UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Modular Framework for Adaptive Scheduling in Grid Application Development Environments

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Holly Janine Dail

Committee in charge:

Professor Francine Berman, Chair
Professor Jeanne Ferrante
Professor Keith Marzullo

2002

The thesis of Holly Janine Dail is approved:

Chair

University of California, San Diego

2002

To my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This work has provided me with the opportunity to interact with and learn from many wonderful people. I would like to make special mention of the following people and institutions, without whom this work would not have been possible.

Fran Berman, my advisor, for her guidance and encouragement. She has been a wonderful role model and I have learned much from her.

Henri Casanova, my co-advisor, who provided the perfect combination of insight, inspiration, and encouragement. He has been exceptionally generous.

Jeanne Ferrante and Keith Marzullo, my committee members, for their guidance throughout my time at UCSD, and for providing insightful feedback on the thesis itself.

Alan Su and Shava Smallen for their support when things were most frustrating, willingness to answer any and all questions, and for careful reviews of the thesis.

Otto Sievert and Graziano Obertelli who worked with me on the GrADS project. Our discussions taught me many things and led to the formulation of this thesis.

Jim Hayes for software engineering advice and for developing practically unbreakable software tools to assist research efforts such as this one and Renata Teixeira for sharing her networking expertise with me.

All those responsible for maintaining the GrADS testbed environment, on which the experiments in this thesis were performed. I am especially grateful to the Innovative Computing Laboratory at UTK for usage of the torc machines, the Pablo group at UIUC for usage of the opus and major machines, Martin Swany for assistance with NWS-related questions, and Sridhar Gullapalli for assistance with MDS-related questions.

Finally, I would like to thank all members of the GrADS research community. I would especially like to thank Mark Mazina and John Mellor-Crummey for, among other things, their interest in exploring compiler / scheduler interactions. I would also like to thank Ruth Aydt, who has provided many insightful comments at every stage.

This material is based upon work supported by the National Science Foundation under Grant No. 9975020. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

ABSTRACT OF THE THESIS

A Modular Framework for Adaptive Scheduling in
Grid Application Development Environments

by

Holly Janine Dail
Master of Science in Computer Science
University of California, San Diego, 2002
Professor Francine Berman, Chair

To achieve improved performance, application schedulers are typically designed to satisfy the resource requirements of specific applications. Consequently, application characteristics and models are often embedded in the scheduler itself. Results have shown that this strategy is effective for achieving improved application performance. However, application-specific schedulers may not be easily retargeted for other applications. In this thesis, we propose a modular application scheduler design that employs detailed application performance models and mapping strategies that promote application performance, but does not embed such components within the scheduler itself.

Our scheduler is both environment-sensitive and configurable. To ensure that schedules are properly targeted for conditions of the target execution environment at run-time, the scheduler can incorporate dynamic resource availability in scheduling decisions. The scheduler also supports a set of configurable scheduling policies that are easily tuned to control scheduler behavior.

We implement a prototype scheduler and use the class of iterative, mesh-based applications to test the prototype. We implement two test applications, *Jacobi* and the *Game of Life*, and develop performance models and mapping strategies for each application. We present experimental results we obtained by applying our scheduling methodology to *Jacobi* and the *Game of Life* in Computational Grid environments. Our testbeds included up to 20 machines organized in 4 clusters at 3 geographically distributed sites. In these experiments, our approach consistently outperforms conventional scheduling approaches.

Chapter I

Introduction

With vast improvements in wide-area network performance and the pervasiveness of commodity resources, distributed parallel computing can benefit from an increasingly rich computational platform. Focused development efforts have been successful in targeting important scientific applications for distributed groups of resources. The majority of these projects have involved large time investments and have required extensive support by distributed computing experts.

In recent years, several large-scale software infrastructure projects [18, 25, 34] have focused on simplifying the usage of distributed, heterogeneous computational platforms, or Computational Grids [19, 20]. Such Grid computing software helps reduce programmer effort, and can improve application performance. However, these efforts generally do not focus on the specific needs of applications; to achieve acceptable performance on the Grid, users must consider the needs of their application and adapt their usage of Grid computing software accordingly. For example, in a typical application development scenario, users are currently obliged to discover available resources, select an application-appropriate subset of these resources, perform staging of binaries on selected machines, and may even need to perform some application monitoring to determine if the application is making progress. For these reasons, Grid application development remains a daunting proposition for the majority of users who could benefit from the extensive resources offered by Computational Grids.

The obvious alternative is to develop software that frees the user of these responsibilities. Application scheduling is one area in which significant progress has been made towards the simplification of application development for the Grid; see [8] for a survey of progress in this area. Application schedulers typically manage discovery of available resources, selection of an application-appropriate resource group, and mapping of application tasks or data to those resources. To effectively provide these services, schedulers must evaluate the target Grid resource environment in terms of the requirements of the application itself. Many projects have successfully developed scheduling strategies for the Grid [1, 2, 12, 42, 48, 49, 50, 52, 53] While these schedulers do consider application requirements, the majority of such efforts embed application-specific details in the scheduling software itself; components that are commonly embedded include application-specific performance models and strategies for mapping application data or tasks to selected resources. This strategy can result in effective service for specific applications, but the scheduler design may not be easily retargeted for other applications.

In this thesis, we present a modular scheduling framework that allows the scheduler to utilize detailed application performance models and mapping strategies, but does not embed these components in the scheduler itself. Our approach is based on an application-independent scheduler framework that is coupled with an application-specific performance model and mapping strategy to create a service that effectively develops schedules appropriate to the needs of the target application. This approach provides a flexible scheduler that can be easily targeted to a variety of applications. Note that we do not expect to achieve the performance of a scheduler that has been highly-tuned for a specific application; instead, our goal is to provide consistently improved performance as compared to conventional scheduling strategies. To develop schedules that appropriately utilize available resources, the scheduler considers the characteristics of the target Grid environment. To do this, Grid resource characteristics are retrieved at run-time and automatically incorporated in scheduling decisions. We focus on the class of iterative, mesh-based applications as a challenging, yet tractable test case for Grid application scheduling. In particular, this class of applications demonstrates relatively predictable performance, making the scheduling problem tractable, yet these applications typically involve interesting communication patterns, thus making the scheduling problem challenging. Furthermore, this class is an important class of applications for science and engineering codes. In summary,

> *In this thesis we propose and prototype a modular, adaptive scheduling methodology designed to promote application performance in Computational Grid environments. We use the class of iterative, mesh-based applications as a test case, and demonstrate the efficacy of our scheduling approach in production Grid environments for realistic usage scenarios.*

This work was performed in the context of the larger Grid application development framework proposed by the Grid Application Development Software Project (GrADS) [7]. The GrADS project seeks to simplify all aspects of Grid application development, and is building software designed to provide an end-to-end application development system for the Grid. These efforts provide new challenges and opportunities for the development of Grid application scheduling strategies, and thus is an interesting framework for this thesis. The scheduler described in this thesis is the first prototype scheduling component developed for the GrADS software infrastructure.

This thesis is organized as follows. We describe the application development software architecture proposed by the GrADS project in Chapter II. We also detail the scheduler specification proposed as part of this software architecture. In Chapter III we present the design of our application-generic scheduling framework. In Chapter IV we describe the characteristics of iterative, mesh-based applications; we then detail two specific test applications from this class; and finally we develop an application-specific performance model and mapping strategy that can be paired with our scheduler framework to provide scheduling for our test applications. In Chapter V, we present experimental results we obtained when applying our scheduling methodology to real applications and real Computational Grid environments. Finally, in Chapter VI we describe related work in the field of application scheduling, consider directions for interesting future work, and we present final conclusions.

Chapter II

Background

One of the largest roadblocks to everyday usage of Computational Grids is the extensive expertise and development time that must be invested in each application before acceptable performance can be achieved. Development of a distributed, Grid-enabled application typically requires a complex and time-consuming process of application creation (or modification), compilation, resource discovery, selection of resources, staging of binaries and data files, execution, and post-mortem analysis. When application performance is critical, many cycles of the development process may be required.

The Grid Application Development Software Project (GrADS) [7, 23] has proposed an ambitious alternative: replace the discrete, user-controlled stages of application preparation and execution with an end-to-end software-controlled process. Our goal is to provide tools that enable the user to focus only on high-level application design without sacrificing application performance. Existing Grid middleware products [18, 25, 34] provide some services required by this system, but are not sufficient. Thus, a primary goal of the GrADS project is to develop **new technologies for Grid application development and execution**. For example, members of the GrADS project are developing software components to provide

- discovery and communication of Grid resource characteristics;
- discovery and communication of application characteristics and run-time requirements;
- run-time application monitoring and processing of application performance data; and
- automatic decision processes to provide adaption to application requirements, Grid characteristics, and user policies.

These new technologies can be used in conjunction with existing solutions to provide the individual services needed to simplify application development for the Grid. However, to enable adaptive and performance-oriented Grid computing, these individual tools must be able to communicate and, more importantly, collaborate. Therefore, another primary goal of the GrADS project is to develop a **unifying system architecture** that provides the user with a comprehensive software solution to application preparation and execution.

In Section II.A, we describe GrADSoft, the first version of the GrADS system architecture. Section II.B, details the GrADSoft SCHEDULER component specification. In Section II.C we describe the current status of the GrADSoft architecture and in Section II.D we summarize the chapter.

II.A GrADSoft

Figure II.1 provides a high-level view of the GrADSoft system architecture [30]. The goal in this system design is to clearly specify the services provided by each component as well as the interfaces that components should support. The component design provides the flexibility necessary to support a variety of application preparation and execution scenarios.

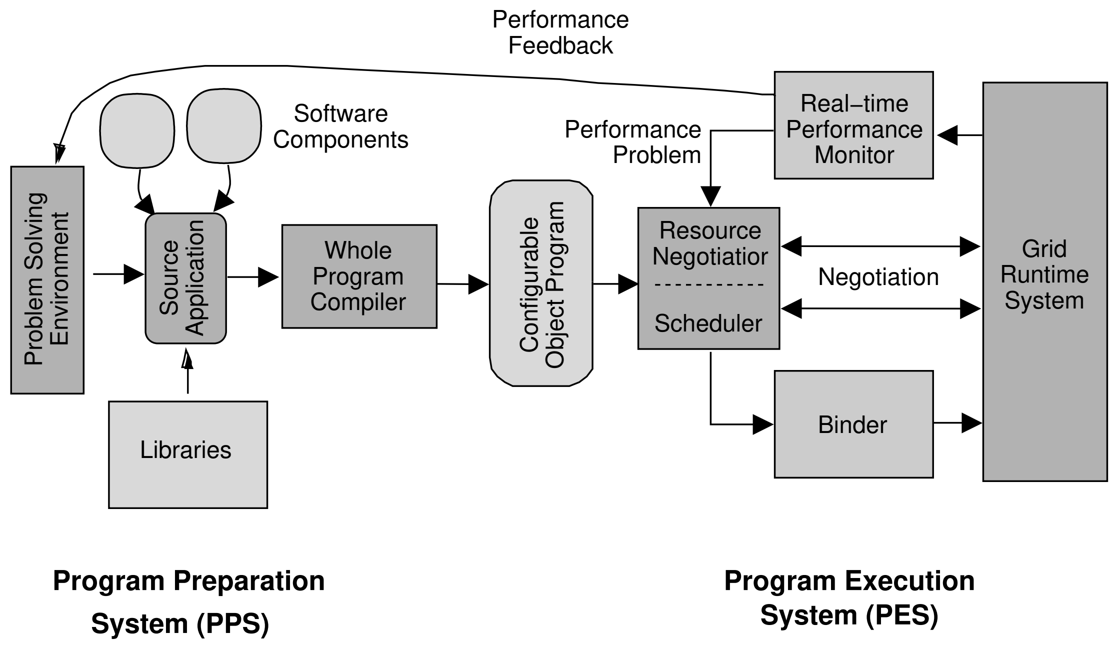


Figure II.1: GrADSoft Architecture.

Since our focus in this thesis is on Grid application scheduling, we are interested primarily in describing the scheduling component of GrADSoft. To understand the role played by the GrADSoft scheduler, it is useful to understand how all of the components diagrammed in Figure II.1 might coordinate to provide a continuous program preparation and execution system. We therefore describe one proposed GrADSoft component interaction scenario. Additional component details and alternative scenarios are described in [30].

As shown in Figure II.1, there are two clear subsystems to the GrADSoft architecture:

- The Program Preparation System (PPS) handles off-line application-development, composition, and compilation.
- The Program Execution System (PES) provides on-line resource discovery, scheduling, binding, and application performance monitoring.

We discuss each of these subsystems in turn.

PPS

To begin the development process, the user interacts with a high-level interface called a problem solving environment (PSE) to assemble a Grid application. Our approach is to develop a collection of libraries that provide not only the base algorithms, but also information and models that describe the resource requirements and performance behavior of each library call. The system will support more general software components in addition to these specialized libraries; however, use of GrADS libraries will provide other GrADSoft components with important application execution performance clues and is therefore likely to provide a run-time performance advantage for the user.

The resulting application and GrADS libraries are passed to the compiler. The compiler then performs program analysis and partial compilation, generates application-wide performance models and mapping strategies, and generates a configurable object program (COP). The COP encapsulates an intermediate representation code (IR code) for the application, an assembly of application behavior models, a mapper, and an application resource requirement specification (refer to Section II.B for details on each COP component). The PPS phase may be performed off-line and sometimes need only be completed once per application; for this reason the COP is a long-lived object that may be re-used for multiple program execution phases.

PES

When the user decides to execute the application, the application COP is retrieved and the PES is invoked by the GrADSoft system. 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 the PERFORMANCE MODEL and MAPPER to select an application-appropriate resource subset and a mapping of the problem data or tasks onto those resources. In a second compilation phase, the BINDER is invoked to perform a final, resource-specific compilation of the INTERMEDIATE REPRESENTATION CODE (recall that this is one of the components of the COP). Next, the executable is launched on the selected Grid resources and a REAL-TIME MONITOR is used to track program performance and detect violation of performance guarantees. Performance guarantees are formalized in a PERFORMANCE CONTRACT.[1] In the case of a performance contract violation, either the BINDER is invoked to reconfigure the program in the current execution environment or the RESCHEDULER is invoked to evaluate alternative resource sets. After program execution is complete, post-mortem performance information will be stored in a repository; this information may then be retrieved by any GrADSoft component to improve application results in future runs.

II.B GrADSoft scheduler specification

The GrADSoft scenario presented in Section II.A provided a brief overview of a specific GrADSoft component interaction scenario. In this section, we describe in greater detail the specification for the GrADSoft SCHEDULER. This specification defines the service the SCHEDULER is to provide as well as the interfaces it should support; it does not rely on any particular interaction scenario.

The role of the SCHEDULER is to select Grid resources appropriate for a particular problem run, where *problem run* is defined by the application itself and by problem configuration parameters such as problem size or input data file. The SCHEDULER should transparently provide service for arbitrary applications and Grid environments. For this reason, characteristics of the application and Grid environment must be available as inputs to the SCHEDULER. Similarly, the selected schedule and associated information must be communicated in a well-defined manner for consumers of the SCHEDULER output.

II.B.1 Inputs

Since the scheduling process is dependent on available application and environment information, we describe in greater detail the format of these SCHEDULER inputs.

[1]When the user interacts with a PSE to develop their application, they also specify what their performance expectations are. For example, they might specify a flexible desired turnaround time or a hard deadline for application completion. These specifications are used to create a performance contract, which formally specifies program performance expectations. More details are available in [30, 51].

Grid information retrieval is necessary in order for the SCHEDULER to develop schedules appropriate for the current Grid environment. Currently, GrADSoft utilizes the popular Grid information services of the Metacomputing Directory Service (MDS) [11, 13] and the Network Weather Service [57, 58]. The NWS and MDS are centralized information servers that provide dynamic and static information about the current state of Grid resources. Since these interfaces are well-known standards for Grid researchers, we do not describe them in detail here.

Application characteristics and models are communicated to the scheduler by way of the COP. Recall that this component encapsulates four distinct sub-components: IR CODE, MAPPER, one or more APPLICATION BEHAVIOR MODELS, and an application resource requirement specification called an ABSTRACT APPLICATION RESOURCE AND TOPOLOGY MODEL (AART). The IR CODE object is used for final program compilation and is not utilized during the scheduling process. The other three components are needed by the scheduler and require further discussion.

Mapper

Given a chosen set of compute resources, the MAPPER determines a performance-efficient assignment of application tasks and/or data for execution on those resources. For example, suppose we have a MAPPER developed for master-slave applications and an input list consisting of two fast machines and two slow ones. Further suppose that we expect the master's workload to be quite high for a certain application. In this case, an appropriate assignment of tasks might place the master on one of the fast machines and a slave on each of the three other machines. To provide load-balancing, the MAPPER might also place extra work on the fastest of the three slaves.

Application behavior models

This group encapsulates any models of application behavior that are required by GrADSoft components. For scheduling purposes, the most useful models are those that provide some measure of desirability for possible resource sets and data mappings. There are many types of behavior models that might be of interest for GrADS in the long-term; for example, models could be equation-based, simulation-based, or history-based and the desirability metric could be predicted execution time, resource usage cost, or throughput. In this thesis, we will consider the metric of predicted execution time as provided by an equational PERFORMANCE MODEL. This is the most common application execution performance metric for run-time schedulers [2, 10, 12, 8, 43, 45, 50, 53].

AART Model

This model provides a structured method for specification of application resource requirements. The need to consider application resource requirements for effective application scheduling on the Grid is clear [8]; however, since many Grid application schedulers are designed for a particular application or application-class, application resource requirements are often embedded in the scheduler design itself [12, 47, 49, 50]. By comparison, a primary GrADSoft design goal is smooth adaptation to a variety of applications. For these reasons, we have developed the AART to support formal specification of resource requirements to the SCHEDULER.

The AART model consists of a collection of resource requirements plus a description of the processor topology required by the application. Examples of such topologies include a one-to-many communication arrangement (star), an all-to-all communication arrangement (fully-connected graph), or a topology where only neighboring processors need to communicate (a mesh). Resource requirements are the mechanism by which specific resource needs are specified. Examples of resource requirements include the minimum aggregate memory needed for the application, the minimum acceptable bandwidth between any two processors, and required software installations. While requirements such as software are descriptive, requirements such as aggregate memory requirements can be specified as parametric models; to be useful these models must be provided with additional information such as problem size.

Since applications frequently require more than one type of resource, any number of resource subsets can be defined in the AART model. In the case that more than one type of resource set is defined, resource requirements can be specified that apply to all resources needed for the application, or they can apply to only a subset of resources needed by the application. In addition, requirements can be applied to a pair of resource subsets; for example, if two resource subsets were defined, A and B, one might also want to specify a minimum bandwidth for any connection between resources in subset A and resources in subset B.

Note that the AART MODEL framework itself is designed to be application generic so that it can be useful for a variety of application types. To instantiate the framework for a specific application, the application's resource needs are grouped into resource subsets and associated resource requirements.

To clarify, we give an example application and show what the AART specification might be for this application. We consider a master-slave application where the master application work must be assigned to a single resource but the slave work can be assigned to any number of resources. Let us assume that the master requires 1 GB of local memory and a CPU speed of 1000 MHz. Additionally, the slaves must contain an aggregate memory amount of 3 GB, though it does not matter how the memory is spread amongst the machines. In this example application, the only communication is between the master and the slaves and more data is sent from the master to the slaves than vice versa. Specifically, the application needs bandwidth capability of 10 megabits per second (Mbps) from the master to the slaves and 3 Mbps from the slaves to the master.

An AART for this application would likely specify a “star” communication topology with the master at the center of the star. The AART would also likely specify that the application requires two distinct subsets of resources, labeled in the AART as Subset 0 for the master and subset 1 for the slaves. Given the above requirements, the resource requirements might be specified formally as:

- Subset 0:
  - Number of resources = 1
  - Local memory requirement $\geq$ 1 GB
  - CPU speed $\geq$ 1000 MHz
- Subset 1:
  - Number of resources $\geq$ 1
  - Aggregate memory requirement $\geq$ 3 GB
- Subset $0 \rightarrow 1$:
  - Bandwidth $\geq$ 10 Mbps
- Subset $1 \rightarrow 0$:
  - Bandwidth $\geq$ 3 Mbps

In this example we provided numeric specifications for resource requirements such as minimum available local memory. For some applications the AART may contain such values. To ensure system generality, the AART structure itself must remain independent of any particular problem run or Grid environment; for this reason, resource requirements will typically be specified as parametric models that accept problem run or resource environment characteristics as input.

II.B.2 Scheduler output

Once the SCHEDULER has selected a final resource set and mapping (a schedule), this information is communicated in a well-defined manner to other GrADSoft components. The SCHEDULER output is called a VIRTUAL MACHINE. The VIRTUAL MACHINE is made up of one or more RESOURCE objects, a NETWORK object, and a TOPOLOGY description.

Resource An object that represents a physical device that can be used to perform work.

Network An object that encapsulates information about each link between members of a given group of RESOURCES. Any number of characteristics can be attached to each "RESOURCE to RESOURCE" link.

Topology A high-level description of the "application to RESOURCE" task or data mapping. For example, given our master slave example, the TOPOLOGY might specify which RESOURCE had been selected for the master and which ones for the slaves.

Virtual machine An object that encapsulates the selected schedule and schedule-time Grid characteristics. The VIRTUAL MACHINE contains a NETWORK, TOPOLOGY, and any number of RESOURCES.

II.C Current status

The GrADSoft system architecture described in Section II.A is an ongoing design effort that is continually evolving as the GrADS project evolves. Efforts are underway to develop each of the new component technologies that will be required to realize the GrADSoft design [7, 23, 51, 3, 29, 56, 37]. While the individual GrADSoft components such as the COMPILER and the PERFORMANCE MONITOR are fundamental to the success of the GrADSoft system, these components must be able to interact and coordinate information flow and decision procedures. To this end, there is also an effort to prototype a unifying software system to provide the necessary information flow and coordination for individual GrADSoft components. At the time of this writing, this system, called the GrADSoft Prototype, includes over ten thousand lines of integrated C++ code [24]. Initial prototypes are complete for all of the PPS / PES interfaces.

II.D Chapter summary

In this chapter we have described GrADSoft, a modular software architecture for Grid application development and execution. We also detailed the basic functionalities and interfaces that should be provided by a GrADSoft SCHEDULER.

The GrADSoft architecture introduces new challenges and opportunities for the development of Grid application scheduling strategies. *This thesis proposes a scheduler design that utilizes and extends the GrADS framework to provide an adaptive scheduling service for Computational Grid environments.*

As part of this work, we developed a prototype SCHEDULER that is integrated into the current GrADSoft prototype. The success of this design in validation experiments demonstrates the efficacy of the GrADS goals. Furthermore, this design is the first instantiation of a major GrADSoft component and is therefore an important proof of concept for the GrADSoft framework itself.

Chapter III

Scheduling methodology

In the previous chapter we presented the GrADSoft architecture and described the specification for the GrADSoft SCHEDULER. In this chapter, we present an adaptive scheduling methodology that is the first instantiation of the GrADSoft scheduler specification. Key challenges of this environment and the approach taken by our methodology are as follows.

- The scheduler should gracefully adapt to a variety of applications. We have designed a highly modular framework that can be easily instantiated for specific applications.
- The quality and quantity of Grid information varies widely and somewhat unpredictably over time and from testbed to testbed. Our approach is to provide best-effort service by adapting to information availability. That is, while the scheduler will likely provide the best service when Grid information is highly available, it should continue to function when information availability is lower.
- Similarly, the quality and quantity of application information and models in GrADSoft will vary widely from application to application. Our goal is again best-effort; that is we seek to provide scheduling service that is commensurate with available information.

We begin in Section III.A with the scope of the scheduler design and definitions of key concepts needed in the rest of the chapter. In Section III.B we describe the scheduler design, in Section III.C we describe the collection and usage of Grid information by the scheduler, and in Section III.D we detail the different scheduling policies supported by the scheduler. Finally, Section III.E provides a chapter summary.

III.A Scope and definitions

III.A.1 Scheduling assumptions

Our methodology is based on a number of assumptions about the target scheduling scenario and environment. Most importantly, the scheduler is designed to support the GrADSoft architecture and to utilize the GrADS Computational Grid environment. Additionally, the methodology is based on a number of assumptions about the target scheduling scenario, application model, and Grid environment. We describe each of these assumptions below.

Scheduling scenario

We assume that the scheduler will be called just before run-time, and that the chosen resource set will be utilized for the entire problem run. Our methodology is *adaptive* to the dynamic conditions of the Grid at run-time; however, it is not adaptive in the sense that the schedule is modified during application execution to adapt to changing Grid conditions.

Our *first scheduling goal* is to ensure that hard application resource requirements are met; for example, for an application with significant and inflexible memory requirements, our foremost concern would be to ensure that those memory requirements are met by the selected resource set. Our *second scheduling goal* is to minimize application execution time. This can be done by optimizing schedule performance based on an application performance model. When a performance model is not available, the scheduler can still make progress in schedule selection by evaluating the quality of each schedule based on heuristic definitions of resource set desirability.

Application model

We assume the target application is parallel and that many problem sizes of interest will require more than one machine for acceptable performance. We also assume a single program, multiple data (SPMD) application model.

Application resource requirements

In Section II.B we introduced the AART MODEL as a formal method for the specification of the type of resource set, or platform, that is likely to be performance-efficient for a particular application. Ideally, the GrADSoft SCHEDULER will handle smoothly any application by automatically parsing the application's AART and thereby understanding the application's resource needs. However, to achieve this ambitious goal, we need to gain experience with simpler versions of this problem. For this thesis, we focus on applications that share the following general characteristics.

i. All processes are able to communicate with all other processes, and the amount of communication performed by the application is significant. An appropriate resource set will therefore provide all-to-all connectivity. Resource sets connected by low-delay networks will provide better performance than those connected by high-delay networks and are therefore preferable.

ii. Application performance is sensitive to the aggregate computational and memory capacity of the target resource set, as well as to the individual capacities of selected machines.

iii. The definition of an appropriate resource set for the application is highly dependent on problem size, environmental characteristics, and other factors. The selection of an appropriate resource set size is therefore not a simple maximize or minimize function.

Application information and models

Recall that the GrADSoft architecture is designed to handle a wide variety of applications; we envision that application characteristics and models will often be derived from library annotations, compiler analysis, and records of historical behavior. Clearly, the sophistication of the resulting application information and models will vary greatly; the scheduler will have to adapt to varying degrees of sophistication of AARTs, performance models, and mappers. We propose a scheduler design that supports two types of application performance model.

- **Memory usage model**: At the coarser level, we assume that only an application memory usage model is available. Provided with problem run information such as problem size, this type of model returns a prediction of the aggregate amount of memory required for the application. We focus on a memory usage model in part because basic compiler analysis of application source code could fairly easily produce such a model. In particular, given some integration, we envision that such a model could be automatically produced by the GrADSoft COMPILER in the near future.

- **Execution time + memory usage model**: At the more sophisticated level, we assume that a full performance model is available. Provided with problem run information, the selected resource set, and a mapping of application tasks or data onto those resources, this type of model returns the predicted execution time in addition to a memory usage prediction. We selected this model as an investigation of how the GrADSoft system could function in the future. While current GrADS PPS technologies are not sophisticated enough to automatically generate an execution time model, this topic is a primary focus of other GrADS researchers [7, 29].

Note that we only specify here the *type* of models that are supported by the scheduler; the exact implementation of such a model is highly application-dependent and therefore can not be embedded in the scheduler design. In the next chapter, we develop an instantiation of the memory usage and execution time models for specific applications.

