A Framework for Effective Scheduling of Data-Parallel Applications in Grid Systems

---

A Thesis
Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
of the requirements for the Degree of
Master of Science in Computer Science

Michael Walker
May 2001

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the
requirements for the degree of Master of Science in Computer Science

____
Michael P. Walker

This thesis has been read and approved by the examining Committee:

____
Thesis advisor: Andrew Grimshaw

____
Committee Chair: Marty Humphrey

____
Minor Representative: Anand Natrajan

Accepted for the School of Engineering and Applied Science:

____
Dean Miksad
School of Engineering and Applied Science
August 2001

Abstract

Grid systems – a unified collection of resources connected by a network – have potential to deliver high performance for many applications and many system users. Achieving high performance in a grid system requires effective resource scheduling. The heterogeneous and dynamic nature of the grid, as well as the differing demands of applications run on the grid, makes grid scheduling complicated. Existing schedulers in wide-area heterogeneous systems require a large amount of information about the application and the grid environment to produce reasonable schedules [Lo88][Wei95][Wei00]. However, this information may not be available, may be too costly to collect, or may increase the run-time overhead of the scheduler such that the scheduler is rendered ineffective. We believe that no one scheduler is appropriate for all grid systems. Instead, grid systems require a *scheduling framework* that produces reasonable schedules for a variety of applications, and allows for more sophisticated scheduling mechanisms to be added to the framework as they are needed.

We propose a scalable, extensible framework for statically[1] scheduling parallel applications in a grid system. By default, a scheduler built within the framework will require only a small amount of information about the application and the state of the network. We then evaluate a scheduler implemented within the framework, and show that it produces reasonable task assignments for a variety of data-parallel applications without consuming much run-time overhead.

[1] In this context, a *static* schedule is produced before job execution, and is not modified during the course of execution.

Table of Contents

Abstract ... **iii**
List of Figures ... **v**
List of Tables ... **vi**
List of Symbols ... **vii**
Chapter 1 Introduction ... **1**
1.1 Thesis ... 5
1.2 Solution Strategy ... 5
Chapter 2 Related Work ... **7**
2.1 Scheduling Taxonomy ... 7
2.2 Grid System Scheduling ... 12
2.3 Existing Scheduling Heuristics ... 15
Chapter 3 Scheduling Framework ... **19**
3.1 Grid Scheduling Solution Space ... 19
3.2 Grid System Model ... 20
3.2.1 Resource Information Collection ... 21
3.3 Application Model ... 22
3.4 Performance Model ... 23
3.4.1 Identifying Scheduling Candidates ... 23
3.4.2 Network Organization Model ... 24
3.4.3 Fitness Equation ... 26
3.6 Scheduling Policy ... 28
Chapter 4 Scheduler Implementation ... **31**
4.1 Resource Information Collection ... 31
4.2 Grid System Model ... 32
4.3 Application Model ... 32
4.4 Performance Model ... 33
4.4.1 Performance Fitness Function ... 33
4.4.2 Network Organization ... 34
4.4.3 Latency Cost Function ... 35
Chapter 5 Evaluation ... **38**
5.1 Evaluation Test Suite ... 38
5.2 Testing Environments ... 39
5.3 Scheduling Policy Evaluation ... 40
5.3.1 Scheduling Policy Evaluation Results ... 41
5.3.2 Scheduling Policy Run Time Evaulation ... 46
5.4 Evaluation Summary ... 47
Chapter 6 Future Work ... **48**
6.1 Further Evaluation ... 48
6.2 Framework Extensions ... 49
References ... **50**

List of Figures

List of Tables

List of Symbols

Symbol	Definition
T	= set of tasks, or a job
k	= number of tasks
t_i	= the ith task
N	= set of machines
n	= number of machines
h_x	= a particular machine
C_i	= the ith machine cluster
p_i	= the ith processor
R	= network router
$\boldsymbol{t}_n$	= network topology
ASSIGNMENT	= set of assignments of tasks to machines
$Performance(t_i, h_x)$	= relative performance function
$Max_performance(t_i, h_x)$	= maximum architectural performance function
$Latency(t_i, h_x)$	= relative communication latency function
$processors(h_x)$	= number of processors function
$speed(h_x)$	= clock speed function
$load(h_x)$	= processor load function
$ASF(t_i, h_x)$	= architectural scale factor function
$Topology(h_x, h_y, \boldsymbol{t}_n)$	= topology communication cost function
ratio	= ratio of job communication to computation
RAND	= random scheduling algorithm
MP	= Max_performance scheduling algorithm
MPL	= Max_performance and latency scheduling algorithm

Chapter 1 Introduction

A *grid computing infrastructure* is a collection of resources connected by a network. In a typical grid infrastructure, the resource collection is a heterogeneous group of computers, and high-speed networking is used to accelerate communication between resources. Each computer may exhibit heterogeneity in its underlying hardware, operating system, file system, or network configuration.

A *grid computing system,* or *grid system*, runs programs on the resources in a grid infrastructure to form a unified system of interacting resources. The programs facilitate interaction with other resources in the system. The collection of programs managing the resource interaction is called *grid system middleware* because it forms a software layer above the native operating system that controls the interaction of the resource in the grid. The grid system middleware manages the underlying resource heterogeneity, and provides the abstraction of a unified system. The grid system middleware may support multiple grid system users.

A grid system user can run applications on various grid resources by running the application on top of the middleware layer. The benefits of exploiting a grid system for user applications are significant. For example, heterogeneous machines can be used for parallel program execution to shorten the overall application execution time. High-performance applications may benefit from the diversity of the grid infrastructure, since different pieces of the application can be run on the resources for which they are best suited.

A typical grid system may run a variety of user applications concurrently. One class of applications typically run on a grid system is a single-program, multiple-data

(SPMD) application, also known as a *data-parallel* application. Data-parallel applications are partitioned into tasks that perform computations on separate pieces of a data set. The tasks work together to process the entire data set, and are collectively called a *job*. The data-parallel program model is often used to solve scientific computing problems. These jobs may run for many hours or days, and can consume a large amount of system resources. The job may perform a large amount of computation, communication between tasks, or both.

A *parameter space study* is a job that repeatedly performs a large amount of computation over a range of program parameters. The total set of parameters can be considered the program data set. Each iteration of the program can be run in parallel on a grid system. Parameter space studies can therefore be considered a compute-intensive data-parallel application.

A *high performance grid system* should strive to maximize the total job throughput of the system, and minimize job execution time. These two goals can sometimes be complementary [Ber99]. For instance, if two jobs require p processors, and the grid infrastructure only provides $2p - 1$ processors, it may not be possible to obtain optimal performance for both jobs simultaneously. If both jobs are run concurrently, then at least two tasks will share the same processor, which increases the execution time of both jobs. However, serial execution of each job will lower the total job throughput of the system.

A *grid resource management system* controls resource usage to leverage the goals of a high performance grid system. A *grid scheduler* is the part of a grid resource management system that uses grid system and job information to produce an assignment

of tasks to machines for a given grid job. The assignment decision is known as task allocation or *task placement*, and the assignment itself is called a *schedule. Effective task placement decisions* produce schedules that strive to minimize job execution time.

Given a collection of jobs and their schedules, the resource management system may then create a separate sequencing of job execution times, or *meta-schedule*, that strives to maximize the total job throughput of the grid system[2]. In this way, scheduling and meta-scheduling work together to facilitate high performance in a grid system. The simplest meta-schedule policy allows all jobs to execute simultaneously. Determining the best meta-schedule policy for a grid system is an open problem, and is the subject of future work. We focus on job scheduling in grid systems.

A grid resource management system must be able to produce schedules with effective task placement decisions. Poor task placement will increase job execution time, which may reduce the total job throughput of the grid system. A grid system may run hundreds of jobs simultaneously. Thus, poor task placement decisions can dramatically reduce the performance of a grid system. However, producing good schedules (i.e. schedules with effective task placement) for grid jobs is a difficult problem.

Task placement is complicated by the composition of the grid system. Certain resources may finish the same task earlier than other resources. Placement decisions that do not consider the performance of a given task on a specific resource can result in increased execution time. The communication topology of the job may favor certain placement strategies over others. The resource characteristics of a grid may change

[2] The concept of meta-scheduling is sometimes known as grid co-scheduling. However, the term co-scheduling has other meanings in the context of networks of workstations (NOWs), and can be confused with the related concepts of co-allocation and advance reservation. The term meta-scheduling is used to emphasize the fact that the execution of individual job schedules is being sequenced in a meta-schedule.

unexpectedly. The changes may include variations in the number and type of resources in the system, resource availability, processor load, disk space, hardware configurations, network traffic, and available memory. Placement decisions that do not consider dynamic information about the grid system may place tasks on machines that cannot currently provide the proper resources to minimize completion time. Finally, a grid system may have multiple administrative domains, each of which allow different access rights to grid users. The placement decision must schedule tasks only on resources to which the user has proper access.

Jobs with different behavior characteristics may require different schedules. For example, differences in computational demands, patterns of communication, memory footprints, or disk I/O rates may influence the effectiveness of a given schedule for a particular job.

The amount of system and application information available to the scheduler determines its ability to make effective task placement decisions. The absence of information makes good scheduling very difficult to achieve. However, detailed application and system information may not be available, or may be too costly to collect. Also, an increase in the amount of information processed by the scheduler may increase the time to produce schedules. A task placement algorithm for grid systems must scale to schedule many tasks on many machines in a reasonable amount of time. Thus, a scheduler should balance the cost of collecting and processing information with the related benefits of the resulting schedule.

1.1 Thesis

The general task placement problem for more than two processors has been shown to be NP-complete [Ull75]. Various sub-optimal scheduling heuristics have been proposed in the literature [Lo88][Sal99][Wei95][Wei00][Zom01]. Each of the solutions is appropriate in certain situations, but may not be appropriate in others. Clearly, no one scheduling algorithm will be equally appropriate for all grid systems and all applications. Instead, grid systems require a *scheduling framework* that produces reasonable schedules for a variety of applications, and allows for more sophisticated scheduling mechanisms to be added to the framework as they are needed.

The main contribution of this thesis is the definition of such a grid scheduling framework and the evaluation of a simple scheduler built within the framework to schedule data-parallel applications. We demonstrate that the simple scheduler produces good schedules for a variety of data-parallel applications without requiring a large amount of program and system information. It adds little run-time overhead to the overall program execution, and will therefore scale well to schedule many jobs on the grid. The scheduling framework is extensible, which allows more sophisticated mechanisms to be added as needed.

1.2 Solution Strategy

The grid scheduling framework consists of (1) a grid system model, (2) an application model, (3) a performance model, and (4) a scheduling policy. The grid system model and application model provide information to influence the task assignment decision. We assume that a grid system may run many jobs concurrently, and this may

influence the grid system state. The grid system model will include information about the current state of the grid system.

The performance model provides a method of evaluating system and application information, and produces estimates of the performance of a given task on a particular resource. The scheduling policy uses the performance estimate to rank placement decisions according to its policy. Each unit of the framework must be extensible. This allows improvements to be added to the framework as they are needed.

