local data set. See Chapter **??** for an additional discussion of specifications of application resources requirements.

We have explored two ways to specify application resource requirements. The first is a primarily internal software object called an abstract application resource and topology (AART) model [20]. Ultimately, the GrADS compiler will automatically generate the AART; however, we do not yet have an automated GrADS compiler in GrADSoft. Currently, application developers must provide a specification of the application requirements before utilizing GrADSoft for a new application. For this purpose, we have developed an alternative approach based on ClassAds (see Section 2.1). With ClassAds, resource requirements can be easily specified in a human-readable text file. Furthermore, some developers are already familiar with ClassAds semantics.

The original ClassAds language is not sufficient for parallel applications. To support parallel application description, we developed a number of extensions to ClassAds.

```
[
   type = “Set”;
   service = “SynService”;
   iter = “100”; α = 100; x = 100; y = 100; z = 100;
   domains = {“cs.utk.edu”, “ucsd.edu”};
   requirements = Sum(other.MemorySize) ≥
            (1.757 + 0.0000138 × x × y × z) &&
            suffix(other.machine, domains);
   computetime = x × y × α/other.cpuspeed × 370;
   comtime = other.RLatency + x × y × 254/other.Lbandwidth;
   exectime = (computetime + comtime) × iter + startup;
   mapper = [type = “dll”; libraryname = “cactus”;
            function = “mapper”];
   rank = Min(1/exectime);
]
```

Figure 1.3. Example of a set-extended ClassAds resource requirements specification.

- A *Type* specifier is supplied for identifying set-extended ClassAds: the expression *Type=“Set”* identifies a set-extended ClassAd.
- Three aggregation functions, *Max*, *Min*, and *Sum*, are provided to specify aggregate properties of resource sets. For example, $Sum(other.memory) > 5Gb$ specifies that the total memory of the set of resources selected should be greater than 5Gb.
- A boolean function *suffix(V, L)* returns *True* if a member of list *L* is the suffix of scalar value *V*.
- A function *SetSize* returns the number of elements within the current resource ClassAd set.

Our extensions, called *Set-extended ClassAds*, are described in full in [24].

Our usage of ClassAds in GrADSoft is illustrated by the resource request for the Cactus application shown in Figure 1.3; the parameters of this resource request are based on performance models developed for Cactus in [30]. In this example, we use $Sum(other.MemorySize) >= (1.751 + 0.0000138 * z * x * y)$ to express an aggregate memory requirement, and $suffix(other.machine, domains)$ to constrain the resources considered to those within domain *cs.utk.edu* or *ucsd.edu*. The execution progress of the Cactus application is generally constrained by the subtask on the slowest processor; we therefore specify $rank = Min(1/exectime)$

so that the rank of a resource set is decided by the longest subtask execution time. The *exectime*, *computetime*, and *comtime* entries define the performance model for Cactus.

To better express the requirements of a variety of Grid applications, we recently developed a new description language, called *Redline* [23], based on Set-extended ClassAds. This language is able to describe resource requirements of SPMD applications, such as Cactus, as well as MIMD applications where different application components may require different types of resources. For example, with Redline one can specify unique requirements for a storage resource, computation resources, and for the network connecting storage and computation resources. We use this added expressiveness to better satisfy application requirements in the scheduling process.

4.3.2 Performance modeling. The performance model is used by the launch-time scheduler to predict some metric of application performance for a given schedule. For ease of discussion we assume the metric of interest is application execution time, but one could easily consider, for example, turnaround time or throughput instead. The performance model object takes as input a proposed schedule and the relevant Grid information and returns the predicted execution time for that schedule.

Other GrADS researchers are developing approaches for automatically generating performance models in the GrADS compiler. In lieu of such technologies, we have experimented with several methods for developing and specifying performance models in GrADS. We generally used the following approach: (i) develop an analytic model for well-understood aspects of application or system performance (e.g. computation time prediction is relatively straightforward), (ii) test the analytic model against achieved application performance in real-world experiments, and (iii) develop empirical models for poorly-understood aspects of application or system behavior (e.g. middleware overheads for authentication and startup may be difficult to predict). We used this approach for the iterative focus applications [13], Cactus [30], and FASTA [26].

For the ScaLAPACK application, we pursued a different approach [27]. The ScaLAPACK LU factorization application is an iterative application consisting of three important phases: the factorization phase involves computation, the broadcast phase involves communication, and the update phase consists of both computation and communication. As the application progresses, the amount of computation and communication in each phase and in the iteration as a whole varies. Analytical models based on simple mathematical formulas can not provide accurate predictions of the cost of the ScaLAPACK application on heterogeneous systems. Instead, we developed a simulation model of the application that simulates the different phases of the application for a given set of resources. The resulting application simulation functions as an effective performance model.

