Chapter 1

Introduction

Current distributed parallel platforms can provide the resources required to execute a scientific application efficiently. However, when these platforms are shared by multiple users, the performance of the applications using the system may be impacted in dynamic and often unpredictable ways. In particular, it becomes challenging to achieve good performance for any single application using the shared resources [SW98b, BCF+98, GLFK98, Lue98].

To use distributed parallel platforms efficiently, adaptive scheduling techniques can be developed to make use of the dynamic performance characteristics of shared resources. A fundamental problem in developing adaptive schedulers is the inability to accurately predict application execution performance.

Accurate performance predictions are necessary both to select a set of target resources and to schedule the application and its data upon them. When selecting resources, a prediction of application behavior may be used to evaluate, and eventually rank, different resource sets according to their likely performance. Application scheduling uses predictions of application behavior to make decisions regarding task and data mapping, as well as to estimate a program or task completion time. If any of these predictions are inaccurate, the resulting schedule may not efficiently use the resources, and application execution times may be unnecessarily increased.

One promising approach to modeling distributed parallel applications is **structural modeling**, which decomposes performance with respect to the functional structure of the application into a top-level model and interacting component models. Component models represent application tasks and application or model developers can select the parameters of a component model (representing both application and system characteristics) to reflect the dynamic time-dependent changes in effective capacities of the resources.

Performance models that do not reflect changing resource conditions are often insufficient for the resource selection and scheduling needs of shared distributed parallel applications. Many conventional models are parameterized by single (point) values. However, in shared environments, point values may provide an inaccurate representation of application behavior due to variations in resource performance. An alternative to using point values is the use of **stochastic values**, or distributions. Whereas a point value provides a single value representation of a quantity, a stochastic value provides a set of possible values weighted by probabilities to represent a range of likely behavior.

Parameters for structural models can be stochastic values (or include other additional information) in order to accurately reflect a range of observed behavior. These parameters, as well as stochastic performance predictions, can be used by schedulers to adapt to the dynamic behavior of shared distributed parallel environments. We take advantage of stochastic information and develop a **stochastic scheduler** to maximize the performance of shared clusters of workstations for parallel applications

This thesis presents three main contributions: an approach to define distributed parallel application performance models called **structural modeling***, the ability to parameterize these models with* **stochastic values** *in order to meet the prediction needs of shared clusters of workstations, and a* **stochastic scheduling policy** *that can make use of stochastic predictions to achieve better application execution times and more predictable application behavior.*

This chapter briefly defines key terms, motivates the need for this research, and outlines the approaches for the following chapters.

1.A High Performance Distributed Parallel Computing

As high performance applications have grown in size and complexity, computational resources have adapted to execute them efficiently as well. One way this has been achieved is through the parallelization of applications and the use of distributed resources. The area that we call **distributed parallel computing** is also known as heterogeneous computing, metacomputing, or computing on a computational grid. Basically, it involves the cooperative use of multiple distributed resources connected by networks to solve a single, resource-intensive application.

Distributed parallel systems exist on a number of scales, from a group of workstations in a lab to collections of high-performance computers and other resources linked across the nation or the world. We include in this category clusters of uniform workstations that may exhibit heterogeneous performance characteristics when shared by multiple users. Distributed parallel systems are often characterized by resources with variable performance and slow or undependable networks. Applications that execute on these networks may be resource-intensive, and have often been adapted from parallel applications previously run on a single MPP.

In order to successfully use distributed platforms for parallel applications, several basic issues must be addressed. First, hardware and software infrastructure is required to provide dependable, consistent, and inexpensive access to computation sites, remote resources, and data archives. For the purposes of this thesis, we assume that the underlying hardware is dependable, that is, we do not explicitly address the possibility of failure except as the perception of inadequate performance of a resource by the application. We also assume that an underlying communication substrate is available. For example, our experiments use either

PVM or MPI as a communication substrate, and target applications running over clusters of shared workstations. Secondly, monitoring tools are needed to observe the use of the system. For example, we make extensive use of the Network Weather Service [Wol96, Wol97, WSP97], a tool that can supply dynamic values for bandwidth, available CPU, and memory on a given system. Thirdly, schedulers are required to best use the available shared resources. That is the issue this thesis addresses.

1.B Performance Prediction Models

In order to build efficient schedules, we must have accurate **performance prediction models** for distributed parallel systems, that is, models that represent the wall-clock execution time of an application or part of an application. A **model** is a mathematical representation of a process, device or concept. Models are used to represent a variety of entities throughout computer science–from architectures to forecasts of resource usage to algorithm costs.

One way predictions and prediction models are used is as part of scheduling to assign work to processors. To illustrate the need for accurate prediction models, we present the following example. Consider a simple two-machine system, consisting of machines A and B, executing an embarrassingly parallel application with 30 units of identical work to be completed. In order to allocate work efficiently, we need an accurate estimate of the execution time per unit of work for each machine. This value is often the result of a prediction model.