The simple scheduler built within the framework uses basic grid system and application models that do not require extensive system benchmarking or application profiling. The performance model makes simplifying assumptions about expected performance of the network and each resource within the network. This allows a performance estimate to be made without extensive information. The scheduling policy uses a simple heuristic to rank task placement decisions.

We examine the benefits of the implemented scheduler, and show that even a simple scheduler can produce schedules that outperform random scheduling for a variety of data-parallel applications and grid infrastructures.

Chapter 2 Related Work

In this section, we present a taxonomy of scheduling techniques. We then describe and classify existing scheduling techniques within the context of that taxonomy. Finally, we place our scheduling framework within the taxonomy.

2.1 Scheduling Taxonomy

We present a scheduling taxonomy for enumerating possible scheduling techniques, classifying existing schedulers, and choosing guidelines for new schedulers. Cassavant and Kuhl present a general taxonomy of scheduling in distributed computing systems [Cas88]. Weismann presents a taxonomy of traditional parallel scheduling techniques [Wei95]. We base our taxonomy on their work.

The general scheduling problem has been interpreted a number of different ways in traditional operating systems and distributed systems [Sil98][Gon77][Wei95]. Running a data-parallel job on a grid system typically involves (1) job partitioning, (2) information collection, (3) task assignment, and (4) instantiation. *Job partitioning* is the division of a job into tasks. The granularity of the division should allow a high degree of parallelism. In grid systems, users may prefer to choose their own granularity, especially since the user may have written the application to be run at a particular granularity. Alternatively, specialized partitioning tools can be used to produce the partition [Wei95].

Information collection is the process of gathering user preferences, application information, and resource information. The collected information can be used to build a simplified model of the application, the grid, and the user preferences. In the *task assignment* stage, the scheduling algorithm then uses the available models to estimate

task performance and produce effective task assignments. The scheduling algorithm may use the collected information to produce better task assignments. Since this information is used in the scheduling algorithm, information collection is an integral part of the grid scheduling framework. Finally, the tasks are *instantiated* when they are executed on resources within the grid. Job instantiation may be user-initiated, or may be controlled by a meta-scheduler to maximize overall job throughput.

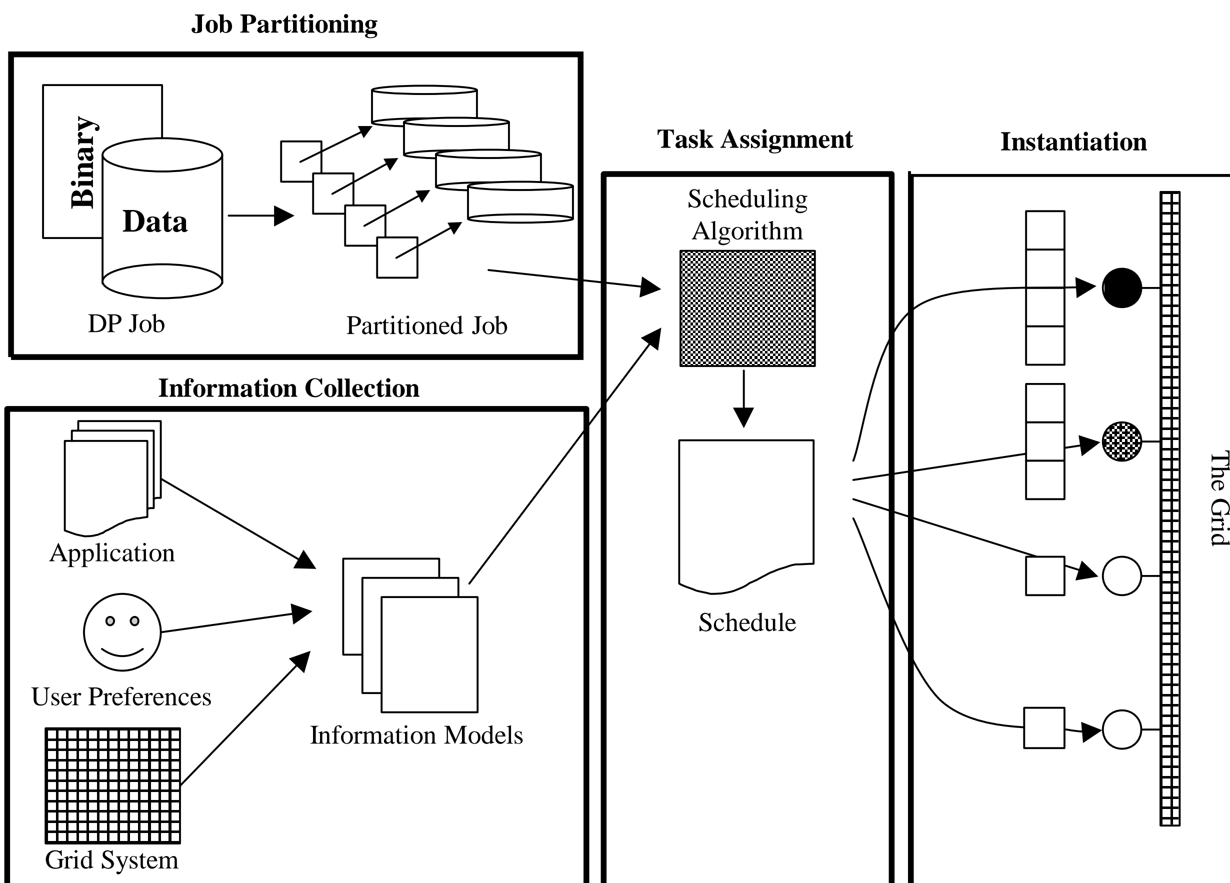


Figure 1: Steps in running a data-parallel job in a grid system. The solid lines surround the stages that comprise the scheduling framework.

This focus of this research is on global scheduling of data-parallel applications. *Global* scheduling is the process of deciding where to execute a job, and is also known as

the scheduling problem in distributed systems literature [Wei95], or more generally, the task placement problem. *Local* scheduling is the allocation of time slices of a single processor to its tasks. Local scheduling is usually managed by an operating system rather than a grid scheduler, and is outside the scope of this work.

Global scheduling strategies can be distinguished by the time at which the schedule is made. *Compile-time* scheduling produces schedules during application compilation, and generally will not formulate placement decisions based on dynamic system information. Unfortunately, compile-time scheduling strategies may not be useful for grid environments where system state varies. For instance, a compile-time scheduling decision may place tasks on unavailable resources, since resource availability information may not be known until run-time. The resultant schedule will not be effective, since some tasks cannot run unless their assigned resources become available.

Run-time scheduling can include dynamic system information in its scheduling decision. Within run-time approaches, scheduling can be done statically or dynamically. A *dynamic* run-time scheduler makes an initial assignment of tasks to machines, but it may change the assignment in response to changes in system performance. Dynamic run-time scheduling consumes some run-time overhead because it must monitor the grid system during job execution. A *static* run-time scheduler will decide task placement at the outset of program execution, and will not transfer tasks to other machines once they have begun execution. Static run-time schedulers may not incur the run-time overhead of the dynamic schedulers, but still require a large amount of program and system information to make a good initial placement decision.

Static run-time scheduling solutions present either an *optimal* solution to a simplified version of the scheduling problem, or a *sub-optimal* solution when the scope of the problem is NP-hard. For instance, optimal scheduling solutions for less than three processors have been formulated using graph-theoretic methods such as the max-flow/min-cut algorithm [Lo88]. However, no existing optimal solution has been demonstrated to extend to general scheduling of parallel applications on heterogeneous distributed systems without becoming computationally intractable.

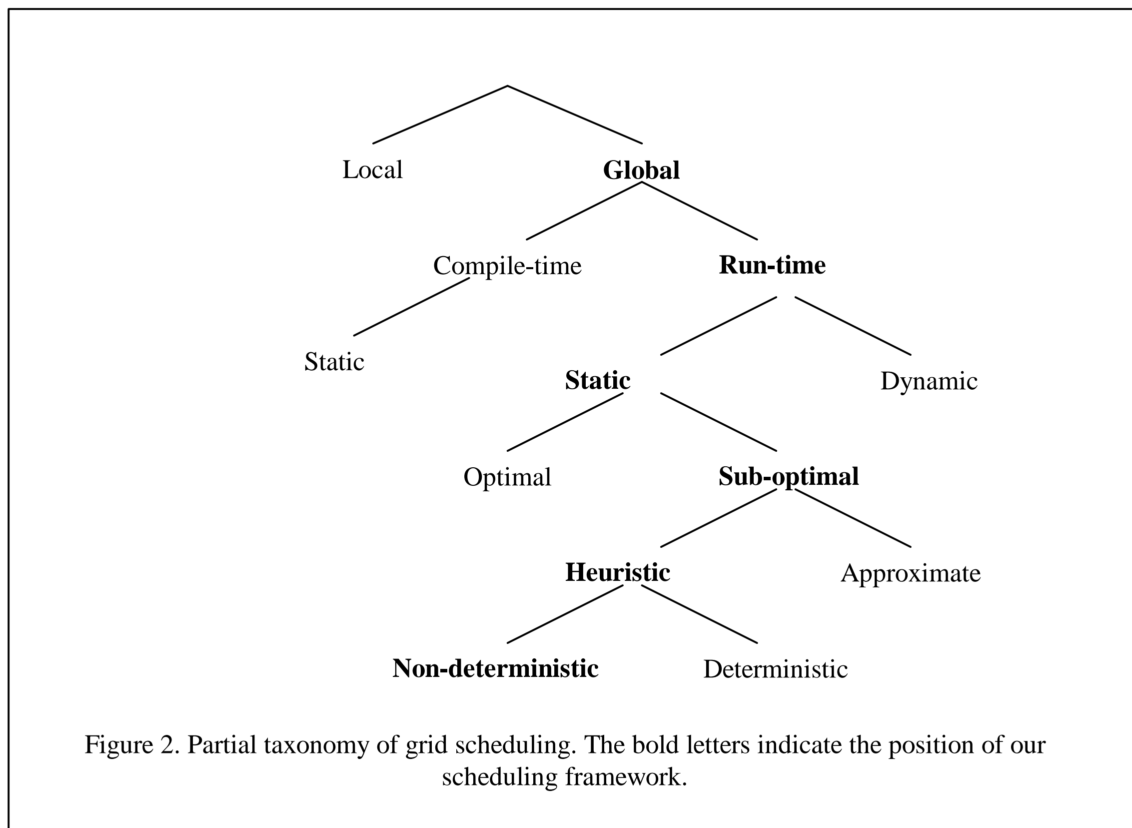


Figure 2. Partial taxonomy of grid scheduling. The bold letters indicate the position of our scheduling framework.