We implemented most of our performance models as shared libraries; GrADSoft dynamically loads the application-appropriate model as needed. This approach allows application-specific performance model code to be used by our application-generic software infrastructure. Using ClassAds, GrADSoft also supports an easier to build alternative: the performance model can be specified as an equation within the ClassAd (see Figure 1.3). While some performance dependencies can not be described in this ClassAds format, we believe this approach will be very attractive to users due to its ease-of-use.

4.4 Mapper

The mapper is used to find an appropriate mapping of application data and/or tasks to a particular set of resources. Specifically, the mapper takes as input a proposed list of machines and relevant Grid information and outputs a mapping of virtual application processes to processors (ordering) and a mapping of application data to processes (data allocation).

Some applications allow the user to specify machine ordering and data allocation; examples include FASTA, Cactus, and our three iterative applications. For these applications, a well-designed mapper can leverage this flexibility to improve application performance. Other applications define ordering and data allocation internally, such as ScaLAPACK. For these applications, a mapper is unnecessary.

4.4.1 Equal allocation. The most straightforward mapping strategy takes the target machine list and allocates data evenly to those machines without reordering. This is the default GrADSoft mapper and can easily be applied to new applications for which a more sophisticated strategy has not been developed.

4.4.2 Time balancing. The Game of Life and Jacobi are representative of applications whose overall performance is constrained by the performance of the slowest processor; this behavior is common in many applications with synchronous inter-processor communication. In [13] we describe our time balance mapper design for the Game of Life and Jacobi in detail; we briefly describe this design here.

The goal of the ordering process is to reduce communication costs. We reorder machines such that machines from the same site are adjacent in the topology. Since the Game of Life is decomposed in strips and involves neighbor-based communication, this ordering minimizes wide-area communications and reduces communication costs. For Jacobi there is no clear effect since its communication pattern is all-to-all.

The goal of the data allocation process is to properly load the machines such that all machines complete their work in each iteration at the same time; this minimizes idle processor time and improves application performance. To achieve this goal we frame application memory requirements and execution time balance considerations as a series of constraints; these constraints form a constrained optimization problem that we solve with a linear programming solver [2]. To use a real-valued solver we had to approximate work allocation as a real-valued problem. Given that maintaining a low scheduling overhead is key, we believe that the improved efficiency obtained with a real-valued solution method justifies the error incurred in making such an approximation.

4.4.3 Data locality. FASTA is representative of applications with data locality constraints: reference sequence databases are large enough that it may be more desirable to co-locate computation with the existing databases than to move the databases from machine to machine. The mapper must therefore (i) ensure that all reference databases are searched completely and (ii) balance load on machines to reduce execution time.

As with the time balance mapper, a linear approximation was made to the more complicated nonlinear FASTA performance model; we again used [2], but here the result is an allocation of portions of reference databases to compute nodes. Constraints were specified to ensure that all the databases were fully handled, that each compute node could hold its local data in memory, and that execution time was as balanced as possible. The load balancing is complicated by the fact that the databases are not fully distributed. For example, a given database may only be available at one compute node, requiring that it be computed there regardless of the execution time across the rest of the compute nodes.

4.5 Search Procedure

The function of the search procedure is to identify which subset of available resources should be used for the application. To be applicable to a variety of applications, the search procedure must not contain application-specific assumptions. Without such assumptions, it is difficult to predict a priori which resource set will provide the best performance for the application. We have experimented with a variety of approaches and in each case the search involves the following general steps: (i) identify a large number of sets of resources that may be good platforms for the application, (ii) use the application-specific mapper and performance model to generate a data map and predicted execution time for those resource sets, and (iii) select the resource set that results in the lowest predicted execution time.

This *minimum execution-time multiprocessor scheduling problem* is known to be NP-hard in its general form and in most restricted forms. Since launch-time scheduling delays execution of the application, maintaining low overhead is a key goal. We have developed two general search approaches that maintain reasonable overheads while providing reasonable search coverage: a resource-aware search and a simulated annealing search.

4.5.1 Resource-aware search. Many Grid applications share a general affinity for certain resource set characteristics; for example, most Grid applications will perform better with higher bandwidth networks and faster processing speeds. For our resource-aware search approach, we broadly characterize these general application affinities for particular resource characteristics; we then focus the search process on resource sets with these broad characteristics.

Over the evolution of the project we have experimented with three search approaches that fall in this category: one developed for the ScaLAPACK application [27], one developed as part of the set-extended ClassAds framework [24], and one developed for the GrADSoft framework [13]. We describe the last search procedure as a representative of this search type.