If the estimate of time per unit of work for a machine is inaccurate, the execution time of the application may be unnecessarily extended. If we assume that the fastest execution time is achieved by having all the processors begin at the same time and finish at approximately the same time, we can achieve an efficient schedule by balancing the number of units of work each processor is assigned accordingly. A poor prediction model might report that Machine A will take 6 seconds and Machine B might take 9 seconds per unit of work. We would then assign 18 units to A and 12 units to B, for a predicted execution time of 108 seconds, as shown in Table 1.1. If Machine A had an actual execution time of 10 seconds per unit and Machine B took 5 seconds per unit, the execution time would be 180 seconds, since the load during execution would be unbalanced. A better data allocation, based on an accurate prediction of work, would be 10 units for Machine A and 20 units for Machine B.

	Machine A	Machine B	Assign A	Assign B	Total Time (Predicted)	Total Time (Actual)
Inaccurate Prediction	6 sec	9 sec	18 units	12 units	108 sec	180 sec
Actual Time	10 sec	5 sec	10 units	20 units	100 sec	100 sec

Table 1.1: Predicted and actual execution times for a unit of work in dedicated mode on two machines.

em In Chapter 3, we explore **structural modeling** as an approach to accurate performance modeling. Structural modeling uses the functional structure of the application to define a set of equations that reflect the time-dependent, dynamic mix of application tasks occurring during execution in a distributed parallel environment. We define **tasks** to be atomic units of either communication or computation[1], since computation and communication comprise the major performance activities associated with distributed parallel applications. In the case of computation, we consider a task to be an atomic unit of computation to be executed on a single machine. For communication, we consider a task to be an atomic grouping of communication calls, typically over short time period. The **functional structure** of an application can be thought of as a program dependence graph consisting of nodes that are communication and computation tasks, and edges that describe their relationship to one another. Also in Chapter 3, we present the steps

[1] Note that we generalize the conventional notion of "task" to include communication.

involved in building a structural model for a given application, and discuss the verification and usage of structural models for generating performance predictions for a variety of applications on three cluster platforms.

We do not model an application and a system as separate entities. Rather, we examine **implementations** of an application for a particular resource environment. Parallel distributed applications are typically not developed for a generic platform; they are tuned to the particular performance characteristics and architecture model of the resources available. For example, if an application includes a rendering routine, there may be several choices of rendering implementations available, each tuned in terms of the cache size, available math libraries, the precision needed for a given set of application data and/or the programming model supported by the resource. The development of a specific implementation will depend on characteristics of the application, its data, and the underlying platform. Therefore a structural model should represent implementations of an application with respect to particular platforms.

A good performance model, like a good scientific theory, is able to explain available observations and predict future events, while abstracting away unimportant details [Fos95]. Our models are represented by equations that are parameterized by both application and system characteristics. These equations allow us to abstract away details into the top level models and the parameters, but still accurately model the performance by showing the inter-relation of application and system factors through the component models and their parameterizations. If the predictions are for use in a timely manner – for example, as part of a runtime scheduling approach – they must be able to be evaluated quickly and with minimal computational overhead as well, a factor that we address throughout.

1.C Point Values and Stochastic Values

Most performance prediction models use parameters to describe system and application characteristics such as bandwidth, available CPU, message size, operation counts, etc. Model parameters are generally represented as a single likely value, which we refer to as a **point value**. For example, a point value for bandwidth might be 7 Mbits/second.

In practice, point values are often a best guess, an estimate under ideal circumstances, or a value that is accurate only for a given time-frame. In some situations it may be more accurate to represent system and application characteristics as a *distribution* of possible values over a range; for example, bandwidth might be reported as a normal distribution having a mean of 7 Mbits/second with a standard deviation of 1 Mbit/second. We refer to values represented as distributions as **stochastic values**. Whereas a point value gives a single value for a quantity, a stochastic value gives a set of possible values weighted by probabilities to represent a range of likely behavior [TK93].

The need for stochastic values in our example system can be seen when comparing predictions in a dedicated (single-user) environment with predictions in a multi-user (shared) environment. In a single-user environment, the time for a machine to perform one unit of work can often be represented as a point value since there is usually a negligible variance in resource availability and runtimes on dedicated machines.[2] For example, to balance the execution times of the example machines in a dedicated setting, where Machine A takes 10 seconds to complete a unit of work and Machine B takes 5 seconds (as shown in Table 1.2), Machine B should receive twice as much work as Machine A.

In a multi-user distributed environment, contention for processors and memories will cause the unit execution times to vary over time in a machine-dependent fashion; hence, simply having a good point value prediction of execution time may not be sufficient. If we determine unit execution time using a mean

[2]This may not be the case if an application is data dependent or non-deterministic.

	Machine A	Machine B	Assign A	Assign B
Single-user	10 sec	5 sec	10 units	20 units
Multi-user (point)	12 sec	12 sec	15 units	15 units
Multi-user (stochastic)	12 sec mean 0.3 sec sd	12 sec mean 1.8 sec sd	?	?