Grid environment

In this thesis we target Computational Grids consisting of heterogeneous, distributed networks of workstations. Workstations may have different processor types and speeds, different amounts of local memory, or different operating systems. While target workstations may have more than one CPU, the scheduler design currently targets only one CPU per resource. Targeted networks include both local-area networks (LANs) and wide-area networks (WANs). Figure III.1 provides an example of a small Grid of this type.

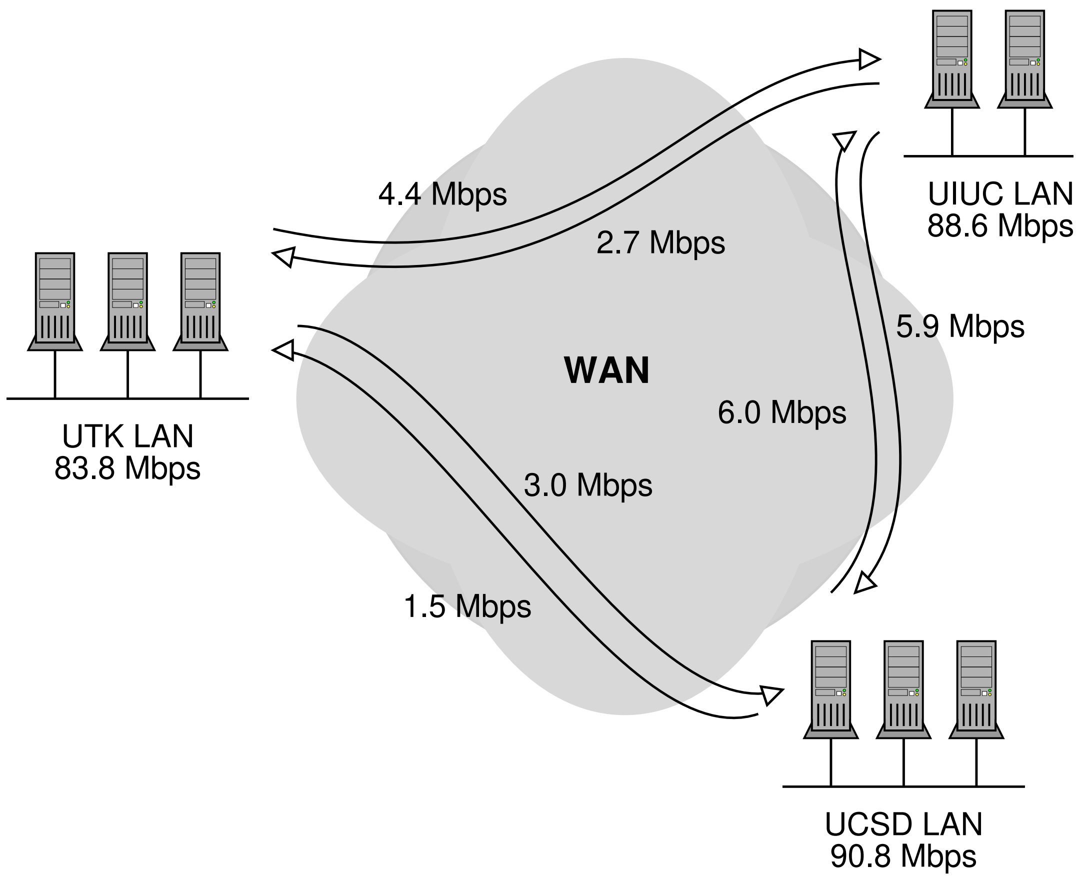


Figure III.1: A heterogeneous, distributed network of workstations. Network links are labeled with available bandwidth in megabits per second; these values were collected by Network Weather Service network monitoring sensors on November 1, 2001 at around 5:30 pm.

III.A.2 Scheduling definitions

The following definitions are used throughout the rest of this chapter to describe our scheduling methodology.

Base machine list The scheduling process begins with a list of all machines available for the current problem run. *Available* can have diverse meanings for different users and applications; in this thesis we define *available* as machines on which the user has an account, that are on-line and accessible, and that have certain Grid middleware services needed for job launching.

Resource pool The resource pool includes all of the machines in the base machine list as well as compute, storage, and network capability characteristics for these machines.

Site A site is a collection of well-connected resources and typically corresponds to a LAN. In practice, intra-site network delays are lower than inter-site delays. For example, Figure III.1 includes three distinct sites: {UCSD, UIUC, UTK}.

Topology-based collections A topology-based collection is a set of machines selected based on their locality. Specifically, given a set of sites, the corresponding topology-based collections can be found by taking the power set of the set of sites. [1] For the example given in Figure III.1, there are seven topology-based collections: {UCSD, UIUC, UTK, UCSD ∪ UIUC, UCSD ∪ UTK, UIUC ∪ UTK, UCSD ∪ UIUC ∪ UTK}. While the power set operation is exponential in the number of sites, Grid users often target a small number of sites.

Candidate resource group (CRG) A group of machines that have been identified as a possible resource set for the current problem run.

Schedule A list of resources and a mapping of data or tasks onto those resources.

Candidate schedule A particular candidate resource group and a mapping of data or tasks onto those resources constitute a candidate schedule.

Final schedule During the scheduling process a specific candidate schedule is eventually selected as the "best" choice, this schedule is then selected as the final schedule. The method by which schedules are compared to find the *best* one will be described in Section III.B.2.

[1] Note that we exclude the null set.

Virtual machine Once a final schedule is selected, information about the selected compute resources, network resources and data or task mapping is encapsulated in a virtual machine object.

III.B Scheduler design

The scheduler design is based on a core schedule search framework that supports a variety of pluggable components (Figure III.2). A base machine list is input to the Grid Info Collector which then retrieves characteristics of the machines and of the networks connecting them. The resulting resource pool is then input to the search procedure along with application characteristics and models in the COP. The scheduling policies object allows automatic configuration of a number of scheduling behaviors. For example, one supported scheduling policy defines whether the scheduler should include cross-site schedules in the search. By default, the scheduler considers cross-site solutions, but this policy could be inappropriate for applications that require a shared file system. After the scheduler has selected a final schedule, the virtual machine is created and returned.

In this section we describe the components of the schedule search procedure. In Section III.C we discuss the Grid info collector, and in Section III.D we enumerate the scheduling behaviors which can be configured via input scheduling policies.

III.B.1 Search Procedure

The scheduler search procedure is at the core of the scheduling methodology; the procedure examines the set of available resources, generates a number of candidate schedules, evaluates the candidate schedules to select a final schedule, and communicates the search results. The procedure takes as input the resource pool, the COP, and the scheduling policies. It outputs the selected schedule in the form of a virtual machine. To find reasonable candidate schedules, the search procedure identifies candidate resource groups (CRGs) and generates a schedule for each. The final schedule is the best of these candidate schedules. In Section III.B.2 we detail many of the individual components of this process. In this section, we examine only the process of generating candidate resource groups (CRGs) from the resource pool.

To guarantee that the optimal CRG will be identified, an exhaustive search over all possible unique resource combinations would be required. However, as we demonstrate momentarily, the cost of such a search is prohibitive. First, note that from the perspective of the scheduler, permutations of the same resource group do not constitute unique CRGs. A performance efficient resource ordering or

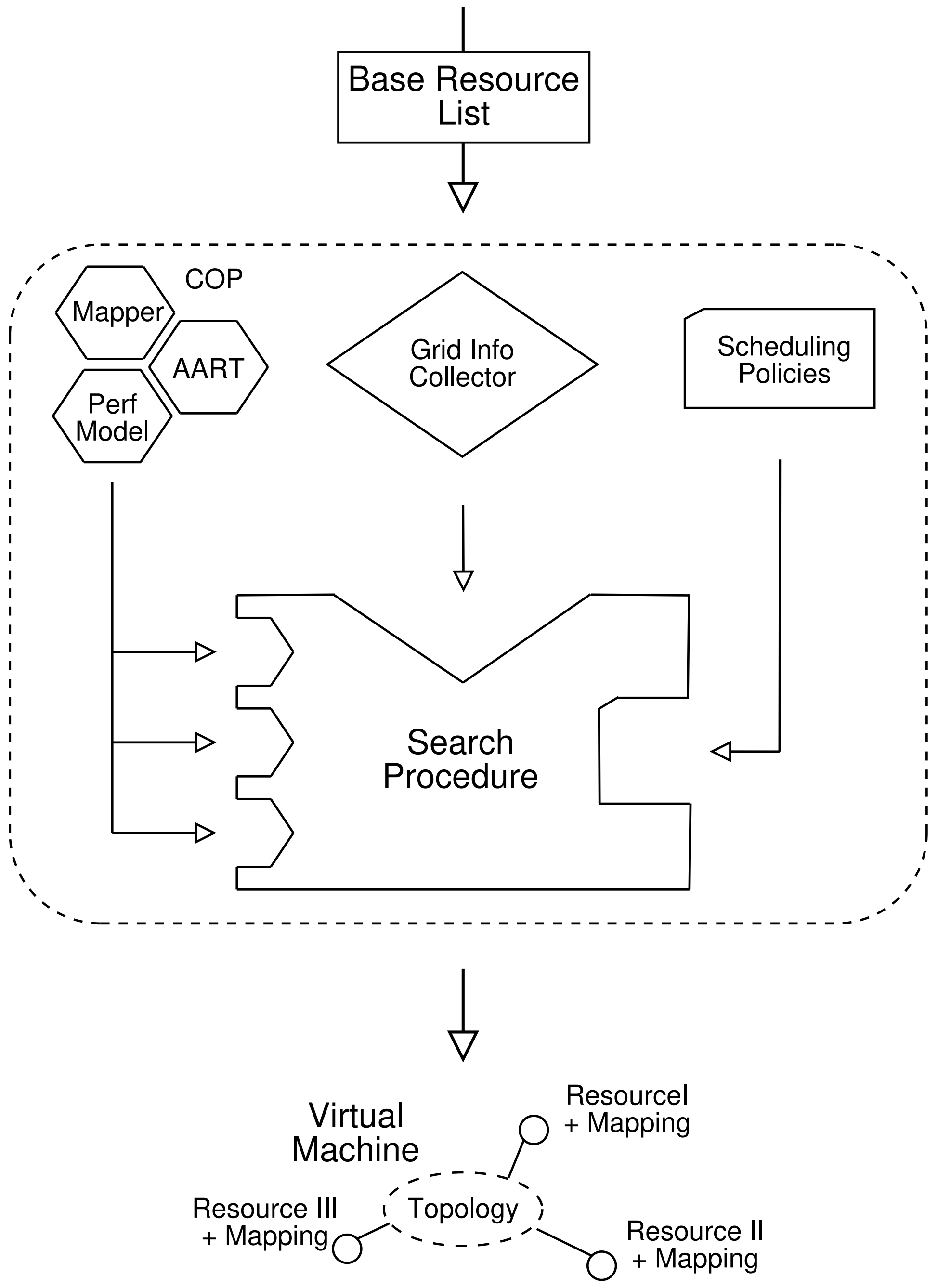


Figure III.2: Scheduler design.

topology is identified during the mapping process and is independent of initial resource ordering; thus, permutations typically result in equivalent schedules. For an exhaustive search, all subsets of size one to the size of the entire resource set must be included in the search. For a resource pool of size n, the number of distinct CRGs that must be included is:

$$numCRGs = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} \tag{III.1}$$

For example, to perform a schedule search in a resource set of 30 machines would require evaluation of $\sum_{k=0}^{30} \frac{30!}{k!(30-k)!} \approx 10^9$ CRGs. For even a reasonably sized resource pool and/or when the mapping process is time intensive the enormous size of the search space makes an exhaustive search simply infeasible.

The search procedure must therefore incorporate extensive pruning of the search space while ensuring that the optimal or near-optimal CRGs are not excluded from the search. Since performance models cannot be used until a candidate CRG has been identified, these models cannot be used to prune the space of possible CRGs. Recall that we address only applications that share certain broad resource requirements (see Section III.A). Our search approach is to prune resource groups that are unlikely to satisfy these requirements. The goal of the search is to ensure that the final list of candidate CRGs includes those CRGs which are likely to be performance efficient platforms for the application.

Pseudo-code for the schedule search procedure is given in Figure III.3. In each *for* loop we refine the list of target CRGs 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.

In the outer-most loop, we identify resource groups that are likely to be more tightly-coupled. We do this by explicitly defining collections of resources based on site topology in the FindSites and ComputeSiteCombos method calls. We discuss in detail our implementation of the FindSites and ComputeSiteCombos method calls in Section III.B.2; however, it is worth mentioning here why these resource groups are likely to provide better connectivity than other possible resource subsets. Since intra-site network delays are typically lower than inter-site delays, it is clear that resources within one site are likely to be more tightly-coupled that resources from multiple sites. However, it is also true that resource groups formed by combining the resources of individual sites are likely to exhibit better connectivity characteristics than randomly selected resource groups of the same size. The primary

```
Algorithm : ScheduleSearch($resourcePool$)

 $sites \leftarrow FindSites(resourcePool)$
 $topologyCollections \leftarrow ComputeSiteCombos(sites)$

 for each $collection$ ($topologyCollections$)

    for each $focus$ ($computation, memory, dual$)

       for $targetSize \leftarrow 1$ to $size(collection)$

          $CRG \leftarrow FindBest(collection, focus, targetSize)$
          $currSched \leftarrow GenerateSchedule(CRG)$
          if $ScheduleCompare(currSched, bestSched) == FirstIsBetter$
            $bestSched \leftarrow currSched$

 return ($bestSchedule$)
```

Figure III.3: Schedule search procedure. We discuss the overall search design in Section III.B.1 and the implementation of individual method calls in Section III.B.2

advantage posed by resource groups formed in this way is that such a resource group will typically span less sites than randomly selected resource groups containing the same number of resources.

In the middle loop of the search procedure we seek to locate resources that exhibit high local memory and computational capacities. Since we cannot know in advance which aspect will be more important for application performance, we define three different search *foci*: the *computation focus* emphasizes the computational capacity of machines, the *memory focus* emphasizes the local memory capacity of machines, and the *dual focus* places equal weight on each factor. The search for machines that satisfy these foci occurs in the FindBest method call; the implementation of this method call is discussed in Section III.B.2.

Finally, in the inner-most loop of our search procedure, we focus on the selection of an appropriately-sized resource group. It is difficult to determine what an appropriate resource set size will be for an application since it depends on application characteristics as well as resource characteristics. For example, for applications with substantial computation and communication, there is no way to know a priori whether a smaller, better connected resource group or a larger, poorly connected resource group will be more performance-efficient. Rather than attempt such a prediction, we include all resource set sizes in the search space. In this way, the selection of an appropriate resource set size is the responsibility of the ScheduleCompare method call; when an execution time model is available, ScheduleCompare can use this model to determine which CRG is most appropriate for application needs (see Section III.B.2 for details). Note that an exhaustive search at this level of the procedure is only feasible due to the extensive pruning performed at the first two levels.

This nested set of refinement methods greatly reduces the search space of CRGs. To demonstrate this we develop an **upper bound on the number of CRGs considered by the search heuristic**. Assuming we have s sites in the resource set under consideration, the process of defining site combinations creates 2^s topology-based collections.[2] We consider three resource orderings for each collection (*computation*, *memory*, and *dual*). Given these $3 * 2^s$ ordered collections, we exhaustively search all possible subset sizes for each. Since the number of resources in each site, and therefore in each topology-based collection, is dependent on the characteristics of each Grid environment, we can not predict a priori the number of resources in each of the $3 * 2^s$ ordered collections. Instead, we develop an upper bound by assuming each collection is of size n, the size of the entire resource pool. For each ordered collection, we assume n distinct subsets will be included. The upper bound on the total number of CRGs identified by the search procedure is therefore $3n2^s$. In contrast to an exhaustive search of the resource set space, the algorithmic complexity of our heuristic is exponential in the number of sites, not the number of computational resources. In the vast majority of scenarios, the number of sites is much smaller than the number of resources so that our heuristic can be expected to significantly reduce search cost.

Consider again the example presented earlier of 30 machines arranged in 3 sites with 10 machines in each site. Recall that the exhaustive search procedure produces 10^9 CRGs for this topology. The upper bound on the number of CRGs produced by our search procedure is $3 * 30 * 8 = 720$; a direct calculation of the number of CRGs generated for this scenario reveals that only 360 are generated.

III.B.2 Search methods

We now describe the algorithms used for each method call in Figure III.3. We discuss the methods in order of their usage in the schedule search procedure.

The **FindSites** method takes the list of available machines and organizes the machines into disjoint subsets such that the network delays within each subset are lower than the network delays between subsets. As a first approximation, our FindSites implementation utilizes a heuristic based on the ma-

[2]In fact, we exclude the null-set leaving $2^s - 1$ such collections.

chine domain names. Domain names are typically assigned to organizations and organizations are often geographically centralized. Additionally, the networks within an organization typically exhibit lower delays than networks between organizations. While these generalities are not true of every organization, they do hold for the GrADS Computational Grid environment and also for most Computational Grids we know of. Based on these assumptions, we group machines into a single site if they share the same domain name and into different sites if they have different domain names. Previous work in web client clustering [31] and content routing [26] have also utilized this heuristic approach to group machines in a similar way. Note that the method fails to distinguish hierarchies of machines within the same domain name. More sophisticated methods of network topology discovery exist [44, 36] and can be used to improve the ability of the FindSites method to discover true network topologies.

The **ComputeSiteCombos** method call takes as input the list of sites discovered via FindSites and builds a list of all possible topology-based collections. We exclude the null set so there are $2^s - 1$ such collections for a set of s sites.

The **FindBest** method takes as input a list of machines (*collection*), a machine type focus (*focus*), and a target resource set size (*targetSize*) and returns the best *targetSize* machines from the *collection* based on a machine preference of *focus*. For example, suppose the input collection contained two fast machines and two slow machines, that the *focus* is computation, and that the *targetSize* is three. In this case, the returned CRG will contain the two fast machines and the faster of the two slow machines.

The **GenerateSchedule** method call takes as input a list of machines and returns a schedule for those machines. Recall that a schedule consists of a list of machines and a data or task mapping onto those machines. Since the mapping process is application-specific, GenerateSchedule retrieves and uses the mapper from the COP. Note that due to constraints such as local machine memory capacities it is not always possible to find a feasible mapping; when this occurs, GenerateSchedule fails and the search for candidate CRGs continues. In the next chapter we present an example of an application-specific mapping strategy that clarifies the concept of GenerateSchedule and the mapper.

The **ScheduleCompare** method takes as input two candidate schedules and returns *FirstIsBetter* if the first schedule is better than the second and returns *SecondIsBetter* otherwise. Recall from Section III.A that we are investigating two sophistication levels for application information and models: a memory usage model and an execution time + memory usage model. The ScheduleCompare method uses different comparison metrics, described below, depending on which type of model is available.

Memory usage model without an execution time model

When only a memory usage model is available, a series of heuristics are used to compare candidate schedules. These heuristics are designed to evaluate how well the given schedules satisfy the broad application resource requirements defined in Section III.A. Figure III.4 provides an overview of the series of schedule comparisons used to select the better of two schedules.

Execution time model + memory usage model

When an execution time is available in addition to the memory usage model, ScheduleCompare uses the model to calculate a predicted execution time for each schedule. An obvious and straightforward approach is to select the schedule with the minimum predicted execution time and return it. A drawback to this absolute best selection methodology is that the selected schedule sometimes targets many more resources than are necessary to achieve acceptable performance. For example, many applications have poor speedup at larger resource set sizes; in these cases the performance advantage of adding more resources can be almost zero, yet an absolute best selection methodology will select larger and larger resource sets until performance actually degrades. In a shared-resource setting such as a Computational Grid, a better choice that balances the performance of individual applications with overall system throughput is to target a smaller number of resources that provide a similar performance level. Performance contracts, introduced in Chapter II, provide a formal specification of acceptable application performance levels; such a contract could be used to determine what target resource set sizes will provide acceptable application performance. Unfortunately, since performance contracts are still under development, we could not easily experiment with this concept.

As an alternative strategy that does not require specification of an absolute performance requirement, we introduce a *performance improvement threshold*. In a shared-resource environment such as the Grid, conservative resource usage should be encouraged to improve the overall system throughput; therefore when a smaller resource set provides *equivalent* performance to a larger set, a "good citizen" approach would target the smaller set. Since each user may have a personal definition of *equivalent*, we provide the *threshold* as a tunable scheduler option. We incorporate this strategy at the inner-most loop of the scheduler search procedure (Figure III.3); the loop is executed once to find the absolute best target resource set size and is then executed a second time to find the smallest resource set that provides a predicted execution time within the configurable performance improvement threshold.

III.C Grid information

Computational Grids are highly dynamic environments where compute and network resource availability can be unstable. When resource performance patterns are not well understood, application

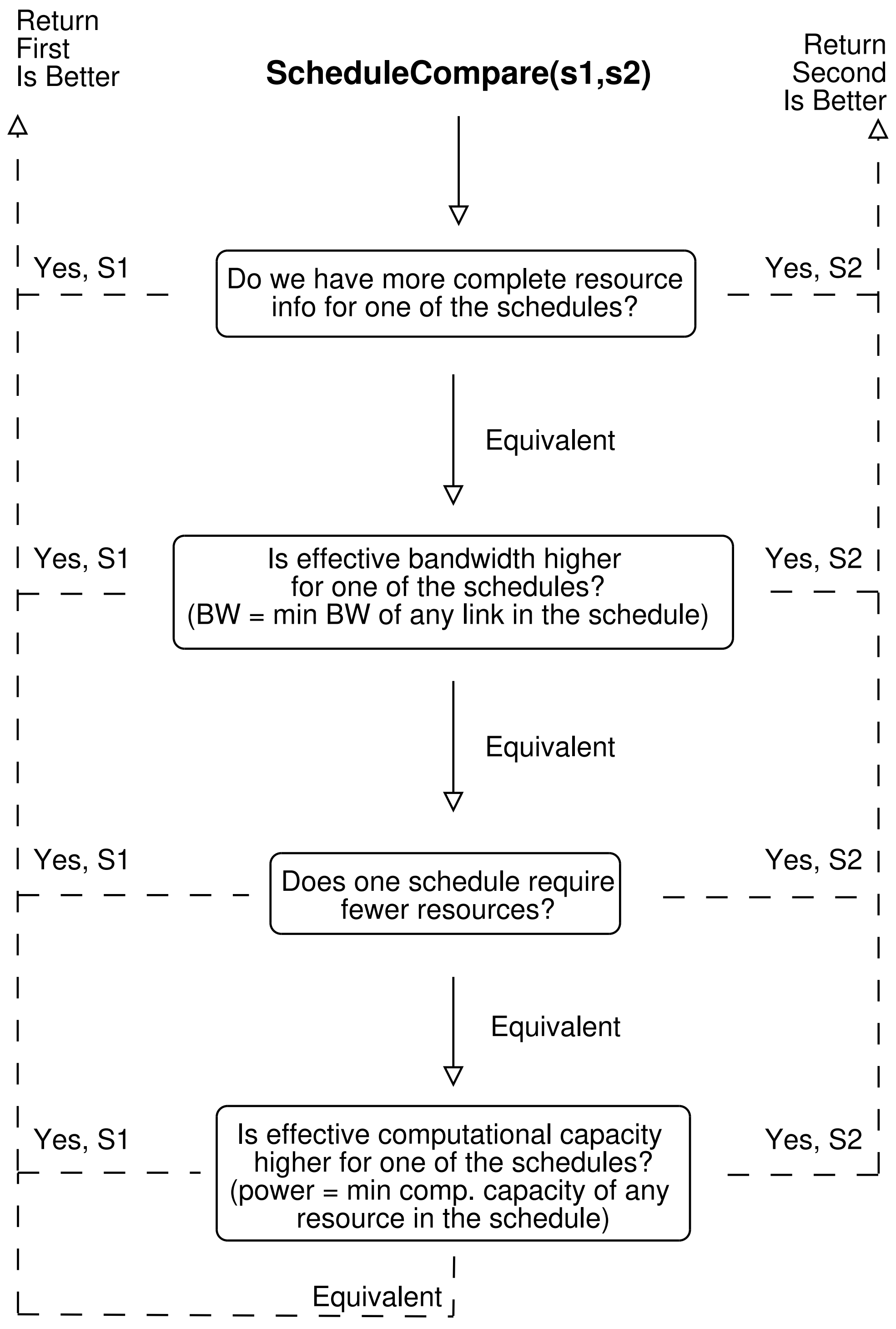

Return First Is Better
ScheduleCompare(s1,s2)
Return Second Is Better
Yes, S1
Do we have more complete resource info for one of the schedules?
Yes, S2
Equivalent
Yes, S1
Is effective bandwidth higher for one of the schedules? (BW = min BW of any link in the schedule)
Yes, S2
Equivalent
Yes, S1
Does one schedule require fewer resources?
Yes, S2
Equivalent
Yes, S1
Is effective computational capacity higher for one of the schedules? (power = min comp. capacity of any resource in the schedule)
Yes, S2
Equivalent

Figure III.4: Schedule comparison methodology when a memory usage model is available but an execution time model is not.

performance suffers. To avoid these problems, a scheduler should make decisions based on up-to-date information about the current Grid environment. Unfortunately, currently available Grid information sources can be periodically unstable or altogether unavailable, thereby precluding the use of any scheduler that depends solely on that information source. Our goal is to provide best-effort service by supporting backup sources, when possible, for each type of information required by the scheduler.

The *Grid Information Collector* is the scheduler component responsible for the collection of resource and network information for scheduling. In III.C.1 we discuss several Grid information sources that can be utilized by the Grid Information Collector. In III.C.2 we describe the types of information required by our scheduling strategy and identify which information sources can be used to satisfy each of these requirements.

III.C.1 Information sources

There are a variety of Grid information collection and dissemination mechanisms and each Computational Grid varies somewhat in its information infrastructure. We provide an overview here of two of the most widely utilized systems, the Metacomputing Directory Service (MDS) [13, 11] and the Network Weather Service (NWS) [57, 58]. See Section V.A for the configuration and usage of these two systems in the GrADS Computational Grid.

The **MDS** is a flexible Grid information management system that is used to collect and publish system configuration, capability, and status information. Essentially, any non-sensitive information that can be retrieved from an operating system could be published in the MDS for retrieval by distributed clients. Grid characteristics that can typically be retrieved from the MDS include

- the set of potentially available resources;
- machine characteristics such as operating system, processor type, processor speed, number of CPUs available, and physical memory size; and
- software availability and installation location.

The **NWS** is a distributed monitoring system designed to track current resource and network conditions [57, 58]. In addition to providing near real-time estimates of deliverable performance, the system supports very short-term (10 seconds ahead) forecasting of future availability. The system supports monitoring of the following system characteristics:

- *availableCpu*: the fraction of CPU available to a newly-started process,

- *currentCpu*: the fraction of CPU available to a process that is already running,
- *freeMemory*: the amount of space unused in memory,
- *freeDisk*: the amount of space unused on disk,
- *connectTimeTcp*: the amount of time required to establish a TCP connection to a remote host,
- *bandwidthTcp*: the speed with which data can be sent to a remote host, and
- *latencyTcp*: the amount of time required to transmit an empty TCP message to a remote host.

III.C.2 Scheduler Grid information requirements

There are several types of Grid information required for the scheduling process: a list of machines available for the run, local computational and memory capacity estimates for each resource, and connectivity estimates for the networks between resources. Wherever possible we support all available types of information.