The goal of the search process is to identify groups of machines that (i) contain desirable individual machines (i.e., fast CPUs and large local memories) (ii) carry desirable characteristics as an aggregate (i.e., low-delay networks); we term these groups of machines candidate machine groups (CMGs). Pseudo-code for the search procedure is given in Figure 1.4. In the first step, machines are divided into disjoint subsets, or sites. Currently, we group machines in the same site if they share the same domain name; we plan to improve this approach so that sites define collections of machines connected by low-delay networks. The *ComputeSiteCollections* method computes the power set of the set of sites. Next, in each *for* loop the search focus is refined based on a different characteristic: connectivity in the outer-most loop, computational and/or memory capacity of individual machines in the second loop, and

```
ResourceAware_Search:
sites ← FindSites( machList )
siteCollections ← ComputeSiteCollections( sites )
foreach (collection ∈ siteCollections )
  foreach (machineMetric ∈ (computation, memory, dual) )
    for (r ← 1:size(collection) )
      list ← SortDescending( collection, machineMetric )
      CMG ← GetFirstN( list, r )
      currSched ← GenerateSchedule( CMG,
              Mapper, PerfModel )
      if (currSched.predTime < bestSched.predTime)
        bestSched ← currSched
 return (bestSched)
```

Figure 1.4. A search procedure that utilizes general application resource affinities to focus the search.

selection of an appropriate resource set size in the inner-most loop. The *SortDescending* function sorts the current *collection* by *machineMetric*. For example, if *machineMetric* is *dual*, the *sortDescending* function will favor machines with large local memories *and* fast CPUs. *GetFirstN* simply returns the first r machines from the sorted *list*. Next, to evaluate each CMG, the *GenerateSchedule* method (i) uses the *Mapper* to develop a data mapping for the input *CMG*, (ii) uses the *Performance model* to predict the execution time for the given schedule (*predtime*), and (iii) returns a schedule structure which contains the CMG, the map, and the predicted time. Finally, predicted execution time is used to select the best schedule, which is then returned.

The complexity of an exhaustive search is 2^p where p is the number of processors. The complexity of the above search is $3p2^s$ where s is the number of sites. As long as $s << p$, this search reduces the search space as compared to an exhaustive search. We have performed in-depth evaluation of this search procedure for Jacobi and Game of Life [13], and validated it for Cactus, FASTA, and ScaLAPACK.

4.5.2 Simulated annealing. Our resource-aware search assumes general application resource affinities. For applications which do not share those affinities, the resource-aware search may not identify good schedules. For example, FASTA does not involve significant inter-process communication and so does not benefit from the resource-aware procedure's focus on connectivity. Instead, for FASTA it is important that computation be co-scheduled with existing databases. For this type

```
**SimulatedAnnealing_Search:**
for r ← 1:size(machList)
   filteredList ← check local memory
   if size(filteredList) < r, continue
   currCMG ← pick r machines randomly from filteredList
   currSched ← GenerateSchedule(currCMG, Mapper, PerfModel)
   foreach temperature ← max:min
      while energy has not stabilized
         % remove machine, add machine and swap order
         newCMG ← randomly perturb currCMG
         newSched ← GenerateSchedule( newCMG,
                 Mapper, PerfModel )
         if newSched.predTime < currSched.predTime
            currCMG ← newCMG
         else if random number < exp(-dE/kT)
            % Probabilistically allow increases in energy
            currCMG ← newCMG
         if currSched.predTime < bestSched.predTime
            bestSched ← currSched
return bestSched
```

Figure 1.5. A search procedure that uses simulated annealing to find a good schedule without making application assumptions.

of application, we have developed a simulated annealing search approach that does not make assumptions about the application, but is a costlier search approach.

Simulated annealing [21] is a popular method for statistically finding the global optimum for multivariate functions. The concept originates from the way in which crystalline structures are brought to more ordered states by an *annealing process* of repeated heating and slowly cooling the structures. Figure 1.5 shows our adaptation of the standard simulated annealing algorithm to support scheduling. For clarity we have omitted various simple heuristics used to avoid unnecessary searches.

In our procedure, we perform a simulated annealing search for each possible resource subset size, from 1 to the full resource set size. This allows us (i) to decrease the search space by including only machines that meet the memory requirements of the application, and (ii) to avoid some of the discontinuities that may occur in the search space. For example, when two local minima consist of different machine sizes, the search space between them is often far worse than either (or may not even be an eligible solution).

For each possible machine size r we create a filtered list of machines that would meet resource requirements. The initial CMG (candidate machine group) is created by randomly selecting r machines out of this list. A schedule is created from the CMG using the *GenerateSchedule* method, which provides the predicted execution time of the schedule.