Table 1.2: Execution times for a unit of work in single-user and multi-user modes on two machines. Multi-user stochastic reports a mean and standard deviation, abbreviated sd.

capacity measure over a 24-hour period for a fixed amount of work, it is possible that A and B will have the same unit execution time on average, say 12 seconds per unit of work. In this case, it would make sense to equally balance the work between the machines.

However, a mean value is only summary statistic for a range of actual values, and may neglect critical information about the distribution of work over time. A stochastic value includes information about the distribution of values over the performance range. For example, it may be that because Machine B is much faster than Machine A, it has more users and therefore a more dynamic load. Because of this, at any given time the unit execution time for Machine B might be a mean of 12 seconds per unit of work, with a standard deviation of 1.8 seconds. If we assume the distributions are normal, then two standard deviations will cover approximately 95% of the values, so the time to complete a unit of work will usually vary over an interval from 8.4 seconds to 15.6 seconds. On the other hand, since Machine A is a slower machine without as many users contending for its resources, the actual unit execution time for this machine might be a mean of 12 seconds per unit of work and a standard deviation of 0.3 seconds (or 11.4 to 12.6 seconds per unit of work in most cases). On average, both machines perform the same, however at any given time their performance may differ radically. The amount of data to be assigned to each machine in this case is no longer a simple decision, but will depend on the goals of the user or application developer and the optimization function of the scheduling policy.

Conventional performance prediction models typically cannot take advantage of knowledge about the distribution of a model parameter value. *In Chapter 4, we define an extension to structural modeling to allow for model parameters that are* **stochastic values***, resulting in application performance predictions that are* **stochastic predictions**. We demonstrate that stochastic values can enhance the information conveyed by a performance prediction. In order to make use of stochastic values in prediction models, we need to know how the value can be represented and how to combine the values arithmetically. We examine three representations of distributional data: summarizing them as normal distributions; representing them as a simple interval of values; and using histograms. Arithmetic formulas for each representation are discussed, as well as the advantages and disadvantages of each approach. Chapter 4 also gives an overview of related approaches and additional research on the use of statistical approaches for pragmatic performance predictions.

1.D Stochastic Scheduling

With additional information provided by a stochastic prediction it is possible to develop more sophisticated application schedulers. As stated in [Ber98], an **application scheduler** may:

1. Select a set of resources on which to run the tasks of the application.

2. Assign application task(s) to compute resources.

3. Distribute data or co-locate data and computation.

4. Order tasks on a compute resource.

5. Order communication between tasks.

In the literature, the first two items are often referred to as **mapping**, the third item is referred to as **allocation**, and items 4, 5, or 1-5 are referred to as **scheduling**. Each of these actions uses some form of prediction.

When a scheduler makes use of stochastic information to determine a schedule for the application, we refer to it as a **stochastic scheduler**. Just as stochastic predictions predict a range of possible execution times, stochastic schedulers must chose from a set of possible schedules. Therefore schedulers of this nature will have the added actions of both defining the set of possible schedules and deciding which schedule in that set is appropriate for a given application, environment and user.

To illustrate using our example, if the application developer required the application to finish by a given time, a scheduler may choose to conservatively assign the data and assign more work to the smaller variance machine (Machine A). If the prediction supplied to the scheduler was a stochastic value represented using a normal distribution, one way to do this might be to use a 95% confidence interval[3] and assign the data using the mean plus two standard deviations as the time per unit of work. This value is chosen for normal distributions since it equates to approximately 95% of the values in the predicted range. In the example setting, we would assign 17 units of work (rounded up from 16.59) for Machine A and 13 units for Machine B, as shown in Table 1.3. If the execution times do indeed fall within the predicted interval (that is, machine A actually varied in performance from 11.4 to 12.6 seconds per unit of work and machine B actually varied in performance from from 8.4 seconds to 15.6 seconds). this data assignment would have a best case completion time of 148.2, but a worse case completion time of only 214.2, based on the given range of possible performance. We call this a **95% conservative schedule** since it corresponds to a 95% probability that each of the tasks will finish in the predicted range of time on their respective processors.

[3]A confidence interval is an interval of plausible values for a characteristic constructed so that the value of the characteristic will be captured inside the interval [DP96a].

	Assign A	Assign B	Best time	Worst time
95% Conservative Prediction	17 units	13 units	148.2 sec	214.2 sec
95% Optimistic Prediction	13 units	17 units	152.1 sec	265.2 sec

Table 1.3: Conservative and optimistic scheduling assignments.

If there was a small penalty for poor predictions, a scheduler might optimistically assign a greater portion of the work to the machine with the maximum potential performance, Machine B. Likewise, if the application developer had reason to believe that the previous distribution of data was too conservative, he/she might want to assign more units of work to Machine B. For example, the opposite of a 95% conservative schedule would be a **95% optimistic schedule**, as shown in line 2 of Table 1.3, corresponding to a 5% conservative schedule, or a 5% confidence interval that the task on a processor will complete in a given time. The 95% optimistic schedule would have a best case completion time of 152.1, but a worse case completion time of 265.2 in this setting.

