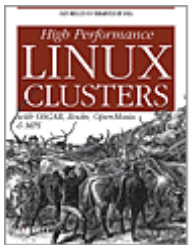


High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI By Joseph D. Sloan

Chapter 1: **Cluster Architecture**

Computing speed isn't just a convenience. Faster computers allow us to solve larger problems, and to find solutions more quickly, with greater accuracy, and at a lower cost. All this adds up to a competitive advantage. In the sciences, this may mean the difference between being the first to publish and not publishing. In industry, it may determine who's first to the patent office.

Traditional high-performance clusters have proved their worth in a variety of uses—from predicting the weather to industrial design, from molecular dynamics to astronomical modeling. `High-performance computing (HPC)` has created a new approach to science—modeling is now a viable and respected alternative to the more traditional experiential and theoretical approaches.

Clusters are also playing a greater role in business. High performance is a key issue in data mining or in image rendering. Advances in clustering technology have led to high-availability and load-balancing clusters. Clustering is now used for mission-critical applications such as web and FTP servers. For example, Google uses an ever-growing cluster composed of tens of thousands of computers.

Because of the expanding role that clusters are playing in distributed computing, it is worth considering this question briefly. There is a great deal of ambiguity, and the terms used to describe clusters and distributed computing are often used inconsistently. This chapter doesn't provide a detailed taxonomy—it doesn't include a discussion of Flynn's taxonomy or of cluster topologies. This has been done quite well a number of times and too much of it would be irrelevant to the purpose of this book. However, this chapter does try to explain the language used. If you need more general information, see the Appendix A for other sources. *High Performance Computing*, Second Edition (O'Reilly), by Dowd and Severance is a particularly readable introduction.

When computing, there are three basic approaches to improving performance—use a better algorithm, use a faster computer, or divide the calculation among multiple computers. A very common analogy is that of a horse-drawn cart. You can lighten the load, you can get a bigger horse, or you can get a team of horses. (We'll ignore the option of going into therapy and learning to live with what you have.) Let's look briefly at each of these approaches.

Modern Computing and the Role of Clusters

Because of the expanding role that clusters are playing in distributed computing, it is worth considering this question briefly. There is a great deal of ambiguity, and the terms used to describe clusters and distributed computing are often used inconsistently. This chapter doesn't provide a detailed taxonomy—it doesn't include a discussion of Flynn's taxonomy or of cluster topologies. This has been done quite well a number of times and too much of it would be irrelevant to the purpose of this book. However, this chapter does try to explain the language used. If you need more general information, see the Appendix A for other sources. *High Performance Computing*, Second Edition (O'Reilly), by Dowd and Severance is a particularly readable introduction.

When computing, there are three basic approaches to improving performance—use a better algorithm, use a faster computer, or divide the calculation among multiple computers. A very common analogy is that of a horse-drawn cart. You can lighten the load, you can get a bigger horse, or you can get a team of horses. (We'll ignore the option of going into therapy and learning to live with what you have.) Let's look briefly at each of these approaches.

First, consider what you are trying to calculate. All too often, improvements in computing hardware are taken as a license to use less efficient algorithms, to write sloppy programs, or to perform meaningless or redundant calculations rather than carefully defining the problem. Selecting appropriate algorithms is a key way to eliminate instructions and speed up a calculation. The quickest way to finish a task is to skip it altogether.

If you need only a modest improvement in performance, then buying a faster computer may solve your problems, provided you can find something you can afford. But just as there is a limit on how big a horse you can buy, there are limits on the computers you can buy. You can expect rapidly diminishing returns when buying faster computers. While there are no hard and fast rules, it is not unusual to see a quadratic increase in cost with a linear increase in performance, particularly as you move away from commodity technology.

Types of Clusters

Originally, "clusters" and "high-performance computing" were synonymous. Today, the meaning of the word "cluster" has expanded beyond high-performance to include `high-availability (HA) clusters` and `load-balancing (LB) clusters`. In practice, there is considerable overlap among these—they are, after all, all clusters. While this book will focus primarily on high-performance clusters, it is worth taking a brief look at high-availability and load-balancing clusters.

High-availability clusters, also called `failover` clusters, are often used in mission-critical applications. If you can't afford the lost business that will result from having your web server go down, you may want to implement it using a HA cluster. The key to high availability is redundancy. An HA cluster is composed of multiple machines, a subset of which can provide

the appropriate service. In its purest form, only a single machine or server is directly available—all other machines will be in standby mode. They will monitor the primary server to insure that it remains operational. If the primary server fails, a secondary server takes its place.