Sub-optimal scheduling solutions provide a near-optimal solution by maximizing a cost function. The cost function usually calculates the time to job completion given information about the tasks and the system. Casavant and Kuhl categorize sub-optimal scheduling as either *approximate* or *heuristic*. An *approximate* algorithm searches a

subset of the solution space to find a schedule that satisfies the cost criteria. A *heuristic* algorithm uses a non-optimal algorithm to make reasonable schedules. Many current solutions use a scheduling heuristic to maximize the cost function, while other existing strategies use genetic algorithms [Zom01], fuzzy logic, mean-field annealing, and simulated annealing [Sal99] methods.

Most static run-time heuristics require a large amount of program and resource information to provide near-optimal schedules. *Deterministic* scheduling heuristics require that precise information about the application and system resources is present before the application is executed. Typically, an application provides information about the task execution time on a given processor, the communication topology of the application, as well as the amount of communication expected of each task. Unfortunately, some parallel applications may exhibit non-deterministic behavior, and information about application behavior may not be completely predictable. Furthermore, a system shared by multiple users will also exhibit non-deterministic behavior unless it can make certain guarantees about the quality of service that it can provide. In cases where application or system behavior is non-deterministic, a *non-deterministic* scheduler must make decisions without the aid of precise application or resource information.

We propose a grid scheduling framework that supports static run-time scheduling in a non-deterministic environment. This framework will allow the opportunity to use system information available at run-time to aid in the scheduling process. The framework does not require deterministic program and system information because this information may not be available, or may be too costly to collect. The task assignment algorithm implemented for the framework uses a sub-optimal heuristic designed to schedule a wide

range of parallel applications in a reasonable amount of time. Its place in the scheduling taxonomy is illustrated in figure 2. The framework is described in detail in chapter 3.

2.2 Grid System Scheduling

A number of grid systems have implemented methods of scheduling grid jobs. The Application-Level Scheduler (AppLeS) [Zag98] is a user-level scheduler designed to run with an existing grid system. It uses dynamic system information, including network performance prediction information provided by Network Weather Service (NWS) [Wol98], to make accurate system performance predictions. AppLeS is a global, run-time scheduler that allows a variety of static and dynamic scheduling policies.

Each application must have its own AppLeS agent to produce schedules. The agent requires an application model, an application-specific resource usage function, and a scheduling policy. The scheduling policy can be user-defined, or can be one of a number of default strategies. The AppLeS scheduler provides extensibility in that the agent can be tailored to meet the needs of the application. The information required by an AppLeS agent may be difficult to provide in some situations, and the overhead of NWS may be a significant factor in the scheduling process. Our framework is different from AppLeS because it does not require run-time services like NWS by default, and because our framework focuses on providing a reasonable default scheduling policy.

The Network-based Information Library (Ninf) [Nak98] provides an extensible global scheduling framework for grid systems. It uses a resource status predictor and a global database of system information to make accurate predictions of system performance. Ninf researchers are currently exploring different scheduling algorithms to

use with the scheduling framework. The Bricks performance evaluation system for grid scheduling algorithms [Tak01] is a useful tool to aid in this exploration.

Nimrod [Abr95] is a tool designed to execute parameterized simulations on distributed workstations, and has similar scheduling goals as grid systems. The Nimrod scheduling framework uses a job distribution manager (JDM) to submit host requirements to an arbitrary trading service. The trading service uses the host requirements, and possibly other outside information, to produce a schedule. The trading service can be any scheduling mechanism. The trading service allows scheduling extensibility for the Nimrod system.

Globus is an integrated toolkit of basic services designed for use in grid systems [Fos97][Fos99]. A Globus resource broker uses grid system and job information to produce a resource selection policy. A Globus co-allocator uses the selection policy to divide the job execution across one or more resource sites that satisfy the policy requirements. A resource site is a collection of machines controlled by some local resource management tools. Typically, the local resource management tools are queueing systems like LSF, NQE, or LoadLeveler, but they can also be tools for managing heterogeneous clusters. A Globus Resource Allocation Manager, or GRAM, runs on each site to interact with the local management tools. The GRAM receives the co-allocator job request and forwards it to the local management tools, which control the site scheduling policy. The scheduling policy of each site may be different. In this way, Globus uses a hierarchy of scheduling mechanisms to achieve global job scheduling.

The layered scheduling structure used in Globus resource management allows local management tools to schedule tasks in any arbitrary fashion. Although this allows

site-level scheduling customization, the lack of global scheduling may lead to poor scheduling decisions. For instance, a simple site scheduler may not take application characteristics into account during task assignment, which may result in poor performance for some applications. The GRAM and co-allocator currently cannot influence the local scheduling decision, although advance reservations may allow the co-allocator more control in future Globus implementations. Additionally, the Globus resource broker cannot modify the resource selection in response to dynamic changes in the grid system. However, grid system state can strongly affect job performance, and should be a factor in the scheduling decision. The ability to modify resource selection may also be added to future Globus implementations [Cza01].

Prophet is a scheduling framework designed for grid systems middleware such as the Legion grid computing system [Wei95]. The Prophet framework presents strategies for job partitioning, task allocation, and instantiation, and makes detailed models of applications and grid systems. Our scheduling framework is largely based on the Prophet framework. However, our framework does not include job partitioning and instantiation, and focuses on providing a simple scheduler that produces effective schedules for a wide variety of data-parallel jobs. Prophet uses system benchmarking and application profiling in its scheduling policy, and makes scheduling decisions based on the assumption that clusters in a grid infrastructure are homogeneous. Our scheduling policy does not require extensive system benchmarking or application profiling, and assumes that clusters in a grid infrastructure may be heterogeneous.

Gallop is a wide-area scheduling system designed to exploit opportunities for high-performance in Internet-based systems [Wei98]. Gallop uses a layered approach to

scheduling parallel applications. A local site with a scheduling request initiates a global scheduler. The global scheduler selects candidate sites for job assignment prioritized by nearness to the local site. No information about intra-site resources is considered in the global scheduling decision. Given the set of sites for job execution, each site uses a local scheduler to produce the local schedule with the lowest projected completion time. The local scheduler may be Prophet, or may be any arbitrary scheduling algorithm. The best local schedule from all selected sites is chosen as the job schedule, and the job is instantiated on that site.

Gallop differs from our scheduling framework because it assumes the jobs are determinstic, and because it uses a layered approach to wide-area scheduling. We assume that jobs may be non-deterministic in their behavior, and that the completion time may not be predictable. Interactive data-parallel jobs may exhibit non-deterministic behavior, and a completion time for such jobs is not predictable. Also, the global scheduling in our framework examines intra-site resource characteristics in the task assignment process.

2.3 Existing Scheduling Heuristics

A detailed discussion of sub-optimal scheduling heuristics is beyond the scope of this work, and further analysis can be found elsewhere in the literature [Lo88][Wei95]. In this section, we present a selection of common heuristics that have been used in distributed systems.

Random and round-robin task assignment can sometimes outperform even complex algorithms, as shown in scheduling work in operating systems literature [Sil98]. These simple algorithms are effective in homogeneous distributed systems, but are less

effective when applied to a heterogeneous computing environment, since certain machines may far outweigh others in their ability to process tasks, and a random or round-robin scheduling strategy would ignore these costs. We explore the effectiveness of random task assignment in chapter 5.

Lo presents a three-part algorithm for task assignment in distributed systems that aims to reduce overall execution and communication costs [Lo88]. This algorithm does not take processor load into account when assigning tasks, and attempts to minimize overall execution and communication costs, rather than overall job completion time. This algorithm has the undesirable effect of scheduling all tasks to the same processor when the system is homogeneous in order to minimize communication costs. This strategy ignores potential parallelism and greatly increases job completion time. A second algorithm is presented to introduce additional interference costs in execution and communication time when multiple tasks are scheduled on the same processor. This solution produces greater task concurrency, but assumes that deterministic information about the tasks and processors are available, including completion time and execution and communication interference costs. In order to determine the interference costs of a set of k tasks $T = \{t_1, t_2, \ldots, t_k\}$ on a set of n processors $P = \{p_1, p_2, \ldots, p_n\}$, one must measure the cost of each pair $\{t_i, t_j\}$ of tasks on every processor in P. In a system of k tasks and n processors, the algorithm for deriving interference costs consists of $\binom{k}{2}n$ interference measurements, which may incur excessive overhead when k or n is sufficiently large and the tasks are non-trivial. A solution for arbitrary interference costs when $n < 3$ is presented, and the general case is left for future work.

Weissman presents heuristics for scheduling parallel computations in heterogeneous environments [Wei95][Wei00]. Three common classes of parallel applications are identified, and reasonable heuristics are presented that assess variations in processor load and network bandwidth. The heuristic for task assignment is a greedy algorithm with complexity $O(nk^2)$, and assumes that k will be small in practice. The heuristic requires knowledge of the computational cost for each task running on each processor, as well as knowledge of the communication costs between two tasks on two processors, for all tasks and all processors. In practice, these exact values may not be known, or it may be intractable to produce when the number of tasks and processors grows too large or dynamic load is taken into account.

Massively parallel processing (MPP) systems bear some resemblance to grid systems running data-parallel applications. MPP scheduling algorithms often coordinate processing, communication, and data access in their strategy [Ber99]. This coordination should also be a part of grid scheduling. However, MPP schedulers often operate under the assumptions that the scheduler controls all resources, the resources are homogeneous, and there is minimal resource contention. Some or all of these statements may be false for grid systems. Thus, MPP schedulers typically do not make good grid schedulers.

Hamidzadeh et al. [Ham95] presents global, run-time, dynamic scheduling techniques for scheduling in heterogeneous computing systems. Their Self-Adjusting Scheduling for Heterogeneous systems (SASH) algorithm uses a variation of the branch-and-bound optimal algorithm, and iteratively creates schedules during job execution. The scheduling process is dynamic, and the SASH algorithm dedicates one processor for scheduling calculations during job execution. The scheduling calculations use processor

type and communication costs in predicting estimated job completion time. This algorithm has been shown to outperform other dynamic scheduling algorithms when the number of available processors is large. However, dynamic scheduling techniques that reserve processors may cause starvation if the number of jobs being run exceed the number of processors in the system. Furthermore, processor reservation may not even be possible in a system where resources are shared.

Biological science techniques have been incorporated into distributed system scheduling to create new scheduling heuristics and approximations. Genetic-based scheduling [Zom01] employs genetic algorithms (GAs) to evolve schedules through schedule reproduction, crossover, and mutation. The schedules "survive" one round of evolution if they meet the cost function criteria (e.g., job completion time). Successive rounds of evolution have been shown to converge upon near-optimal schedules in a short period of time when the number of tasks is small. The convergence time for GAs for large-scale scheduling may be considerably larger, although a sufficiently parallel GA scheduler has been suggested to reduce the cost. Exactly how this parallel scheduler would be scheduled has not been discussed. Furthermore, the GA solutions have operated under the assumption that application behavior is deterministic. A GA heuristic that assumes a non-deterministic environment may be useful for grid scheduling, and is the subject of future work.

Chapter 3 Scheduling Framework

In this section, we present a scalable, extensible framework for scheduling in a grid system. First, the solution space of the scheduling framework is discussed. Next, each component of the scheduling framework is presented. The scheduling framework is composed of (1) a grid system model, (2) an application model, (3) a performance model, and (4) a scheduling policy. Each of these components can be extended to support sophisticated scheduling techniques, or simplified to a base framework.