As third option, we could consider allowing the percentage of conservatism to vary for the two platforms, that is, instead of a 95% optimistic or 95% conservative schedule, we could vary these percentages along the entire spectrum. Knowing the environment in order to determine the percentage of conservatism needed is as important as having valid information when scheduling with stochastic data, as the possible variations are extensive. A scheduling approach that can make these decisions and provide good execution performance is the ultimate goal of the work described in this thesis.

In Chapter 5, we present an approach to developing a **stochastic scheduler**. The stochastic scheduling policy we present is based on time balancing and uses a system of benefits and penalties to define the values to use in the set of possible schedules. This approach is similar to the scheduling approach of Sih and Lee [SL90b], in which the amount of work is increased for a processor with desirable characteristics (such as a light load, fast processor speed, etc.), called

benefits, or decreased when a machine has undesirable properties (such as poor network connectivity, small memory capacity, etc.), called **penalties**. The resulting schedules often achieve a faster application execution time than comparative schedules built without the additional stochastic data, and frequently result in more predictable application execution behavior for the experimental applications.

1.E Summary

The three main research contributions presented in this thesis are: an approach to performance modeling called structural modeling, the ability to parameterize these models using stochastic information, and a stochastic scheduling policy that adapts to the performance characteristics available resources of shared clusters of workstations. The goal of this dissertation is to present a technique for modeling distributed parallel applications and a scheduling approach that can utilize enhanced modeling information to determine performance-efficient application schedules.

1.F Dissertation Outline

The thesis is organized as follows: Chapter 2 presents the experimental environment used for the thesis. It details the platforms used, the workload characterization, and the test suite of applications. Chapter 3 defines structural modeling. Chapter 4 presents stochastic parameters and predictions. Chapter 5 describes the use of the stochastic predictions in a stochastic scheduling policy based on benefits and penalties. Chapter 6 presents conclusions and contributions, and discusses the extensive possible future work.

Chapter 2

Experimental Environment

In order to experimentally validate the claims in this thesis, we define an experimental space with a three axes: platform, workload and application. This chapter defines these axes, and details the domain for each of them.

2.A Experimental Platforms

We focus on environments where resources available to a user will have variable, possibly unpredictable, performance due to the presence of other users on the system. Although performance variation can occur on a single machine or on a space-shared parallel platform (where sets of nodes are assigned to a single user at a time), we focus specifically on shared clusters of workstations. In particular, our experiments were run on the following platforms:

UCSD Parallel Computation Lab (PCL) Cluster. This cluster consists of four Sparc workstations located in the UCSD Parallel Computation Lab. The machines are connected with either 10 Mbit (slow) or 100 Mbit (fast) ethernet over an Intel switch, and all run Solaris. The machines in the PCL cluster are:

Thing1: 200MHz Sun Ultra 1, 128MBytes RAM, fast ethernet connection

Thing2: 200MHz Sun Ultra 1, 128MBytes RAM, fast ethernet connection

Picard: 40MHz Sparc 10, 32MBytes RAM, slow ethernet connection

Lorax: 85MHz Sparc 5, 32MBytes RAM, slow ethernet connection

These machines are workstations on student's desks, and are used for both day-to-day activities like reading email, using Netscape, and running compilers, as well as for more compute-intensive experimental work.

Application-Level Scheduling (AppLeS) Research Group Linux Cluster. This cluster consists of four PC's located in the Parallel Computation Lab. They are connected using full-duplex, 100 megabit (fast) switched ethernet.

This cluster is made up of:

Soleil: 400 MHz Pentium II, Debian GNU/Linux, 384 MBytes RAM

Saltimbanco: 400 MHz Pentium II, Debian GNU/Linux, 256 MBytes RAM

Mystere: 400 MHz Pentium II, Debian GNU/Linux, 384 MBytes RAM

Quidam: 400 MHz Pentium II, Debian GNU/Linux, 256 MBytes RAM

The Linux Cluster is primarily used as a computation resource, and not used on an individual basis. In addition, this is a lab-administered machine, so extensive dedicated experiments were possible on this platform.

San Diego Supercomputer Center (SDSC) Alpha Farm. This platform consists of eight DEC Alpha 3000/400 workstations running OSF Unix located at the San Diego Supercomputer Center. They each run at 133 MHz, and have an 8K primary cache and a 512K secondary cache, along with 128MBytes of RAM. They are connected via high-speed ethernet.

This shared platform has been used by various researchers in a compute-intensive manner, usually as a system, but at times individually by machine.