The base machine list is a list of the machines to be considered in the scheduling process and is the most important information requirement. On some testbeds a list of all machines in the testbed can be retrieved from an MDS, but unfortunately this list typically includes machines on which the user does not have accounts. The currently deployed MDS technology provides no support for secure publishing of account information, which is generally too sensitive to publish in an insecure way. However, secure MDS mechanisms are currently in the beta stage and should be available in the near-term. Until that technology is available, we obtain the list of available machines from the user directly.

Local memory capacity information is utilized by the scheduler in two ways. First, the information is used by the **FindBest** method call to sort machines when the focus parameter is *memory* or *dual*. Second, most reasonable mappers will require local capacity information to ensure that the local application requirements do not exceed local capacities. The scheduler supports usage of either total physical memory values, which can be retrieved from the MDS, or free memory values, which can be retrieved from the NWS. In future discussions we abbreviate these information types by TOTAL and FREE, respectively. Note that only machines for which local memory information is available are included in the final schedule.

Local computational capacity information is also utilized by the scheduler in two ways. As with local memory, the **FindBest** method uses computational capacity information to sort machines

Component	Parameter	Options	Default
Grid Info Collector	usergrid	{MACHINE_LIST}	no default
	memType	FREE, TOTAL	FREE
	compType	AVAIL, MHZ, AVAIL_MHZ	AVAIL_MHZ
	nwsDataType	LAST_VALUE, PRED	PRED
Search Procedure	perfModel	MEMORY, EX_TIME	EX_TIME
	resConserve	$[0.0, \infty)$	0.05
	crossSiteOK	YES, NO	YES

Table III.1: Configurable scheduling policies.

when the focus parameter is *computation* or *dual*. In addition, many mappers and performance models will incorporate performance predictions requiring computational capacity estimates. The scheduler supports usage of three information types: processor speed (available from the MDS), available CPU estimates (available from the NWS), or available processor speed (computed by the Grid Information Collector as a multiplicative combination of available CPU and processor speed). Recall that the scheduler design currently only targets one CPU per resource; NWS CPU availability values can be well over 100% for multi-processor resources so we therefore cap all NWS CPU availability estimates at 100%. Hereafter we abbreviate these three information types by MHZ, AVAIL, and AVAIL_MHZ.

Network performance information is used primarily by the scheduler to provide input to the mapper and performance model. For example, many mappers will incorporate performance predictions that require network characteristics. The scheduler supports usage of bandwidth and latency data, both of which are available from the NWS. In future discussions, these will be abbreviated as BAND and LAT.

III.D Scheduling policies

In this chapter we have discussed many configurable scheduler characteristics. For each of these characteristics a default value is defined; to support flexibility in scheduler usage the scheduler also supports the specification of alternative configurations via the scheduling policies scheduler input. A summary of scheduler configuration characteristics, available options, and default values is given in Table III.1.

Only one of the scheduling policies must be specified: the user must provide a list of Grid resources on which he or she has an account, hereafter called *usergrid*. As the security of Grid information services improves this requirement will be lifted.

III.E Chapter summary

In this chapter we have described a modular, adaptable scheduler design that can be applied to many types of applications. This scheduler must be paired with an application-specific performance model and mapper. In the next chapter we develop these important components for a specific application class.

Chapter IV

Iterative, mesh-based applications

In Chapter III we described an adaptable scheduling framework that, when combined with an application-specific performance model and mapper, provides automatic matching of application requirements with available Grid resources. In this chapter we describe two specific applications and detail an implementation of the mapper and performance model for each. The performance models, mappers, and applications themselves will be used in the next chapter to demonstrate our scheduling methodology in validation experiments.

For our test applications we have chosen two examples from the class of iterative, mesh-based applications. In Section IV.A, we describe the characteristics of this application class and detail the test applications. Section IV.B describes our performance model design, and Section IV.C describes our mapper design. Finally, in Section IV.D we provide a chapter summary.

IV.A Application characteristics

The class of iterative, mesh-based applications is critical to many fields of science and engineering, including for instance particle simulations, partial differential equation solvers, and circuit simulations [21]. We have chosen to focus on applications in this class both because the class is important and because this class typically exhibits relatively predictable performance.

Iterative applications are characterized by a single sequence of operations that is repeated many times over the course of execution. Many iterative applications can be classified as *loosely synchronous*, meaning that in each iteration the participating processors are synchronized in some way before continuing to the next iteration [21]. We focus on loosely synchronous iterative applications where the data domain can be represented by a one, two, or three-dimensional mesh. We also assume that the workload per iteration and the workload per unit of the data mesh are both constant (traditionally labeled a *regular iterative model*).

Many iterative, mesh-based applications are discrete approximations to continuous space-time problems; examples include binary cellular automatons [14], atmospheric simulations [15], and heat transfer in a solid. For these applications, each iteration represents a small, fixed period of time and the data mesh represents the physical space for the problem. Linear system solvers are another application domain with many iterative, mesh-based solutions [6]; examples include Gauss-Seidel, Successive Over-relaxation, and Conjugate Gradient.

We focus on two of these applications for development and testing purposes: **Game of Life** and **Jacobi**. We have selected these applications as our initial test cases because they are well-known, straightforward to describe, and share many performance characteristics with other iterative, mesh-based applications. We implemented each test application as a SPMD-style computation using C and the Message Passing Interface (MPI) [38]. Traditionally, MPI programs are restricted to groups of machines that share a file system and have similar architectures. However, the MPICH implementation of MPI [27, 28] provides support for execution of unmodified MPI programs across heterogeneous architectures and wide-area networks; specifically, this support is provided through a Globus-enabled version of MPICH called MPICH-G [16, 17]. We use MPICH-G for each of our applications to allow experimentation across Computational Grids.

IV.A.1 Game of Life

Conway's Game of Life is a well-known binary cellular automaton whereby fixed rules are applied to determine a next generation of cells based on the state of the current generation [14]. A two-dimensional mesh of pixels is used to represent the environment, each pixel represents a cell, and values of 0 and 1 indicate a dead and living cell, respectively. In each iteration, the state of every cell is updated based on the current status of the cell itself and of its eight nearest neighbors (a *9-point stencil*); a dead cell with exactly three live neighbors comes alive and living cells survive only if they have two or three living neighbors.

Game of Life is a straightforward application to implement in parallel because the update of each pixel depends only on the values of the cell's immediate neighbors. Data can be partitioned in numerous ways including block, block-cyclic, row-based strip, and column-based strip. Each processor manages a portion of the array and defines a 1-pixel wide set of *ghost-cells* along data grid edges. Each iteration consists of a computational phase in which each processor updates their portion of the data array and a communication phase in which each processor re-initializes their ghost cell data with information from its neighbors.

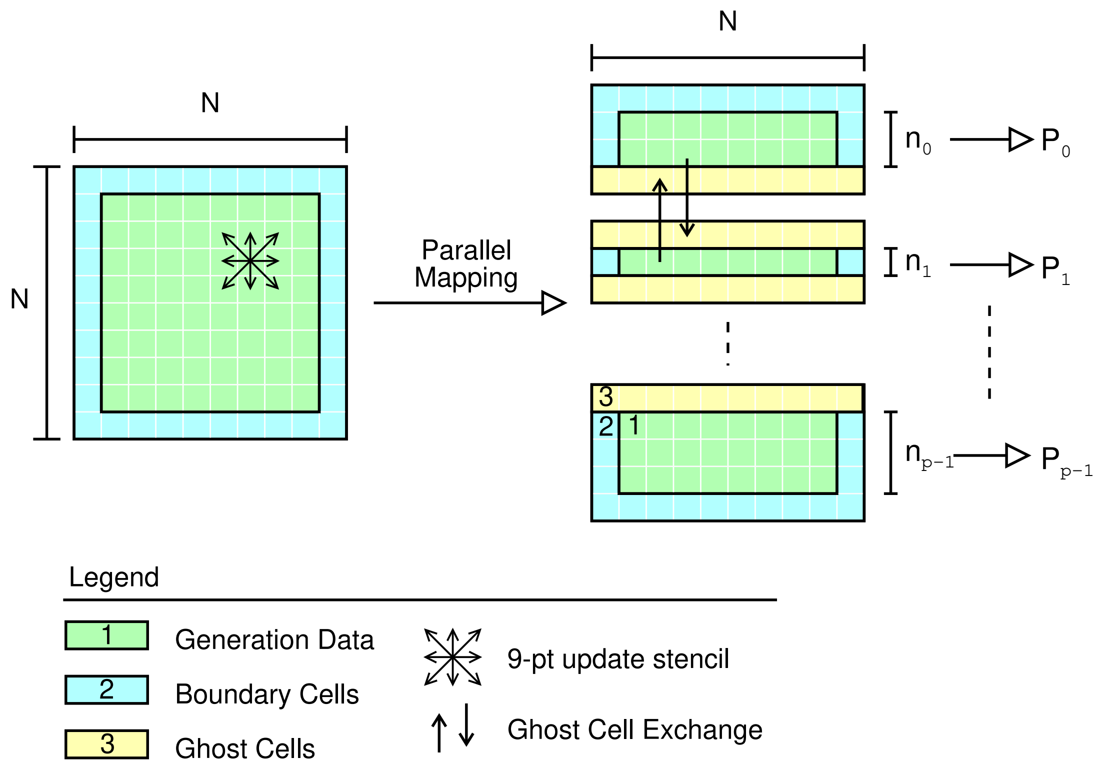


Figure IV.1: Game of Life application structure.

Key features of our Game of Life implementation are diagrammed in Figure IV.1, a listing of variables and their definitions is given in Table IV.1, and pseudo-code for the iterative portion of our implementation is presented in Figure IV.2. We selected a row-based strip data partitioning strategy because this strategy typically exhibits lower communication costs than other partitioning schemes, an important consideration for Grid computing. To allow experimentation with load-balancing work allocation strategies, we incorporated support for irregularly-sized data partitions. The ghost cell exchange phase is implemented with non-blocking sends (MPI_Isend), non-blocking receives (MPI_Irecv), and final wait method calls to ensure the return of all communication calls (MPI_Wait). Our implementation does not include termination detection, and it executes for a configurable, but fixed number of iterations.

```
**Algorithm :** GameOfLife($procID, p, N, nLocal, numIts$)

**define** $DEAD$ 0, $ALIVE$ 1
**define** $FIRST\_ID$ 0, $LAST\_ID$ $p-1$

// Arrays must include space for boundary and ghost cells
**local** $currGen[nLocal+2][N]$, $nextGen[nLocal+2][N]$
$currGen \leftarrow GetInitialGeneration(i, nLocal)$

**for** $k \leftarrow 0$ **to** $numIts - 1$

  // COMPUTATION PHASE
  // We exclude boundary values from update
  // i.e.  row 0, row nLocal+1, col 0, and col N+1
  **for** $r \leftarrow 1$ **to** $nLocal$
    **for** $c \leftarrow 1$ **to** $N$
      $count \leftarrow currGen[r-1][c-1] + currGen[r-1][c] +$
              $currGen[r-1][c+1] + currGen[r][c-1] +$
              $currGen[r][c+1] + currGen[r+1][c-1] +$
              $currGen[r+1][c] + currGen[r+1][c+1]$
      **if** $currGen[r][c] == ALIVE$ **and** $count \in \{2, 3\}$
        $nextGen[r][c] = ALIVE$
       **else if** $currGen[r, c] == ALIVE$
        $nextGen[r][c] = DEAD$
       **else if** $currGen[r, c] == DEAD$ **and** $count == 3$
        $nextGen[r][c] = ALIVE$
       **else**
        $nextGen[r][c] = DEAD$

  // COMMUNICATION PHASE
  **if** $procID \neq FIRST\_ID$
    // Re-initialize ghost cell data with lower-indexed proc
    $AsyncSend(nextGen[1][:], procID - 1)$
    $AsyncRecv(nextGen[0][:], procID - 1)$
    $WaitAll()$
  **if** $procID \neq LAST\_ID$
    // Re-initialize ghost cell data with higher-indexed proc
    $AsyncSend(nextGen[nLocal][:], procID + 1)$
    $AsyncRecv(nextGen[nLocal+1][:], procID + 1)$
    $WaitAll()$

  // Swap data array pointers [no mem copy]
  $SwapDataPtrs(currGen, nextGen)$

**return** $(currGen)$
```

Figure IV.2: Game of Life application pseudo-code.

Name	Type	Definition
p	int	total number of processors
P_i	NA	processor i where $0 \leq i < p$
N	int	number of rows & columns in the global data matrix
n_i	int	number of rows in the local data matrix of P_i
numIts	int	total number of iterations to perform
procID	int	MPI processor ID where $0 \leq$ procID $< p$
FIRST_ID	int	ID of the top-most processor (usually 0)
LAST_ID	int	ID of the bottom-most processor (usually $p-1$)
currGen	int[ ][ ]	2-dim matrix for initial data in each iteration
nextGen	int[ ][ ]	2-dim matrix for result data in each iteration

Table IV.1: Summary of variables used in Game of Life description.

IV.A.2 Jacobi

The Jacobi method is a simple algorithm for the solution of a system of linear equations by iteration [6, 41]. Jacobi is often explained in the context of solving Laplace's equation. We instead describe the general linear system solver version; note that the Jacobi method involves more communication for this case than in the solution to Laplace's equation (which requires only neighbor-based communication). A linear system of equations can be represented as $ax = b$ where a is a square N x N matrix of coefficients, $x = (x_0, x_1, ..., x_r, ..., x_{N-1})^T$ is a vector of unknowns, and $b = (b_0, b_1, ..., b_r, ..., b_{N-1})^T$ is a vector containing the constant coefficients. To avoid confusion with the notation we use for processors (P_i), in this discussion we use r as a shorthand for row number and c as a shorthand for column number. The method begins with an initial guess for the solution vector x^0 and in each iteration successive approximations x^k, $k = 1, 2, ..., numIts - 1$ to the solution are calculated. The value of x_r in iteration number $k + 1$ is given by:

$$x_r^{k+1} = \frac{1}{a_{rr}}(b_r - \sum_{c \neq r} a_{rc} x_c^k) \tag{IV.1}$$

This solution method is guaranteed to converge only if the system is diagonally dominant. That is, convergence is guaranteed if:

$$\forall r \in \{0 : N-1\}, |a_{rr}| > \sum_{c \neq r} |a_{rc}| \tag{IV.2}$$

Name	Type	Definition
p	int	total number of processors
P_i	NA	processor i where $0 \leq i < p$
N	int	number of rows (unknowns) in the vector x
n_i	int	number of rows of x assigned to P_i
$numIts$	int	total number of iterations to perform
$localA$	double[ ][ ]	2-dim matrix with n_ixN sized local portion of a
$localB$	double[ ]	vector with n_i sized local portion of b
$r0$	int	global location of the first row of this proc's unknowns
$currX$	double[ ]	initial values for x in each iteration
$nextX$	double[ ]	result values for x in each iteration
$localX$	double[ ]	result values for local portion of x
$distance$	double	a metric for convergence detection

Table IV.2: Summary of variables used in Jacobi description.

We have chosen Jacobi because it is straightforward to describe and shares many performance characteristics with other, more popular algorithms.

An efficient parallel data decomposition for the Jacobi method is to assign a portion of the unknowns to each processor. The vector x is then decomposed into an assignment of work ($map = (n_0, n_1, ..., n_i, ..., n_{p-1})$) to processors ($procList = (P_0, P_1, ..., P_i, ..., P_{p-1})$). Each processor need only store a rectangular sub-matrix of a of size n_i x N. Each iteration begins with every processor computing new results for their n_i-sized portion of the unknowns. Next, each processor must distribute their updated portion of x to every other processor so that at the end of the communication phase every processor has a fully updated x vector. The final phase in each iteration is a termination detection phase. The method is *stationary*, meaning that the matrix a is fixed throughout the application; for this reason, each processor need only know the values in its sub-matrix of a. The Jacobi method does not require ghost cells or updates for the matrix a.

Pseudo-code for our implementation of the Jacobi method is given in Figure IV.3, and Table IV.2 provides definitions for variables referenced in the pseudo-code.

IV.B Application performance modeling

As discussed in Chapter III, the scheduler is dependent on the availability of a performance model in the form of either a memory usage model or an execution time + memory usage model. In this section we develop an instantiation of each of these performance models for our test applications, Jacobi and

```
**Algorithm :** JACOBI($localA, localB, p, N, nLocal, r0, numIts$)

 local $currX[N]$, $nextX[N]$, $localX[nLocal]$
 local $distance$

 // Initialize currX with b values
 $currX[r0 : r0 + nLocal] \leftarrow bLocal$
 for $i \leftarrow 0$ to $p - 1$
   $Broadcast(currX, i)$

 for $k \leftarrow 0$ to $numIts - 1$

    // COMPUTATION PHASE
    for $r \leftarrow 0$ to $nLocal - 1$
       $localX[r] \leftarrow localB[r]$
       for $c \leftarrow 0$ to $N - 1$
          if $c \neq r + r0$
             $localX[r] \leftarrow localX[r] - localA[r][c] * currX[r + r0]$
       $localX[r] \leftarrow localX[r] \;/\; localA[r][r + r0]$

    // COMMUNICATION PHASE
    $nextX[r0 : r0 + nLocal] \leftarrow localX$
    for $i \leftarrow 0$ to $p - 1$
      $Broadcast(nextX, i)$

    // TERMINATION DETECTION
    // Application runs for fixed numIts, but we include
    // code to ensure realistic performance results
    $distance \leftarrow 0$
    for $r \leftarrow 0$ to $N - 1$
      $distance \leftarrow distance + (nextX[r] - currX[r])^2$
    $distance \leftarrow \sqrt{distance}$

    // Swap data pointers [no mem copy]
    $SwapDataPtrs(currX, nextX)$

 return $(currX)$
```

Figure IV.3: Jacobi application pseudo-code.

the Game of Life.

IV.B.1 Memory usage model

When the active, local data set of an application does not fit in the physical memory of each participating processor, application progress can grind nearly to a halt due to paging of memory to disk. Since the performance effects of memory usage can be quite severe, it is very important that schedulers not allocate more work to each processor than will fit in local memory. However, it is difficult to predict the performance effects of application memory usage. For these reasons, we do not include memory usage as part of a performance metric, but rather as a *schedule constraint*: for a schedule to be considered *feasible* the work allocated to every processor must be predicted to fit within the local memory capacity of that processor.

Based on examination of data structures allocated in each application, we predict the *memUnit* bytes of storage that will be allocated per pixel of the data mesh. The aggregate memory requirement, *aggMemReq*, is then dependent on the problem size, *memUnit*, and on d, the number of dimensions in the data mesh. In megabytes, the aggregate memory requirement can be written:

$$aggMemReq = \frac{memUnit * N^d}{2^2 0}. \tag{IV.3}$$

To predict an application memory requirement for processor P_i, we incorporate the size of the local data partition, n_i:

$$memReq_i = \frac{memUnit * n_i * N^{d-1}}{2^2 0}. \tag{IV.4}$$

Recall that data for local processor memory availability, mem_i, can be supplied by total physical memory values from the MDS or free memory values from the NWS. Theoretically, a simple comparison of $memReq_i$ to mem_i should be sufficient to determine if the application's memory requirements are satisfied by a machine's local memory. In practice, a close match of the two factors provides an overly "tight" fit and the application's memory demand will frequently exceed local memory capacity. There are several reasons for this effect:

- While most memory allocation by these applications is for the data arrays, every application requires some additional static memory allocation. These memory requirements are not considered explicitly in our memory usage model.

- The operating system and other background processes require a significant portion of the available physical memory. When mem_i is based on total physical memory, this contention is not considered.
- All of our target machines are time-shared. If we run a job that requires all of the local memory, then when another user runs even a small job our job will be swapped out to disk.

To avoid this problem, we propose an alternative: a given work allocation should only be considered feasible if each processor provides some additional memory over the amount specified by the memory usage model. We incorporate a tunable parameter in the memory usage model called the *memFactor*; this factor specifies the percentage of additional memory that should be available on each processor and in aggregate. The user of the model must select a value that provides a reasonable tradeoff given the target resource set; lower values will improve the chance of finding a reasonable resource set while higher values will reduce the chance of the application's resource demands exceeding the capacity of the targeted resources. Based on early experimental results and memory usage benchmarks, we identified 20% as a value that provides a reasonable tradeoff for the GrADS Computational Grid environment; for all experimental studies presented in Chapter V the memory usage model is configured with this value.

To fully instantiate this memory model, we need to determine appropriate *memUnit* and d parameters for each test application. In the **Game of Life** application, a two-dimensional integer array of size n_i x N is allocated for both the *currGen* matrix and the *nextGen* matrix. For all discussions in this thesis we assume that four bytes of storage are required for each integer and eight bytes of storage are required for each double; while this assumption is accurate for all machines currently in the GrADS testbed, to accurately handle a broader variety of architectures the storage requirements should be treated as inputs to the memory model. Overall, the Game of Life parameters are $d = 2$ and $memUnit = 2\ ints/pixel * 4\ bytes/int = 8\ bytes/pixel$. For the **Jacobi method**, the primary data allocation for each processor is the 2-dimensional array of doubles of size n_i x N for the local component of a. Each processor also allocates a size N vector for $currX$ and $nextX$ and a size n_i vector for $localX$ and $localB$. Since the size of these vectors will be much smaller than the a matrix, we focus only on the matrix memory allocation. Therefore, for the Jacobi method, $d = 2$ and $memUnit = 1\ double/pixel * 8\ bytes/int = 8\ bytes/pixel$.

IV.B.2 Execution time model

Iterative, mesh-based applications share a number of characteristics that enable relatively straightforward performance modeling. Recall that we assume a regular, synchronous iteration model. This model allows us to make a number of additional *assumptions*:

1. The iterative phase dominates execution time.

2. The computational cost per iteration is constant for each processor.

3. The progress of the slowest machine defines the progress of all participating machines.

Based on these assumptions, the execution time of the application will be proportional to the application iteration time. Furthermore, due to the synchronization of processors in each iteration, the application iteration time will be equal to the time of the slowest participating processor. If we represent the predicted iteration time on processor i as $itTime_i$ and the overall predicted application iteration time as $itTime_{app}$ this metric can be formalized as:

$$itTime_{app} = max\{itTime_0, itTime_1, ..., itTime_{p-1}\}. \tag{IV.5}$$

Since iteration time is independent of the number of iterations performed it is slightly simpler to model and discuss than execution time; for this reason, we compare schedules based on the following *schedule evaluation metric*: the most desirable schedule is the schedule with the lowest predicted iteration time. Note that for a schedule to be considered feasible, it must still satisfy the memory usage model schedule constraint.

For iterative, mesh-based applications, iteration time is typically dominated by one or more phases. For example, the Game of Life is dominated by a computation phase (updating the value of each cell) and a communication phase (sharing of ghost cells with neighbors). For each of our test applications, these phases are serialized for each processor (i.e. we have not implemented overlap of communication and computation). We can therefore model the iteration time on processor i ($itTime_i$), as a combination of that processor's computation time ($compTime_i$) and communication time ($commTime_i$):

$$itTime_i = compTime_i + commTime_i. \tag{IV.6}$$

Since we assume a regular iterative application model, we can expect that iteration time is relatively constant during application execution. Furthermore, since we assume that execution time is dominated by the iterative phase, the full execution time is proportional to the iteration time.

Note that our implementation of Jacobi includes a termination detection phase, but this phase involves only computation and thus can be included directly in the $compTime_i$ model. In the following paragraphs we describe a model for the computation time and communication time for each processor. Unless otherwise noted, the units of time are seconds.

Computation

The computation phase for our test applications primarily consists of the pixel update process in each iteration. For the Jacobi implementation, the termination detection phase is purely computational and so we consider it part of the computation phase as well.

As we did in the memory model, we introduce a base unit of computation: *compUnit*. The *compUnit* is the number of processor cycles performed by the application per pixel of the mesh per iteration. Recall that the computational capacity of a processor, $comp_i$, can be represented by the CPU speed (MHZ) or by the available CPU speed (AVAILMHZ), each of which has units of 10^6 cycles per second. The computation time per iteration on processor i can then be modeled as

$$compTime_i = \frac{compUnit * n_i * N^{d-1}}{10^6 * comp_i}. \tag{IV.7}$$

To fully instantiate this model we need to determine an appropriate *compUnit* value for each test application. Unfortunately, obtaining an accurate count of the number of cycles required for a code segment is non-trivial. One simple method is to examine the source code and manually count operations; this method typically over-predicts operation counts because it does not account for compiler optimizations or the effect of caching. Another option is to base the operations count estimate on assembly code; this method accounts for compiler optimizations but is also inaccurate because (1) it assumes the processor is able to sustain a fixed rate in cycles per operation throughout execution and (2) it fails to identify variations in operation counts by architecture. We instead utilize an empirical approach to determine a *compUnit* value for each application. We selected a range of problem sizes and ran the computational portion of the application on each processor configuration included in our testbed. These tests were run in unloaded conditions and 100 iterations were performed per run. We calculate an average iteration time, *itTimeAv*, and use the processor speed (MHZ) to convert to cycles per pixel per iteration: $compUnit = MHZ * itTimeAv * 10^6$. Finally, for each application we average the *compUnit* values determined for each problem size and generate a final application *compUnit* value. For the processor configurations targeted in this thesis, the *compUnit* values determined in this manner were relatively similar; for example, the Game of Life benchmarks resulted in average *compUnit* values ranging from 68.7 to 74.8. We therefore selected a single value and used it in all experiments; for the Game of Life we selected $compUnit = 72$, and for Jacobi we selected $compUnit = 36$. For a testbed with greater processor configuration heterogeneity, the scheduler should utilize a different value for each processor configuration.

Communication

In this section we describe a model of the communication behavior for the Game of Life and Jacobi. Our model is based primarily on the cost of message transfers; we do not model additional overheads such as synchronization.

The **Game of Life** communication phase consists of a swap of ghost cells between processors. For all processors except the lowest indexed processor in the data decomposition (i.e. P_0), a message is sent to and received from the $i-1$ processor. Similarly, for all processors except the highest indexed processor in the data decomposition (i.e. P_{p-1}), a message is sent to and received from the $i+1$ processor (see Figure IV.1). These messages each include N pixels of information, each pixel is represented by an integer, and we assume each integer requires four bytes of storage. The size of each message, $msgSize$, is therefore $4 * N$. Since our Game of Life implementation uses non-blocking sends and receives, all of the messages in each iteration could theoretically be overlapped. In practice, however, processors can not simultaneously participate in four message transfers at once without a reduction in performance for each message and, more importantly, processors do not reach the communication phase of each iteration at the same moment. As an initial approximation, we assume that messages with a particular neighbor can be overlapped, but that communication with different neighbors occurs in distinct phases which are serialized. Since we assume that communications with a particular neighbor can be overlapped, we assume that the cost of each such communication is determined by the maximum predicted delay of the two involved message transfers. Note that while these messages are between the same two hosts, the delay of each message must be determined independently because network performance for sending and receiving to and from the same host can be asymmetric. If we represent the time to send a message from processor a to processor b as $msgTime_{a,b}$, then the communication cost for each processor can be calculated as shown in Figure IV.4.

In the **Jacobi** communication phase, each processor broadcasts their n_i sized portion of the vector x to every other processor. For MPICH, the MPI implementation used in this thesis, a broadcast is

```
Algorithm : CalculateCommGOL($procID, p$)

 define $FIRST\_ID$ 0
 define $LAST\_ID$ $p - 1$

 $commTime_i = 0$
 if $procID \neq FIRST\_ID$
   $commTime_i = max(msgTime_{i,i-1}, msgTime_{i-1,i})$
 if $procID \neq LAST\_ID$
   $commTime_i += max(msgTime_{i,i+1}, msgTime_{i+1,i})$

 return ($commTime_i$)
```

Figure IV.4: Game of Life communication cost calculation.

composed of individual MPI_Send and MPI_Recv calls. The broadcast begins at the root node (the root is whichever processor is the initiator of the broadcast), and is sent to all other processors via a binomial tree [5]. The binomial tree broadcast structure is designed to minimize the number of serialized messages that must proceed before the broadcast is complete; theoretically at most $log_2(p)$ messages are serialized in each broadcast. Figure IV.5 illustrates a binomial tree broadcast structure for seven processors with P_0 as the root node.