3.1 Grid Scheduling Solution Space

The grid scheduling problem is the problem of mapping a set of tasks to a set of machines, or nodes. We consider a set of k tasks $T = \{t_1, t_2, \ldots, t_k\}$ and a set of n nodes $N = \{h_1, h_2, \ldots, h_n\}$. The set T comprises the total job to be scheduled, and has been previously partitioned into k tasks by the user, the application writer, or an automated partitioning tool. The scheduling framework produces some mapping of tasks to nodes $ASSIGNMENT = \{(t_1, h_x), \ldots, (t_k, h_y)\}$. Each of the k elements in *ASSIGNMENT* map each unique task in T to any node in N, so multiple tasks can be assigned to the same node (see Figure 3).

We focus on scheduling data-parallel jobs with multiple interacting components. The framework is suitable for parallel jobs with non-trivial amounts of communication, as well as massively parallel jobs such as parameter space studies, which do not incur high communication overhead.

We assume that the behavior of the network and the hosts is dynamic and non-deterministic. We also assume that multiple users may share the grid system. Multiple

schedulers may be simultaneously producing assignments, and some nodes may support time-sharing. Thus, the scheduling framework must support co-scheduling and tolerate non-reservable resources that may have significant workloads.

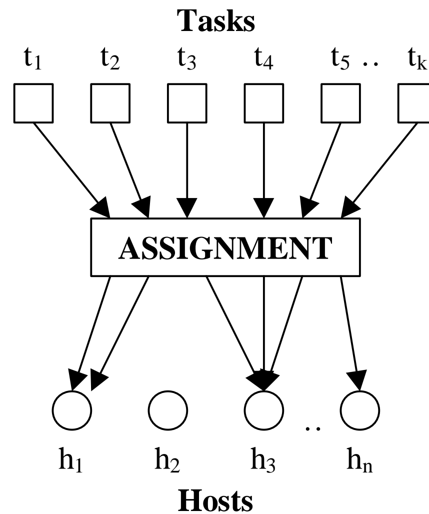


Figure 3: A set of tasks is mapped to hosts through an assignment.

3.2 Grid System Model

The grid system model provides a representation of grid system resources. The model includes information about resource characteristics and availability, as well as the topological information about the network configuration. The grid model provides information used by task assignment algorithms within the scheduling framework.

A *grid resource* is typically an individual host machine with one or more processors of the same type. For example, a grid resource might be a workstation, a vector multiprocessor, a mesh multicomputer, or a network attached storage server. Each resource has a number of characteristics that may vary over time. We can enumerate typical characteristics of a grid resource in different categories:

- Processors (type, number, speed, cache size)
- Processor load (current, peak, sustained)
- Memory (type, real, virtual, swap, peak and sustained bandwidth, latency, current available)
- Operating system (type, version, configuration)
- Storage devices (type, size, peak and sustained bandwidth, latency)
- Communication devices (type, peak and sustained bandwidth, latency)
- Network location (nearest cluster, nearest router, IP address, geographic location)

Clearly, the characteristics that define resource type cannot be fully enumerated, since there is a potentially infinite set of different characteristics. Thus, a grid system model must necessarily be extensible enough to support arbitrary resource characteristics.

3.2.1 Resource Information Collection

Information agents provide the system data used in the grid system model and the performance model. All resource information is, in one sense, dynamic. Processor load and available memory are clearly dynamic entities, but so is node processor type, operating system, or network topology. The configuration of the grid or resources within the grid may change as resource owners add, change, or remove components. These resource characteristics do not change with the frequency of highly dynamic characteristics like available memory, but should not be considered static. Each grid resource uses an information agent to take a snapshot of the current resource state. Snapshots are static lists of resource information used by the scheduling framework.

Snapshots can be periodically updated to reflect dynamic changes in resource characteristics.

The information gathered by an information agent may include the typical resource characteristics detailed in section 3.1. However, the information acquisition can be tuned to aid in scheduling specific jobs. The information agent must relay some minimum performance and network information to the task assignment algorithm as specified by the scheduling policy. The performance information should allow scheduling policy to produce an ordering of nodes according to performance. Usually, this is determined by a combination of system information. The network information should allow the scheduling policy to produce an estimate of network communication costs. One implementation of an information collection system is discussed in section 4.1.

3.3 Application Model

The application model provides a representation of application behavior. A grid scheduler can use the application model as a way to guide the scheduling process. The user or the application developer familiar with the program normally provides such information. In typical grid scheduling algorithms, large amounts of application-specific information are required to produce good schedules. Some of the typical information may include:

- The number of partitioned tasks k
- The architecture(s) for which the task binaries are valid
- The topology of task communication (e.g., linear, broadcast, tree)
- The ratio of communication to computation for each node

- Profiling statistics of the application run on different architectural configurations
- The expected number and size of network packets sent and received by each node

Application information is necessary only if a task assignment algorithm requires it. A simple algorithm may choose to require some or none of the listed application characteristics, and a specialized scheduler may require different characteristics of the application. The only exception to this rule is the partitioning information, which must be supplied if the scheduler is to assign a set of tasks to nodes. By default, very little application information is required.

3.4 Performance Model

The performance model (1) identifies valid scheduling candidates, (2) builds any necessary performance prediction models, and (3) defines a fitness function for performance prediction.

3.4.1 Identifying Scheduling Candidates

The performance model must identify which resources are valid candidates for task placement. To be a valid scheduling candidate, a machine must be able to run the task binary. The performance model requires a small amount of information from the grid system and application models to identify valid scheduling candidates.

The grid system model must provide a small amount of information about each computing resource by default:

- Architecture type
- Operating system

The architecture type and operating system type are used to determine valid scheduling candidates, and are necessary components of the grid system model.

The application model must also provide a small amount of information by default:

- The number of partitioned tasks k
- The architecture(s) for which the task binaries are valid

The number of tasks and valid task architectures are both necessary for producing a valid schedule.

3.4.2 Network Organization Model

Grid resources are fundamentally connected in a *network organization*. The network abstraction of a grid resource is called a *node*. In a traditional network organization, a node is connected to the majority of other nodes through a wide-area network, and may be connected to a smaller collection of nodes in a local-area network. If there are no other nodes in the local area, the "local-area network" consists of only the node itself.

Because network resources in a grid system are heterogeneous, the arrangement of nodes is more complex. The network organization in a grid system usually forms a *supercluster* (or cluster of clusters). Each *cluster* is effectively a local-area network of nodes, connected to other clusters by routers in an increasingly wider area. Nodes in a

cluster are identified by their shared local communication link, and not by their architecture. Thus, heterogeneous, or *federated,* clusters can compose a grid supercluster.

The performance model may use grid system model information to assemble a network organization. The model then produces an estimate of the expected bandwidth and latencies between any two nodes in the system to increase the overall accuracy of the performance estimation. Accurate prediction of network performance is complicated by many factors. The composition of each cluster in the grid system may be quite different. For instance, two LANs may exhibit widely disparate bandwidths and latencies due to differences in the number of routers and gateways, fiber and ethernet bandwidths, or network traffic. Even within a level of the network organization, various sub-levels may perform differently. Katramatos et al. [Kat01] have shown that even homogeneous clusters can exhibit significant variations in latency. The grid system model should be able to represent the differences in network performance at every level. Unexpected changes in network traffic may skew estimates of network performance dramatically, but a grid system shared between many users may experience such unexpected changes frequently.

A number of research groups are investigating services for network performance prediction in superclusters [Wol98][Glo99][Kat01][Sup99]. Many of these services collect static information about network configuration, information about observed performance, and dynamic information about current network traffic. The services may prove useful for predicting network performance in a complex network environment such as the grid. The extensibility of the performance model includes support for any number

of tools for network performance prediction. One simplified tool for network performance prediction is described in section 4.2.2.

3.4.3 Fitness Equation

The task assignment framework produces an estimated performance for a task on each node in the system model by using a cost or *fitness equation*. The fitness equation is the method by which the scheduling policy ranks nodes and chooses an appropriate schedule. The scheduling policy calls a function $Performance(t_i, h_x)$ to obtain a relative estimate of the performance, or fitness, of a task t_i on a particular node h_x. *Performance* has access to information provided in the grid system model, the application model, and the network organization model, and may use the information to produce performance estimates. *Performance* can be composed of a number of functions that aid in the performance estimation. In its most general form, *Performance* uses a function *Max_performance* to determine the maximum performance of a task on the given host (in cycles/second), and uses a function *Latency* to estimate the cost of network communication latency (in seconds):

$$Performance(t_i, h_x) = Max_performance(t_i, h_x) / (1 + Latency(t_i, h_x))$$

For example:

Let $Max_performance(t_i, h_x)$ = 1Ghz, or 10^9 cycles/sec.

Let $Latency(t_i, h_x)$ = 1 sec.

Then, $Performance(t_i, h_x) = 10^9 / (1 + 1) = 5 * 10^8$ cycles/sec., or 500 Mhz.

This simple cost equation forms the basis for evaluating task assignment, but does not dictate how each part of the equation should be calculated. The division of performance into separate maximum performance and latency predictions allows for a natural division of architecture-based and network-based prediction functions. Most existing tools for benchmarking and profiling are usually divided along these lines. For instance, benchmarking suites like SPECint95 [Spe95] are designed for architectural evaluation, whereas tools like NWS [Wol98] are designed for network performance prediction. Both performance metrics are influential in the overall performance of the node, and they are both included in the fitness function.

An arbitrary combination of prediction functions can be used to calculate the expected maximum performance and latency. This extensibility allows the cost function to be refined to calculate accurate performance metrics as needed.

For example, see figure 5. *Max_performance* might be a weighted combination of performance estimations. *Architecture* produces an estimated performance measurement based on architecture benchmarks. The results of benchmarks like Linpack and SPECint95 may be weighted according to their effectiveness in performance prediction. In addition, application profiling may be used to produce an estimated performance measurement based on previous runs of the program on the given machine. The combination of application profiling and system benchmarking may produce a more accurate prediction for Max_performance.

Likewise, the Latency function may be extended to make accurate communication latency estimations. *Communication_cost* produces an estimated latency based on MPI

benchmarking. This estimate is combined with the Network Weather Service estimate to produce an accurate estimate of predicted communication latency.