At each temperature T, we sample the search space repeatedly to ensure good coverage of the space. To generate a new CMG, the current CMG is perturbed by adding one machine, removing one machine, and swapping their ordering. The new CMG is evaluated using *GenerateSchedule* and the difference between the predicted execution time and that of the current CMG is calculated as dE. If the predicted execution time is smaller ($dE < 0$), then the new CMG becomes the current CMG. If $dE > 0$ then we check if the new CMG should be probabilistically accepted. If a random number is less than the Boltzmann factor $exp(-dE/T)$, then accept the new CMG, else reject the new CMG. Using this distribution causes many upward moves to be accepted when the temperature is high, but few upward moves to be accepted when the temperature is low. After the space has been sampled at one temperature, the temperature is lowered by a constant factor α ($T \leftarrow \alpha T, \alpha < 1$). This search procedure allows the system to move to lower energy states while still escaping local minima. The schedule with the best predicted execution time is returned. We have performed a validation and evaluation of this search procedure for the ScaLAPACK application in [37].

5. RESCHEDULING

Launch-time scheduling can at best start the application with a good schedule; over time, other applications may introduce load in the system or application requirements may change. To sustain good performance for longer running applications, the schedule may need to be modified during application execution. This process, called rescheduling, can include changing the machines on which the application is executing (migration) or changing the mapping of data and/or processes to those machines (dynamic load balancing).

Rescheduling involves a number of complexities not seen in launch-time scheduling. First, while nearly all parallel applications support some form of launch-time scheduling (selection of machines at a minimum), very few applications have built-in mechanisms to support migration or dynamic load balancing. Second, for resource monitoring we have to differentiate between processors on which the application is running

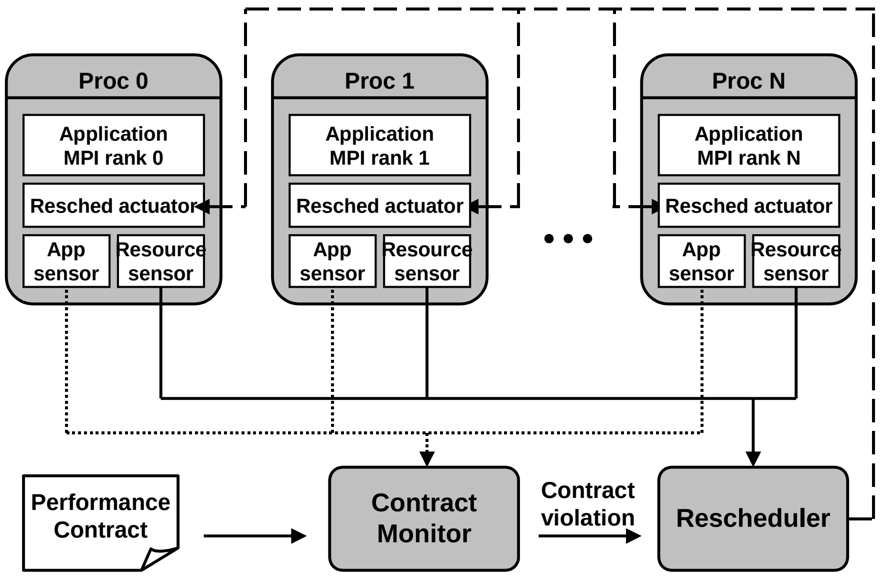

Figure 1.6. GrADS rescheduling architecture.

and processors on which the application is not running. Measurements from resource monitors such as NWS CPU sensors can not be directly compared between active and inactive processors. Third, the overheads of rescheduling can be high: monitoring for the need to reschedule is an ongoing process and, when a rescheduling event is initiated, migration of application processes or reallocation of data can be very expensive operations. Without careful design, rescheduling can in fact hurt application performance.

We have experimented with a variety of rescheduling approaches to address these issues, including an approach developed for the Cactus application [6], an approach called application migration that has been validated for the ScaLAPACK application [34], and an approach called process swapping that has been validated for the Fish application [31]. Through this evolution we have identified a rescheduling architecture. Note that our rescheduling efforts were all conducted for iterative applications, allowing us to perform rescheduling decisions at each iteration. In the following sections we describe this architecture and key contributions we have made to each component of the design.

5.1 Architecture

Figure 1.6 shows our rescheduling architecture and we depict an application already executing on N processors. We also assume that a *performance contract* has been provided by earlier stages of GrADS; this contract specifies the performance expected for the current application execution. While the application is executing, *application sensors* are co-located with application processes to monitor application progress. Progress can be measured, for example, by generic metrics such as flop rate or by application-intrinsic measures such as number of iterations completed. In order that these sensors have access to internal application state, we embed them in the application itself. We work to minimize the impact of these sensors on the application's footprint and execution progress. Meanwhile, *resource sensors* are located on the machines on which the application is executing, as well as the other machines available to the user for rescheduling. For evaluating schedules, it is important that measurements taken on the application's current execution machines can be compared against measurements taken on unused machines in the testbed.