The idea behind a load-balancing cluster is to provide better performance by dividing the work among multiple computers. For example, when a web server is implemented using LB clustering, the different queries to the server are distributed among the computers in the clusters. This might be accomplished using a simple round-robin algorithm. For example, *Round-Robin DNS* could be used to map responses to DNS queries to the different IP addresses. That is, when a DNS query is made, the local DNS server returns the addresses of the next machine in the cluster, visiting machines in a round-robin fashion. However, this approach can lead to dynamic load imbalances. More sophisticated algorithms use feedback from the individual machines to determine which machine can best handle the next task.

Keep in mind, the term "load-balancing" means different things to different people. A high-performance cluster used for scientific calculation and a cluster used as a web server would likely approach load-balancing in entirely different ways. Each application has different critical requirements.

Distributed Computing and Clusters

While the term *parallel* is often used to describe clusters, they are more correctly described as a type of `distributed computing`. Typically, the term parallel computing refers to tightly coupled sets of computation. Distributed computing is usually used to describe computing that spans multiple machines or multiple locations. When several pieces of data are being processed simultaneously in the same CPU, this might be called a parallel computation, but would never be described as a distributed computation. Multiple CPUs within a single enclosure might be used for parallel computing, but would not be an example of distributed computing. When talking about systems of computers, the term parallel usually implies a homogenous collection of computers, while distributed computing typically implies a more heterogeneous collection. Computations that are done asynchronously are more likely to be called distributed than parallel. Clearly, the terms parallel and distributed lie at either end of a continuum of possible meanings. In any given instance, the exact meanings depend upon the context. The distinction is more one of connotations than of clearly established usage.

Since cluster computing is just one type of distributed computing, it is worth briefly mentioning the alternatives. The primary distinction between clusters and other forms of distributed computing is the scope of the interconnecting network and the degree of coupling among the individual machines. The differences are often ones of degree.

Clusters are generally restricted to computers on the same subnetwork or LAN. The term `grid computing` is frequently used to describe computers working together across a WAN or the Internet. The idea behind the term "grid" is to invoke a comparison between a power grid and a computational grid. A computational grid is a collection of computers that

provide computing power as a commodity. This is an active area of research and has received (deservedly) a lot of attention from the National Science Foundation. The most significant differences between cluster computing and grid computing are that computing grids typically have a much larger scale, tend to be used more asynchronously, and have much greater access, authorization, accounting, and security concerns. From an administrative standpoint, if you build a grid, plan on spending a lot of time dealing with security-related issues. Grid computing has the potential of providing considerably more computing power than individual clusters since a grid may combine a large number of clusters.

Limitations

While clusters have a lot to offer, they are not panaceas. There is a limit to how much adding another computer to a problem will speed up a calculation. In the ideal situation, you might expect a calculation to go twice as fast on two computers as it would on one. Unfortunately, this is the limiting case and you can only approach it.

Any calculation can be broken into blocks of code or instructions that can be classified in one of two exclusive ways. Either a block of code can be parallelized and shared among two or more machines, or the code is essentially serial and the instructions must be executed in the order they are written on a single machine. Any code that can't be parallelized won't benefit from any additional processors you may have.

There are several reasons why some blocks of code can't be parallelized and must be executed in a specific order. The most obvious example is I/O, where the order of operations is typically determined by the availability, order, and format of the input and the format of the desired output. If you are generating a report at the end of a program, you won't want the characters or lines of output printed at random.

Another reason some code can't be parallelized comes from the data dependencies within the code. If you use the value of x to calculate the value of y, then you'll need to calculate x before you calculate y. Otherwise, you won't know what value to use in the calculation. Basically, to be able to parallelize two instructions, neither can depend on the other. That is, the order in which the two instructions finish must not matter.

Thus, any program can be seen as a series of alternating sections—sections that can be parallelized and effectively run on different machines interspersed with sections that must be executed as written and that effectively can only be run on a single machine. If a program spends most of its time in code that is essentially serial, parallel processing will have limited value for this code. In this case, you will be better served with a faster computer than with parallel computers. If you can't change the algorithm, big iron is the best approach for this type of problem.

My Biases

The material covered in this book reflects three of my biases, of which you should be aware. I have tried to write a book to help people get started with clusters. As such, I have focused primarily on mainstream, high-performance computing, using open source software. Let me explain why.