These three platforms span a wide range of cluster system characteristics. In terms of performance, the Linux cluster machines were significantly faster than the PCL cluster machines. The PCL cluster machines were faster than the Alphas, although there were twice as many machines in the Alpha Farm as the PCL cluster. In terms of networking, both the Alpha Farm and the Linux Cluster used high speed ethernet connections, but communication costs were typically much more a factor in the performance on the Linux cluster due to it's faster workstations. In addition, while the Alpha Farm and Linux Cluster both have homogeneous processors, allowing us to make some simplifying modeling assumptions, the PCL platform had three types of machines and different network connections as well. Also, although the machines in the Alpha Farm and the Linux cluster were homogeneous, they were not used uniformly, as each had a single workstation used more often than the others to connect to the system. Finally, these systems all had different operating systems, different C and Fortran compilers, etc.

2.B Workload Characterization

The second axis of our experimental space is workload. By **workload** we mean the set of applications running on a given platform within a specific period. Workload is typically measured in a performance model by the effect it has on the platform resource capacities by such measures as *bandwidth*, *available CPU*, *latency*, etc. One of the primary sources of performance variation on a production platform is the fluctuation of available CPU and bandwidth at a given time. By *available CPU* we mean the percentage of a machine available for use by a given application, as used by Wolski [Wol96]. These are also the parameters that are the most difficult to control in an environment where a user cannot limit the access of other users to the machines or networks of interest. Note that a "typical" workload can vary radically from one site to another, from one time interval to another, and from one resource to another, so there is no universal typical workload.

We addressed the lack of a typical workload in two ways. First, we ran experiments on platforms where we could control the outside use of the resources – unlike the average user – and could simulate additional workloads on them. This helped in validating and verifying the performance prediction strategies and models by providing repeatable results. Second, we ran experiments on production platforms in competition with other users to validate and verify our predictive and scheduling techniques in a realistic setting. Details on the workload simulations and production workloads are discussed as part of the experimental results in Chapters 4 and 5.

2.C Applications

The third axis of our experimental space is the application. Distributed parallel applications have some similarity with both parallel and distributed applications. Two classes of distributed parallel applications, **Master-Slave** and **Regular SPMD**, are detailed in the next two sections, along with the representative applications and implementations for each class that were used as a test suite for this thesis.

Parallel applications targeted to distributed resources differ from parallel applications targeted to MPPs in several ways. Characteristics of parallel distributed applications include:

Coarse granularity. In order to amortize the high communication costs of commercial networks and off-the-shelf operating systems, distributed parallel applications are typically coarse-grained. Many MIMD or SPMD applications that were originally developed for MPP's port well to distributed parallel platforms, but SIMD applications require synchronization that cannot be performed efficiently in this environment. Many vector codes have been reworked to suit networked resources, but this can be a difficult task.

Small number of tasks. Distributed parallel applications generally consist of a small number of tasks [CS95, DS95, DTMH96, PRW94, MMFM93, WK93, YB95]. The size of individual tasks vary from application to application.

Modular Implementation. Many distributed parallel applications are actually several parallel applications joined together to solve a larger problem. For example, the Global Climate Modeling group at UCLA combined an ocean physics code [Cox84], an atmospheric physics code [AL77, MMFM93] and an atmospheric chemistry code [DS95] to more accurately investigate global weather patterns. Each of these codes were developed for separate platforms and but are executed in tandem over a wide-area high-performance distributed parallel platform to solve a larger problem [MFS+95]. Distributed parallel applications such as this may also use a mixture of languages and computational paradigms (data parallel, sequential, vector), and may even use different underlying communication substrates (PVM, MPI, NEXUS, etc.).

Tasks may have multiple implementations. A task in a parallel distributed application may have multiple implementations, each perhaps tuned to a different platform or performance characteristic. For example, if an application uses a rendering routine, there may be several choices of rendering routines targeted to different possible platforms, each differing in performance capability and algorithmic approach.

Many parallel distributed applications fall within the classes of Master-Slave and Regular SPMD applications, described in the following sections. Within these classes, we attempted to vary other characteristics as well. The applications in our test suite are written in C, Fortran, or a mix of both. Some run over PVM, some use MPI. Some were developed by students, some are kernels of scientific codes, and others are part of scientific benchmarking suites. The communication and computation ratios vary from computation-intensive to communication-intensive, although most are coarse-grained, with a high computation to communication ratio as seen in most practical codes for distributed platforms. The codes we are using have different synchronization styles, and address data and work distribution issues differently. Future work includes extending the techniques in this thesis for other application classes, such as pipelined and multi-level codes.

2.D Master-Slave Applications

One class of application that achieves good performance on clusters of workstations is the **Master-Slave** class of applications. These applications are structured so that a Master process controls the data for multiple Slave processes, with all the Slaves running the same code. The Master-Slave paradigm is widely used in the cluster computing setting because it is easy for the Master to dynamically adjust the data decomposition across the Slave computations, and to achieve good performance on systems with changing characteristics. This is especially important for multi-user clusters of heterogeneous workstations. Programming approaches vary for Master-Slave computations. We denote the two most common Master-Slave paradigms as **Assign** and **Request**, and characterize their programming approaches below.