Application sensor data and the performance contract are passed to the *contract monitor*, which compares achieved application performance against expectations. When performance falls below expectations, the contract monitor signals a violation and contacts the rescheduler. An important aspect of a well-designed contract monitor is the ability to differentiate transient performance problems from longer-term issues that warrant modification of the application's execution. Upon a contract violation, the *rescheduler* must determine whether rescheduling is profitable, and if so, what new schedule should be used. Data from the resource sensors can be used to evaluate various schedules, but the rescheduler must also consider (i) the cost of moving the application to a new execution schedule and (ii) the amount of work remaining in the application that can benefit from a new schedule.

To initiate schedule modification, the rescheduler contacts the *rescheduling actuators* located on each processor. These actuators use some mechanism to initiate the actual migration or load balancing. *Application support for migration or load balancing* is the most important part of the rescheduling system. The most transparent migration solution would involve an external migrator that, without application knowledge, freezes execution, records important state such as register values and message queues, and restarts the execution on a new processor. Unfortunately, this is not yet feasible as a general solution in heterogeneous environments. We have chosen application-level approaches as a feasible first step. Specifically, the application developer simply adds a few lines to their source code and then links their application against our MPI rescheduling libraries in addition to their regular MPI libraries.

5.2 Contract Monitoring

GrADS researchers have tried a variety of approaches for application sensors and contract monitoring. The application sensors are based on Autopilot sensor technology [29] and run in a separate thread within the application's process space. To use these sensors, a few calls must be inserted into the application code. The sensors can be configured to read and report the value of any application variable; a common metric is iteration number. Alternatively, the sensors can use PAPI [25] to access measurements from the processor's hardware performance counters (i.e., flop rate) and the MPICH profiling library for communication metrics (i.e., bytes sent per second).

The performance contracts [35] specify predicted metric values and tolerance limits on the difference between actual and predicted metric values. For example, contracts can specify predicted execution times and tolerance limits on the ratio of actual execution times to predicted execution times. The contract monitor receives application sensor data and records the ratio of achieved to predicted execution times. When a given ratio is larger than the upper tolerance limit, the contract monitor calculates an average of recently computed ratios. If the average is also greater than the upper tolerance limit, the performance monitor signals a contract violation to the rescheduler. Use of an average reduces unnecessary reactions to transitory violations.

In the following two sections, we present two approaches to rescheduling for which contract monitoring provides a fundamental underpinning.

5.3 Rescheduling Via Application Migration

Application rescheduling can be implemented with application migration, based on a stop / restart approach. When a running application is signaled to migrate, all application processes checkpoint important data and shutdown. The rescheduled execution is launched by restarting the application, which then reads in the checkpointed data and continues mid-execution.

To provide *application support* for this scenario we have implemented a user-level checkpointing library called **S**top **R**estart **S**oftware (SRS) [33]. To use SRS, the following application changes are required.

- SRS_Init() is placed after MPI_Init() and SRS_Finish() is placed before MPI_Finalize().

- The user may define conditional statements to differentiate code executed in the initial startup (i.e., initialization of an array with zeros) and code executed on restart (i.e., initialization of array with values from checkpoint). SRS_Restart_Value() returns 0 on start and 1 on restart and should be used for these conditionals.

- The user should insert calls to SRS_Check_Stop() at reasonable stopping places in the application; this call checks whether an external component has requested a migration event since the last SRS_Check_Stop().

- The user uses SRS_Register() in the application to register the variables that will be checkpointed by the SRS library. When an external component stops the application, the SRS library checkpoints only those variables that were registered through SRS_Register(). SRS_Read() is used on startup to read checkpointed data.

- SRS_Read() also supports storage of data distribution and number of processors used in the checkpoint. On restart, the SRS library can be provided with a new distribution and/or a different number of processors; the data redistribution is handled automatically by SRS.

- The user must include the SRS header file and link against the SRS library in addition to the standard MPI library.

At run-time, a daemon called *Runtime Support System* (RSS) is started on the user's machine. RSS exists for the entire duration of the application, regardless of migration events. The application interacts with the RSS during initialization, to check if the application needs to be stopped during SRS_Check_Stop(), to store and retrieve pointers to the checkpointed data, and to store the processor configuration and data distribution used by the application.

We use a modified version of the *contract monitor* presented in Section 5.2. Specifically, we added (i) support for contacting the migration rescheduler in case of a contract violation, (ii) interfaces to allow queries to the contract monitor for remaining application execution time, and (iii) support for modifying the performance contract dynamically. For the third issue, when the rescheduler refuses to migrate the application, the contract monitor lowers expectations in the contract. Dynamically adjusting expectations reduces communication with the contract monitor.