First, there are many approaches and applications for clusters. I do not believe that it is feasible for any book to address them all, even if a less-than-exhaustive approach is used. In selecting material for this book, I have tried to use the approaches and software that are the most useful for the largest number of people. I feel that it is better to cover a limited number of approaches than to try to say too much and risk losing focus. However, I have tried to justify my decisions and point out options along the way so that if your needs don't match my assumptions, you'll at least have an idea where to start looking.

Second, in keeping with my goal of addressing mainstream applications of clusters, the book primarily focuses on high-performance computing. This is the application from which clusters grew and remains one of their dominant uses. Since high availability and load balancing tend to be used with mission-critical applications, they are beyond the scope of a book focusing on getting started with clusters. You really should have some basic experience with generic clusters before moving on to such mission-critical applications. And, of course, improved performance lies at the core of all the other uses for clusters.

Finally, I have focused on open source software. There are a number of proprietary solutions available, some of which are excellent. But given the choice between comparable open source software and proprietary software, my preference is for open source. For clustering, I believe that high-quality, robust open source software is readily available and that there is little justification for considering proprietary software for most applications.

While I'll cover the basics of clusters here, you would do well to study the specifics of clusters that closely match your applications as well. There are a number of well-known clusters that have been described in detail. A prime example is Google, with literally tens of thousands of computers. Others include clusters at Fermilab, Argonne National Laboratory (Chiba City cluster), and Oak Ridge National Laboratory. Studying the architecture of clusters similar to what you want to build should provide additional insight. Hopefully, this book will leave you well prepared to do just that.

Chapter 2: **Cluster Planning**

This chapter is an overview of cluster planning. It begins by introducing four key steps in developing a design for a cluster. Next, it presents several questions you can ask to help you determine what you want and need in a cluster. Finally, it briefly describes some of the software decisions you'll make and how these decisions impact the overall architecture of the cluster. In addition to helping people new to clustering plan the critical foundations of their cluster, the chapter serves as an overview of the software described in the book and its uses.

Designing a cluster entails four sets of design decisions. You should:

1. Determine the overall mission for your cluster.
2. Select a general architecture for your cluster.
3. Select the operating system, cluster software, and other system software you will use.
4. Select the hardware for the cluster.

While each of these tasks, in part, depends on the others, the first step is crucial. If at all possible, the cluster's mission should drive all other design decisions. At the very least, the other design decisions must be made in the context of the cluster's mission and be consistent with it.

Selecting the hardware should be the final step in the design, but often you won't have as much choice as you would like. A number of constraints may drive you to select the hardware early in the design process. The most obvious is the need to use recycled hardware or similar budget constraints. Chapter 3 describes hardware consideration is greater detail.

Defining what you want to do with the cluster is really the first step in designing it. For many clusters, the mission will be clearly understood in advance. This is particularly true if the cluster has a single use or a few clearly defined uses. However, if your cluster will be an open resource, then you'll need to anticipate potential uses. In that case, the place to start is with your users.

Design Steps

Designing a cluster entails four sets of design decisions. You should:

1. Determine the overall mission for your cluster.
2. Select a general architecture for your cluster.
3. Select the operating system, cluster software, and other system software you will use.
4. Select the hardware for the cluster.

While each of these tasks, in part, depends on the others, the first step is crucial. If at all possible, the cluster's mission should drive all other design decisions. At the very least, the other design decisions must be made in the context of the cluster's mission and be consistent with it.

Selecting the hardware should be the final step in the design, but often you won't have as much choice as you would like. A number of constraints may drive you to select the hardware

early in the design process. The most obvious is the need to use recycled hardware or similar budget constraints. Chapter 3 describes hardware consideration is greater detail.

Determining Your Cluster's Mission

Defining what you want to do with the cluster is really the first step in designing it. For many clusters, the mission will be clearly understood in advance. This is particularly true if the cluster has a single use or a few clearly defined uses. However, if your cluster will be an open resource, then you'll need to anticipate potential uses. In that case, the place to start is with your users.

While you may think you have a clear idea of what your users will need, there may be little semblance between what you think they should need and what they think they need. And while your assessment may be the correct one, your users are still apt to be disappointed if the cluster doesn't live up to their expectations. Talk to your users.

You should also keep in mind that clusters have a way of evolving. What may be a reasonable assessment of needs today may not be tomorrow. Good design is often the art of balancing today's resources with tomorrow's needs. If you are unsure about your cluster's mission, answering the following questions should help.