Assign. In an Assign-style Master-Slave application, the Master process assigns all of the data to the Slave processes before they begin to execute. The Slaves run in parallel, and send their results back to the Master process. This usually repeats for some number of iterations.

An example of an Assign Master-Slave application is shown in Figure 2.1 and the pseudocode is given in Figure 2.2. This type of application consists of a Master task that determines how much work each of P Slave tasks performs. The Master communicates to the Slaves generally using a Scatter routine, perhaps a multicast, and receives information back from the Slaves with a Gather routine, perhaps a series of receives. Synchronization occurs every iteration at the Master task, but may also occur for each communication or computation task depending on the implementation.

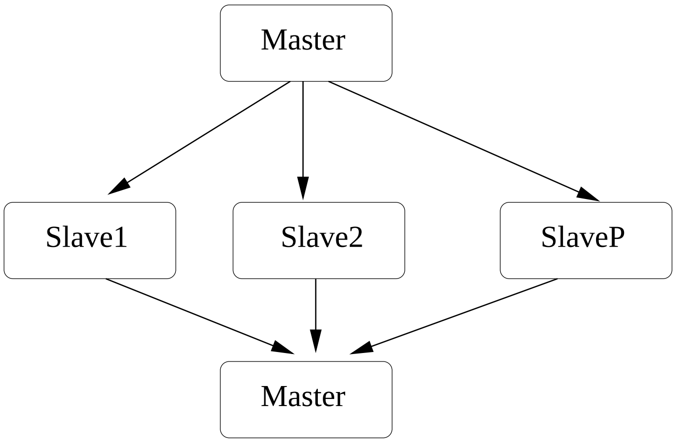

Figure 2.1: Graph of one iteration of an Assign Master-Slave application.

```
Master Computation:
   Start Slaves
   For i = 1 to MaxIterations
      Broadcast Data to Slaves
      Receive Data from all Slaves
      End of iteration computation
   Kill Slaves

Slave Computation:
   While (True)
      Receive Data
      Compute Slave Work
      Send Data to Master
```

Figure 2.2: Pseudocode for an Assign Master-Slave application.

Request. In a Request-style Master-Slave application, the Master task holds the data and Slave tasks request portions of it. Each piece of work can be thought of as a **job**. Output from the job can either be returned a piece at a time when the Slave requests another job, or all at once when the pool of jobs is empty. There is no synchronization between the Slaves, but there may be a synchronization between iterations as the Master determines a new set of jobs to be handed out. This style of Master-Slave application is also called *work queue*, *task farming*, or *self-scheduling* [Wil95]. This is characterized by the graph in Figure 2.3 and the pseudocode in Figure 2.4.

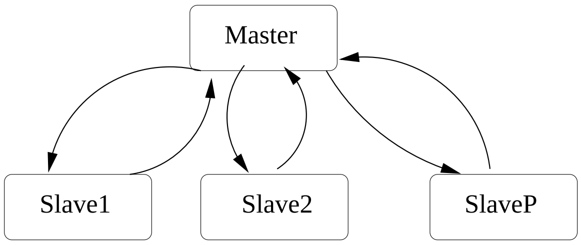

Figure 2.3: Graph of one iteration of a Request Master-Slave application.

```
Master Computation:
   Start Slaves
   For i = 1 to MaxIterations
      Send initial Data to each Slave
      Receive Data from all Slaves
      While there is Data uncomputed
         Send Data set to Slave
         Receive Data from all Slaves
      End of iteration computation
   Kill Slaves

Slave Computation:
   While (True)
      Receive Data
      Compute Slave Work
      Send Data to Master
      Ask Master for more Data
```

Figure 2.4: Pseudocode for a Request Master-Slave application.

In the following subsections we present the Master-Slave distributed applications that are used as the test suite codes for this thesis.

2.D.1 Genetic Algorithm Application

Genetic algorithms are search algorithms that model survival over generations of populations. In a typical genetic algorithm, there is a "population" of possible solutions that are "bred" with one another to create "child" solutions. The child solutions are evaluated, and the "fittest" (best) of the new "generation" of solutions are used as "parents" for the next generation. Genetic algorithms have been used extensively by the artificial intelligence community as an optimization technique for NP-complete and NP-hard problems. They are now being used by several groups of computational scientists [DTMH96, SK95, PM94] to address problems such as protein folding.

We implemented a genetic algorithm (GA) heuristic for the Traveling Salesman Problem (TSP) [LLKS85, WSF89]. Our distributed implementation of this genetic algorithm uses a global population and synchronizes between generations [Bha96]. It was written in C using PVM. This application has a typical Assign Master-Slave structure. All of the Slaves operate on a global population (each member of the population is a solution to the TSP for a given set of cities) that is broadcast to them by the Master using a PVM multicast routine. Each Slave works in isolation to create a specified number of children (representing new tours), and to evaluate them. This data is sent back to the Master. Once all the sets of children are received by the Master, they are sorted (by efficiency of the tour), some percentage are chosen to be the next generation, and the cycle begins again. Figure 2.5 shows the pseudocode for this application. In our implementation, a Slave process runs on the Master processor. Data assignment is done statically at runtime.