The migration *rescheduler* operates in two modes: *migration on request* occurs when the contract monitor signals a violation, while in *opportunistic migration* the rescheduler checks for recently completed applications (see [34] for details) and if one exists, the rescheduler checks if performance benefits can be obtained by migrating the application to the newly freed resources. In either case, the rescheduler contacts the NWS to obtain information about the Grid; our application migration approach does not involve special-purpose resource sensors. Next, the rescheduler predicts remaining execution time on the new resources, remaining execution time on the current resources, and the overhead for migration and determines if a migration is desirable. We have evaluated rescheduling based on application migration in [34] for the ScaLAPACK application.

5.4 Rescheduling Via Process Swapping

Although very general-purpose and flexible, the stop / migrate / restart approach to rescheduling can be expensive: each migration event can involve large data transfers and restarting the application can incur expensive startup costs. Our process swapping approach [31] provides an alternative trade-off: it is light-weight and easy to use, but less flexible than our migration approach.

To enable swapping, the MPI application is launched with more machines than will actually be used for the computation; some of these machines become part of the computation (the *active* set) while some do nothing initially (the *inactive* set). The user's application sees only the active processes in the main communicator (MPI_Comm_World); communication calls are hijacked and work is performed behind the scenes to convert the user's communication calls in terms of the active set to real communication calls in terms of the full process set. During execution, the system periodically checks the performance of the machines and swaps loaded machines in the active set with faster machines in the inactive set. This approach is very practical: it requires little application modification and provides an inexpensive fix for many performance problems. On the other hand, the approach is less flexible than migration: the processor pool is limited to the original set of machines and we have not incorporated support for modifying the data allocation.

To provide *application support* for process swapping we have implemented a user-level swapping library. To use swapping, the following application changes are required.

- The iteration variable must be registered using the swap_register() call.
- Other memory may be registered using mpi_register() if it is important that their contents be transferred when swapping processors.

- The user must insert a call to MPI_Swap() inside the iteration loop to exercise the swapping test and actuation routines.
- The user must include mpi_swap.h instead of mpi.h and must link against the swapping library in addition to the standard MPI library.

In addition to the swap library, swapping depends on a number of run-time services designed to minimize the overheads of rescheduling and the impact on the user. The *swap handler* fulfills the role of application sensor, resource sensor, and rescheduling actuator in Figure 1.6. The swap handler measures the amount of computation, communication, and barrier wait time of the application. The swap handler also measures passive machine information like the CPU load of the machine and active information requiring active probing like current flop rate.

The *swap manager* performs the function of the rescheduler in Figure 1.6. A swap manager is started for each application and is dedicated to the interests of that application only. Application and resource information is sent by the swap handlers to the swap manager. Acting in an opportunistic migration mode, the swap manager analyzes performance information and determines when and where to swap processes according to a swap policy. When a swap is necessary, the swap manager triggers the swap through the swap handlers.

When a swap request is received, the swap handler stores the information. The next time the application checks if swapping is necessary by calling MPI_Swap(), the swap is executed. From the application's perspective, the delay of checking for a swap request is minimal: the information is located on the same processor and was retrieved asynchronously.

We have evaluated our process swapping implementation with the Fish application and experimental results are available in [31].

6. METASCHEDULING

Our launch-time scheduling architecture (Figure 1.2) schedules one application at a time and, except for the usage of dynamic NWS data on resource availability, does not consider the presence of other applications in the system. There are a number of problems with this application-centric view. First, when two applications are submitted to GrADS at the same time, scheduling decisions will be made for each application ignoring the presence of the other. Second, if the launch-time scheduler determines there are not enough resources for the application, it can not make further progress. Third, a long running job in the system can severely impact the performance of numerous new jobs entering the

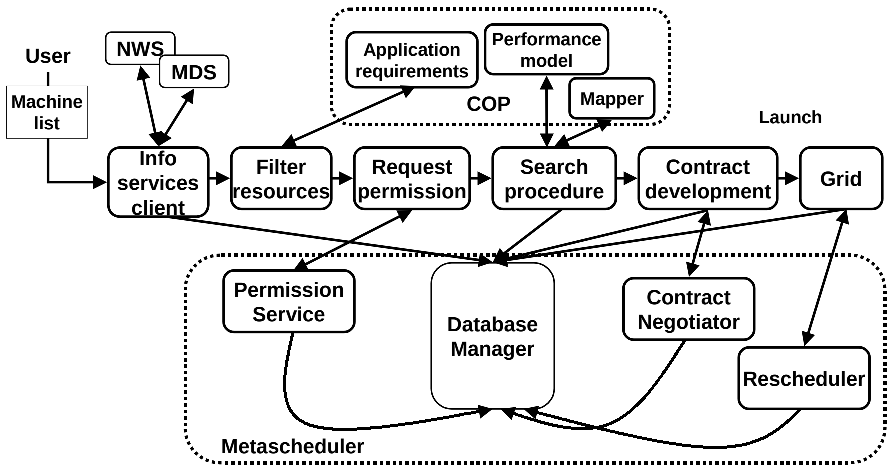

Figure 1.7. Metascheduler and interactions.

system. The root cause of these and other problems is the absence of a metascheduler that possesses global knowledge of all applications in the system and tries to balance the needs of the applications.