Since there are p broadcasts per iteration we assume that $p * log_2(p)$ messages are sent per iteration. Due to the (possibly) irregular data partitions, each processor's broadcast could be of a different size. Additionally, since the root node will be different for each broadcast in an iteration, the connections involved in the broadcast will vary from broadcast to broadcast. One modeling approach is presented in [5]; in this work the authors propose directly calculating the cost of each path in the binomial broadcast tree to determine the longest path, which is then taken to be the predicted broadcast time. This methodology has not been tested on heterogeneous, wide-area resources sets and it does not address many factors that can impact broadcast time (e.g. wait times, message overlaps, network contention). As a first approach, we decided to use a more efficient and simpler communication model. We calculate an average message cost, $msgTime_{avg}$, and then approximate the communication cost as:

$$commTime_i = p * log_2(p) * msgTime_{avg}. \tag{IV.8}$$

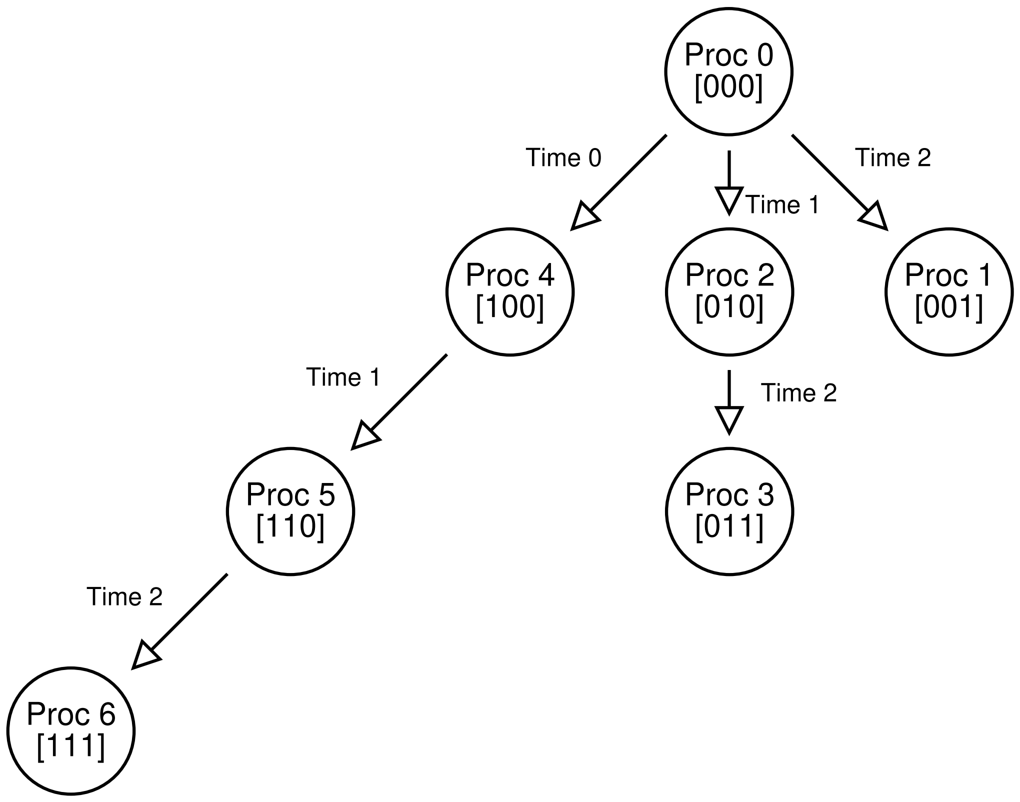


Figure IV.5: Example of a binomial tree as used in the MPICH broadcast implementation. Circles represent processors and arrows represent messages. Processor IDs are also listed in binary to illustrate the way in which the tree is built.

To develop a value for $msgTime_{avg}$, we first assume messages will all contain an average number of mesh pixels, N/p. Each pixel contains a double and we assume that each double requires eight bytes of storage; therefore, we assume all messages will be of size $8N/p$. We also assume that within the selected resource set, the usage of any processor to processor link is equally likely. Then, we individually calculate the cost of sending a size $8N/p$ message from each processor to every other processor and, finally, we take the average. We believe that this methodology provides a good tradeoff between model complexity and model accuracy.

It still remains to develop a model for the **cost of sending a message** between two processors. Suppose the startup time for sending a message from a to b is $\alpha_{a,b}$ and the available network bandwidth is given by $\beta_{a,b}$. A common and simple model for the time to send a message from a to b is

$$msgTime_{a,b} = \alpha_{a,b} + msgSize/\beta_{a,b}. \tag{IV.9}$$

In our target environment, the GrADS Computational Grid, near real-time measurements are provided by the Network Weather Service (NWS) for network latency (α) and bandwidth (β). We discuss the configuration of this measurement infrastructure in Section V.A; in essence, network performance measurements and predictions are based on the cost of transferring a fixed amount of data in a TCP/IP communication stream [55]. Note that these measurements include the overhead necessary to initiate a TCP/IP communication stream, which can be significant [55]. In initial benchmarking experiments we tested the prediction accuracy of the above message time model parameterized by NWS network performance predictions. In these experiments we observed that the message-passing costs experienced by our test applications were significantly lower than those predicted by the model for $msgTime_{a,b}$. We also experimented with other message time models, and found that a bandwidth only model provided better message time prediction accuracy. We therefore selected the bandwidth only model and the revised message time model is:

$$msgTime'_{a,b} = msgSize/\beta_{a,b}. \tag{IV.10}$$

IV.C Mapper

The function of the mapper is to determine an appropriate mapping of work onto processors for a given candidate resource group, or CRG. For our test applications, the resulting map consists of an allocation of mesh rows $n_0, n_1, ..., n_{p-1}$ to processors $P_0, P_1, ..., P_{p-1}$. In order to find such a mapping, two distinct subproblems must be solved: what **topological arrangement** of processors should be used (e.g. which physical processor should be assigned to logical processor position P_0) and what **allocation of work** to those processors is appropriate (e.g. exactly how many rows of the data mesh should be assigned to process P_0).

Our primary goal in finding a **topological arrangement** of processors is to order the processors such that communication costs are minimized. There are a variety of ways to solve this problem. It is possible to perform an exhaustive search and try each processor arrangement (using an execution time model, when available, to select the best one), but since the mapper is called frequently we do not wish to incur the large cost of such a search process. We instead opted for a simple and computationally inexpensive methodology: we group resources by site and then arrange processors in the topology such that machines from the same site are placed next to each other. For an application such as Game of Life that involves primarily neighbor-based communication, this topological arrangement tends to reduce the number of messages transferred over the wide-area when compared to a random processor arrangement. A simple improvement to this approach which we have not implemented would arrange the sites based on network delays in order to avoid the use of poorly performing links.

Our primary goal in finding an **allocation of work** onto processors is to ensure application resource-requirements are met. A secondary goal is to reduce application execution time by evenly balancing the workload on each processor. Determination of an application-appropriate work allocation is highly dependent on application performance characteristics. Since our scheduler design supports two levels of application performance models, we have designed two work-allocation strategies that take advantage of available information in each performance modeling context. The *equal allocation mapper* utilizes the application information available in the memory usage model context; the *time-balance mapper* utilizes the application information available in the execution time + memory usage model context.

IV.C.1 Equal allocation mapper

When only memory usage application information is available, a sophisticated mapping strategy cannot be employed. Our equal allocation mapping strategy is to simply allocate work to processors uniformly. Each processor is assigned $n_i = N/p$ rows of the mesh and the total number of pixels assigned to each processor is $n_i * N$. To ensure that application resource requirements are met, the mapper verifies that local memory availability is sufficient to support application memory requirements;

in the case that local memory capacities are not sufficient for application needs, the mapper simply returns with failure to find a map. In the scheduling context described in Chapter III, the current CRG is removed from the list of candidate CRGs and the search process continues.

Note that not every problem size will evenly decompose on every resource set size. In the case that the target application supports unequal partitions, we simply ensure that partitions are as similar as possible, but do not require exactly equal partitions; this is the approach taken for our test applications. In the case that the target application does not support unequal work partitions, candidate CRGs are restricted to those resource sets for which work can be divided into identically-sized partitions.

IV.C.2 Time balance Mapper

When an application execution time model is available, it is possible to utilize more performance-efficient mapping strategies. Our approach is to formalize important resource requirements and performance considerations as a series of constraints. Work-allocation can then be framed as a constrained optimization problem. A solution consists of an allocation of work onto resources; the goal of the optimization problem is to find a work-allocation that minimizes application iteration time. Recall that for this application class, iteration time is determined by the slowest processor. A work allocation that perfectly balances load among participating processors assigns less work to slower processors and more work to faster processors in a way that minimizes overall iteration time.

The variables in this constrained optimization problem constitute the mapping of rows to processors. Since each of the variables is constrained to an integer value, the system of constraints can be framed as an integer programming problem [54]. Unfortunately, the integer programming problem is NP-complete, rendering the solution computationally expensive to compute. During the scheduling process, a mapping is calculated for each candidate resource group; since the scheduler considers a relatively large number of CRGs (see Section III.B), a computationally expensive mapper will result in high overheads for scheduling. A much more efficient alternative is provided by linear programming solvers [54]; in this case the solution is real-valued and can only provide an approximate solution for an integer problem. However, even for small iterative, mesh-based applications of interest for the Grid, the maximum error that can be caused by the usage of a real-valued solution is quite small. For example, consider a Jacobi problem size of 1000 x 1000, which very easily runs on a single processor. Suppose this problem is decomposed on 2 processors. In this case, the maximum error that can be introduced by using a real-valued solution is 1/1000, or 0.1%. Since the introduced error is small and the solver efficiency is much higher, we frame the mapping problem as a linear programming problem. Many linear programming solvers are freely available [33]; we selected the **lp_solve** package [35], a freely available linear programming solver which is based on the simplex method.

The problem formulation begins with the specification of an *objective function.* For this problem, an ideal objective function is the iteration time of the slowest processor. Unfortunately, there is no way to specify this objective in a linear formulation. Instead, our formulation minimizes the computation time on the first processor in the topology, P_0, and utilizes a series of constraints to ensure load-balance among processors (we will discuss these constraints momentarily). We use the execution time model to formally specify the **objective function**:

$$minimize(\frac{N^{d-1} * compUnit}{comp_0} * n_0). \tag{IV.11}$$

The second component of problem formulation is a specification of **bounds** on the variables. In this case, each processor must be assigned a non-negative amount of work not to exceed the total problem size, N. Formally:

$$\forall i \in \{0 : p-1\}, 0 \leq n_i \leq N. \tag{IV.12}$$

The rest of the problem formulation consists of a series of **constraints**. Unlike some packages, the **lp_solve** package supports specification of constraints as equalities or inequalities. We therefore formalize each constraint in whichever format is the most natural. First, the total amount of work allocated must be equal to the total number of rows, N:

$$\sum_{i=0}^{p-1} n_i = N. \tag{IV.13}$$

Next, we use the specification of memory requirements from the memory usage model to ensure that the data allocated to each processor fits within that processor's local memory:

$$\forall i \in \{0 : p-1\}, N^{d-1} * memUnit * n_i \leq mem_i. \tag{IV.14}$$

Finally, we need to formally specify that processor iteration times should be balanced. To do this, one can select a reference processor and specify that the iteration time on every other processor must be equal to the iteration time on the reference processor. Since the objective function involves minimizing computation time on processor 0 we select P_0 as the reference processor; the constraints can then be written: $\forall i \in \{1 : p-1\}, |itTime_i - itTime_0| = 0$. When local memory capacities are sufficient, this constraint specification will result in a perfectly balanced mapping. However, when local memory capacities are more limited, the solver may fail to find a solution. In some of these cases, local memory availabilities are sufficient to support application requirements, but only if the data mapping is not perfectly balanced. Clearly, an unbalanced solution is preferable to no solution at all; we therefore introduce a relaxation factor, R in the time balancing constraints. The revised constraints specify that the iteration time on every processor must be within some relaxation factor of the iteration time on P_0: $\forall i \in \{1 : p-1\}, |itTime_i - itTime_0| \leq R * itTime_0$. To specify this constraint in a linear form, we again refer to the full performance model. Recall that communication requirements are independent of the data mapping for our test applications; communication costs can therefore be specified as a constant and moved to the right hand side of the linear constraint specification. Since absolute values can not be specified in a linear formulation, we use two inequalities. The formal constraint specification is:

$$\begin{aligned} \forall i \in \{1 : p-1\}, -(1+R) * compTime_0 + compTime_i \\ \leq (1+R) * commTime_0 - commTime_i \end{aligned} \tag{IV.15}$$

$$\begin{aligned} \forall i \in \{1 : p-1\}, (1-R) * compTime_0 - compTime_i \\ \leq (-1+R) * commTime_0 + commTime_i \end{aligned} \tag{IV.16}$$

Since the optimal mapping solution is found for an R factor of zero, we first formulate the problem with $R = 0$ and use **lp_solve** to try to find a solution. If a solution is found, the mapping is returned. When a perfectly balanced solution is not found, we want to find the mapping that satisfies local memory requirements while providing the best possible load-balance. To find this mapping we have to find the minimum R factor for which a solution is possible. We utilize a binary search method that begins with a minimum value of $R = 0$ and a configurable maximum value (for this thesis we use a maximum of 10). If a solution is impossible for the maximum R value, the mapper returns with failure to find a map. If a solution is found, a binary search is used to search for a R value that is close to optimal. The search ends when the difference between two successive R factors in the search is smaller than some tolerance (for this thesis we use a tolerance of 0.01).

IV.D Chapter summary

This chapter has focused on the general characteristics of iterative, mesh-based applications and on the specifics of two applications from this class, the Jacobi method and the Game of Life. We described in detail our implementation of each of these applications, and discussed the performance impacts of our implementation choices. We also presented an application-specific performance model design for memory usage prediction and execution time prediction. Finally, we described the function of the mapper and presented our implementation of an equal allocation mapper and a time balance mapper.

In the next chapter we present experimental results demonstrating the efficacy of our performance model and mapper designs as well as the scheduling methodology itself.

Chapter V

Experiments

In this chapter, we describe experimental results we obtained when applying our methodology in realistic scheduling runs of Jacobi and the Game of Life. Our validation approach is two-fold. First, we present a suite of experiments that test the application-specific execution time model and mapping strategies we developed in Chapter IV. Second, we present a suite of experiments designed to directly test our scheduling methodology itself. This two-phase approach is useful because we expect the performance of the scheduling methodology to be highly dependent on the type of application performance model and mapping strategy provided to it. It is therefore useful to understand the performance impact of each application-specific component before examining scheduler performance results.

This chapter is organized as follows. In Section V.A we describe the architecture and software configurations of the resources used in our validation experiments. We also describe the configuration of the Grid information services used to obtain information about these resources. In Section V.B, we present validation results for the application-specific performance models developed in Section IV.B. Likewise, in Section V.C, we present validation results for the mapping strategies developed in Section IV.C. Section V.D describes experimental results we obtained when applying our methodology in realistic scheduling runs of our two test applications. Section V.E explores the costs incurred in the process of scheduling. Finally, in Section V.F we summarize our findings.

	Circus cluster (UCSD)	Torc cluster (UTK)	Opus cluster (UIUC)	Major cluster (UIUC)
Size	6	8	4	6
Domain	ucsd.edu	cs.utk.edu	cs.uiuc.edu	cs.uiuc.edu
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
CPU	450 MHz PIII dralion nouba 400 MHz PII others	550 MHz PIII	450 MHz PII	266 PII
CPU Count	1	2	1	1
Memory	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 V.1: Summary of testbed resource characteristics.

V.A Experimental methodology

V.A.1 Testbeds

At the date of this writing, there are approximately 40 machines in the GrADS testbed; resources include machines located at Indiana University (IU), University of California at Santa Barbara (UCSB), University of Tennessee at Knoxville (UTK), Rice University (Rice), University of Illinois at Urbana-Champaign (UIUC), and University of California at San Diego (UCSD). For the thesis, we focus on a subset of the GrADS testbed including resources at UTK, UIUC, and UCSD. At UIUC the resources that we target are in two distinct clusters; we target a single cluster at each of the other sites. A snapshot of the network bandwidth within and between these sites is shown in Figure III.1. Characteristics of the targeted testbed resources are summarized in Table V.1.

We have selected two target resource groups, or testbeds, from this group of machines: a **one-site testbed** consisting of the UCSD subset of the target resource group and a **three-site testbed** consisting of all target resources. The one-site testbed serves as an example of a small testbed with relatively homogeneous workstations and a low-delay network. The three-site testbed serves as an example of a larger testbed with greater resource heterogeneity and larger network delays.

V.A.2 Software requirements

Our experimental methodology uses a variety of software packages; we discuss in turn those needed to launch and run the application itself, those needed only by the resource on which the scheduler will run, and those needed to support Grid information collection and dissemination.

In order to **launch and run the application**, every compute resource must be running Globus [22] and MPICH-G [39] and all of the required sub-packages. All of the resources targeted in this thesis are currently running Globus V1.1.3 (with threads enabled) and MPICH-G V1.1.2.

The **scheduler** itself requires a larger number of packages. The scheduler is implemented in C++; we used GNU autoconf and GNU make to configure and build the scheduler. To support the requirements of the time balance mapper, the lp_solve package [35] must be installed; we used version 3.2. The scheduler code also uses many building blocks from the AppleSeeds library [4] and the GrADSoft prototype [24]. NWS V2.0 [40] is used to query an on-line NWS nameserver for Grid information (we discuss the NWS configuration in the next section). LDAP is used by the scheduler to query an on-line MDS server for Grid information.

V.A.3 Grid information services

The NWS resource measurement infrastructure [40] includes measurement sensors on every resource of interest and a centralized *nameserver* that provides a single point of contact for consumers of NWS information. Measured resource attributes can be retrieved directly, or a next step prediction can be generated from measurement series [58]; in all experiments presented in this chapter NWS values are based on a next step prediction. The **GrADS NWS nameserver** is currently running on host `fender.cs.utk.edu`. For most testbed machines, measurements of local resource characteristics such as available CPU and free memory are collected every 10 seconds. The GrADS NWS infrastructure is also configured to collect network performance information (e.g. bandwidth and latency) between hosts. To avoid the large number of network measurements required for collection of all-to-all values, network measurements are gathered in a hierarchical fashion. Within each site (e.g. the UCSD Circus machines), measurements are collected in an all-to-all manner between all hosts. For wide-area network information, a particular workstation is selected at each site to act as a site representative; measurements are then collected in an all-to-all manner between site representatives. On the GrADS testbed, network performance measurements are based on the transfer of a TCP message of 1 MB in size, and measurements are repeated every 5 minutes. This measurement frequency is relatively low; use of a low measurement frequency is one way to mitigate the intrusiveness of the network measurement technique and to reduce interference between network measurements.

For the purposes of this thesis, we also maintained a **locally-controlled NWS nameserver and measurement infrastructure**. The local NWS nameserver is running on host `dralion.ucsd.edu` and is used only for NWS series needed for this thesis. We decided to use a locally-controlled NWS infrastructure because we found that we were able to maintain a more consistent level of information availability. To ensure that our results directly map to the general testbed infrastructure, wherever possible we have configured the measurement methodology in a similar way to the GrADS infrastructure. In Section V.E we discuss the overhead associated with retrieval of information from both the GrADS NWS nameserver and the locally-controlled nameserver.

To support retrieval of information from the MDS, there are duplicate **GrADS MDS servers** running at `grads.isi.edu`, port 3890 and `castanet.cs.uiuc.edu`, port 4444. As we discuss later in this chapter (Section V.E), the time required to retrieve information from either of these MDS servers can be high; additionally, the servers can be quite unstable. An upcoming release of the MDS will likely solve many of these problems. As a short-term workaround, we have implemented a local caching mechanism for MDS data; if MDS caching is enabled in the scheduler (this is a configurable option), then newly retrieved data values are stored in the local cache file. With caching enabled, later information requests will check the cache before accessing the MDS server. Local caching would be unacceptable if the information we retrieve were changing frequently; fortunately, the resource attributes we retrieve from the MDS, such as processor speed and physical memory, change very slowly.

V.A.4 Timing methodology

In the rest of this chapter, we present a variety of experimental results, most of which involve application performance results. For all such application performance results, we use a consistent timing methodology. The MPI function MPI_Wtime is used to record current time at various code locations. The resolution of this timing function (as reported by the MPI function MPI_Wtick) on the systems we targeted was 1 μsec. We use MPI_Wtime calls at the beginning and ending of each iteration to measure the time of each iteration. We record the computation and communication time per iteration in the same way. In order to exclude initial synchronization costs from average iteration times, the application is configured to run for four iterations before collected timings are incorporated into the average; after these warmup iterations, the application executes 100 iterations. At the end of this iterative phase, each processor calculates an average iteration time, an average computation time per iteration, and an average communication time per iteration. The overall average application iteration time is given by the maximum average iteration time reported by any processor. The procedure is the same for determining average application computation and communication times. Since processors are synchronized at each iteration, recorded iteration times are typically very similar. However, computation and communication times recorded on each processor can vary widely. The measurement of communication time is difficult because there is no simple way to differentiate between time actually spent communicating and time spent waiting for a communication peer to reach the communication phase. For this reason, we report timings only for the iteration time and the computation time. The difference between the iteration time and computation time can be interpreted as the communication time plus time for overheads such as waiting for communication peers. Iteration time results do not include the costs of scheduling. Instead, we address scheduling latency in Section V.E. With the exception of Section V.E, all timings reported in this chapter are in seconds per iteration.

V.B Performance model validation

Our scheduling methodology depends on a performance model to compare candidate schedules; the success of the scheduler in selecting performance-efficient schedules is therefore dependent on the ability of the performance model to correctly predict application resource usage. Recall that the scheduler design supports two levels of performance model: a memory usage model and an execution-time + memory usage model. The goal of this section is to evaluate the prediction capability of the application-specific execution time model that we presented in Section IV.B. While the prediction capability of the memory usage model is also important for scheduler performance, we feel that the application-specific memory usage model described in Section IV.B is straightforward. We have performed simple sanity checks on the validity of this model and are confident that it predicts the memory usage of our test applications well. [1]

[1]Specifically, we compared predicted application memory requirements with actual memory usage as reported by the UNIX command `ps`.

V.B.1 Experimental design

Approach

To evaluate the predictive capability of our execution time model we compare predicted performance to actual performance for a variety of conditions. For each such comparison we (1) select an application, testbed, problem size, an exact target resource set (a machine list), and a data mapping onto the selected resources; (2) use the execution time model to predict application performance; and (3) run the application and measure actual performance.

The execution time model supports a variety of configurations (see Section III.C.2 for details). For testing purposes we have selected a single model configuration. The selected configuration uses available processor speed (compType = AVAIL_MHZ) for the computational capacity of each machine and predicted values for all NWS information inputs (nwsType = PRED).

Testbeds

We include experiments performed on the one-site and three-site testbeds described in Section V.A, with one modification. For this set of experiments we wanted roughly equal numbers of resources from each site so we did not include the Opus cluster at UIUC in the three-site testbed.

Experimental procedure

For each testbed, we define an *experiment series* consisting of a reasonable selection of problem sizes and resource set sizes for that testbed. For the one-site testbed, an experiment series consists of problem sizes of $N = \{600, 1200, 2400, 4800, 7200, 9600\}$ and resource set sizes of $p = \{1, 2, 3, 4, 5, 6\}$. For the three-site testbed, an experiment series consists of problem sizes of $N = \{630, 1260, 2448, 4500, 7200, 9000\}$ and resource set sizes of $p = \{3, 6, 9, 12, 15, 18\}$. We selected a different set of problem sizes for each testbed to increase the frequency with which the problem size could be decomposed into identically sized partitions on the target resource set sizes.

To begin each experiment series, we ran the predictor and application for the smallest problem size on the smallest resource set size. We then ran increasingly larger resource set sizes for that problem size, and eventually continued on to the next larger problem size. For each application-testbed combination we completed three *repetitions* of the experiment series.

Runs of the application in an experiment series were performed within roughly the same period of time (e.g. within 4-12 hours) while repetitions of the experiment series were sometimes separated by a relatively long interval (e.g. 2 weeks). We therefore expect the Grid environment to be more similar within an experiment series than between experiment series. Note that if we executed application runs in an immediately back-to-back fashion, NWS resource availability predictions would be biased by the resource utilization of the previous run; this bias is due to the fact that, as with all measurement systems, there is a slight delay between changes in actual behavior and the recording of those changes. To avoid this undesirable interaction we included a three minute sleep phase between application runs; we selected three minutes because in practice it typically proved sufficient.

The configuration of an application run requires more than selection of a testbed, application, problem size, and resource set size; we also needed to select a specific resource set (a machine list) and a mapping of work onto those resources. Since the goal of these experiments was to evaluate only the execution time model, we favored a straightforward evaluation environment; we therefore predefined the target resource sets and data mappings for each configuration. In Section V.D we describe scheduling experiments in which the resource set and mapping were determined at run-time. For the one-site testbed, the compute and network resources were fairly homogeneous; we therefore simply selected a random resource ordering and used it for all runs. For the three-site testbed, the resources and networks were more heterogeneous; in this case, for each target resource set size we randomly selected an equal number of resources from each site and arranged them by site in the communication topology. For example, for the six-processor, three-site case, the processor arrangement we used was {torc3.cs.utk.edu, torc7.cs.utk.edu, fmajor.cs.uiuc.edu, hmajor.cs.uiuc.edu, dralion.ucsd.edu, soleil.ucsd.edu}. For the selection of a mapping of work onto the selected resources, we again favored a straightforward validation environment by using the equal allocation mapping strategy described in the previous chapter (Section IV.C). In Section V.C we describe experiments that compare the performance achieved with the equal allocation mapper and the time balance mapper.

V.B.2 Results

Since the execution time model is different for both applications, we present results for the Game of Life and Jacobi separately. Additionally, since the resource characteristics of the two target testbeds are quite different we also present results for each testbed separately. In the following sections we present results for each of the four resulting application-testbed combinations. Refer to Section V.A.4 for details on the timing methodology used for these experiments.