2.D.2 High Temperature Superconductor Kernel

The High Temperature Superconductor Kernel (HiTemp) is used as part of the suite of applications used to test new versions of PVM, and is a simplified

```
Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Broadcast entire population to Slave
    Receive Data from all Slaves
    Sort children
    Select next generation
  Kill Slaves
  Return best child
Slave Computation:
  While (True)
    Receive population from Master
    For j = 1 to NumberofChildren
      Randomly pick two parents
      Cross parents to create child
      Evaluate child
    Send all children to Master
```

Figure 2.5: Pseudocode for GA Application

kernel of a larger superconductor code [Koh97]. The Master in this application spawns Slaves that do matrix inverses of various sizes using specially-tuned BLAS routines.

HiTemp uses a "pool of jobs" approach, typical of the Request-style Master-Slave applications. The Master is started, and it spawns a Slave task on each available host. Each Slave then requests individual jobs from the available work pool held by the Master, in this case the matrix information, until the pool of jobs is empty. In this application, the actual jobs vary in size nondeterministically to simulate superconductor environmental data. This code is used as a test code for PVM, and is written in C and Fortran using PVM library calls. Data decomposition is done dynamically at runtime by the Slaves, and the only synchronization is between iterations. A Slave process runs on the Master process, but this interferes at most minimally with the Slave execution. Pseudocode for this application is given in Figure 2.6.

```
Master Computation:
  Start Slaves
  For i = 1 to MaxIterations
    Send one job to each Slave
    While there are jobs in work pool
      Receive result of job from Slave
      Send job to Slave
    Receive result of last job from each Slave
  Kill Slaves

Slave Computation:
  While (True)
    Receive job
    Compute matrix inverse
    Send job result to Master
```

Figure 2.6: Pseudocode for HiTemp kernel

2.D.3 N-body Code

N-body simulations, also known as particle simulations, arise in astrophysics and celestial mechanics (gravity is the force), plasma simulation and molecular dynamics (electrostatic attraction or repulsion is the force), and computational fluid dynamics (using the vortex method). These problems were some of the first problems addressed in parallel computation and continue to be studied in distributed parallel environments. The papers collected in the SIAM minisymposium [HT97] and its predecessor [Bai95] offer ample evidence of the breadth and importance of N-body methods.

The problem our N-body implementation addresses is the tracking of the motion of planetary bodies. We scale the number of bodies involved, using a two-dimensional space with boundaries. Our implementation is an exact N^2 version that calculates the force of each planet on every other, and then updates their velocities and locations accordingly for each time step. It is based on an Assign-style Master-Slave approach [CI98] and written in C using PVM. For each iteration, the Master sends the entire set of bodies to each Slave, and also assigns a portion of the bodies to each processor to calculate the next iteration of values. The Slaves calculate the new positions and velocities for their assigned bodies, and then send this data back to the Master process. Data decomposition is done statically at runtime by the Master, which also reads in an initial data file. Figure 2.7 shows pseudocode for this application.

```
Master Computation:
  Start Slaves

  For i = 1 to MaxIterations
    Broadcast all body data to Slaves
    Send each slave sector information
    Receive data for each sector from Slaves
    Sort bodies according to new positions
  Kill Slaves

Slave Computation:
  While (True)
    Receive velocity, force, position data
    Compute new data for bodies in sector
    Send sector data to Master
```

Figure 2.7: Pseudocode for the N^2 N-body Code.

2.E Regular SPMD Applications

In conjunction with the class of Master-Slave applications, we also focused on the class of **Regular SPMD** applications. Regular SPMD applications are comprised of multiple identical tasks (with no "lead" task) working on different data sets. We focus on applications that have regular, well-structured communication patterns. Such codes are widely used in the scientific community. A few examples include: NASA's High Speed Civil Transport design code [GW], an application to model the San Diego Bay [HT], as well as matrix codes and other mathematical functions at the heart of many scientific codes, including successive over-relaxation codes, multigrid codes and Fourier transforms [BHS+95].

We implemented a test suite of Regular SPMD applications as examples of this class. In the following subsections, we address the methods of data distribution, the communication patterns, and synchronization issues for each.

2.E.1 Successive Over-Relaxation Code

Successive Over-Relaxation (SOR) is a Regular SPMD code that solves Laplace's equation. Our implementation uses a red-black stencil approach where the calculation of each point in a grid at time t is dependent on the values in a stencil around it at time $t-1$, as depicted in Figure 2.8. The application is divided into "red" and "black" phases, with communication and computation alternating for each. In our implementation, these two phases repeat for a predefined number of iterations. Pseudocode for the SOR is given in Figure 2.9.