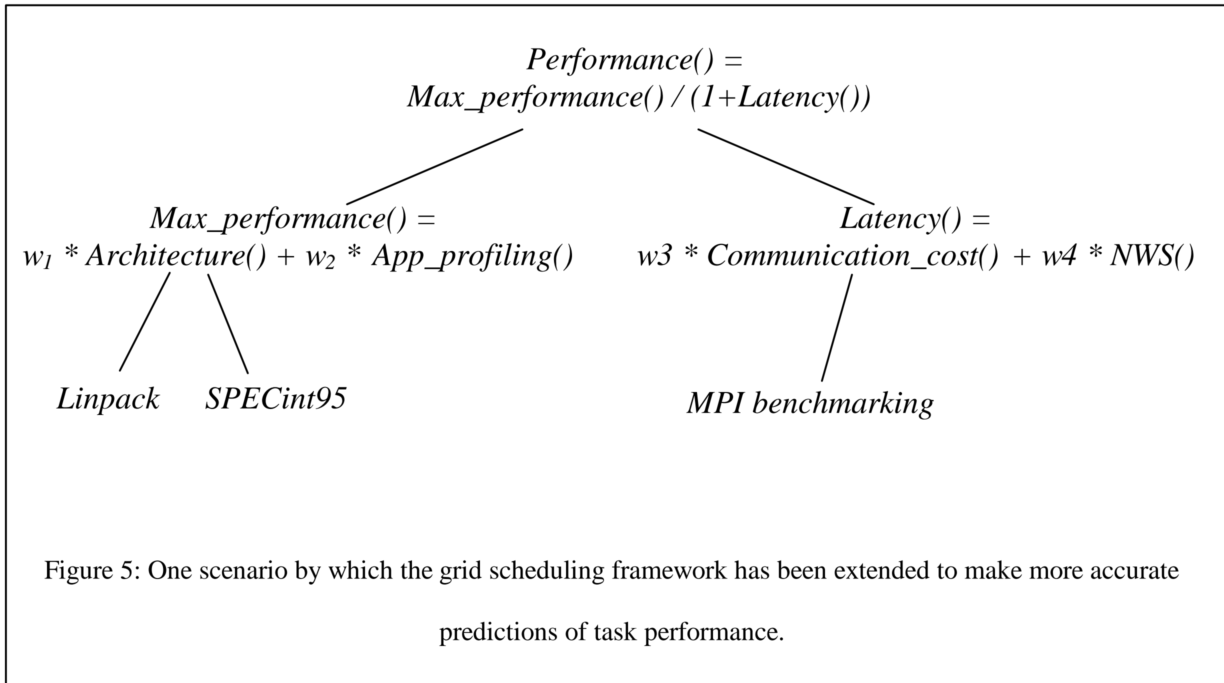


Figure 5: One scenario by which the grid scheduling framework has been extended to make more accurate predictions of task performance.

3.6 Scheduling Policy

The scheduling policy uses the performance model estimates to rank machines and choose a schedule. In our scheduling framework, a simple scheduling heuristic is used to produce an assignment of tasks to machines:

Let $Performance(t_i, h_x)$ produce the current cost of assignment for a given task t_i, and initialize $ASSIGNMENT = \emptyset$

(1) For each task t_i

(2) Let $Fitness(t_i) = -1$

(3) For each node h_x, *0* **£***x* **£***n*

(4) If $Performance(t_i, h_x) > Fitness(t_i)$, then

(5) $Fitness(t_i) = Performance(t_i, h_x)$

(6) $ASSIGNMENT = ASSIGNMENT \cup \{(t_i, h_x)\}$

(7) Increment h_x processor load by one.

This is a greedy algorithm with complexity O(nk), similar to previous work in heterogeneous scheduling algorithms [Lo88][Wei00]. The algorithm iteratively assigns each task to the machine that produces the best performance. After the best machine for that round is selected, the machine load on that machine is increased, and the pairing is added to the *ASSIGNMENT* set. Machine load is initially a weighted average of recent load as reported by the operating system, and may effect *Max_performance* estimates. We assume data-parallel jobs to be compute-intensive, and increase the new load of a chosen node by one to reflect this in future task assignments of the job. However, this possible overestimation does not affect future job schedules, since the load value is not persistent over multiple scheduling runs. The possible overestimation only affects future task placement decisions for the current job. Once a task has been assigned to a node, the task assignment cannot be changed.

The algorithm is different from the Lo and Weismann algorithms because it produces a relative performance estimate, rather than an absolute predicted completion time. This can be useful for scheduling jobs with non-deterministic behavior, such as interactive applications.

This simple heuristic is designed for scalability. The run-time overhead of the algorithm is small even when *n* or *k* grows. This type of scalability is necessary for grid

systems, where the number of nodes may range in the thousands, and the task partitioning may be equally large. The heuristic is also extensible because the fitness function can be extended to provide the desired accuracy of performance prediction. However, extensions to the fitness function may add to the algorithmic complexity of the heuristic, since it may increase the running time of *Performance*. Thus, the algorithm provides a ranking method that is not algorithmically complex by default, and it can be extended to provide more accurate placement decisions as desired.

One weakness of the algorithm is found in its iterative method of task placement. The fitness function typically estimates latency by calculating the aggregate communication costs between a given node h_x and the nodes listed in *ASSIGNMENT*. This latency estimate will not effect the performance estimates when *ASSIGNMENT* is initially empty, because the aggregate communication costs will be zero. The algorithm may make a poor initial placement decision because it will not initially identify nodes with poor communication facilities as poor candidates.

Modifying the behavior of the fitness function can eliminate this weakness. For example, the number of partitioned tasks in a job can be used to estimate latency costs even when *ASSIGNMENT* is initially empty. Machine latency can be non-zero, even when no tasks are currently scheduled, in order to avoid a poor local selection. One solution to this weakness is found in section 4.4.3.

Chapter 4 Scheduler Implementation

This chapter describes a grid scheduler implemented for the Legion grid computing system. Legion is an object-based grid system that allows a grid infrastructure to be used as a single virtual machine [Gri97]. Legion uses the grid infrastructure to provide much of the functionality of an operating system, such as a unified persistent name space and a file system. Legion also provides a high performance computing environment for a variety of applications.

The goal of implementation was to produce a scheduler that requires only a small amount of information about the application and the state of the network. In order to avoid run-time overhead, the scheduling cost function does not require any detailed benchmarking statistics about the application or the grid system.

In this chapter, we review the resource information collection process in Legion. Then, the grid system model, application model, and performance model are defined.

4.1 Resource Information Collection

In Legion, a *collection* acts as a repository for grid system information [Cha98]. The collection stores information records as a set of Legion object attributes, and supports both push and pull models of data collection. In Legion, everything is represented as an object. Legion object attributes can be arbitrary information about any object, including any grid resource. This flexibility allows new scheduling policies to choose what grid system information is needed to create schedules.

4.2 Grid System Model

The grid system model obtains resource information by querying the collection. The characteristics required by the grid system model do not require extensive system benchmarking. They are:

- Architecture type
- Number of processors
- Processor speed
- Processor load
- Operating system
- Network location

The architecture type and operating system are necessary components of the grid system model, as discussed in section 3.4.1. The number of processors, processor speed, and processor load allows the performance model the basic information to determine the peak and current processing power of a node. Finally, the network location allows the performance model to have a networked representation of the grid nodes, which allows estimation of the cost of communication between nodes. The network representation is detailed in section 4.4.

4.3 Application Model

The application model is built from user-provided application characteristics that do not require extensive application profiling. They are:

- The number of partitioned tasks k

- The architecture(s) for which the task binaries are valid
- The ratio of communication to computation for each node

The number of tasks and valid task architectures are both necessary for producing a valid schedule, as discussed in section 3.4.1. The ratio of communication to computation gives the performance model a method of weighing the relative importance of communication rates and computational power for the application without requiring extensive application profiling.

4.4 Performance Model

The performance model bases the fitness equation on cost estimates provided by simple *Max_performance* and *Latency* functions. The *Latency* function uses a simple network organization model to predict the estimated latency between two grid system nodes.

4.4.1 Performance Fitness Function

Max_performance requires relatively little information about nodes and the application:

$$Max_performance(t_i, h_x) = speed(h_x) * \frac{processors(h_x)}{load(h_x)+1} * ASF(t_i, h_x)$$

such that:

- $processors(h_x)$ produces the number of processors on node h_x,
- $speed(h_x)$ produces the processor clock speed in cycles/second,
- $load(h_x)$ is a weighted average of the currently reported 5-, 10-, and 15-minute load on the host, and

- *ASF*(t_i, h_x) returns the architectural scale factor, or estimated relative performance of the task on the host.

For instance, a machine with two 750 MHz processors, a current weighted load of one, and a relative architecture scale factor of 1.2 can expect to contribute approximately $750 * \frac{2}{1+1} * 1.2 = 900$ weighted clock cycles per second.

The estimates for *processors*(h_x), *speed*(h_x), and *load*(h_x) are provided by the collection. The *architecture_scale_factor*(t_i, h_x) is a user-provided estimate of the relative performance of their application on different architectures. This allows applications with strong affinities towards particular architectures to receive the proper task assignment.

4.4.2 Network Organization

Traditional network classifications, such as WAN and LAN classifications, do not sufficiently reflect the complexities and differences in the levels of network organization in a grid. One simple method of organization that allows for these differences is the distance-based hierarchical model. Each cluster represents a level in the hierarchy, and the routing distance between any two nodes can easily be derived from the hierarchy. This organization is extensible because it can accommodate any number of levels of hierarchy, rather than a limited number of classifications such as WAN, MAN, and LAN.

A hierarchical model simplifies certain network characteristics. For instance, it assumes that the communication costs between two clusters will be the same for packets travelling in either direction. Also, the abstraction does not include specific information about each cluster's network performance. However, this simple model allows rough

communication cost estimates to be made without gathering network statistics. This simplification reduces the overall cost of scheduling.

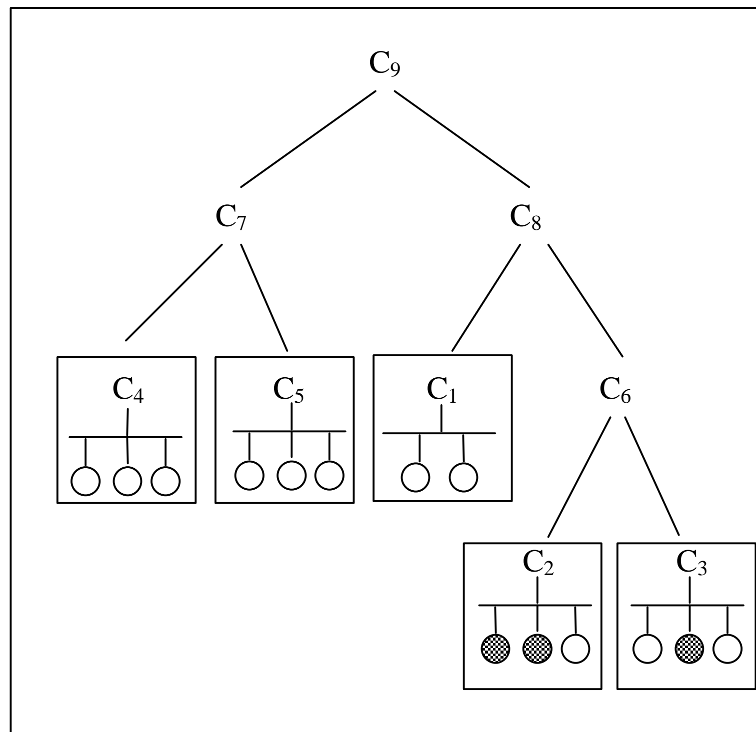


Figure 4: A hierarchical representation of a grid topology. The leaves in the tree (boxed) are clusters containing nodes, and the non-leaf nodes represent clusters of clusters.