Game of Life, one-site testbed

Recall that for the one-site testbed, an experiment series consists of problem sizes of $N = \{600, 1200, 2400, 4800, 7200, 9600\}$ and resource set sizes of $p = \{1, 2, 3, 4, 5, 6\}$. Figure V.1 presents

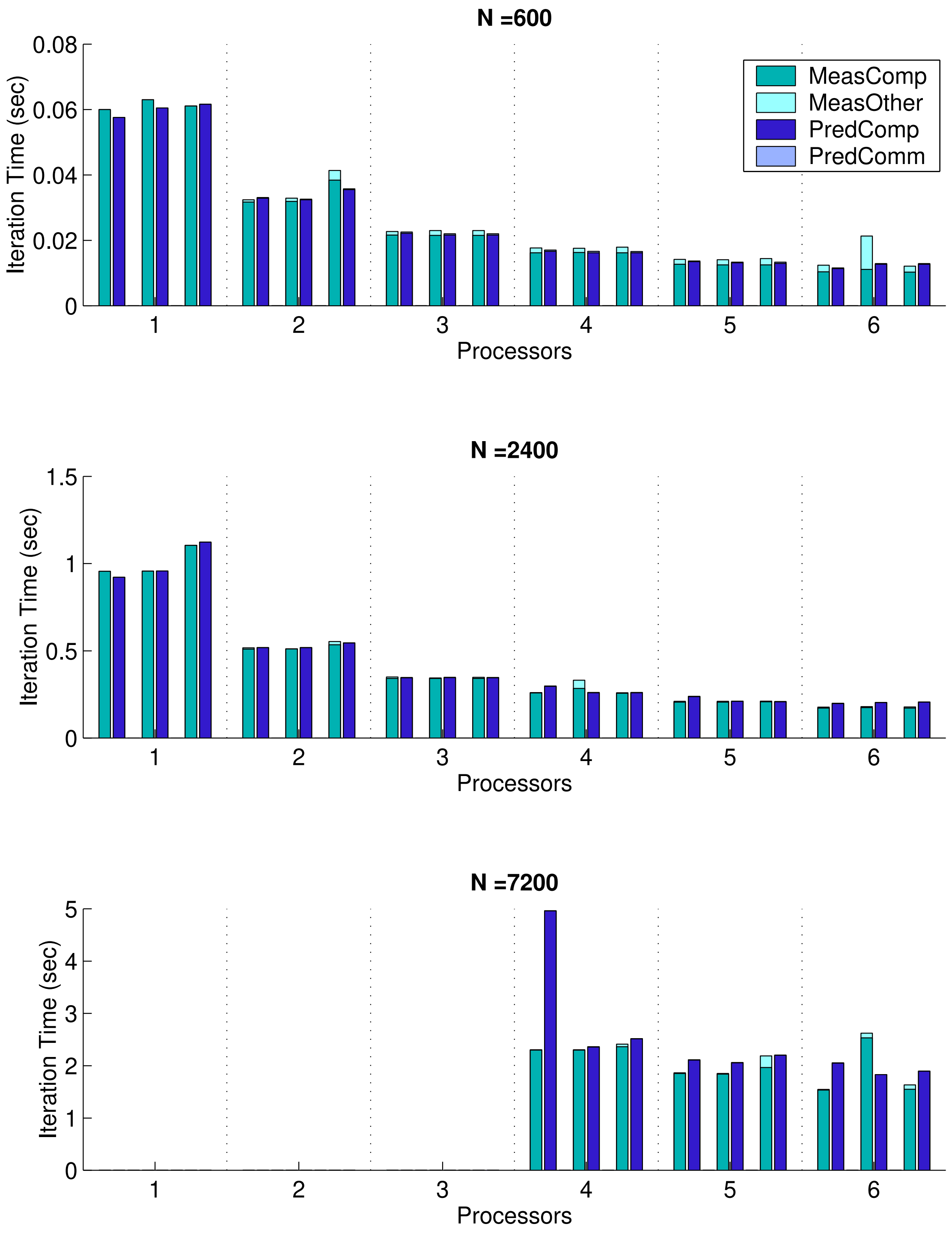


Figure V.1: Results of execution time model validation experiments for the Game of Life application on the one-site testbed, problem sizes of 600, 2400, and 7200.

results for three of the six problem sizes for the Game of Life experiment series on the one-site testbed; for the three sizes shown, $N = \{600{,}2400{,}7200\}$, all repetitions are shown for all target resource set sizes. For each resource set size, the three independent repetition results are presented as three pairs of vertical bars; each pair of bars represents one execution time prediction and application run pair. The full left-hand bar represents measured application iteration time; the darker, lower portion of the bar (*MeasComp*) is the measured computation time per iteration; and the lighter, upper portion of the bar (*MeasOther*) is the difference between the average iteration time and the average computation time. In this set of runs, iteration time is clearly dominated by computation time; computation is dominant here because the Game of Life does not involve very costly communications, and the one-site testbed provides relatively low-cost communication since all messages are transferred across the local area 100 Mbps Ethernet.

The right-hand bar of each bar pair represents predicted application iteration time; the darker, lower portion of the bar represents the predicted computation time and the lighter, upper portion of the bar is the difference between the predicted iteration time and the predicted computation time. A comparison of the overall bar heights for each bar pair indicates the prediction accuracy of the execution time model. Similarly, a comparison of the application computation time with the predicted computation time indicates the prediction accuracy of our computation time model. Finally, a comparison of the *MeasOther* time with the *PredComm* time indicates how well the communication time model (which only predicts the cost of message transfer) compares with the application time spent for message transfers, synchronization, and other overheads.

Notice that in Figure V.1 there are no results for a problem size of 7200 and resource set sizes of 1, 2, and 3. We excluded application runs from a series if the physical memory availability of the targeted processors is not large enough to support application needs; for this testbed, a problem size of 7200 requires at minimum four processors. One result is striking in this set: the predicted iteration time for $N = 7200$, $p = 4$, $rep = 1$ is more than twice as large as the measured iteration time. For this run, the predicted CPU availability retrieved from the NWS was less than 47% for one of the targeted processors; our methodology therefore predicted that the computation time on this processor would be more than twice as long as if the processor were unloaded. The actual results indicate that our application received nearly 100% of the CPU during execution; we hypothesize that either the load conditions changed during the short time from prediction to actual run, or the original CPU availability measurements were inaccurate.

Overall, the results shown in Figure V.1 indicate that the prediction accuracy of our execution time

	600	1200	2400	4800	7200	9000
Attempted Runs	18	18	18	18	18	18
Completed Runs	18	18	18	15	9	3
Median % Error	4.4	4.6	1.5	10.9	13.4	25.1
Mean % Error	6.7	6.7	5.8	15.4	25.2	25.0
StdDev % Error	8.7	5.9	7.1	19.3	35.6	6.1

Table V.2: Summary of execution time model prediction errors for the Game of Life on the one-site testbed.

model is fairly high for this application-testbed combination. This is not surprising since (1) the target resource set is relatively homogeneous, and (2) the application is dominated by computation time which is traditionally easier to model than communication time. To obtain a more quantitative analysis, we use the following method to calculate the relative percent prediction error for each run:

$$predError = 100 * \left| \frac{predTime - actualTime}{actualTime} \right| . \tag{V.1}$$

In Table V.2, summary results are provided for all six of the tested problem sizes. For each problem size, the experiment series included six resource set sizes and three repetitions; accordingly, the *attempted* row shows that 18 runs were attempted for each problem size. The *completed* row indicates how many of these runs were actually completed. The most common reason for an incomplete run is that the run was simply skipped due to limited memory availability during the experimental period; for this set of results, all of the incomplete runs can be attributed to this cause. More typically, some number of runs will fail due to other causes as well; incomplete runs can usually be attributed to an off-line target machine, a communication time-out in Globus, or insufficient memory availability on one or more of the target resources. Table V.2 also gives the *median*, *mean*, and *standard deviation* of all prediction error results for each problem size.

In summary, the mean prediction error of our execution time model is low for the three smaller problem sizes (error $< 7\%$) and moderate for the larger problem sizes (error values were between 15% and 25%).

Game of Life, three-site testbed

For the three-site testbed, each experiment series consists of problem sizes of $N = \{630, 1260, 2448, 4500, 7200, 9000\}$ and target resource set sizes of $p = \{3, 6, 9, 12, 15, 18\}$. In Figure V.2 we show results for problem sizes of $N = \{630, 4500, 9000\}$.

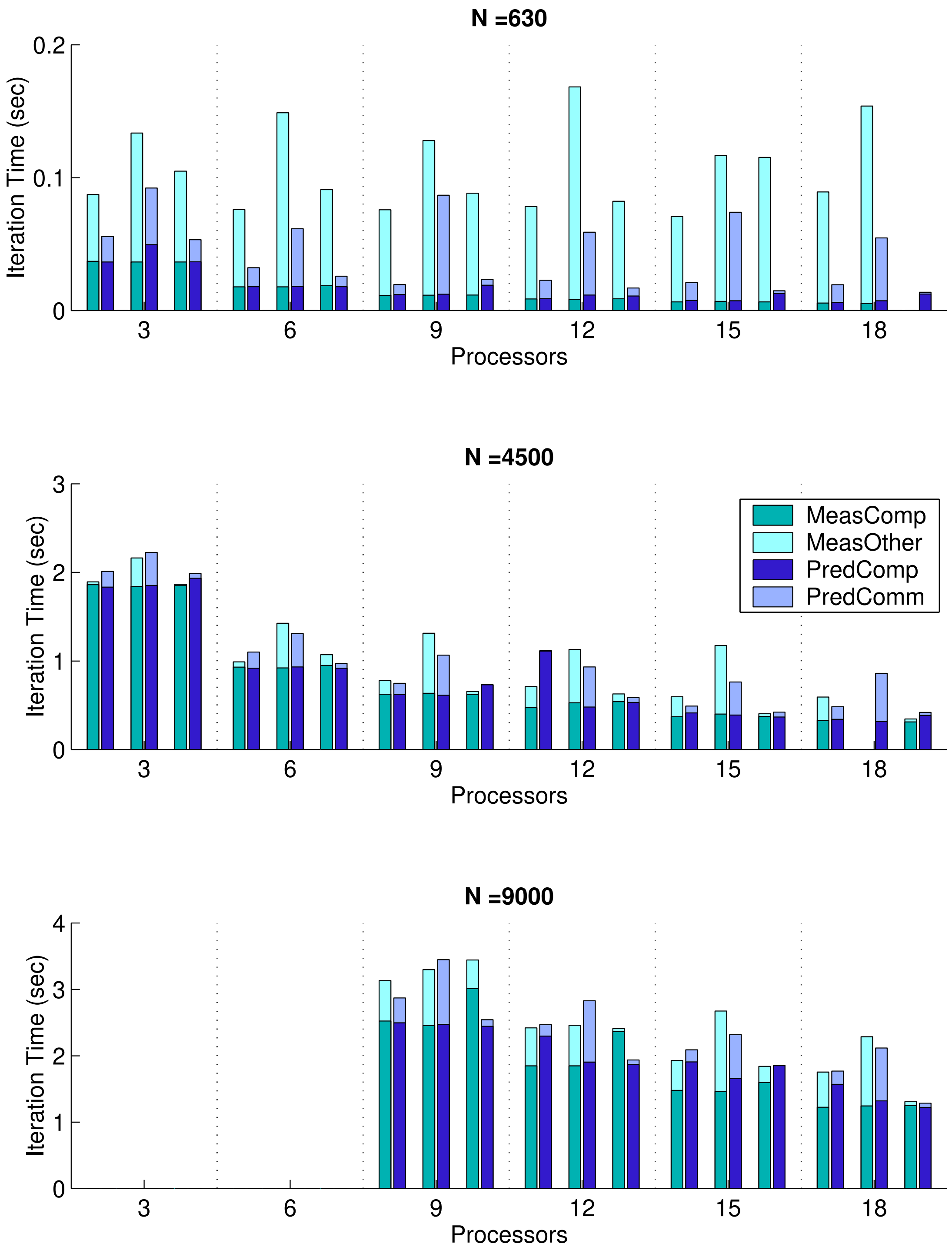


Figure V.2: Results of execution time model validation experiments for the Game of Life application on the three-site testbed, problem sizes of 630, 4500, and 9000.

	630	1260	2448	4500	7200	9000
Attempted Runs	18	18	18	18	18	18
Completed Runs	17	17	17	17	15	12
Median % Error	65.0	36.8	15.9	11.3	4.8	7.9
Mean % Error	60.9	34.6	24.3	15.1	7.6	9.0
StdDev % Error	17.8	19.9	32.1	13.5	8.5	8.1

Table V.3: Summary of execution time model prediction errors for the Game of Life on the three-site testbed.

Upon comparison of these results with those from the one-site testbed (Figure V.1), two differences are immediately obvious: communication costs are higher for the three-site testbed, and the model prediction accuracy is lower. The computation time predictions are generally fairly accurate for both sets of runs; the increased prediction errors are primarily attributable to communication time misprediction.

Recall that the communication model we use is a bandwidth-only model and notice that communication time misprediction is most severe for the smallest problem sizes (Figure V.1). For the transfer of smaller message sizes in the wide-area, communication cost is often dominated by message latency; it is therefore likely that misprediction for smaller problem sizes is attributable to our usage of a bandwidth-only model. However, in initial tests we found that the bandwidth-only model performed better *in general* than a model including both latency and bandwidth (see Section IV.B). Regardless, for the purposes of comparing candidate schedules, the most important characteristic of a model is that it correctly track *trends* in application performance. While the communication model is not particularly accurate, it does successfully track changing network performance; for example, for all target resource set sizes run for a problem size of $N = 4500$ the network bandwidth during repetition one was significantly higher than the bandwidth during repetition two. The communication model correctly predicts increased communication costs for repetition two.

Table V.3 summarizes prediction error statistics for all problem sizes. On average, the mean model prediction error is much higher for the smaller sizes than for the larger sizes (e.g. 60% for $N = 630$ versus 9% for $N = 9000$, respectively). Overall, average prediction errors are moderate for the Game of Life on the three-site testbed.

Jacobi, one-site testbed

Figure V.3 shows the results of the Jacobi experiment series for problem sizes of $N = \{600, 2400, 7200\}$ using the one-site testbed. Comparison of Figure V.3 with Figure V.1 reveals that even in the more tightly-coupled one-site testbed, communication costs are significant for the Jacobi application,

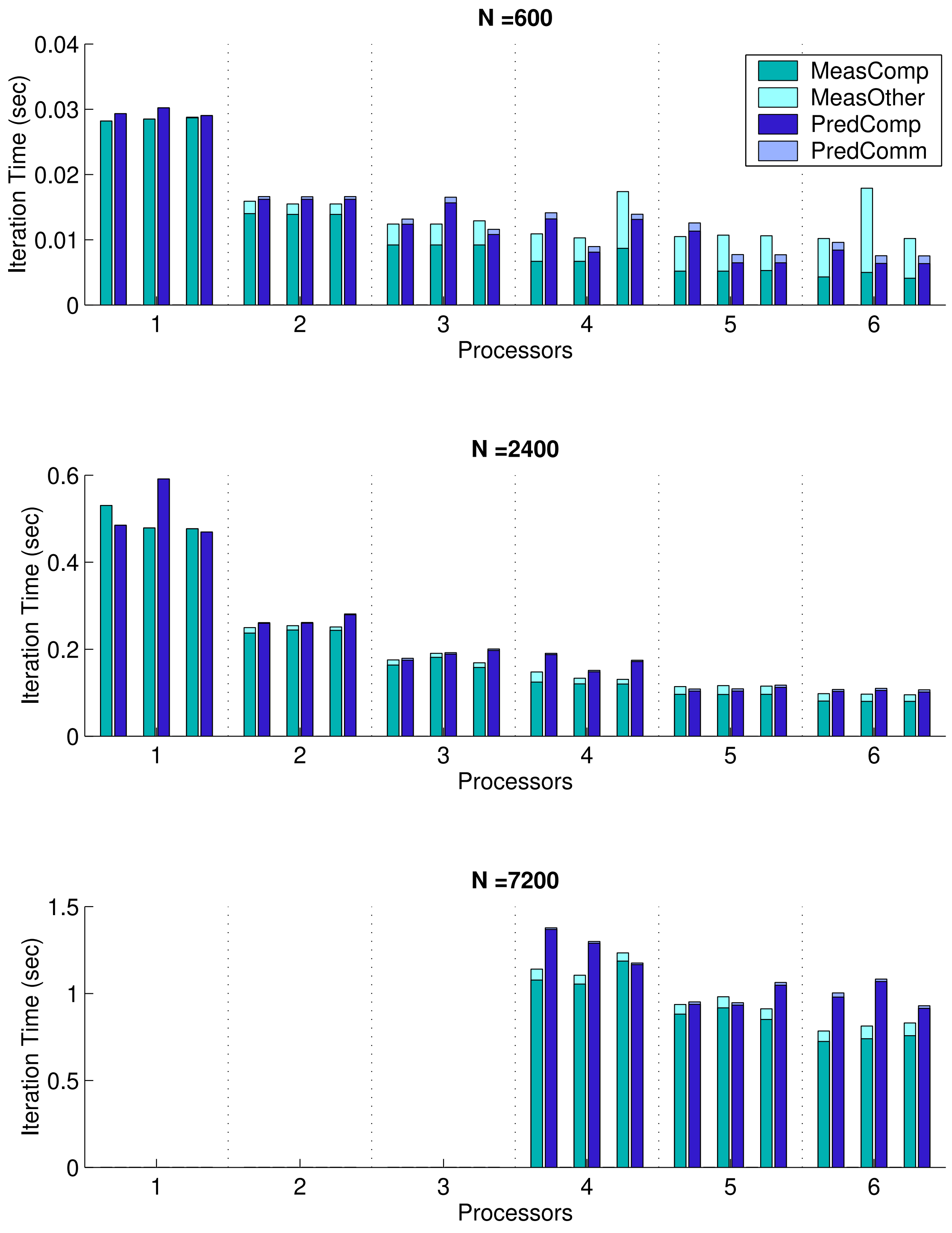


Figure V.3: Results of execution time model validation experiments for the Jacobi application on the one-site testbed, problem sizes of 600, 2400, and 7200.

	600	1200	2400	4800	7200	9000
Attempted Runs	18	18	18	18	18	18
Completed Runs	18	18	18	15	9	3
Median % Error	11.5	6.5	9.1	2.7	16.6	14.4
Mean % Error	17.0	12.5	11.1	9.4	15.3	16.3
StdDev % Error	14.6	20.2	9.7	12.1	11.0	13.1

Table V.4: Summary of execution time model prediction errors for Jacobi on the one-site testbed.

	630	1260	2448	4500	7200	9000
Attempted Runs	18	18	18	18	18	18
Completed Runs	17	13	18	16	13	10
Median % Error	74.9	52.8	36.5	31.7	18.0	16.5
Mean % Error	72.2	52.2	44.6	33.4	25.6	25.8
StdDev % Error	11.5	7.8	37.6	17.4	18.7	19.8

Table V.5: Summary of execution time model prediction errors for Jacobi on the three-site testbed.

which was not the case for the Game of Life. Recall that in our execution time model development (Section IV.B) we predicted that Jacobi would involve about half as much computation per iteration as the Game of Life, but that Jacobi's communication requirements would be much higher. Another noticeable difference between Figure V.3 and Figure V.1 is that the computation time prediction error seems to be higher for this data series. This degradation in prediction accuracy could be caused by a number of factors. The two most plausible are as follows.

1. We have included the purely computational termination detection phase in computation time measurements and predictions. However, for Jacobi the computational cost of termination detection grows as the size of N while our computation model assumes that all computation time grows as the size of N^2.

2. The increased communication activity in Jacobi might have affected computation times.

Table V.4 gives summary prediction error statistics for all problem sizes for Jacobi on the one-site testbed. The mean model prediction error is generally low for this testbed-application combination with values ranging from 9% to 17%.

Jacobi, three-site testbed

The final application-testbed combination is the Jacobi application on the three-site testbed. Experimental results for problem sizes of $N = \{630, 4500, 9000\}$ are shown in Figure V.4, and summary prediction error statistics are shown in Table V.5. Communication time is clearly a larger fraction

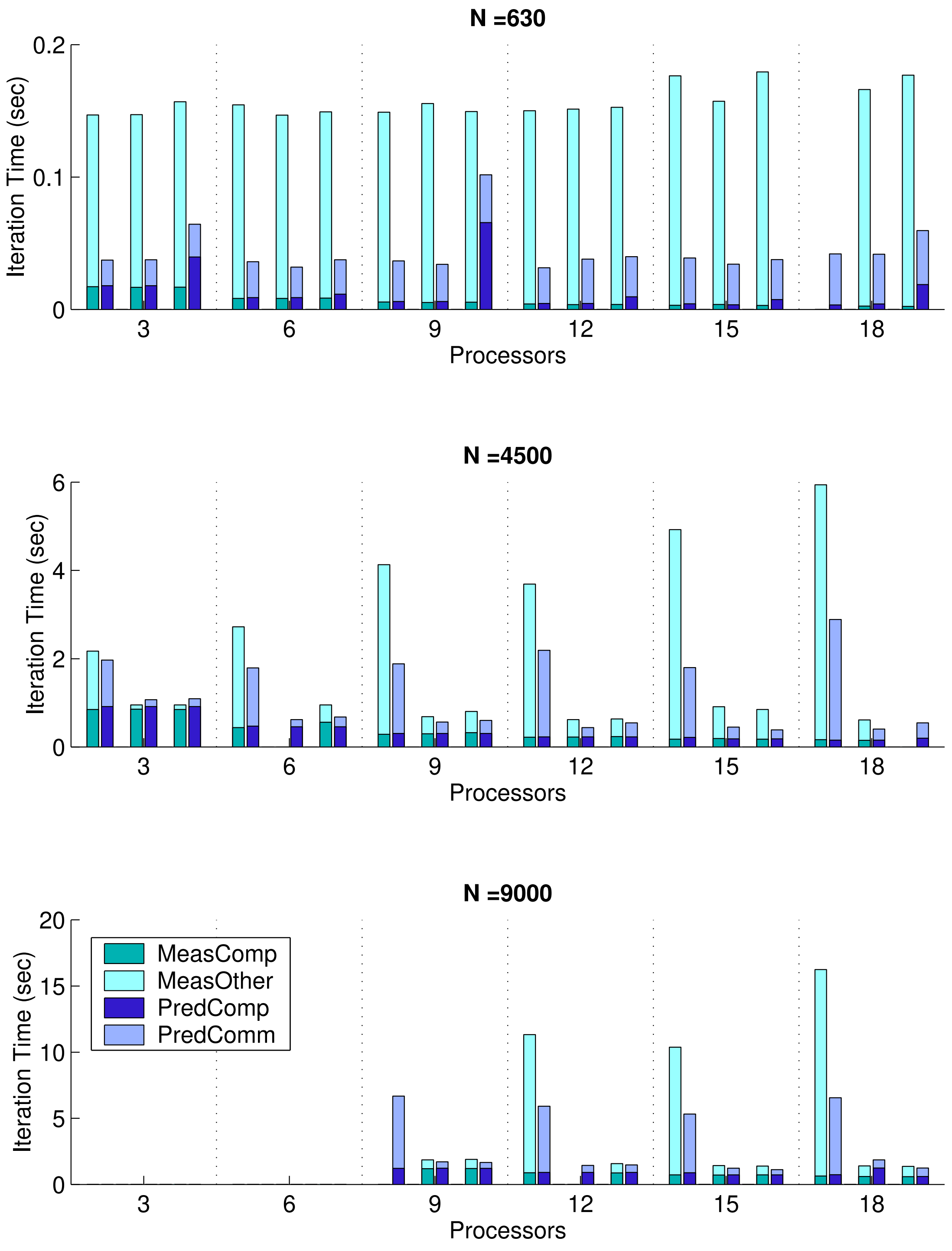


Figure V.4: Results of execution time model validation experiments for the Jacobi application on the three-site testbed, problem sizes of 630, 4500, and 9000.

of iteration time for this testbed-application combination than for the other three. This is to be expected for two reasons: (1) each Jacobi iteration includes a communication-intensive broadcast phase, and (2) for this testbed, each of those broadcasts includes a number of high-delay wide-area message transfers. As was the case for the Game of Life, the communication model typically under-predicts communication time and is especially inaccurate for smaller problem sizes. As discussed for the Game of Life, this under-prediction is likely attributable to our usage of a bandwidth-only model. Notice that the communication model does correctly track the increase in communication cost with increasing numbers of processors; this trend is especially apparent for repetition 1, problem sizes of 4500 and 9000. Measured iteration times indicate that network performance was degraded in repetition 1 as compared to the other 2 repetitions; the execution time model clearly distinguishes between these conditions and correctly tracks the improvement in network performance.

V.B.3 Summary

The experiments presented in this section were designed to evaluate the prediction accuracy of our execution time model in a wide variety of realistic Grid conditions and for an array of reasonable application configurations. For the one-site testbed, average performance prediction errors were less than 26% for both applications; average prediction errors were more typically 5% to 15%. For the three-site testbed, average performance prediction errors were less than 75% for both applications; more typically, average prediction errors were 10% to 50%. Given the highly dynamic nature of realistic Grid environments and the simplicity of our execution time model, prediction errors in these ranges are not unreasonable, nor are they unexpected.

V.C Mapper validation

Our core schedule search procedure, presented in Section III.B.1, is dependent on the availability of a reasonable mapping strategy for the development of feasible candidate schedules. The quality of generated schedules, as measured by the resulting application iteration time, can therefore be expected to be related to the execution time impact of the chosen mapping strategy.

In Section IV.C we presented two application-specific mapper designs, the equal allocation mapper and the time balance mapper. In Section V.B we utilized the equal allocation mapper in execution time model validation experiments. In this section we present experiments that investigate the performance characteristics of both the equal allocation mapper and the time balance mapper. We also examine the impact of mapper choice on application iteration times.

V.C.1 Experimental design

Approach

To evaluate the two mapping strategies we compared application iteration times achieved with each mapper under a variety of realistic Grid conditions. For each such comparison we (1) selected an application, testbed, problem size, and exact target resource set (a machine list); (2) used the equal allocation mapping strategy to find an equal allocation data map; (3) ran the application with the equal allocation data map on the resource set defined in step 1; (4) used the time balance mapper to find a time balance data map; and (5) ran the application with the load-balanced data map on the resource set defined in step 1.

Recall that the mappers support a variety of information inputs (see Section III.D). For testing purposes, we selected a single set of information inputs. Both mappers require local memory capacity information for all targeted resources; in these experiments the mappers utilized free memory values (memType = FREE). The time balance mapper also requires local computational capacity information; for these tests we selected available processor speed (AVAIL_MHZ). Finally, all NWS information inputs were next step predictions (nwsType = PRED).

Testbeds

For these experiments, we targeted the same real testbeds used for the performance model validation experiments.

Experimental procedure

To include a variety of problem sizes and target resource set sizes in our experiments, we used the same *experiment series* defined for each testbed in Section V.B. For the one-site testbed, we excluded the one-processor test cases since both mappers would develop exactly the same mapping, precluding interesting comparison. Since the goal of these experiments was to investigate the performance impact of the mapping strategies, and not to test resource selection, we targeted the same predefined resource sets as were used in Section V.B. As before, we included a *sleep* interval of three minutes between each application run. We again performed three repetitions of each experiment series for each application-testbed combination.

V.C.2 Results

Rather than present a large number of experimental runs as in Section V.B, we instead present summary results and describe in greater detail only the most significant results. In order to examine the impact of mapper choice on application iteration times, we consider the relative *percent improvement* of the time balance mapper over the equal allocation mapper:

$$percentImp = 100 * \frac{itTime_{equal} - itTime_{balanced}}{itTime_{equal}}. \tag{V.2}$$

When the time balance data map generates a shorter iteration time than the equal allocation map, the percent improvement metric will be positive; similarly, when application iteration time is shorter with the equal allocation map than with the time balance map, the percent improvement metric will be negative.