In designing a cluster, you must take into consideration the needs of all users. Ideally this will include both the potential users as well as the obvious early adopters. You will need to anticipate any potential conflicting needs and find appropriate compromises.

The best way to avoid nasty surprises is to include representative users in the design process. If you have only a few users, you can easily poll the users to see what you need.

If you have a large user base, particularly one that is in flux, you will need to anticipate all reasonable, likely needs. Generally, this will mean supporting a wider range of software. For example, if you are the sole user and you only use one programming language and parallel programming library, there is no point in installing others. If you have dozens of users, you'll probably need to install multiple programming languages and parallel programming libraries.

Architecture and Cluster Software

Once you have established the mission for your cluster, you can focus on its architecture and select the software. Most high-performance clusters use an architecture similar to that shown in Figure 1-5. The software described in this book is generally compatible with that basic architecture. If this does not match the mission of your cluster, you still may be able to use many of the packages described in this book, but you may need to make a few adaptations.

Putting together a cluster involves the selection of a variety of software. The possibilities are described briefly here. Each is discussed in greater detail in subsequent chapters in this book.

One of the first selections you will probably want to make is the operating system, but this is actually the final software decision you should make. When selecting an operating system, the fundamental question is compatibility. If you have a compelling reason to use a particular piece of software and it will run only under a single operating system, the choice has been made for you. For example, openMosix uses extensions to the Linux kernel, so if you want openMosix, you must use Linux. Provided the basic issue of compatibility has been met, the primary reasons to select a particular operating system are familiarity and support. Stick with what you know and what's supported.

All the software described in this book is compatible with Linux. Most, but not all, of the software will also work nicely with other Unix systems. In this book, we'll be assuming the use of Linux. If you'd rather use BSD or Solaris, you'll probably be OK with most of the software, but be sure to check its compatibility before you make a commitment. Some of the software, such as MPICH, even works with Windows.

There is a natural human tendency to want to go with the latest available version of an operating system, and there are some obvious advantages to using the latest release. However, compatibility should drive this decision as well. Don't expect clustering software to be immediately compatible with the latest operating system release. Compatibility may require that you use an older release. (For more on Linux, see Chapter 4.)

Cluster Kits

If installing all of this software sounds daunting, don't panic. There are a couple of options you can consider. For permanent clusters there are, for lack of a better name, *cluster kits*, software packages that automate the installation process. A cluster kit provides all the software you are likely to need in a single distribution.

Cluster kits tend to be very complete. For example, the OSCAR distribution contains both PVM and two versions of MPI. If some software isn't included, you can probably get by without it. Another option, described in the next section, is a CD-ROM-based cluster.

Cluster kits are designed to be turnkey solutions. Short of purchasing a prebuilt, preinstalled proprietary cluster, a cluster kit is the simplest approach to setting up a full cluster. Configuration parameters are largely preset by people who are familiar with the software and how the different pieces may interact. Once you have installed the kit, you have a functioning cluster. You can focus on using the software rather than installing it. Support groups and mailing lists are generally available.

Some kits have a Linux distribution included in the package (e.g., Rocks), while others are installed on top of an existing Linux installation (e.g., OSCAR). Even if Linux must be installed first, most of the configuration and the installation of needed packages will be done for you.

There are two problems with using cluster kits. First, cluster kits do so much for you that you can lose touch with your cluster, particularly if everything is new to you. Initially, you may not understand how the cluster is configured, what customizations have been made or are possible, or even what has been installed. Even making minor changes after installing a kit can create problems if you don't understand what you have. Ironically, the more these kits do for you, the worse this problem may be. With a kit, you may get software you don't want to deal with—software your users may expect you to maintain and support. And when something goes wrong, as it will, you may be at a loss about how to deal with it.

CD-ROM-Based Clusters

If you just want to learn about clusters, only need a cluster occasionally, or can't permanently install a cluster, you might consider one of the CD-ROM-based clusters. With these, you create a set of bootable CD-ROMs, sometimes called "live filesystem" CDs. When you need the cluster, you reboot your available systems using the CD-ROMs, do a few configuration tasks, and start using your cluster. The cluster software is all available from the CD-ROM and the computers' hard disks are unchanged. When you are done, you simply remove the CD-ROM and reboot the system to return to the operating system installed on the hard disk. Your cluster persists until you reboot.