4.4.3 Latency Cost Function

The *Latency* cost function uses the simple network organization model to produce latency cost estimates without extensive network or application information. It does not require measurements of peak and sustained network bandwidth and latency from the collection, and does not use current information about network traffic. Furthermore, it does not require the application writer to define the communication topology of the

application. It merely requires the user to submit an approximate ratio of communication to computation (from 0 to 1) that is expected in the application run, *ratio*, and uses that ratio to weigh the estimated cost of network communication.

Typically, network latency increases with the "wideness" of the network, or decreases as the span becomes more localized. A wide-area parallel processing study [Wei95] demonstrated that there is approximately an order of magnitude degradation in communication capacity between different layers of wide-area connectivity. The study measured bandwidth and latency over department-wide, campus-wide, metropolitan-wide, and nation-wide area networks. The *Topology* function exploits this observation:

$Topology(h_x, h_y, \boldsymbol{t}_n) = 10^n,\ n \geq 0$, where

- n is number of network levels between h_x and h_y, and
- $\boldsymbol{t}_n$ is the hierarchical network topology.

Topology calculates the number of levels of network hierarchy n between the two nodes h_x and h_y given the network topology $\boldsymbol{t}_n$, and returns an order-of-magnitude value, 10^n, to estimate the communication costs between the two nodes.

The result of *Topology* is used to calculate the approximate network latency:

$Latency(t_i, h_x)$

$cost = 0$

for each h_y in *ASSIGNMENT*

$cost \mathrel{+}= Topology(h_x, h_y, \boldsymbol{t}_n) * ratio$

return cost

This algorithm calculates the communication cost for each node in *ASSIGNMENT* using *Topology*, and returns the estimated cost. Clearly, as more tasks are assigned to nodes, the cost of scheduling a new task many levels removed from other nodes increases. By default, when no nodes are in *ASSIGNMENT*, the calculated latency is zero.

To compensate for this problem, *Topology* uses the number of partitioned tasks and the network organization model to produce a special latency cost for a given node h_x even when *ASSIGNMENT* is empty. In this case, *Topology* calculates the latency by creating a temporary assignment of tasks to nodes "closest" to the given node h_x. Clearly, the closest node to h_x is h_x itself, so *Topology* makes the temporary additional requirement that only one task is scheduled per processor. This requirement is not related to the final schedule, or to future calls to *Topology*. The method merely provides a ranking of nodes by closeness to other nodes, scaled by the total number of tasks in the job. This estimate helps avoid poor scheduling decisions during the initial placement.

Chapter 5 Evaluation

In this chapter, we present the evaluation of the simple scheduler implemented in our framework. First, we describe the evaluation test suite. The parameters of the evaluation are then presented. Finally, the testing results are presented and discussed.

5.1 Evaluation Test Suite

In order to evaluate the implemented scheduler under a variety of conditions, we created a small test suite to simulate the behavior of a variety of data-parallel applications. The tests were written using the Legion MPI programming interface for synchronization. The suite supports six different task communication patterns typical of such applications. These patterns are linear, two-way ring, mesh, tree, star, and full-way communication.

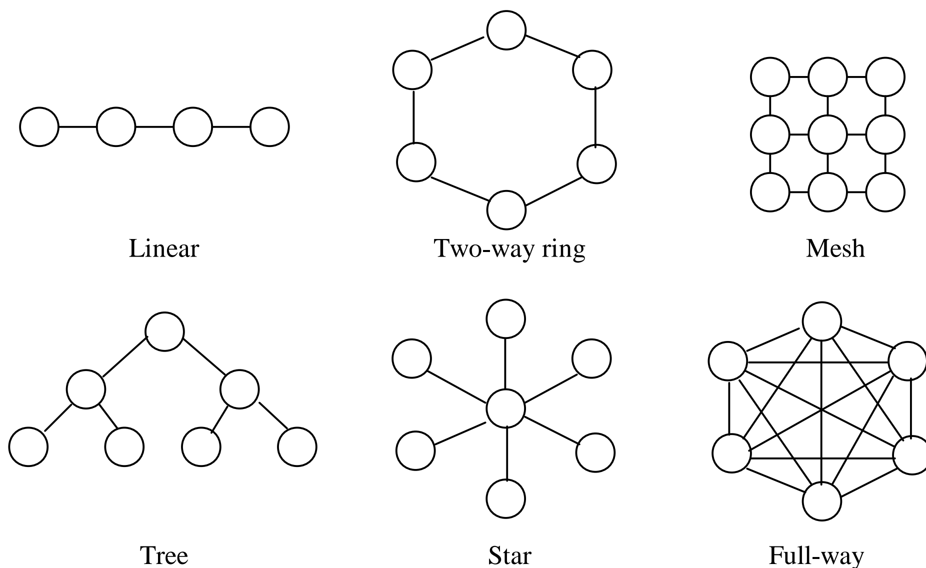


Figure 6: Different communication patterns typical of DP jobs.

Each run of the program partitions *k* tasks to run for a preset amount of time. Each task divides its time between communicating and computing according to the given communication to computation ratio and the communication pattern. For instance, one run of the program might partition 20 tasks to run for 10 seconds with a communication to computation ratio of 1:4 in linear communication mode. Each task will spend the first two seconds passing messages in a linear fashion, and then spend the remaining eight seconds doing matrix calculations. The number of communication and computation cycles completed after each run is tabulated.

The test suite is used to evaluate the relative performance of tests scheduled using different scheduling algorithms. In the evaluation procedure, each algorithm is given the same snapshot of the system state from the collection, as well as a small amount of information about the current test job. This information is the number of tasks (ranging from five to 40), the ratio of communication to computation, and the architecture scale factor. From this information, each algorithm produces a schedule. The job is run on a grid testbed using each generated schedule, and the relative amount of communication and computation produced by each run forms a basis for comparison between the algorithms.

5.2 Testing Environments

The evaluation procedure was run in two testing environments. *Npacinet-1* is a subset of the National Partnership for Advanced Computing Infrastructure (NPACI) [Npa01] Legion testbed. Npacinet-1 consists of approximately 95 nodes connected through four layers of network hierarchy. All nodes in Npacinet-1 run Linux on an Intel

Pentium-based architecture. Apart from the base architecture, the node characteristics (e.g., processor speed, physical memory, etc.) vary. One cluster, based at the University of Minnesota, consists of four dual-processor 533-MHz Pentium machines running Linux 2.2.12-20smp with 256 MB of physical memory, connected by Giganet. The second cluster is a subset of the Centurion cluster at the University of Virginia. It consists of 128 dual-processor 400-MHz Pentium II machines running Linux 2.2.14-1.3.0 with 512 KB cache and and 256 MB physical memory. Each node is connected to a 100Mbps fast Ethernet switch, and these switches are joined via gigabit Ethernet switches. Thus, the network configuration of Centurion is a cluster of clusters. The disparity in architecture and network configuration are good examples of grid system heterogeneity, and makes Npacinet-1 a useful testbed for scheduling algorithm evaluation.

Npacinet-2 is a larger subset of the NPACI Legion testbed. Npacinet-2 contains Npacinet-1, as well as 64 533-Mhz DEC Alpha machines running Linux 2.2.14-1.3.0 with 256 MB memory, and 6 100-Mhz SGI O_2 machines running IRIX 6.5 with 128 MB memory. The DEC Alpha machines are clustered with Myrinet, and the SGI O_2 machines are connected with Ethernet. The machines are interactively shared with other users, and exhibit variance in resource usage. Npacinet-2 was used because it is representative of a typical grid infrastructure. It exhibits heterogeneity in architecture, dynamic resource usage, and network configuration.

5.3 Scheduling Policy Evaluation

The evaluation tested the effectiveness of three scheduling policies: (1) RAND, (2) MP, and (3) MPL. RAND is a scheduler that uses a random host selection algorithm.

This algorithm is sometimes often used in operating systems or homogeneous distributed systems, but may not extend well to a grid system. It serves as a baseline for comparing the other algorithms. The MP algorithm uses a fitness function that only uses Max_performance to make performance estimates. Finally, MPL uses both Max_performance and Latency to make performance estimates.

Although RAND provides a baseline for evaluation, we cannot compare the produced schedules to an optimal schedule in this test environment. The complexity and dynamic nature of the grid system makes finding the optimal schedule impossible. For instance, the differences in processor load and available memory across the machines in the testbeds change unpredictably. Finding the optimal schedule requires a way to predict these changes in order to choose the optimal node for each task. The prediction of these characteristics in even a simple grid system is difficult, if not impossible, to achieve. Therefore, we compared our scheduling policies with a baseline policy, rather than with an optimal one.

MP is useful for comparison with MPL because it should expose the strengths and weaknesses of using the latency estimation method described in section 4.4.3. However, a latency-only scheduler is not necessary for testing, since it will always schedule all tasks on the same processor to minimize the latency costs.

5.3.1 Scheduling Policy Evaluation Results

In the scheduling policy evaluation, we compare (1) MP vs. RAND, (2) MPL vs. MP, and (3) MPL vs. RAND. In each comparison, the two scheduling policies are given the same job and system information, and generate schedules according to their

respective policy. Each comparison scheduled jobs ranging from five to 40 tasks, with six ranges of communication to computation ratios, over two testbeds. Each comparison ran approximately 190 tests per testbed, for a total of approximately 380 tests.

Table 5.1 illustrates the relative performance of the test suite run with schedules generated by MP and RAND scheduling policies. In both testbeds, MP-scheduled jobs consistently produced 20-60% more computation than similar jobs scheduled with RAND. The *Max_performance* estimation allowed the MP scheduler to predict the processor availability more effectively than a random scheduling policy. Neither RAND nor MP evaluated latency costs as part of their performance model, and the resultant communication performance was unpredictable because of this fact. On average over both testbeds, MP and RAND produced similarly poor schedules for communicating tasks. The variance in communication performance is due to the lack of communication criteria, which results in random communication performance. In these testbeds, RAND may have had a slight advantage over MP. MP specifically scheduled tasks on nodes with the highest *Max_performance*, but RAND chose from a random sampling of the available nodes. Because the majority of nodes in Npacinet-1 (approximately 90 out of 95) were part of the same Centurion cluster, RAND had a majority of nodes fall in the same cluster. This sampling resulted in slightly better communication performance.