The goal of our metascheduling work [32] is to investigate scheduling policies that take into account both the needs of the application and the overall performance of the system. To investigate the best-case benefits of metascheduling we assume an idealized metascheduling scenario. We assume the metascheduler maintains control over all applications running on the specified resources and has the power to reschedule any of those applications at any time. Our metascheduling architecture is shown in Figure 1.7. The metascheduler is implemented by the addition of four components, namely database manager, permission service, contract negotiator and rescheduler to the GrADS architecture.

The *database manager* acts as a repository for information about all applications in the system. The information includes, for example, the status of applications, application-level schedules determined at application launch, the predicted execution costs for the end applications. As shown in Figure 1.7, the application stores information in the database manager at different stages of execution. Other metascheduling components query the database manager for information to make scheduling decisions.

In metascheduling, after the filter resources component generates a list of resources suitable for problem solving, this list is passed, along with the problem parameters, to the *permission service.* The permission service determines if the filtered resources have adequate capacities for problem solving and grants or rejects permission to the application for proceeding with the other stages of execution. If the capacities of the resources are not adequate for problem solving, the permission service tries to proactively stop a large resource consuming application to accommodate the new application. Thus, the primary objective of the permission service is to accommodate as many applications in the system as possible while respecting the constraints of the resource capacities.

In the launch-time scheduling architecture shown in Figure 1.2, after the search procedure determines the final schedule, the application is launched on the final set of resources. In the metascheduling architecture shown in Figure 1.7, before launching the application, the final schedule along with performance estimates are passed as a performance contract to the *contract developer*, which in turn passes the information to the metascheduling component, *contract negotiator.* The contract negotiator is the primary component that balances the needs of different applications. It can reject a contract if it determines that the application has made a scheduling decision based on outdated resource information. The contract negotiator also rejects the contract if it determines that the approval of the contract can severely impact the performance of an executing application. Finally, the contract negotiator can also preempt an executing application to improve the performance contract of a new application. Thus the major objectives of the contract negotiator is to provide high performance to the individual applications and to increase the throughput of the system.

More details about our metascheduling approach and an initial validation for the ScaLAPACK application are available in [32].

7. STATUS AND FUTURE WORK

In this chapter we have described approaches developed in GrADS for launch-time scheduling, rescheduling, and metascheduling in a Grid application development system. In developing these approaches we have focused on a Grid environment consisting of distributed clusters of shared workstations; unpredictable and rapidly changing resource availabilities make this a challenging and interesting Grid environment. Previous Grid scheduling efforts have been very successful in developing general scheduling solutions for embarrassingly parallel applications (e.g. [11]) or serial task-based applications (e.g. [28]). We focus instead on data-parallel applications with non-trivial inter-processor communications. This application class is challenging and interesting because to obtain reasonable performance, schedulers must incorporate application requirements and resource characteristics in scheduling decisions.

We have incorporated many of our launch-time scheduling approaches in GrADSoft and the resulting system has been tested for all of the focus applications described in Section 3. Currently, we are working to generalize our rescheduling results and incorporate them in GrADSoft. Similarly, researchers at UIUC are integrating more sophisticated contract development technologies and researchers at Rice University are working to integrate initial GrADS compiler technologies for automated generation of performance models and mappers. GrADSoft will be used as a framework for testing these new approaches together and extending our scheduling approaches as needed for these new technologies.

In collaboration with other GrADS researchers we plan to extend our approaches to support more Grid environments and other types of applications. Rich Wolski and Wahid Chrabakh, GrADS researchers at the University of California, Santa Barbara, have recently developed a reactive scheduling approach for a Satisfiability application. This application's resource requirements dynamically grow and shrink as the application progresses and the reactive scheduling approach dynamically grows and shrinks the resource base accordingly. As this work evolves, their results can be used to extend our rescheduling approaches to consider changing application requirements. Anirban Mandal and Ken Kennedy, GrADS researchers at Rice University, are working to develop compiler technologies for work-flow applications; we plan to extend our scheduling approaches for this application type.

Acknowledgments

The authors would like to thank the entire GrADS team for many contributions to this work.

References

[1] *FASTA package of sequence comparison programs.* ftp://ftp.virginia.edu/pub/fasta.