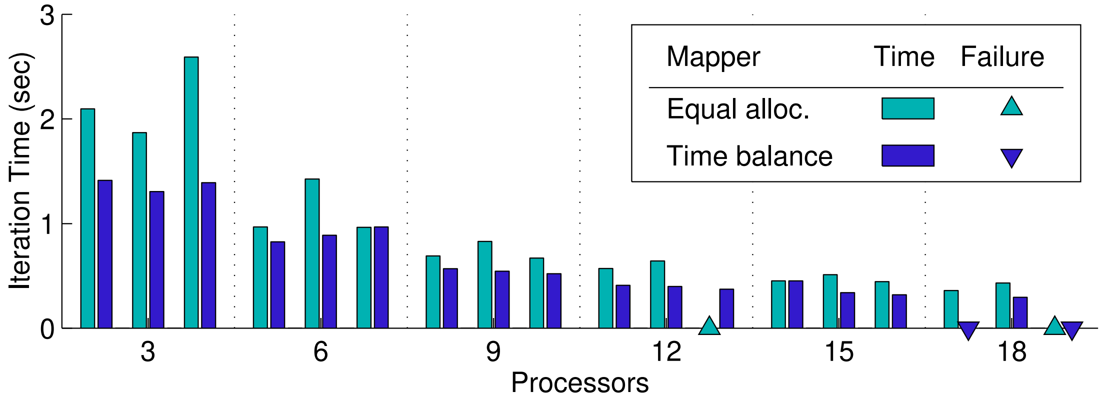


Figure V.5: Experimental results for the equal allocation and time balance mappers for the Game of Life application on the three-site testbed, problem size 4500.

Game of Life

Figure V.5 presents results for the Game of Life with a problem size of $N = 4500$ on the three-site testbed; results are shown for all target resource set sizes and for all three repetitions. This set of results is representative of the results for the other problem sizes tested. For each bar pair in this figure, the left-hand bar represents the application iteration time achieved with the equal allocation map, and the right-hand bar represents the application iteration time achieved with the load-balanced map. For the results presented in this figure, the time balance mapper generally results in better (i.e. shorter) execution times when compared with the equal allocation mapper.

	600	1200	2400	4800	7200	9600
Attempted runs	15	15	15	15	15	15
Completed equal alloc.	13	13	12	7	4	1
Completed time balance	13	13	13	11	7	3
Comparable runs	13	13	12	7	3	1
Median % improve	3.6	5.4	9.3	14.2	9.7	-19.1
Mean % improve	8.7	18.8	18.0	21.7	13.3	-19.1
StdDev % improve	10.6	21.8	18.8	18.7	67.7	0

Table V.6: Summary of percent improvement of the time balance mapper as compared to the equal allocation mapper for Game of Life on the one-site testbed.

The triangles in Figure V.5 mark places where there was either a mapper failure (e.g. the mapper found that local processor memory capacities were insufficient or an application failure (e.g. a Globus communication error caused the application itself to actually fail). For this series of runs, the failure causes were as follows.

- The equal allocation mapper failure at $p = 12$, $rep = 3$ was caused by an application failure of unknown origin.
- In the case of the balanced failure at $p = 18$, $rep = 1$, one of the target machines was so heavily loaded that NWS predicted CPU availability was zero and the load-balanced mapper could not find a valid mapping.
- For $p = 18$, $rep = 3$ both mappers failed to find a map because memory availability information was unavailable for one of the target machines (most likely the machine was off-line).

Note that the last two failures occurred because the list of target resource sets is predefined for these experiments. When the scheduler is allowed to select target resource sets, such resources are simply avoided and the mappers will not exhibit this type of failure.

Table V.6 and Table V.7 present summary information for all Game of Life mapper comparison runs on the one-site and three-site testbeds, respectively. The first row, *Attempted runs*, indicates the number of attempted application runs; the *Completed equal alloc.* and *Completed time balance* rows indicate how many of those runs were successfully completed for each mapper. Recall that failures can be related to either a mapper failure or an application failure. Row *Comparable runs* records how many of the attempted runs included a successful equal allocation mapper run **and** a successful time balance mapper run. We can only determine a percent improvement metric for those runs in which

	630	1260	2448	4500	7200	9000
Attempted runs	18	18	18	18	18	18
Completed equal alloc.	15	17	17	16	12	9
Completed time balance	16	16	17	16	16	16
Comparable runs	14	16	17	15	11	8
Median % improve	9.2	14.0	22.0	30.1	31.1	35.6
Mean % improve	15.8	13.6	24.8	26.3	29.1	30.6
StdDev % improve	26.8	13.5	14.6	13.4	18.5	22.3

Table V.7: Summary of percent improvement of the time balance mapper as compared to the equal allocation mapper for Game of Life on the three-site testbed.

both mappers ran successfully; for this reason the statistics in the rest of the table are based only on the number of runs listed in the *comparable runs* row. The last rows of the tables, *Median % improve*, *Mean % improve*, and *StdDev % improve*, present the median, mean, and standard deviation of the percent improvement of the time balance mapper as compared to the equal allocation mapper. For the majority of problem sizes on each testbed, the time balance mapper provides a substantial improvement over the equal allocation mapper; the one exception, $N = 9600$ on the one-site testbed, is based on only one comparable set of mapper runs and the application happened to perform very badly with the time balance map in this run. Notice that the performance advantage provided by the time balance mapper is more significant for the three-site testbed.

For the Game of Life application there are two primary factors that contribute to the success of the time balance mapper in improving application execution time: (1) the mapper balances computational load based on the computational capacities of the targeted resources and (2) the mapper overlaps communication on some processors with computation on others. In the following paragraphs we describe in greater detail how each factor improves application execution time.

Computational load-balancing. The computational speed of the resources in each testbed are heterogeneous and since the resources are shared, the load on targeted CPUs can be quite different. When the equal allocation mapper is used, the slower or more highly-loaded machines slow the entire computation down. When the time balance mapper is used, the load is reduced on the slower and/or heavily-loaded machines and increased on the faster and/or more lightly-loaded machines. Assuming that NWS CPU availability predictions are accurate, this load adjustment tends to minimize overall application execution time by assigning appropriate workloads to each processor. Previous work in application-specific schedulers [9, 12] has demonstrated similar success with computational load-balancing for iterative, mesh-based applications.

Overlapping communication and computation. For the Game of Life, the communication phase for each processor involves only communication with neighboring processors. Recall that during the mapping process we arrange the processor topology to minimize the amount of data sent over slow, wide-area links; we do this by placing resources from the same site adjacent to each other in the topology. When a work allocation is found for this processor topology, processors in the same site receive contiguous strips of data. Only processors that share a strip edge with a processor from a remote site must participate in time-consuming wide-area message transfers; we call these processors *site edges* in the processor topology. Since we use a strip decomposition, at most two processors from each site can be site edges. Our implementation of the Game of Life communication phase uses non-blocking calls; processors that are not side edges can therefore continue computation while site edge processors are involved in wide-area transfers. The time balance mapper correctly handles this communication heterogeneity by allocating less work to site edge processors. For example, the time balance mapper achieved a large performance improvement over the equal allocation mapper for all three repetitions of $p = 3$ in Figure V.5. In this case, there are three resources, {torc3.cs.utk.edu, cmajor.cs.uiuc.edu, quidam.ucsd.edu}, and three sites; every processor must therefore participate in wide-area communications. However, cmajor.cs.uiuc.edu will spend more time communicating since it must communicate with both torc3.cs.utk.edu and quidam.ucsd.edu. The mapper correctly allocates less work to this resource. Consider repetition three, in which the resources were assigned roughly 42%, 22%, and 35% of the total work. The middle processor, cmajor.cs.uiuc.edu, was allocated the least work due to higher communication costs and the first processor, torc3.cs.utk.edu, was assigned a larger portion of the work than the third processor, quidam.ucsd.edu, because of differences in processor speed (550 MHz versus 400 MHz).

There is another advantage to the time balance mapper that is not immediately obvious in the summary statistics: for larger problem sizes, the time balance mapper runs are significantly more likely to complete than the equal allocation mapper runs (compare the *Completed equal alloc.* and *Completed time balance* rows in Tables V.6 and V.7). As shown in Table V.1, the physical memory sizes vary widely for the three-site testbed. Additionally, since resources are shared, the free memory available on targeted resources can vary widely for both testbeds. The time balance mapper can adapt to this heterogeneity by adjusting the load assigned to each processor, thus ensuring application memory requirements are met. By comparison, the equal allocation mapper will fail when any one of the targeted resources does not have the local memory capacity to handle an equal share of the application workload.

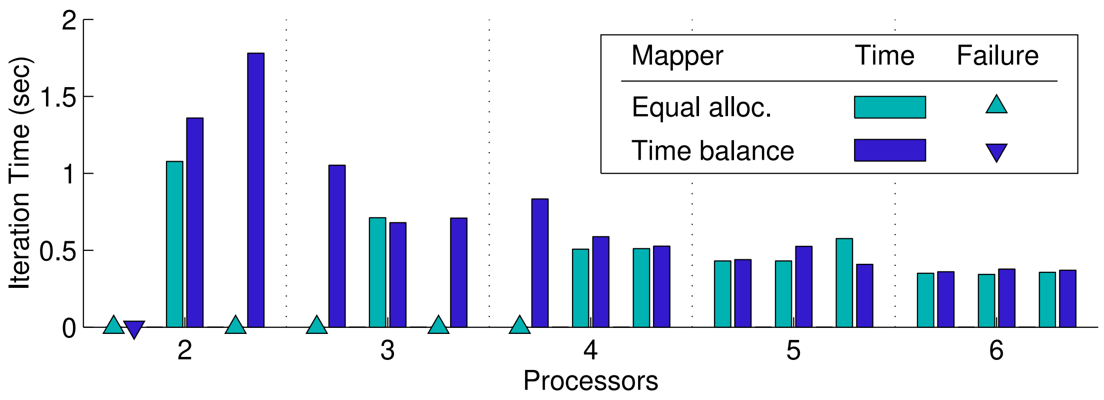


Figure V.6: Experimental results for the equal allocation and time balance mappers for the Jacobi application on the one-site testbed, problem size 4800.

Figure V.6 presents mapper validation results for Jacobi with a problem size of $N = 4800$ on the one-site testbed; results are shown for all three repetitions on all target resource set sizes. The relative performance of the two mappers seen in this set of results is fairly representative of the results for other problem sizes and for the three-set testbed. For most application runs shown in this graph, application performance was comparable with the equal allocation and time balance mappers; for some runs the performance was better with the equal allocation mapper, although never dramatically. Note that in this set of runs, the number of failures was much lower for the time balance mapper than for the equal allocation mapper.

Table V.8 and Table V.9 present summary information for all Jacobi mapper comparison runs on the one-site and three-site testbeds, respectively. For the one-site testbed, use of the time balance mapper generally resulted in a modest degradation of performance as compared to use of the equal allocation mapper, but the time balance mapper runs were significantly more likely to complete successfully. Suprisingly, the performance and completion patterns were reversed for the three-site testbed: in these experiments, the time balance mapper resulted in a modest performance improvement but was less likely to complete successfully. The higher failure rate for the load balance mapper is notable, particularly since the load balance mapper typically succeeds in finding a suitable data map when the equal allocation mapper cannot. During these experiments, one or more resources were heavily loaded and the NWS reported memory availability predictions of 0%. The load balance mapper was unable to find a suitable mapping in which application memory requirements were met by the target resource group and so failed. When the scheduler is allowed to select resources, dynamic NWS memory availability predictions allow

	600	1200	2400	4800	7200	9600
Attempted runs	15	15	15	15	15	15
Completed equal alloc.	15	15	15	10	4	2
Completed time balance	15	15	15	14	10	5
Comparable runs	15	15	15	10	4	2
Median % improve	-1.9	-4.5	-6.4	-3.5	13.1	-5.1
Mean % improve	-1.2	-7.3	-14.1	-5.3	12.9	-5.1
StdDev % improve	8.7	11.1	29.9	15.4	12.1	17.1

Table V.8: Summary of percent improvement of the time balance mapper as compared to the equal allocation mapper for Jacobi on the one-site testbed.

	630	1260	2448	4500	7200	9000
Attempted runs	18	18	18	18	18	18
Completed equal alloc.	18	18	16	15	11	10
Completed time balance	16	17	13	12	12	15
Comparable runs	16	17	11	10	7	7
Median % improve	-1.6	-2.1	4.3	7.7	13.9	11.5
Mean % improve	-0.2	5.3	8.8	-10.5	16.3	7.9
StdDev % improve	11.5	23.5	18.2	57.2	9.2	13.9

Table V.9: Summary of percent improvement of the time balance mapper as compared to the equal allocation mapper for Jacobi on the three-site testbed.

the scheduler to avoid heavily loaded machines altogether.

It is also notable that the time balance mapper provided more of a performance advantage for the Game of Life than for Jacobi. Recall that the time balance mapper improved performance for the Game of Life by overlapping communication on some processors with computation on others and by balancing computational load on participating processors. Since our implementation of the Jacobi communication phase uses a series of broadcasts, all processors must participate in the entire communication phase, and the time spent communicating per processor is essentially independent of the work allocation. For these reasons, the time balance mapper cannot improve performance by overlapping computation on some processors with communication on others. Nonetheless, the mapper should be able to take advantage of computational load balancing to improve Jacobi performance. An important limitation of this capability is that the computation phase is sometimes a relatively minor portion of application iteration time (see Figure V.4); in these cases balancing computational load affects only a small portion of the iteration time. It is surprising that the time balance mapper does not provide a significant performance advantage for the Jacobi application in those cases where computation time does constitute a significant portion of application execution time (see Figure V.3).

V.C.3 Summary

In this section we have demonstrated the utility of both the time balance and the equal allocation mappers, thereby encouraging us to consider both mappers for scheduling experiments. Additionally, for the Game of Life we demonstrated that the time balance mapper achieved a significant performance advantage as compared to the equal allocation mapper; the performance advantage was more moderate for the Jacobi application. The time balance mapper requires more sophisticated application information than the equal allocation mapper; the performance advantage shown for the time balance mapper therefore demonstrates that we can take advantage of more sophisticated application information to improve application performance. We also found that, overall, the time balance mapper found a suitable data map more often than the equal allocation mapper.

V.D Scheduler validation

The focus of this thesis is the development of a flexible, application-targetable scheduling methodology for Grid environments. In this section we present experiments that explore the efficacy of our methodology. In particular, we investigate the following questions.

i. What is the impact of our scheduling methodology on application execution times as compared to conventional scheduling approaches? We hope that our methodology reduces execution times.

ii. What is the impact of application information availability on scheduler performance? Specifically, (a) can this methodology develop reasonable schedules despite limited application information and models, and (b) can this methodology take advantage of more sophisticated information and models to promote application performance? We hope that the answer to both questions is yes.

iii. How is application performance affected when dynamic resource information is available to our scheduling methodology? Can the methodology develop reasonable schedules when only static resource information is available? We hope that the scheduler can take advantage of dynamic resource information, and that it can continue to function when such information is not available.

To investigate these questions we developed four scheduling strategies that we describe in the next section. We then studied the performance of each strategy in a wide span of usage scenarios including a variety of problem sizes, applications, testbeds, and ambient load conditions; the design of these experiments is also described in the next section. In Section V.D.2 we describe the results of these experiments and in Section V.D.3 we summarize our findings.

V.D.1 Experimental design

Approach

To help us investigate the questions listed above, we developed four scheduling strategies based on realistic Grid scheduling scenarios: *user*, *basic*, *static*, and *dynamic*. We developed the user strategy to emulate the scheduling process that a typical user might employ. The remaining three strategies are variations of our scheduling methodology based on resource and application information availability scenarios. In the following paragraphs we describe the motivation behind, and the design of, each scheduling strategy.

To answer question (i), we needed to compare the performance achieved with our methodology with that achieved by a conventional approach. Unfortunately there is no standard Grid scheduler that is effective for the applications and environments that we target. In fact, the conventional approach for the majority of Grid users is to develop a simple scheduling strategy based on basic information about their application's performance characteristics and the Grid they wish to run it on. We therefore developed a **user strategy** to emulate the decision making process that a user might employ. First, we had to decide what application performance metric a user would be likely to employ. For the applications targeted in this thesis, the easiest application performance metric to obtain is an estimate of application memory usage; to estimate memory usage one need only find the memory allocation commands in the application and determine how problem size, problem dimensions, and data type will affect application memory requirements. We believe that the development of a full application performance model, including parameterization for the target Grid of interest, would be too large of an investment for most Grid users. For these reasons, our user strategy predicts application resource requirements based solely on our memory usage model developed in Section IV.B.1. To use this model the user requires access to memory capacity information for the resources she wishes to target. The memory capacity of Grid resources is based on physical memory sizes; this is a practical strategy for users as the information can be obtained directly by logging in to the machines of interest or by requesting the information from the resource manager (i.e. a systems administrator). Finally, we assume that each Grid user has a local resource set that she preferentially accesses, typically because she is more familiar with the computational environment or because she is more comfortable as a resident on her home resources than as a guest on remote resources. The preferential resource ordering that is assumed by our user strategy is {UCSD, UTK, UIUC}.

So how do all of these assumptions combine to form the user scheduling strategy? Our user strategy (1) predicts application memory requirements for the application and problem size of interest; (2) selects the minimum number of target resources that will satisfy application memory requirements (resources are selected in order from the preferential resource ordering); (3) determines an equal, or nearly equal, allocation of work onto the selected resources; and (4) runs the target application with the selected schedule. Note that our user strategy uses the same *memFactor* of 20% that is used by our memory usage model; inclusion of this factor emulates the fact that most Grid users are aware of the performance problems inherent in allocating all of the physical memory on a shared resource.

In order to answer questions (i), (ii), and (iii), we developed a number of strategies based on the scheduling methodology presented in this thesis. First, we wanted to study how our methodology performed with full application and Grid resource information. In this strategy, called the **dynamic strategy**, the scheduler is provided with the full execution time + memory usage model described in Section IV.B.2; this model is the more sophisticated of the two performance models developed for our test applications. Furthermore, the scheduler utilizes dynamic Grid resource information, which we categorize as a more sophisticated level of resource information utilization than static information. In keeping with the availability of sophisticated application information, the scheduler utilizes the time balance mapper (described in Section IV.C.2) for the allocation of work to processors. In this strategy, schedules are developed at run-time to take advantage of dynamic resource performance information.

To address question (ii), we study a strategy in which full Grid resource information is available, but application information is limited. In this strategy, called the **basic strategy**, our scheduling methodology is provided with only the memory usage model, but is given access to dynamic resource availability information. Since this strategy assumes that application information is limited, the scheduler employs the equal allocation mapper, described in Section IV.C.1. Free memory predictions are used in conjunction with the equal allocation mapper to ensure that the application data is mapped in a way that does not overflow local processor memory capabilities. Recall from Section III.B that we create candidate resource groups (CRGs) based in part on three resource-oriented sorting foci: *computation*, *memory*, and *dual*. Predictions of dynamic CPU availability and free memory capacity are used by this sorting method to select desirable resources. Schedules are developed at run-time to take advantage of dynamic resource performance information.

Finally, to address question (iii) we study a strategy in which full application performance information is available, but Grid resource information is more limited. In this strategy, called the **static strategy**, the scheduler is provided with the execution time + memory usage model and the time balance mapper, but is provided with only static resource information. Specifically, the computational capacity of resources is given by the full processor speed and the memory capacity of resources is given by the full physical memory; these characteristics are retrieved from the MDS. The NWS provides the only network performance estimates that can be retrieved from GrADS information sources; the scheduler is therefore configured to utilize NWS bandwidth estimates, but the scheduler is run off-line so that it cannot take advantage of run-time performance estimates.

Figure V.7 summarizes the application and Grid information usage by each of the four scheduling strategies. In addition to the configuration options we detailed above for each of the three variations on our scheduler, there are other scheduler configuration options which we did not vary in these experiments but which affect scheduler behavior. In particular, we selected a performance improvement threshold of 5% for these experiments (the threshold is defined in Section III.B.2). The predicted performance of selected schedules will therefore always fall within 5% of the predicted performance of the best schedule examined. We believe that 5% is a conservative estimate of what a typical user would define as equivalent performance in Computational Grid environments.

Testbeds

For these experiments, we ran experiments on the one-site and three-site testbeds described in Section V.A. Note that, as described in Section V.A, the three-site testbed included the Opus cluster at UIUC.

Experimental procedure

The focus of these experiments is to compare application iteration times achieved by each scheduling strategy in a variety of application, testbed, problem size, and ambient load conditions. For the basic and dynamic strategies, which utilize dynamic resource information, schedule development was performed at run-time. For the user and static strategies, which utilize only static resource information, schedule development was performed off-line and schedules were retrieved at run-time. To complete a *scheduling strategy comparison experiment*, the four strategies were run in a back-to-back manner in the following order: user, basic, static, and dynamic. To avoid each application run from affecting the decisions of the following scheduling strategy (due to the affect on NWS dynamic resource predictions) we included a three minute sleep between application runs.

Each scheduling strategy comparison experiment is defined by a selection of a test application, a

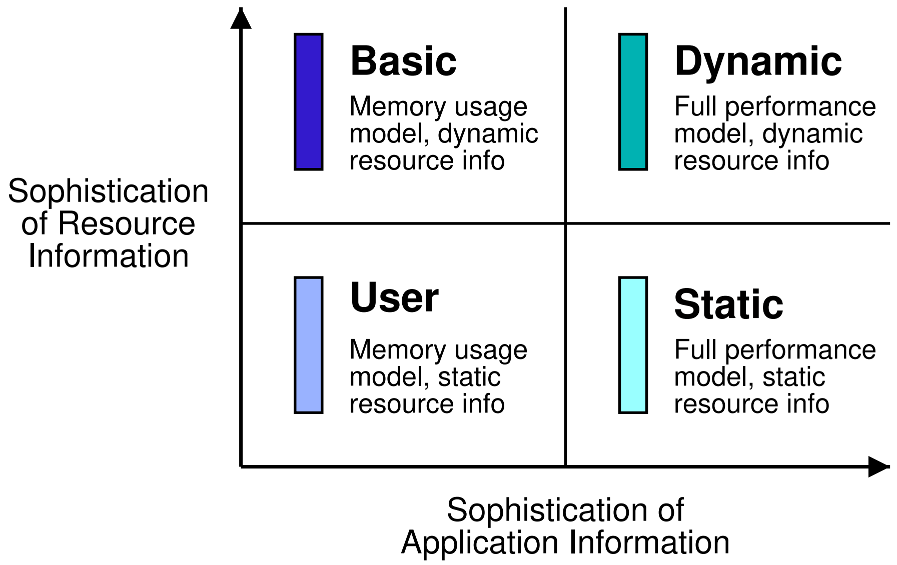

Basic
Memory usage model, dynamic resource info
Dynamic
Full performance model, dynamic resource info
User
Memory usage model, static resource info
Static
Full performance model, static resource info
Sophistication of Resource Information
Sophistication of Application Information

Figure V.7: Summary of user, basic, static, and dynamic scheduling strategies. For each strategy we note the availability of sophisticated application and resource information. Bars correspond to the colors used for each strategy in our scheduling results graphs.

testbed, and a problem size. We performed these experiments for each of the test application on each of the testbeds (four application-testbed pairs). For each testbed, we selected six test problem sizes to span a broad range of application scenarios. For the one-site testbed we use the same problem sizes as in the performance model and mapper validation experiments, $N = \{600, 1200, 2400, 4800, 7200, 9600\}$. For the three-site testbed, we wanted to explore a broader range of problem sizes than we used for the performance model and mapper validation experiments. On this testbed we used $N = \{600, 4800, 9600, 14400, 16800, 19200\}$. We performed experiment series for each application-testbed combination; each experiment series consisted of a scheduling strategy comparison for each problem size, or 6 comparison experiments where each comparison experiment involved the testing of each of the 4 scheduling strategies. We completed 10 repetitions of each experiment series. Overall, we completed 60 comparison experiments for each of the 4 application-testbed scenarios. Since each of these 240 comparison experiments included the testing of 4 scheduling strategies, we completed a total of 960 scheduling strategy tests.

Performance metrics

To provide a quantitative comparison of the application performance achieved by each scheduling strategy, we utilize two comparison metrics: the rank and the percent degradation from best. Both of these metrics are commonly used for the comparison of scheduling strategies [32]. A **rank** is an integer value between 1 and 4 indicating the relative performance of each strategy in a scheduling strategy comparison experiment; the strategy that achieved the best application iteration time was assigned a 1 while the strategy that achieved the worst application execution time was assigned a 4. If a scheduling strategy failed to find a suitable schedule, or the application itself failed, the worst rank, a 4, was assigned to that strategy.

To calculate the **percent degradation from best** we first find the lowest iteration time achieved by any of the strategies, $itTime_{min}$. For each scheduling strategy we then calculate the percent degradation from best as:

$$degFromBest = 100 * \frac{itTime - itTime_{best}}{itTime_{best}}. \tag{V.3}$$

The strategy which achieved the minimum iteration time will be assigned a percent degradation from best value of zero. Note that if we could include an optimal scheduler in the experiments it would consistently achieve a 0% degradation from best. When one of the scheduling strategies failed to find a schedule or its corresponding application run failed, that scheduler was not assigned a percent degradation from best value.

V.D.2 Results

The first set of results we present is a summary comparison of the application performance achieved by the four scheduling strategies outlined in the previous section. Later in this section, we present individual scheduling strategy comparison experiments to highlight signficant points.

To report summary results, we aggregated all 6 problem sizes and 10 repetitions into a single group of results for each application-testbed scenario; for each application-testbed scenario we ran a total of 60 scheduling strategy comparison experiments. Figure V.8 presents the average rank assigned to the scheduling strategies for each scenario. For all but one of the scenarios, the dynamic strategy achieved the best (i.e. lowest) average rank; the exception was the Jacobi application on the three-site testbed where the static strategy achieved a better rank by a slight margin. Compare the sophistication of available Grid information and application performance models (see Figure V.7) with the average rank for each scheduling strategy. Several important points are revealed.

- The two strategies that achieved the best average ranks were the static and dynamic strategies. These strategies utilize a more sophisticated application performance model than the other two

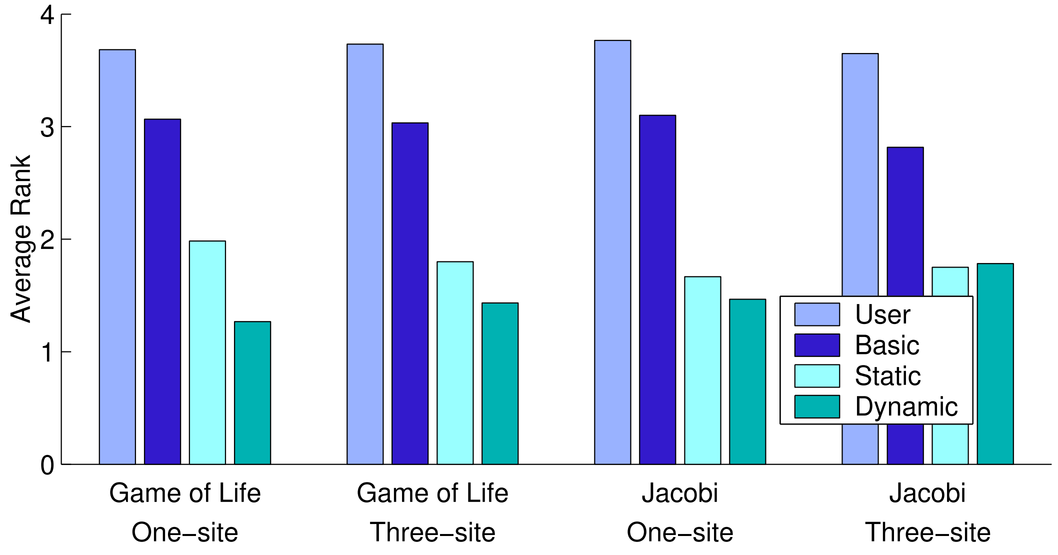


Figure V.8: Average rank of each scheduling strategy for all application-testbed scenarios. The best possible average rank is 1 and the worst is 4.