Table 5.1: MP versus RAND

	Npacinet-1		Npacinet-2	
384 Trials total	Communication (%)	Computation (%)	Communication (%)	Computation (%)
Linear	75.38836	120.2711	63.66857	156.9209
Two-way ring	136.2604	120.8976	99.86789	157.1913
Mesh	85.69966	126.9933	125.9631	159.0397
Tree	148.1161	124.8453	165.0605	159.0735

Star	78.04896	120.4064	74.89542	153.7328
Full	139.3748	118.4857	246.6633	156.0611
Mean:	107.7419	121.7487	87.81705	156.7893

Table 5.2 illustrates the relative performance of the test suite run with schedules generated by MPL and MP scheduling policies. In Npacinet-1, MPL-scheduled jobs performed 5.4% more computation that MP-scheduled jobs, and MPL-scheduled jobs in Npacinet-2 performed at 90.2% of the computational capacity of MP jobs. The slight decrease in overall computational performance was compensated by the large differences in communication performance, which ranged from 97-535% over MP. Clearly, the *Latency* calculations made a significant impact on job communication in most cases, without negatively influencing the produced computation.

Table 5.2: MPL versus MP

	Npacinet-1		Npacinet-2	
384 Trials total	Communication (%)	Computation (%)	Communication (%)	Computation (%)
Linear	215.6294486	101.4318	457.8429	90.55876
Two-way ring	175.1978118	109.9872	348.457	89.84642
Mesh	96.98183761	94.91566	534.8136	90.03901
Tree	398.836372	98.87442	266.4326	91.80503
Star	242.4777107	103.6321	402.4	89.88453
Full	153.8510937	117.997	135.3367	89.70192
Mean:	213.6160746	105.4309	378.1468	90.22894

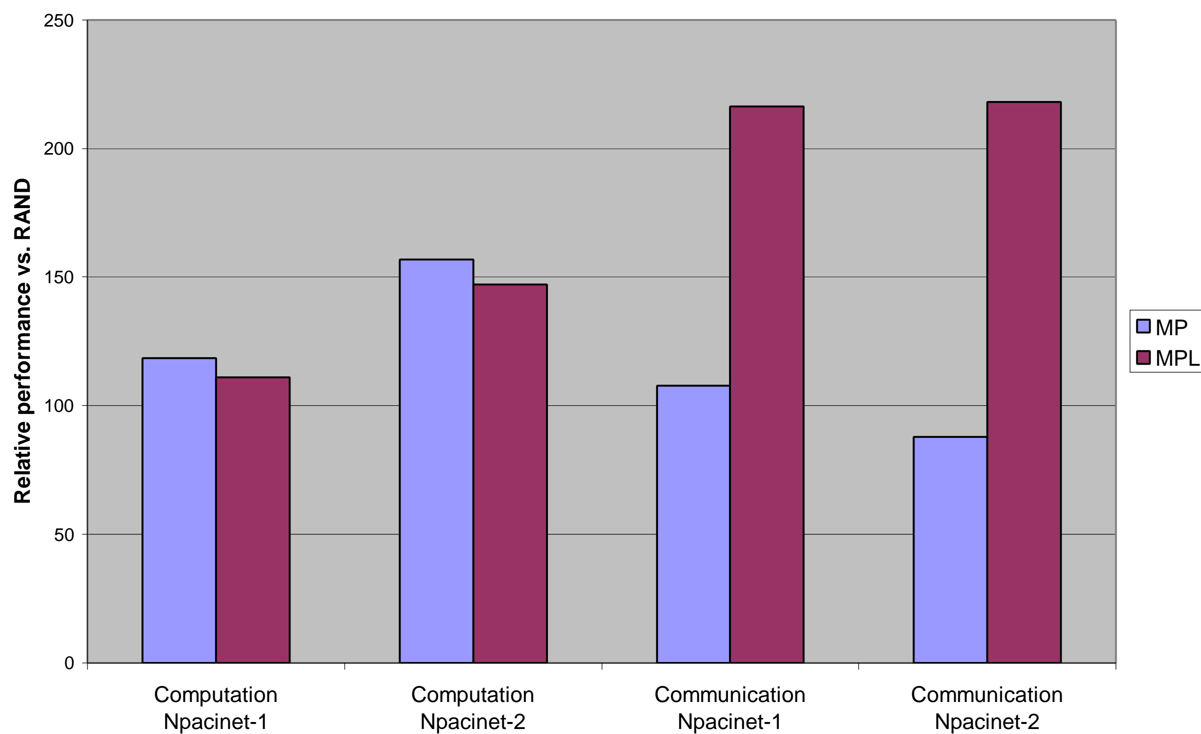
Table 5.3 illustrates the relative performance of the test suite run with schedules generated by MPL and RAND scheduling policies. MPL-scheduled jobs consistently outperformed RAND-scheduled jobs in communication performance (ranging from 133 – 333%), with moderate gains in computation performance as well (ranging from 96-169%).

Table 5.3: MPL versus RAND

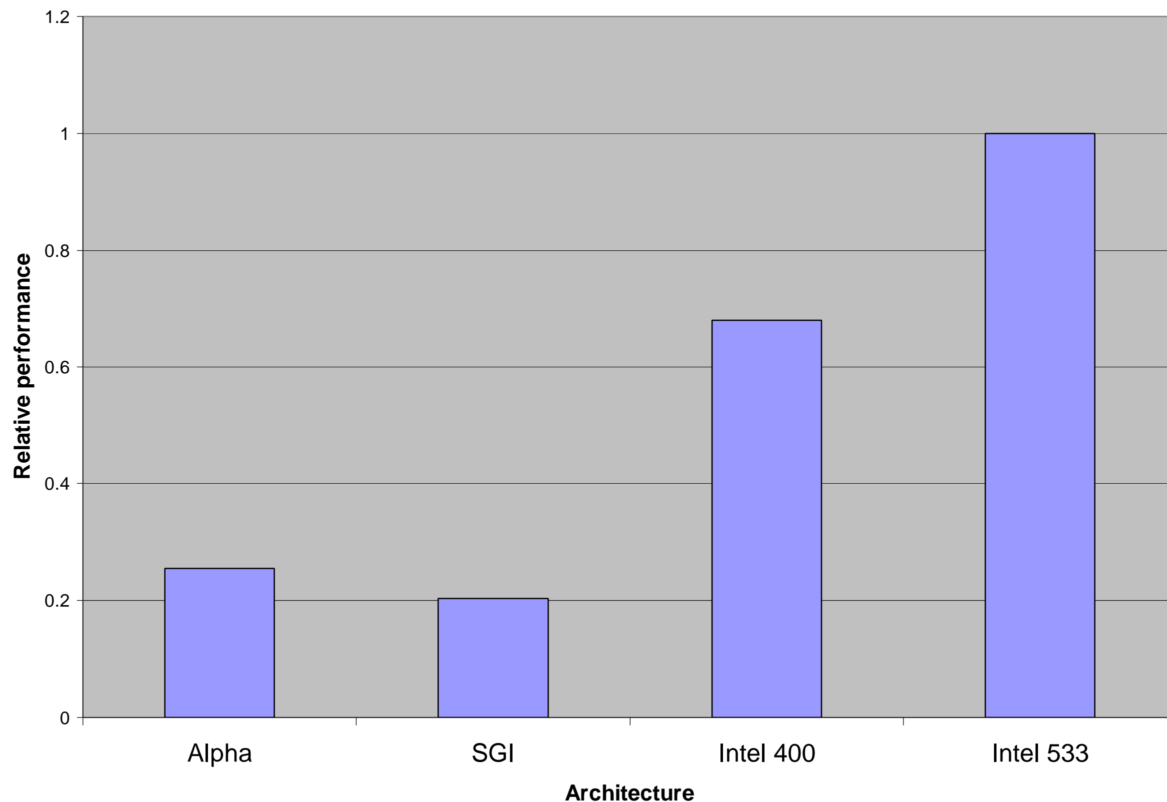
	Npacinet-1		Npacinet-2	
384 Trials total	Communication (%)	Computation (%)	Communication (%)	Computation (%)
Linear	174.5401	96.00306	253.9795	140.506
Two-way ring	162.6376	114.0959	213.358	142.9694
Mesh	332.8688	120.1422	263.6982	153.0446
Tree	290.8362	118.3486	207.0874	169.8817
Star	182.589	113.1457	200.7836	138.7187
Full	132.902	108.5371	143.8444	147.0409
Mean:	216.3261	111.0118	218.0809	147.0974

Figure 7 summarizes MP- and MPL-scheduled job performance against RAND-scheduled jobs. Both MP and MPL beat RAND computation performance on both nets.

Figure 7: Schedule Performance Summary Statistics



The higher computational performance yield on Npacinet-2 may be due to the wider differences in architectural performance. Figure 8 illustrates the relative computational



performance of the test suite on the three architectures in Npacinet-2. The SGI O_2 machines performed approximately 20% of the computation produced by an Intel 533 MHz, and the DEC Alpha machines performed approximately 25% as much computation as the Intel on average. In Npacinet-1, the 400-Mhz Intel produced approximately 70% of the computation of the 533-Mhz machine. Thus, MP and MPL had more opportunity to outperform RAND in computational performance in Npacinet-2, because of the architectural variety.

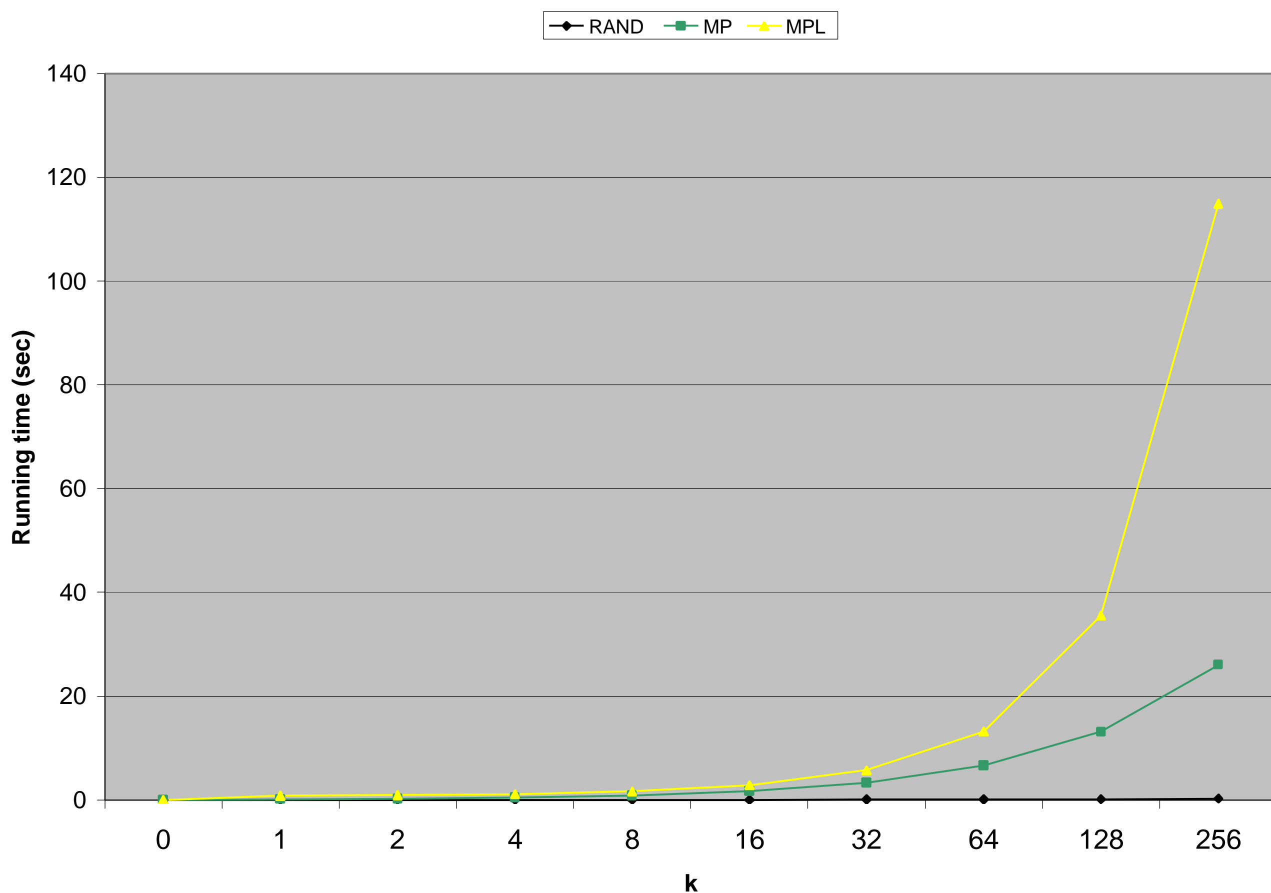
Figure 7 also demonstrates the added value of the *Latency* calculation. MPL-scheduled jobs performed an average of 200% more communication on both testbeds, whereas MP-scheduled jobs performed similarly to RAND jobs. Clearly, the *Latency*