[2] *lp_solve FTP site.* ftp.es.ele.tue.nl/pub/lp_solve.

[3] *Maui Scheduler homepage.* http://www.supercluster.org/maui.

[4] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with nimrod/g: Killer application for the global grid? In *International Parallel and Distributed Processing Symposium, May 2000.*, 2000.

[5] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. *Data management and transfer in high-performance computational grid environments.* Parallel Computing, 2002.

[6] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. *The Cactus Worm: Experiments with dynamic resource discovery and allocation in a grid environment.* International Journal of High Performance Computing Applications, 2001.

[7] G. Allen, W.Benger, C. Hege, J. Masso, A. Merzky, T. Radke, E. Seidel, and J. Shalf. *Solving Einstein's equations on supercomputers.* IEEE Computer Applications, 1999.

[8] F. Berman. *The Grid, Blueprint for a New computing Infrastructure chapter 12.* Morgan Kaufmann Publishers,Inc.Edited by Ian Foster and Carl Kesselman, 1998.

[9] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, L. J. Dennis Gannon, K. Kennedy, C. Kesselman, D. Reed, L. Torczon, and R. Wolski. *The GrADS project: Software support for high-level grid application development.* International Journal of High-performance Computing Applications, 15(4), Winter 2001., 2001.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics.* Philadelphia, PA, 1997.

[11] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template: User-level middleware for the grid. In *Proceedings of Super Computing '00 (Denver, 2000).*, 2000.

[12] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE 5'gmposium on High-Performance Distributed Computing, August 2001.*, 2001.

[13] H. Dail, F. Berman, and H. Casanova. *A decoupled scheduling approach for grid application development environments.* Journal of Parallel and Distributed Computing. To appear., 2003.

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

[15] Message Passing Interface Forum. *MPI: A message-passing interface standard.* Technical Report UT-CS-94-230, 1994.

[16] I. Foster and C. Kesselman. The globus project: A status report. In *Proceedings of the 7th Heterogeneous Computing Workshop, IEEE Press*, 1998.

[17] A. S. Grimshaw, A. J. Ferrari, F. Knabe, and M. A. Humphrey. *Wide-Area Computing: Resource Sharing on a Large Scale.* IEEE Computer, May 1999., 1999.

[18] IBM. *Using and Administering LoadLeveler.* IBM Document #SA227881-00. Available at http://www.ibm.com.

[19] N. Karonis, B. Toonen, and I. Foster. *MPICH-G2: A grid-enabled implementation of the message passing interface.* Journal of Parallel and Distributed Computing. To appear., 2003.

[20] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, R. Aydt, D. Reed, D. Gannon, J. Dongarra, S. Vadhiyar, L. Johnsson, C. Kesselman, and R. Wolski. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop, International Parallel and Distributed Processing Sgmposium*, 2002.

[21] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. *Optimization by simulated annealing.* Science, 220, 1983.

[22] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[23] C. Liu and I. Foster. A constraint language approach to grid resource selection. Technical Report University of Chicago TR-2003-07, 2003.

[24] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resourceselection framework for grid applications. In *Proceedings of the 11 th IEEE Symposium on HighPerformance Distributed Computing*, 2002.

[25] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, 2001.

[26] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the United ,States of America*, 1988.

[27] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical libraries and the grid. In *Proceedings of IEEE SC'01 Conference on High-performance Computing*, 2001.

[28] R. Raman, M. Livny, and M. Solomon. *Matchmaking: An extensible framework for distributed resource management.* Cluster Computing, 2(2), 1999.

[29] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, 1998.

[30] M. Ripeanu, A. Iamnitchi, and I. Foster. *Performance predictions for a numerical relativity package in Grid environments.* International Journal of High Performance Computing Applications, 15(4), 2001.

[31] O. Sievert and H. Casanova. A simple mpi process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 2003. To appear.

[32] S. Vadhiyar and J. Dongarra. A metascheduler for the grid. In *Symposium on High Performance Distributed Computing*, 2002.

[33] S. Vadhiyar and J. Dongarra. Srs - a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 2003. In review.

[34] S. Vadhiyar and Jack J. Dongarra. A performance oriented migration framework for the grid. In *International Symposium on Cluster Computing and the Grid, 2003. To appear.*, 2003.

[35] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed. Performance contracts: Predicting and monitoring grid application behavior. In *Proceedings of the end International Workshop on Grid Computing*, 2001.

[36] R. Wolski, N. Spring, and J. Hayes. *The network weather service: A distributed resource performance forecasting service for metacomputing.* Future Generation Computer Systems, 15(5-6), 1999.

[37] Asim YarKhan and Jack J. Dongarra. Experiments with scheduling using simulated annealing in a grid environment. In *Proceedings of the 3rd International Workshop on Grid Computing*, 2002.