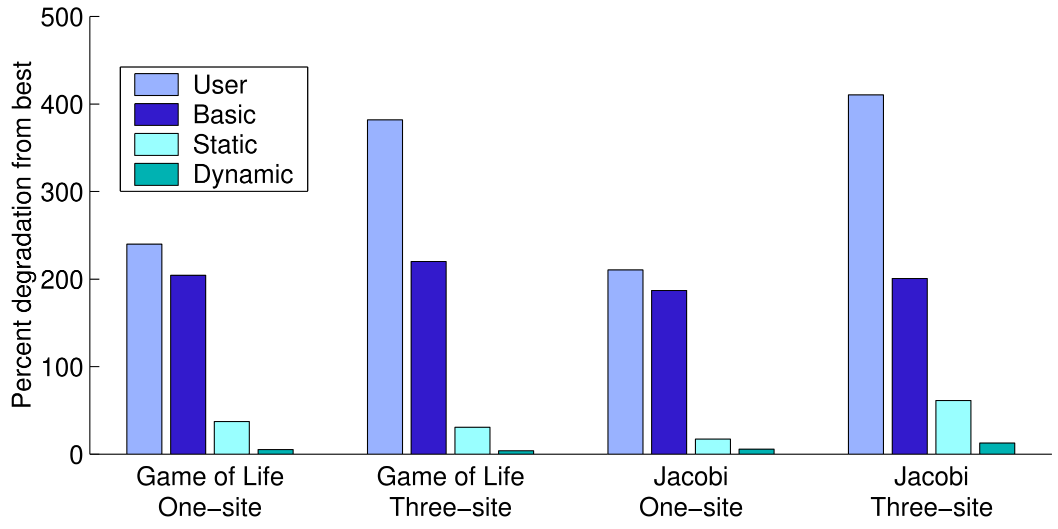


Figure V.9: Average degradation from best for each scheduling strategy for all application-testbed scenarios. For reference, an optimal scheduling strategy would average 0% degradation from best.

strategies. This trend demonstrates the importance of accurate application performance models in developing performance-efficient schedules.

- The dynamic strategy outperformed the static strategy in three out of four application-testbed scenarios, and the basic strategy outperformed the user strategy in all four scenarios. The dynamic and basic strategies utilize more sophisticated Grid information than the other strategies. This trend is suggestive of the importance of dynamic Grid information in developing performance-efficient schedules.

- The static strategy consistently outperformed the basic strategy. The static strategy utilizes a more sophisticated application performance model, but less sophisticated Grid information. For these experiments, we conclude that availability of an accurate application performance model had a more significant impact on the development of performance efficient schedules than did the availability of dynamic Grid information.

Figure V.9 reports the average percent degradation from best for each scheduling strategy in all application-testbed scenarios and Table V.10 reports summary statistics for the same data set. With one exception, we see the same ordering of the strategies as we saw for average ranks in Figure V.8; the exception is for the Jacobi application on the three-site testbed where the relative ordering of the dynamic and static strategies are reversed from Figure V.8 to Figure V.9. While the order of the strategies is quite similar between the two sets of results, Figure V.9 provides more information about the performance impact of each scheduling strategy.

For example, the average percent degradation for the user model is higher for the three-site testbed cases than for the one-site cases. Recall that our user strategy assumes a particular preference ordering of target resources; this ordering is designed to emulate the preference a user typically shows for machines in their own administrative domain. Specifically, the ordering assumed in our user strategy is {UCSD, UTK, UIUC}. In reality, the UTK resources have the fastest processors and largest memories. The basic, static, and dynamic scheduling strategies automatically identify resources with the fastest processor speeds and largest physical memories, and, assuming the UTK resources are not overly loaded, often select machines in the UTK set first. Since the one-site testbed is more homogeneous than the three-site testbed, the effect is not as noticeable.

Summary statistics are useful for demonstrating overall trends, but can only provide a partial picture of the behavior of the four scheduling strategies. In the following sections we present a detailed examination of results for each application-testbed scenario.

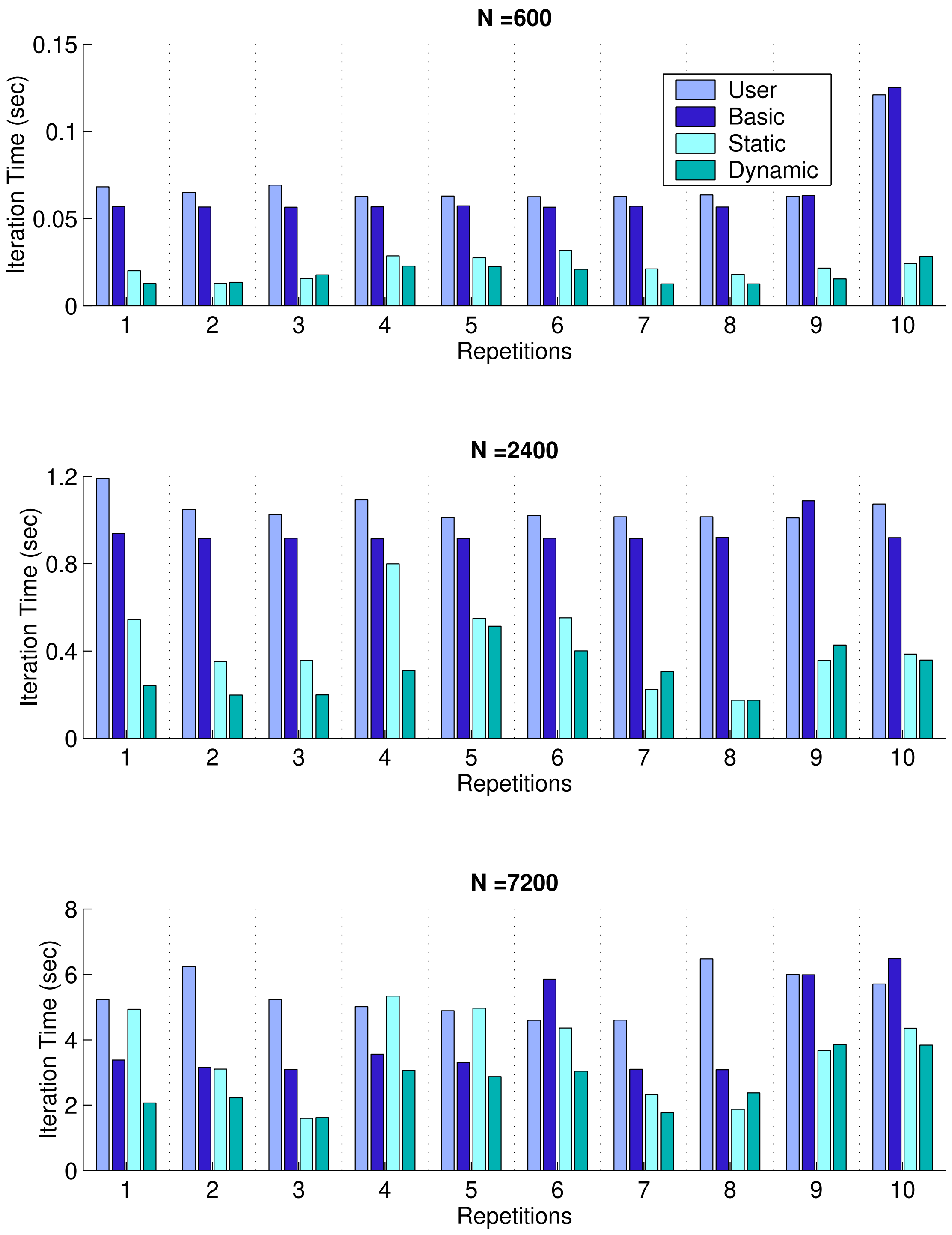


Figure V.10: Game of Life iteration times for the user, basic, static, and dynamic scheduling strategies. Experiments targeted the one-site testbed with problem sizes of 600, 2400, and 7200.

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

Table V.10: Summary statistics for percent degradation from best for each scheduling strategy over all application-testbed scenarios.

Game of Life

A subset of the scheduling strategy comparison experiments we ran for the Game of Life on the one-site testbed are shown in Figure V.10; all repetitions are shown for N = {600, 2400, 7200}, three of the six problem sizes tested for this testbed. For $N = 600$, performance of the user and basic strategies are relatively similar as are the performance of the static and dynamic strategies; the static and dynamic strategies achieved significantly improved performance as compared to the user and basic strategies. These results indicate that for this scenario and problem size, the availability of more sophisticated application information had a more significant performance impact than the availability of sophisticated Grid resource information. By comparison, for several of the repetitions for $N = 7200$ (specifically repetitions 1, 4, and 5), the basic and dynamic strategies perform significantly better than the user and static strategies; for these comparison runs, the availability of dynamic Grid information had a more significant performance impact than did more sophisticated application information.

Table V.11 reports the average rank and degradation from best for each scheduling strategy; results are reported for each problem size. In each series, 10 repetitions are performed; typically no more than 1 or 2 repetitions per series failed. Failures are occasionally caused by a scheduling strategy's inability to satisfy application memory requirements. More often, the application itself fails due to an error in allocating memory, an authentication error, or a Globus communication error. Recall that ranks are

	User		Basic		Static		Dynamic	
	Rank	Deg.	Rank	Deg.	Rank	Deg.	Rank	Deg.
N = 600	3.8	326.4%	3.2	286.7%	1.7	31.2%	1.3	3.7%
N = 1200	3.9	309.7%	3.1	277.4%	1.8	35.6%	1.2	11.8%
N = 2400	3.9	296.3%	3.1	251.5%	1.8	49.2%	1.2	5.6%
N = 4800	3.9	308.4%	3.1	273.6%	1.7	23.1%	1.3	2.0%
N = 7200	3.6	126.5%	2.9	59.5%	2.2	41.3%	1.3	3.3%
N = 9600	3.0	54.2%	3.0	63.7%	2.7	44.0%	1.3	4.4%

Table V.11: Average rank and percent degradation from best for each scheduling strategy for the Game of Life on the one-site testbed.

	User		Basic		Static		Dynamic	
	Rank	Deg.	Rank	Deg.	Rank	Deg.	Rank	Deg.
N = 600	3.8	143.1%	2.6	70.1%	1.8	49.6%	1.8	10.9%
N = 4800	3.8	1036.6%	3.0	660.5%	1.9	31.5%	1.3	2.2%
N = 9600	3.6	454.0%	3.2	325.3%	1.9	44.6%	1.3	1.9%
N = 14400	3.7	352.0%	3.1	134.1%	1.6	23.9%	1.6	0.7%
N = 16800	3.6	131.4%	3.3	89.5%	1.8	21.7%	1.3	4.5%
N = 19200	3.9	141.3%	3.0	44.3%	1.8	16.5%	1.3	3.2%

Table V.12: Average rank and percent degradation from best for each scheduling strategy for the Game of Life on the three-site testbed.

assigned to failed runs (a failure receives the lowest ranking); average ranks are therefore computed over 10 runs. Percent degradation from best values are not computed for failed runs; average values for degradation from best are therefore averages over successful runs only.

In Figure V.11 we present scheduler comparison runs for the Game of Life on the three-site testbed; three of the six problem sizes tested are shown: $N = \{600,\ 9600,\ 16800\}$. In these results it is striking that the relative performance of the four scheduling strategies is quite variable across repetitions and problem sizes. Nonetheless, the general trend is an improvement in application performance from the user strategy to the dynamic strategy. Table V.12 reports average rank and average degradation from best for all six problem sizes tested for this scenario.

Jacobi

In Figure V.12 we present a subset of the scheduling strategy comparison experiments we ran for the Jacobi application on the one-site testbed. As was the case for the results presented in Figure V.10, these results suggest that the availability of more sophisticated Grid information did not have a significant performance impact at the smaller problem sizes. Table V.13 summarizes the average rank and average

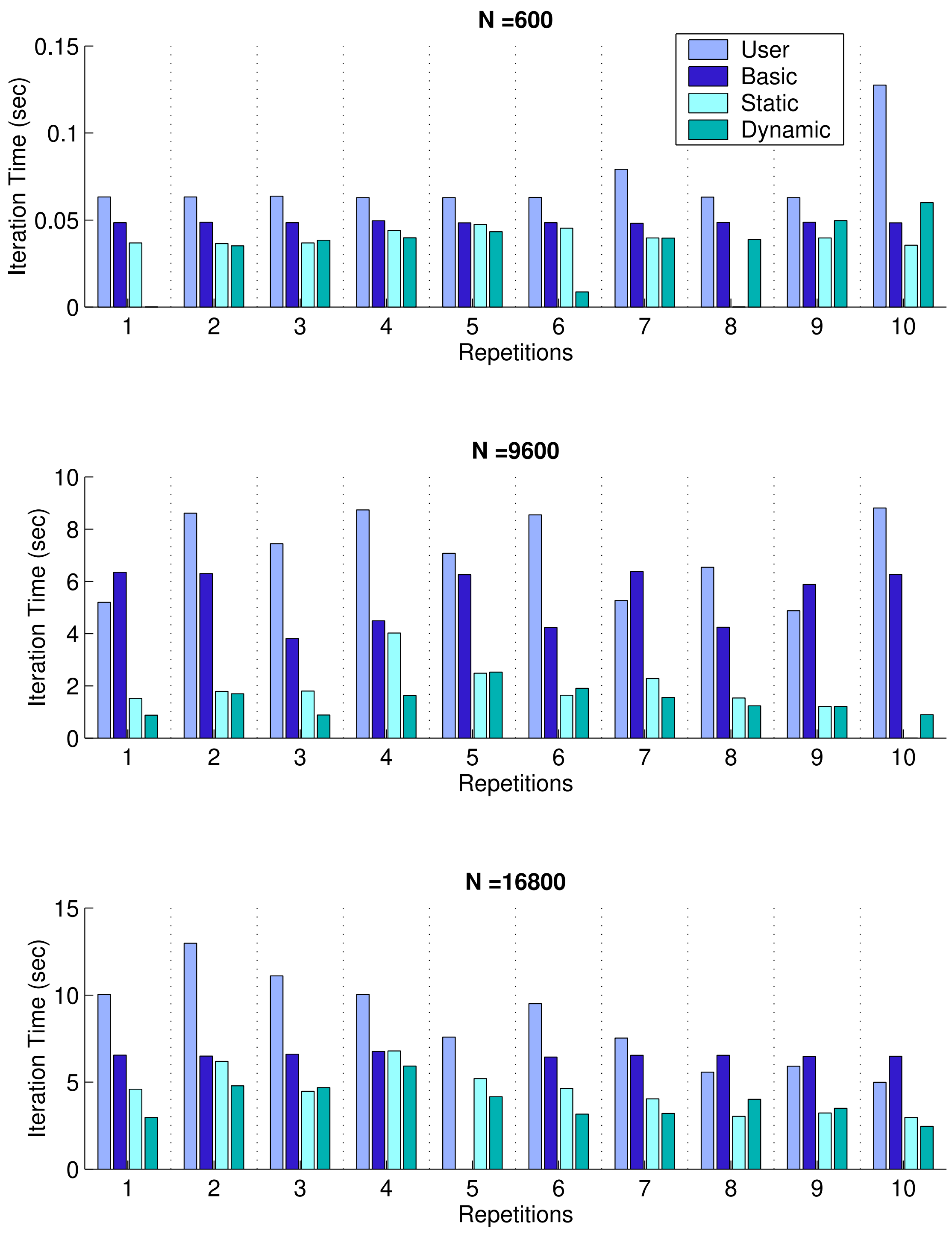


Figure V.11: Game of Life iteration times for the user, basic, static, and dynamic scheduling strategies. Experiments targeted the three-site testbed with problem sizes of 600, 9600, and 16800.

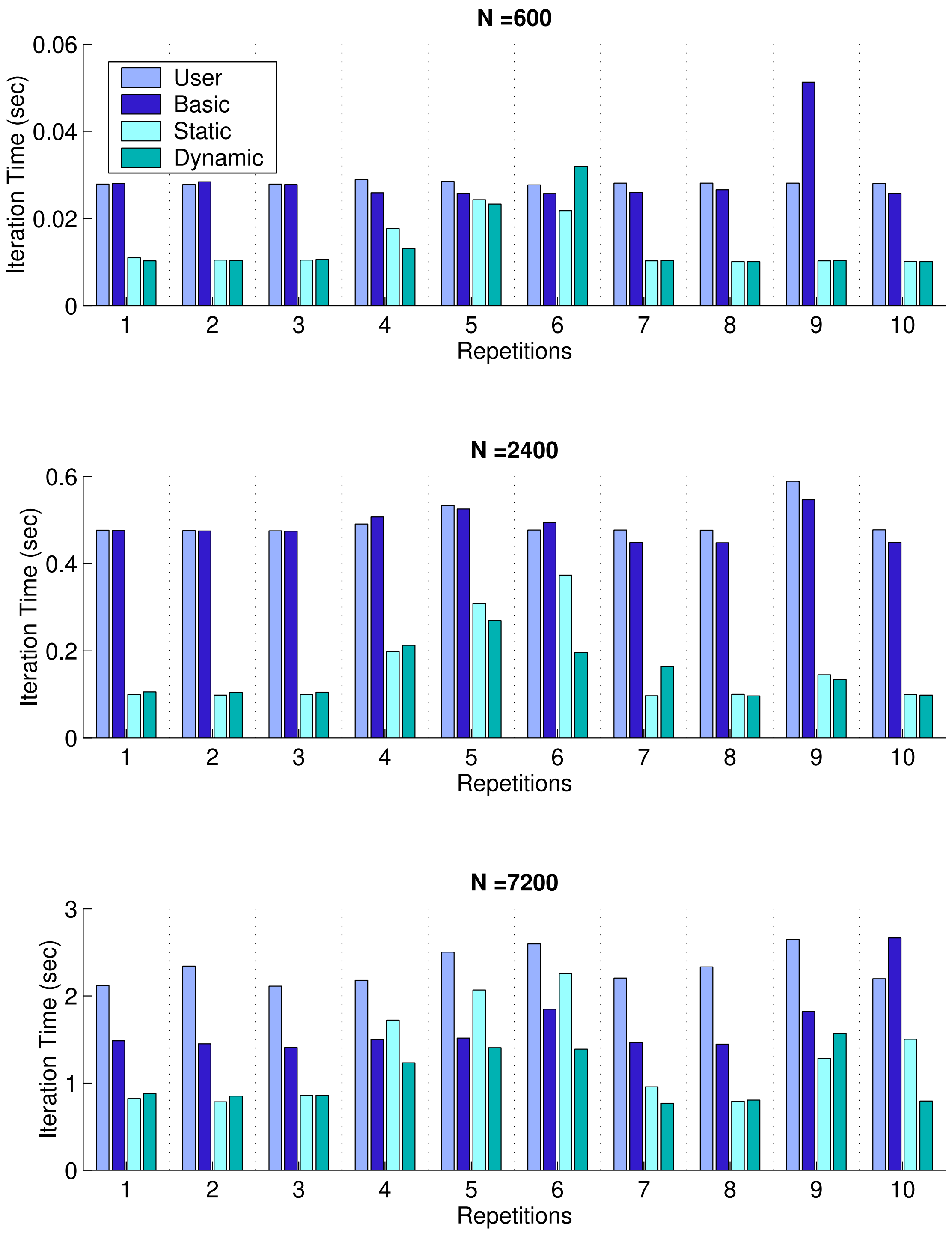


Figure V.12: Jacobi iteration times for the user, basic, static, and dynamic scheduling strategies. Experiments targeted the one-site testbed with problem sizes of 600, 2400, and 7200.

	User		Basic		Static		Dynamic	
	Rank	Deg.	Rank	Deg.	Rank	Deg.	Rank	Deg.
N = 600	3.6	137.5%	3.2	150.5%	1.5	4.8%	1.7	5.0%
N = 1200	3.9	238.6%	3.1	214.9%	1.7	20.9%	1.3	4.9%
N = 2400	3.8	303.2%	3.2	292.3%	1.5	11.8%	1.5	9.4%
N = 4800	3.8	355.7%	3.2	342.1%	1.6	14.2%	1.4	5.0%
N = 7200	3.9	140.5%	2.8	74.1%	1.9	26.3%	1.4	3.9%
N = 9600	3.6	55.5%	3.1	47.3%	1.8	25.0%	1.5	6.1%

Table V.13: Average rank and percent degradation from best for each scheduling strategy for Jacobi on the one-site testbed.

	User		Basic		Static		Dynamic	
	Rank	Deg.	Rank	Deg.	Rank	Deg.	Rank	Deg.
N = 600	3.7	249.3%	2.7	159.4%	1.9	100.5%	1.7	23.3%
N = 4800	3.9	706.2%	3.1	577.4%	1.3	17.5%	1.7	5.8%
N = 9600	3.3	238.9%	3.2	174.1%	1.8	16.7%	1.7	16.8%
N = 14400	3.9	441.0%	2.8	93.7%	1.5	22.6%	1.8	8.3%
N = 16800	3.7	401.2%	2.5	26.4%	2.7	204.3%	1.1	0.6%
N = 19200	3.4	—	2.6	188.3%	1.3	0.1%	2.7	22.9%

Table V.14: Average rank and percent degradation from best for each scheduling strategy for Jacobi on the three-site testbed.

degradation from best for each scheduling strategy and each problem size for Jacobi on the one-site testbed.

Figure V.13 presents a subset of the scheduling strategy comparison experiments performed for the Jacobi application on the three-site testbed. In this set of experiments, application performance with the user, basic, and static strategies was highly variable as compared to the results for the other three application-testbed scenarios. With the exception of the first repetition, the dynamic strategy resulted in consistent application iteration times. In Section V.B we demonstrated that a greater fraction of iteration time was typically dedicated to communication for the Jacobi application on the three-site testbed than for any other application-testbed scenario. Variations in wide-area network performance could explain the application iteration time behavior of the user, basic, and static strategies. This explanation would suggest that the dynamic strategy effectively avoided wide-area links with degraded performance. More experiments are needed to fully substantiate this hypothesis. Table V.14 reports average rank and average degradation from best values for each scheduling strategy for Jacobi on the three-site testbed.

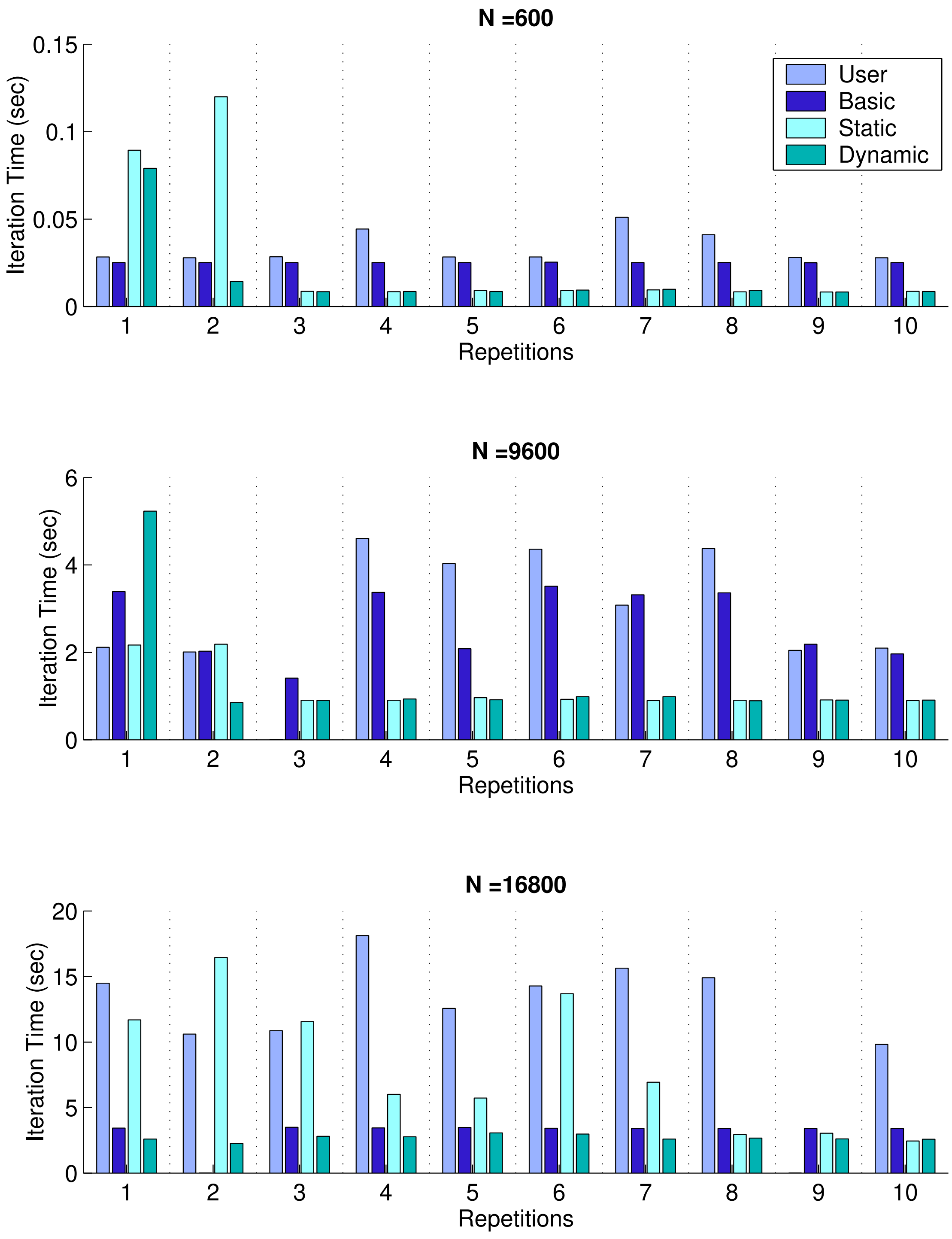


Figure V.13: Jacobi iteration times for the user, basic, static, and dynamic scheduling strategies. Experiments targeted the three-site testbed, problem sizes of 600, 9600, and 16800.

V.D.3 Summary

Our goal in this section was to evaluate the efficacy of our scheduling methodology. In particular, we asked the following three questions in the beginning of the section, repeated here for reference.

i. What is the impact of our scheduling methodology on application execution times as compared to conventional scheduling approaches?

ii. What is the impact of application information availability on scheduler performance? Specifically, (a) can this methodology develop reasonable schedules despite limited application information and models, and (b) can this methodology take advantage of more sophisticated information and models to promote application performance?

iii. How is application performance affected when dynamic resource information is available to our scheduling methodology? Can the methodology develop reasonable schedules when only static resource information is available?

To answer question (i), we introduced a user scheduling strategy and compared its performance to the basic, static, and dynamic strategies, each of which was based on a different configuration of our scheduler design. We presented experimental results showing that the basic, static, and dynamic strategies all consistently outperformed the user strategy.

To answer question (ii), we compared the performance of strategies that used our execution time + memory usage model (the static and dynamic strategies) against the performance of strategies that used only the memory usage model (user and basic strategies). On average, the static and dynamic strategies outperformed the user and basic strategies for all application-testbed scenarios, showing that scheduler was able to utilize more sophisticated application performance models to promote application performance. The consistent performance advantage provided by the basic strategy as compared to the user strategy suggests that our scheduling methdology is able to develop reasonable schedules despite limited application information and models.

To answer question (iii), we compared the performance of strategies that used dynamic resource availability information (the basic and dynamic strategies) against strategies that used only static resource information (the user and static strategies). We found that, on average, the basic strategy outperformed the user strategy and the dynamic strategy outperformed the static strategy, showing that availability of dynamic resource availability information improved scheduler peroformance. Since the static strategy consistently outperformed the basic strategy, we conclude that performance model sophistication had a larger impact on application performance than did availability of dynamic resource information.