estimations guided the scheduling process towards better communication performance without adversely affecting computational performance.

5.3.2 Scheduling Policy Run Time Evaluation

The complexity of each scheduling policy determines its run time. RAND is complexity O(k) because it randomly chooses an assignment for each task. MP is complexity O(nk), because it uses the scheduling algorithm from section 3.6 without adding any significant computational complexity in its calculation of *Max_performance*. MPL adds a latency calculation that iterates over the assigned tasks, and has algorithmic complexity $O(nk^2)$. Figure 9 illustrates a comparison of the algorithm run times as *k* increases.

Figure 9: Scheduling algorithm running time



The running times were gathered on a Pentium-II 400 Mhz machine with 128 MB memory. The rapid increase in running time for MPL is due to its exponential complexity, and makes a large difference in running time when the number of tasks increases. The MPL policy may not be desirable when the number of tasks is very large, the expected job running time is very short, or the amount of communication is small.

5.4 Evaluation Summary

MPL was proven to produce effective schedules for a wide-variety of data-parallel applications. MP and MPL outperformed RAND in producing schedules that maximized job computation, and MPL outperformed MP and RAND in producing schedules that maximized job communication. MPL schedules did not significantly sacrifice computational performance to achieve communication gains in the two testbeds used for evaluation.

MPL, MP, and RAND running times reflected their respective algorithmic complexities. RAND may be a useful scheduling technique for very short running jobs, or for jobs without high communication or computation rates. MP running time also makes it a good candidate for scheduling short running jobs, or jobs without high communication rates. The MPL running time grew quickly with k, making it costly for scheduling short-lived jobs. However, a user may find MPL to be valuable when job run time is long. Clearly, no one scheduling policy is appropriate for all grid systems and applications.

Chapter 6 Future Work

In this chapter, we present topics worthy of future investigation. Firstly, we discuss further methods of evaluating the scheduling framework. We then discuss possible extensions to the scheduling framework.

6.1 Further Evaluation

Our scheduling framework allows the user to provide rough estimates of program communication and computation rates. These estimates may often be incorrect, and the impact of the error on job performance is unknown. Further testing should measure the changes in job performance caused by mistakes in characterizing job behavior. By measuring the degradation in performance, we can determine guidelines for estimating job behavior.

Data-parallel jobs in grid systems can be partitioned into hundreds or thousands of tasks. As grid computing matures, the number of tasks in a data-parallel job may grow even more. Further testing should increase k values to determine job performance for larger schedules created in the framework. Better scheduling strategies should be developed for scheduling in O(nk) time.

The scheduling framework should be tested to determine how well it extends to different program classes. Multiple-instruction multiple-data (MIMD) jobs may pose different scheduling requirements. Also, a variety of practical grid jobs should be assembled into a test suite for scheduling. The programs should be real applications run by the grid community with different job characteristics. The test suite could serve as a

means of benchmarking scheduling policies, much like the SPECint95 suite is used to benchmark architectural performance.

Specialized tools are currently being developed to evaluate grid scheduling algorithms. Bricks [Tak01] can be used to compare the effectiveness of our framework, as well as extensions to the framework. This evaluation may be valuable to users that may choose from a variety of available extensions to the framework.

6.2 Framework Extensions

The scheduling framework is designed to be extensible. More powerful performance predictors should be added to the framework, and their contribution to the performance model should be evaluated. NWS [Wol98] should be tested as a predictor of future network performance. The Cost-Benefit Estimation Service (CBES) [Kat00] is a promising developing method of predicting grid performance, including network costs. Vampir [Vam01] may be useful for evaluating performance for MPI jobs, and could be used in the framework. Genetic algorithms are being employed in grid schedulers [App01], and may be a useful extension to the framework.

Advance reservations and co-scheduling may be a promising method of meeting quality of service requirements for certain grid applications [Cza99]. Co-scheduling is the simultaneous execution of a group of tasks. Grid resources like queueing systems require advance reservations to enable co-scheduling. An advance reservation system may prove useful for extending the grid scheduling framework.

References

[App01] AppLeS: Application-Level Scheduler. http://apples.ucsd.edu. April 2001.

[Ber97] Berman, Francine. "High-Performance Schedulers". In *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, pp. 279-309, 1999.

[Cas88] Casavant, Thomas, and Jon Kuhl. "Taxonomy of Scheduling in General-Purpose Distributed Computing Systems". *IEEE Transactions on Software Engineering*, Vol. 14, Number 2, pp. 141-154, February 1988.

[Cha98] Chapin, Steve J., Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. "Resource Management in Legion". *Future Generation Computer Systems*, Vol. 15, pp. 583-594, 1999.

[Cza99] Czajkowski, Karl, Ian Foster, and Carl Kesselman. "Resource Co-Allocation in Computational Grids". *Proceedings of the 8th IEEE Transactions on High-Performance Distributed Computing*, pp. 219-228, 1999.

[Cza01] Czajkowski, Karl. Personal communication, April 2001.

[Fos97] Foster, Ian, and Carl Kessleman. "Globus: A metacomputing infrastructure toolkit". *International Journal of Supercomputer Applications*, Vol. 11, Number 2, pp. 115-128, January 1997.

[Fos99] Foster, Ian, and Carl Kessleman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[Fre89] Freund, F, and H.J. Siegel. "Heterogeneous Processing". *IEEE Computer*, June 1993.

[Glo99] GloPerf webpage. http://www-fp.globus.org/details/gloperf.html. April 1999.

[Gon77] Gonzalez, M. J. "Deterministic Processor Scheduling". *ACM Computing Surveys*, Vol. 9, Number 3, pp. 173-204, September 1997.

[Gri97] Grimshaw, Andrew S., William A. Wulf, and the Legion team. "The Legion Vision of a Worldwide Virtual Computer". *Communications of the ACM*, Vol. 40, Number 1, pp. 39-45, January 1997.

[Ham95] Hamidzadeh, Babak, David J. Lilja, and Yacine Atif. "Dynamic Scheduling Techniques for Heterogeneous Computing Systems". *Concurrency: Practice and Experience*, October 1995.

[Kat00] Katramatos, Dimitrios, Deepak Saxena, Nehal Mehta, and Steve J. Chapin. "A Cost/Benefit Model for Dynamic Resource Sharing". *Heterogeneous Computing Workshop 2000*.

[Kat01] Katramatos, Dimitrios, Marty Humphrey, Cheol-Min Hwang, and Steve Chapin. "Developing a Cost/Benefit Estimating Service for Dynamic Resource Sharing in Heterogeneous Clusters: Experience with SNL Clusters". To appear in *CCGRID 2001*, report 8519.0616.

[Lo88] Lo, Virginia. "Heuristic Algorithms for Task Assignment in Distributed Systems". *IEEE Transactions on Computers*, Vol. 27, Number 11, pp. 1384-1397, November 1988.

[Nak98] Nakada, Hidemoto, Mitsuhisa Sato, and Satoshi Sekiguchi. "Design and Implementations of Ninf: towards a Global Computing Infrastructure". *Future Generation Computer Systems, Metacomputing Issue,* 1999.

[Npa01] National Partnership for Advanced Computing Infrastructure. http://www.npaci.edu. April 2001.

[Sal99] Salleh, Shaharuddin, and Albert Zomaya. *Scheduling in Parallel Computing Systems: Fuzzy and Annealing Techniques*. Kluwer Academic Publishers, 1999.

[Sil98] Silberschatz, A., and P. Galvin. *Operating System Concepts*, 5th ed. Addison-Wesley, 1998.

[Spe95] SPECint95. The Standard Performance Evaluation Corporation, Manassas, VA, USA. http://open.specbench.org

[Sup99] de Supinski, B.R., and N.T.Karonis. "Accurately Measuring MPI Broadcasts in a Computational Grid". *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing.* 1999.

[Tak01] Takefusa, A. "Bricks: A Performance Evaluation System for Scheduling Algorithms on the Grids". *JSPS Workshop on Applied Information Technology for Science (JWAITS 2001)*. January 2001.

[Ull75] Ullman, J. "NP-complete scheduling problems," *Journal of Computing System Science*, Vol. 10, 1975.

[Vam01] Vampir: Visualization and Analysis of MPI programs. www.pallas.de/pages/vampir.htm. April 2001.

[Wei95] Weissman, Jon. *Scheduling Parallel Computations in a Heterogeneous Environment.* PhD thesis. University of Virginia, August 1995.

[Wei98] Weissman, Jon. "Gallop: The Benefits of Wide-Area Computing for Parallel Processing". *Journal of Parallel and Distributed Computing*, Vol. 54, Number 2, November 1998.

[Wei00] Weissman, Jon. "Scheduling Multi-component Applications in Heterogenous Wide-Area Networks." *Proceedings of the 9th Heterogeneous Computing Workshop*, April 2000.

[Wol98] Wolski, Rich. "Dynamically Forecasting Network Performance Using the Network Weather Service", *Cluster Computing*, 1998.

[Zag98] Zagorodnov, Dmitrii, Francine Berman, and Rich Wolski. "Application Scheduling on the Information Power Grid". Submitted to *International Journal of High Performance Computing Applications*.

[Zom01] Zomaya, A., Richard Lee, and Stephan Olariu. "An Introduction to Genetic-Based Scheduling in Parallel-Processor Systems". In *Solutions to Parallel and Distributed Computing Problems*, ed. A. Zomaya, F. Ercal, and S. Olariu. John Wiley & Sons, 2001.