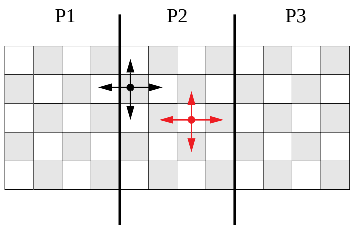

Figure 2.8: Strip decomposition for Red-Black SOR

We used a strip decomposition to partition the code onto the workstation clusters. In this implementation, all the processors are equal peers, as opposed to the Master and Slave set-up, and each is responsible for an individual data set. Communication is done twice an iteration, and there is only loose synchronization between neighboring processors.

2.E.2 NPB LU Benchmark

We implemented distributed forms of several of the NAS Parallel Benchmarks (NPB) [BHS+95]. These benchmarks were developed to embody key compu-

```
Setup, including decomposition
For i = 1 to MaxIterations
  RED PHASE
    Receive right and left data
    For j = FirstRow to LastRow
      For k = FirstCol to LastCol
        Compute red values
    Share left column with left neighbor
    Share right column with right neighbor
  BLACK PHASE
    Receive right and left data
    For j = FirstRow to LastRow
      For k = FirstCol to LastCol
        Compute black values
    Share left column with left neighbor
    Share right column with right neighbor
```

Figure 2.9: Pseudocode for the SOR benchmark.

tational and data-movement characteristics of typical processing in computational fluid dynamics (CFD) calculations, and have gained wide acceptance as a standard indicator of supercomputer and parallel distributed platform performance. They are carefully coded to use modern algorithms, avoid unnecessary computation, and generally represent real-world codes. They perform reasonably well across many platforms.

The **LU benchmark** is a simulated CFD application that solves a block-lower-triangular/block-upper-triangular system of equations. This system of equations is the result of an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. The LU benchmark finds lower triangular and upper triangular matrixes such that $L \cdot U = A$ for an original matrix A. The NPB version of the LU benchmark is based on the NX reference implementation from 1991 [BBB$^+$91]. It consists of a startup phase, a lower/upper triangular-solving function that is iterated, and then a verification phase. The lower-upper triangular solving function is an iterative implicit method that partitions the left-hand side matrix into a lower triangular matrix and an upper triangular matrix.

A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical columns of data assigned to the individual processors. The computation then proceeds to compute from one corner of a given z plane, on diagonals, to the opposite corner of the same z plane, and then proceeds to the next z plane. Communication of boundary data occurs after completion of the computation on all diagonals that contact an adjacent partition. This constitutes a diagonal pipelining method and is called a "wavefront" method by its authors [BFVW93]. This is depicted for one Z plane in Figure 2.10. It results in a relatively large number of small communications of 5 words each.

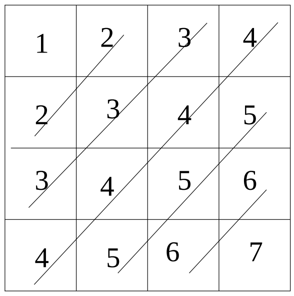

Figure 2.10: Wavefront execution of the LU benchmark, shown for a single z plane, labeled with order of execution possible.

Although this algorithm is not an optimal solution for the problem, it is used as a NPB benchmark because it is very sensitive to the small-message communication performance of an MPI implementation. It is the only benchmark in the NPB 2.0 suite that sends large numbers of very small (40 byte) messages. For this reason we included it in our Regular SPMD test suite.

2.E.3 SP Benchmark

The SP Benchmark, part of the NPB suite [BHS+95], is a simulated CFD application that solves systems of equations resulting from an approximately factored implicit finite-difference discretization of the Navier-Stokes equations. The SP code solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme. It solves three sets of un-coupled systems of equations, first in the x, then in the y, then in the z direction. It uses a multi-partition algorithm [BC88] in which each processor is responsible for several disjoint sub-blocks of points ("cells") of the grid. The cells are arranged such that for each direction of the "line solve phase", the cells belonging to a certain processor will be evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent.

Structurally, this code has a small initialization phase, followed by a broadcast of the data to all the processors. Then the bulk of the work is done in a phase that repeats for ten iterations and consists of copying faces (overlapping data) between each processor, solving in the x direction, the y direction, and the z direction, and an update task. There is loose synchronization between nodes for each iteration.

2.E.4 EP Kernel

We include the NPB Embarrassingly Parallel (EP) Kernel [BHS+95] as an example of a Regular SPMD application with no synchronization and minimal communication.

The NPB EP benchmark generates 2^n pairs of random numbers, and tests them to see whether Gaussian random deviates can be constructed form them. The pseudo-random number generator used in this is of the linear congruential recursion type. This problem is typical of many Monte Carlo simulation applications. The only requirement for communication is a small gathering of information at the end.

There is no synchronization. The data decomposition, in this case the assignment of a number of pairs to generate, is done statically at compile time.