V.E Scheduling overhead

A scheduler design is practical only if the overhead of the scheduling process is reasonable when compared to application execution times. In previous sections of this chapter we have used application iteration time as a performance metric and have therefore not investigated the overheads introduced by the scheduler itself. In this section we describe results that quantify scheduler overhead. Recall that our scheduler design consists of two distinct activities: the collection of Grid resource information (described in Section III.C) and the search for candidate schedules (described in Section III.B). We examine the cost of each of these activities as well as the total cost of scheduling. To quantify the cost of Grid information collection under different information source scenarios, we include test scenarios in which information is retrieved from the GrADS NWS, the GrADS MDS, the local NWS nameserver, and the local MDS cache; each of these collection mechanisms is described in Section V.A.

Note that the cost of scheduling is not fixed; instead, it is dependent on a wide variety of factors including, for example, problem run configuration, the selected testbed, the target application, the complexity of the chosen performance model and maper, and variable load on the GrADS MDS server and NWS nameserver. For example, the cost of retrieving resource information grows as the number of resources in the testbed and the cost of retrieving network information grows as the square of the number of sites in the testbed. For the schedule search procedure, the cost of scheduling increases with the number of resources and the number of sites (because the number of candidate schedules that must be considered increases) but it decreases as the amount of information about each resource decreases (because resource sets with insufficient resource information are pruned from the search space).

V.E.1 Experimental design

Approach

We focus on two representative scenarios which provide a broad picture of scheduling overhead and also demonstrate the general patterns that would be seen for other configurations. For both scenarios we use the Jacobi application and the scheduler is configured to use the execution time + memory usage model (see Section IV.B). The testbed and problem size assumptions for each scenario are as follows.

- In scenario 1 we target the one-site testbed and use a problem size of $N = 4800$.
- In scenario 2 we target the three-site testbed and use a problem size of $N = 14400$.

To examine the cost of retrieving information from a variety of sources, we test each of the following Grid information source modes for each of scenarios 1 and 2.

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

Experimental Procedure

Experiments for each scheduling scenario (i.e. testbed - problem size combination) were performed independently. For each scenario, we ran the scheduler with each of the three information source modes in a back-to-back manner; we completed 10 such triplets. For each run, we measured the time required for the entire scheduling execution ($TotalTime$) and the time required for Grid information collection ($CollectTime$); we consider the cost for the schedule search ($SearchTime$) to be all scheduling time that is not spent in information collection: $SearchTime = TotalTime - CollectTime$.

V.E.2 Results

One-site testbed, N = 4800

In this scenario, the scheduler selects amongst machines in the one-site testbed and the target problem size is $N = 4800$. For reference, in our scheduling experiments for this testbed and problem size, application iteration times were typically between 0.4 seconds and 2 seconds (0.4 seconds for the dynamic and static mode schedulers, 2 seconds for the basic mode and user scheduling strategies). Since we ran 100 iterations in those experiments the application's iterative phase typically took between 40 and 200 seconds.

Figure V.14 presents all 10 repetitions of the experiments performed for this scenario. The lower chart presents the same dataset as the upper chart, but with an expanded y-axis to provide detail

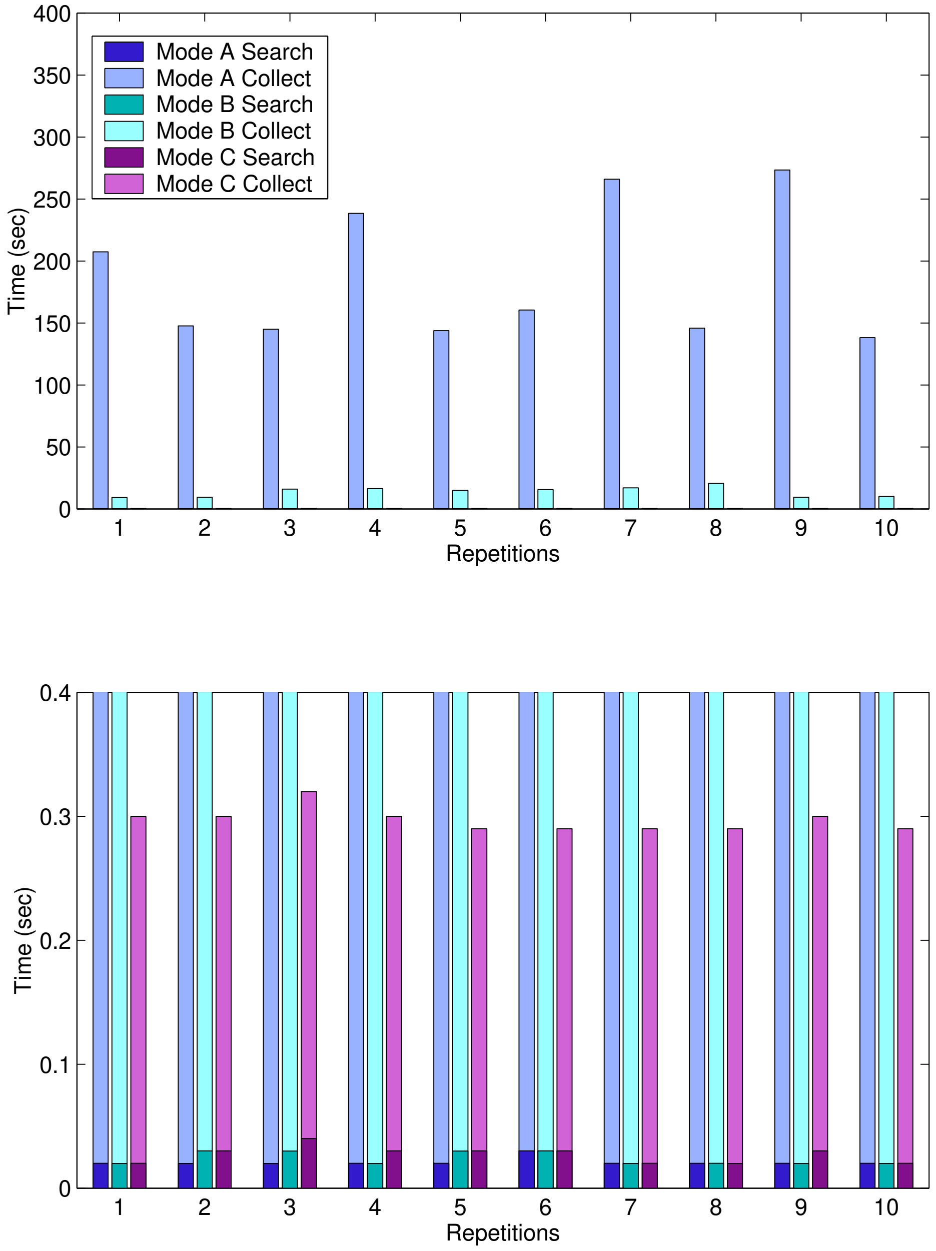


Figure V.14: Summary of Grid information collection and schedule search times for the one-site testbed, $N = 4800$. The upper graph shows the full y-scale; the lower graph shows the same data set with an expanded y-scale.

	Mode A	Mode B	Mode C
Collect Time, Average	186.6	13.8	0.27
Collect Time, StdDev.	54.5	4.0	0.01
Search Time, Average	0.021	0.024	0.027
Search Time, StdDev.	0.003	0.005	0.007
Total Time, Average	186.6	13.8	0.30
Total Time, StdDev.	54.5	4.0	0.01

Table V.15: Scheduling overhead times for the one-site testbed, $N = 4800$.

of smaller scheduling overheads. The full height of each bar is the total scheduling overhead for that information collection mode; the lower, darker portion of each bar (visible only in the lower chart) is the search time and the upper portion of each bar is the cost of Grid information collection. Table V.15 presents summary results over all 10 repetitions for the mean and standard deviation of the collection times, the schedule search times, and the total scheduling time.

The cost of **Grid information collection** is clearly the primary scheduling overhead for all three collection modes. For an application that is expected to run for roughly 40-200 seconds, the cost of Grid information collection in mode A is prohibitive and, in practice, would likely prevent usage of this scheduling methodology. The cost is also significant for mode B, but is acceptable given the performance advantages one could expect to achieve with our scheduling methodology. Notice that information collection times vary significantly across repetitions for modes A and B; this is probably due to (1) variations in wide-area network performance between the scheduler and the remote servers and (2) variations in the load on the servers themselves. Finally, the overhead of information collection in mode C is very low. Overall, these results indicate that until retrieval times are reduced for the MDS, local caching of MDS information will be necessary. For this thesis, the information we retrieve from the MDS changes on the order of weeks or months so local caching is an acceptable solution. Since Grid information collection times are reasonable for mode B, which includes access to the GrADS NWS nameserver, we conclude that usage of a remote NWS nameserver is a reasonable information collection strategy.

The cost of the **schedule search process** is quite low and it is less than 0.05 seconds for all three collection modes. This low search time overhead is due to (1) the low computational complexity of our execution time model and mapping strategy and (2) the extensive search pruning performed during the search process. Notice that schedule search times do vary somewhat across repetitions and information retrieval modes; this is likely due to the effect that missing information has on the schedule

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

Table V.16: Scheduling overhead times for the three-site testbed, $N = 14400$.

search procedure; each information collection mechanism can have different information availabilities and schedule search space pruning is partially based on information availability. When less resource information is available, the schedule search process can generally be expected to take less time.

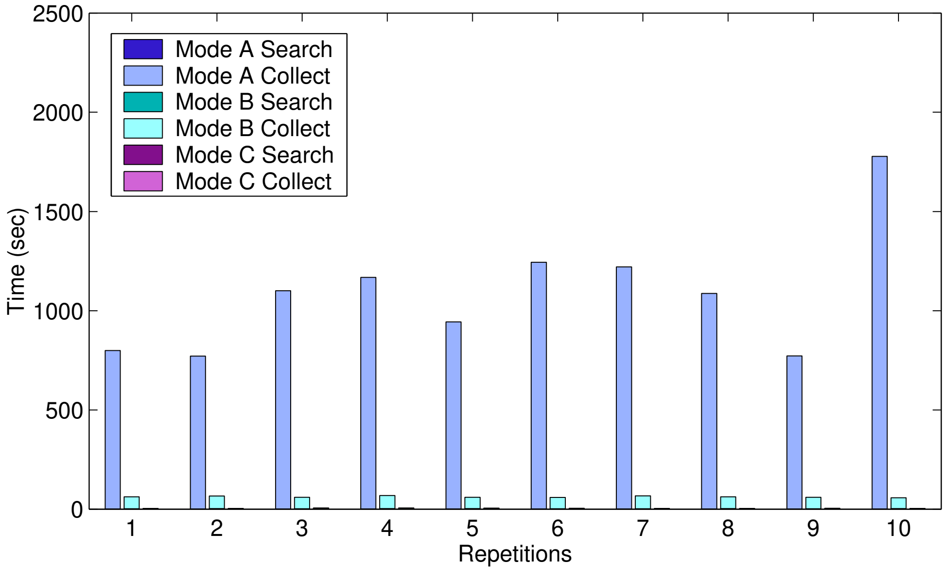
Three-site testbed, $N = 14400$

In this scenario, the scheduler selects amongst machines in the three-site testbed and the target problem size is $N = 14400$. For reference, in our scheduling experiments for this testbed and problem size, the four scheduling strategies typically achieved application iteration times between 1.8 and 11 seconds. Since we ran 100 iterations in those experiments the application's iterative phase occupied 180 to 1100 seconds.

Figure V.15 and Table V.16 present the results of experiments performed for this scenario. Notice that all scheduling overheads have increased for this scenario when compared with the one-site scenario. Both Grid information collection times and schedule search times increased in part because this testbed contains over three times as many resources and three times as many sites. Also notice that schedule search times are much lower for mode A than for the other modes; this is because the scheduler was unable to retrieve some resource characteristics from the GrADS MDS, thus leading to extensive pruning of the search space. Overall, we see that the overhead for the search process is still quite low but that the overhead of Grid information collection is prohibitive for mode A. These results substantiate our earlier claim that usage of a local MDS cache is necessary to provide scheduling with reasonably low overhead.

V.E.3 Summary

In this section we presented a quantitive evaluation of the overheads associated with our scheduling methodology. We specifically examined the overhead of Grid information collection and the schedule search process itself. We found that the cost of the schedule search process is insignificant when



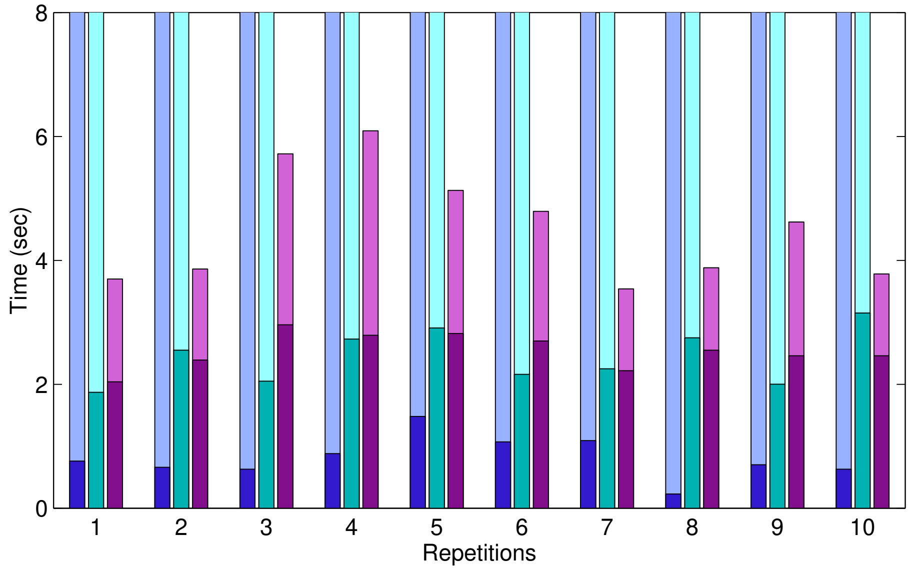


Figure V.15: Summary of Grid information collection and schedule search times for the three-site testbed, $N = 14400$. The upper graph shows the full y-scale; the lower graph shows the same data set with an expanded y-scale.

compared with application execution times. We also showed that the cost of resource information retrieval is reasonable when the information source is the GrADS NWS, the local NWS, or the local MDS cache. On the other hand, Grid information collection times are on the same order as application execution times when the remote GrADS MDS is utilized. Since these overheads are unacceptable and the information we retrieve from the MDS is relatively static, we conclude that usage of a local MDS caching mechanism is an appropriate alternative in the current GrADS testbed environment.

V.F Chapter summary

Our primary goal in this chapter was to investigate the impact of our scheduling methodology on application performance. Since our scheduling methodology is highly dependent on the availability of an application performance model and mapping strategy, we presented a suite of experiments designed to specifically test the application-specific execution time model and mapping strategies we developed in Chapter IV. We concluded that our execution time model provided reasonable prediction accuracy, and was able to correctly track application performance trends. We also presented experiments that verified the utility of each of our mapping strategies and showed that, on average, the time balance mapper provided an application performance advantage when compared to the equal allocation mapper.

After validating our application specific execution time model and mappers, we presented experiments to test our scheduling strategy itself. In these experiments, we demonstrated that our scheduling methodology provides a significant performance advantage over a more conventional scheduling strategy. We also showed that the scheduler is able to develop adequate schedules despite limited application or resource information, but that it is also able to take advantage of more sophisticated information to promote application performance.

The last set of results that we presented tested the overhead associated with the scheduling process itself. We found that the overheads associated with the schedule search process and Grid information collection from the GrADS NWS, local NWS, and local MDS cache are reasonable, while the cost of information retrieval from a remote GrADS MDS server is unacceptable given typical run-times for our applications. We concluded that usage of a local MDS caching mechanism is an appropriate alternative for the purposes of this thesis.

Chapter VI

Discussion

In this thesis we propose an adaptive, run-time scheduling methodology designed to promote the performance of iterative, mesh-based applications in Computational Grid environments. In this chapter we present a final discussion of the thesis. Specifically, in Section VI.A we summarize the thesis and reiterate our findings. In Section VI.B we describe related work in the field of application scheduling. Finally, in Section VI.C we describe possible extensions to our work.

VI.A Summary and findings

The scheduling design proposed in this thesis was developed in the context of the larger Grid Application Development Software project (GrADS). In Chapter II we described the design of the Grid application development infrastructure, termed GrADSoft, proposed by the GrADS project. We also presented the specification for the GrADSoft scheduler.

In Chapter III we presented a design for a scheduler framework that satisfies the GrADSoft scheduler specification and is the first prototype of a scheduler for the GrADSoft infrastructure. Our scheduler design incorporates an "intelligent" schedule search procedure that considerably prunes the search space of possible resource groups while ensuring that desirable resource groups are not excluded. The scheduler utilizes dynamic and static Grid resource information to target schedules to the conditions of Grid resources at run-time. We also described a number of scheduling policies that enable straightforward configuration of scheduler behavior. The scheduler framework presented in this chapter is application-generic; it is designed to be coupled with an application-specific performance model and mapping strategy.

In the beginning of Chapter IV we described the general characteristics of iterative, mesh-based applications. Next, we described two such applications in detail (Jacobi and Game of Life) and presented a detailed performance analysis of each application. We then developed two application-specific performance models, a memory usage model and an execution time + memory usage model. We also detailed two application-specific mapping strategies, an equal allocation mapper and a time balance mapper.

In Chapter V we presented experiments that demonstrated the efficacy of our scheduling methodology for realistic applications, testbeds, and usage scenarios. We took a two-fold validation approach. First, we presented experiments that demonstrated the prediction accuracy of our execution time model and the utility of each of our mapping strategies. Second, we defined four reasonable scheduling strategies for our target applications and environments, and then presented experiments that compared the application performance achieved with each scheduling strategy. We showed that our scheduler design provided significantly enhanced application performance as compared to a conventional scheduling strategy. We also demonstrated that our methodology was able to (1) take advantage of sophisticated application and resource information to promote application performance, and still (2) provide a reasonable scheduling service when only limited application and resource information was available. Finally, we presented experiments to examine the overheads associated with the scheduling process itself. We found that the overhead of our schedule search process was nominal, and that overheads associated with Grid information collection were, for the most part, also acceptable. We found the overheads associated with the retrieval of data from a remote MDS to be unacceptable for the needs of run-time application scheduling; we observed that usage of a local MDS cache was a reasonable solution for our purposes.

VI.B Related work

Many of the strategies utilized by our scheduling design are based upon experience gained in previous Application-Level Scheduling (AppLeS) efforts [9, 10, 12, 48, 50]. Two of these efforts targeted structurally similar applications and are therefore particularly relevant [9, 12]. The first focused on the scheduling of a Jacobi solver for the finite-difference approximation to Poisson's equation [9]. The second effort focused on scheduling of a parallel magnetohydrodynamics simulation (PMHD3D), which is also classified as an iterative, mesh-based application [12]. Each of these efforts demonstrated significant improvements in application performance as compared to conventional scheduling efforts. For the design presented in this thesis, we drew on the experiences gained in these efforts. There are a number of novel aspects to the current work.

- Our scheduler design provides a separation of the application-generic scheduling mechanisms from application-specific performance models and mappers; we expect the scheduler will be more easily targeted to new applications that the Jacobi and PMHD3D schedulers.
- Our schedule search procedure is based on a more general heuristic that we believe is more likely to discover all desirable resource sets.
- Our design has been thoroughly tested on both a local-area network of workstations and a heterogeneous Computational Grid including wide-area links; each of the previous efforts targeted resources at a single site.

Another related effort is Prophet, a run-time scheduling system designed for parallel applications written in the Mentat programming language [53, 52]. This scheduling system is similar to our work in that it exploits application structure and system resource information to promote application performance. Prophet was demonstrated for both SPMD applications and applications based on task-parallel pipelines; the scheduler design was tested in heterogeneous, local-area environments. If possible, we would like to compare the performance of our strategies to those of Prophet, though it may be difficult to find a suitable scenario for comparison that satisfies the requirements of each scheduling strategy. For example, Prophet requires the target application be written in Mentat and we have not used Mentat in our efforts.

There are a number of additional scheduling projects that are notable for targeting a variety of applications or an entire application class [10, 42, 52, 1, 46]. Many of these efforts focus on embarrassingly parallel or master-slave applications which do not have significant communication costs [10, 1, 46]. We describe selected projects that focus on application classes that involve significant communication costs.

The Prophet scheduling system is also a notable example of a scheduler design that targets a variety of applications [53, 52]. Prophet requires modification of application source code and has not been tested in the wide-area. As mentioned earlier, a performance comparison of the two strategies would be quite interesting.

Another project of interest is the Condor matchmaking system [42]. In the matchmaking system, users specify the resource requirements of their application to the system, resource providers similarly specify the capabilities of their resources, and a centralized matchmaker is used to *match* application resource requirements with appropriate resources. This design is quite general and can therefore be applied to many different types of applications. The matchmaking strategy, while more general that the scheduler presented in this thesis, differs in that it is primarily a resource discovery mechanism and is not able to provide detailed schedule development.

VI.C Future work

We plan to extend our work to support other applications and other application classes. For applications that share the broad application resource requirements described in Chapter III.A, our design should be directly applicable. We plan to verify this assertion by testing our methodology for additional applications; for each application, an application-specific performance model and mapping strategy will be required. To support cases where application performance is heavily dependent on the selection of several distinct resource groups, our design must be extended. In particular, we will modify the search procedure to independently search for resources to satisfy each resource group requirement. In this modification we will need to ensure that resources selected to satisfy one group requirement are excluded from the search for other group requirements.

Another direction in which our work could be extended involves the type of application information and models used by our scheduling methodology. For the purposes of this thesis, we designed and built the application performance models and mapping strategies. However, if Grid application development is to be accessible to a larger number of users, then we cannot expect such users to provide detailed performance models and mapping strategies. Recognizing this, other members of the GrADS research community are investigating the feasibility of compiler generation of application information and performance models [29] as well as the inclusion of such models in Grid-enabled libraries [29, 37]. As this work matures we are interested in experimenting with the usage of such models for application scheduling. These models may not be equational in form; in this case we will need to extend our methodology to support additional performance model types.

Bibliography

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

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

[3] Gabrielle Allen, David Angulo, Ian Foster, Gerd Lanfermann, Chuang Liu, Thomas Radke, Ed Seidel, and John Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a grid environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[4] AppleSeeds website at `http://gridlab.ucsd.edu/appleseeds`.

[5] Mohammad Banikazemi, Jayanthi Sampathkumar, Sandeep Prabhu, Dhabaleswar K. Panda, and P. Sadayappan. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *Proceedings of the 8th Heterogeneous Computing Workshop*, April 1999.

[6] Richard Barrett, Michael W. Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[7] Fran Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software support for high-level Grid application development. *International Journal of Supercomputer Applications*, 15(4):327–344, 2001.

[8] Francine Berman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 12: High-Performance Schedulers, pages 279–309. Morgan Kaufmann Publishers, Inc., 1999.

[9] Francine Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*, November 1996.

[10] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the Grid. In *Proceedings of Supercomputing*, November 2000.

[11] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing*, August 2001.

[12] Holly Dail, Graziano Obertelli, Francine Berman, Rich Wolski, and Andrew Grimshaw. Application-aware scheduling of a magnetohydrodynamics application in the Legion Metasystem. In *Proceedings of the 9th Heterogenous Computing Workshop*, May 2000.

[13] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A directory service for configuring high–performance distributed computations. In *Procedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, August 1997.

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

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

[16] Ian Foster, Jonathan Geisler, William Gropp, Nicholas Karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the Message Passing Interface. *Parallel Computing*, 24(12):1735–1749, 1998.

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

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

[19] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., 1999.

[20] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

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

[22] Globus webpage at `http://www.globus.org`.

[23] Grid Application Development Software Project webpage at `http://hipersoft.cs.rice.edu/grads`.

[24] GrADSoft Prototype webpage at `http://gridlab.ucsd.edu/~grads/GrADSoft_htmldoc`.

[25] Andrew Grimshaw, Adam Ferrari, Frederick Knabe, and Marty Humphrey. Wide-Area Computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–37, May 1999.

[26] Mark Gritter and David R. Cheriton. An architecture for content routing support in the internet. In *Proceedings of USENIX USITS*, March 2001.

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

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

[29] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing.* Accepted for Publication.

[30] Ken Kennedy, Mark Mazina, Ruth Aydt, Celso Mendes, Holly Dail, and Otto Sievert. 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.

[31] Balachander Krishnamurthy and Jia Wang. On network-aware clustering of web clients. In *Proceedings of ACM SIGCOMM*, August 2000.

[32] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[33] Linear Programming FAQ webpage at `http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html`.

[34] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

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

[36] Nancy Miller and Peter Steenkiste. Collecting network status information for network-aware applications. In *INFOCOM'00*, March 2000.

[37] Dragan Mirkovic, Rishad Mahasoom, and Lennart Johnsson. An adaptive software library for fast fourier transforms. In *Proceedings of the 2000 International Conference on Supercomputing*, 2000.

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

[39] MPICH-G webpage at `http://www.niu.edu/mpi`.

[40] Network Weather Service webpage at `http://nws.cs.utk.edu`.

[41] Peter S. Pacheco. *Parallel Programming With MPI*, chapter 10, pages 218–225. Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1997.

[42] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.

[43] Jennifer M. Schopf and Francine Berman. Performance prediction in production environments. In *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, 1998.

[44] Gary Shao, Fran Berman, and Rich Wolski. 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.

[45] Gary Shao, Rich Wolski, and Fran Berman. Predicting the cost of redistribution in scheduling. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[46] Gary Shao, Rich Wolski, and Francine Berman. Master/slave computing on the Grid. In *Proceedings of the 9th Heterogenous Computing Workshop*, May 2000.

[47] Shava Smallen, Henri Casanova, and Francine Berman. Applying scheduling and tuning to on-line parallel tomography. In *Proceedings of Supercomputing Conference*, November 2001.

[48] Shava Smallen, Walfredo Cirne, Jaime Frey, Francine Berman, Rich Wolski, Mei-Hui Su, Carl Kesselman, Steve Young, and Mark Ellisman. Combining workstations and supercomputers to support Grid applications: The parallel tomography experience. In *Proceedings of the 9th Heterogenous Computing Workshop*, May 2000.

[49] Neil Spring and Rich Wolski. Application level scheduling of gene sequence comparison on metacomputers. *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998.

[50] Alan Su, Francine Berman, Richard Wolski, and Michelle Mills Strout. Using AppLeS to schedule simple SARA on the Computational Grid. *International Journal of High Performance Computing Applications*, 13(3):253–262, 1999.

[51] Frederik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance Contracts: Predicting and monitoring grid application behavior. In *Proceedings of the 2nd International Workshop on Grid Computing*, Nov 2001.

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

[53] Jon Weissman and Xin Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1(1):109–118, 1998.

[54] H.P. Williams. *Model Building in Mathematical Programming.* Wiley, Chichester, New York, second edition, 1995.

[55] Rich Wolski. Dynamically forecasting network performance using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[56] Rich Wolski, James S. Plank, John Brevik, and Todd Bryan. Analyzing market-based resource allocation strategies for the computational Grid. *International Journal of High Performance Computing Applications*, 15(3):258–281, 2001.

[57] Rich Wolski, Neil Spring, and Chris Peterson. Implementing a performance forecasting system for metacomputing: The Network Weather Service. In *Proceedings of Supercomputing Conference*, November 1997.

[58] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *The Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.