Clearly, this is not an approach to use for a high-availability or mission-critical cluster, but it is a way to get started and learn about clusters. It is a viable way to create a cluster for short-term use. For example, if a computer lab is otherwise idle over the weekend, you could do some serious calculations using this approach.

There are some significant difficulties with this approach, most notably problems with storage. It is possible to work around this problem by using a hybrid approach—setting up a dedicated system for storage and using the CD-ROM-based systems as compute-only nodes.

Several CD-ROM-based systems are available. You might look at ClusterKnoppix, **`http://bofh.be/clusterknoppix/`**, or Bootable Cluster CD (BCCD), **`http://bccd.cs.uni.edu/`**. The next subsection, a very brief description of BCCD, should give you the basic idea of how these systems work.

BCCD was developed by Paul Gray as an educational tool. If you want to play around with a small cluster, BCCD is a very straightforward way to get started. On an occasional basis, it is a viable alternative. What follows is a general overview of running BCCD for the first time.

The first step is to visit the BCCD download site, download an ISO image for a CD-ROM, and use it to burn a CD-ROM for each system. (Creating CD-ROMs from ISO images is briefly discussed in Chapter 4.) Next, boot each machine in your cluster from the CD-ROM. You'll need to answer a few questions as the system boots. First, you'll enter a password for the default user,

Benchmarks

Once you have your cluster running, you'll probably want to run a benchmark or two just to see how well it performs. Unfortunately, benchmarking is, at best, a dark art. In practice, sheep entrails may give better results.

Often the motivation for benchmarks is hubris—the desire to prove your system is the best. This can be crucial if funding is involved, but otherwise is probably a meaningless activity and a waste of time. You'll have to judge for yourself.

Keep in mind that a benchmark supplies a single set of numbers that is very difficult to interpret in isolation. Benchmarks are mostly useful when making comparisons between two or more closely related configurations on your own cluster.

There are at least three reasons you might run benchmarks. First, a benchmark will provide you with a baseline. If you make changes to your cluster or if you suspect problems with your cluster, you can rerun the benchmark to see if performance is really any different. Second, benchmarks are useful when comparing systems or cluster configurations. They can provide a reasonable basis for selecting between alternatives. Finally, benchmarks can be helpful with planning. If you can run several with differently sized clusters, etc., you should be able to make better estimates of the impact of scaling your cluster.

Benchmarks are not infallible. Consider the following rather simplistic example: Suppose you are comparing two clusters with the goal of estimating how well a particular cluster design scales. Cluster B is twice the size of cluster A. Your goal is to project the overall performance for a new cluster C, which is twice the size of B. If you rely on a simple linear extrapolation based on the overall performance of A and B, you could be grossly misled. For instance, if cluster A has a 30% network utilization and cluster B has a 60% network utilization, the network shouldn't have a telling impact on overall performance for either cluster. But if the trend continues, you'll have a difficult time meeting cluster C's need for 120% network utilization.

Chapter 3: **Cluster Hardware**

It is tempting to let the hardware dictate the architecture of your cluster. However, unless you are just playing around, you should let the potential uses of the cluster dictate its architecture. This in turn will determine, in large part, the hardware you use. At least, that is how it works in ideal, parallel universes.

In practice, there are often reasons why a less ideal approach might be necessary. Ultimately, most of them boil down to budgetary constraints. First-time clusters are often created from recycled equipment. After all, being able to use existing equipment is often the initial rationale for creating a cluster. Perhaps your cluster will need to serve more than one purpose. Maybe

you are just exploring the possibilities. In some cases, such as learning about clusters, selecting the hardware first won't matter too much.

If you are building a cluster using existing, cast-off computers and have a very limited budget, then your hardware selection has already been made for you. But even if this is the case, you will still need to make a number of decisions on how to use your hardware. On the other hand, if you are fortunate enough to have a realistic budget to buy new equipment or just some money to augment existing equipment, you should begin by carefully considering your goals. The aim of this chapter is to guide you through the basic hardware decisions and to remind you of issues you might overlook. For more detailed information on PC hardware, you might consult *PC Hardware in a Nutshell* (O'Reilly).

While you may have some idea of what you want, it is still worthwhile to review the implications of your choices. There are several closely related, overlapping key issues to consider when acquiring PCs for the nodes in your cluster:

- Will you have identical systems or a mixture of hardware?
- Will you scrounge for existing computers, buy assembled computers, or buy the parts and assemble your own computers?

Design Decisions

While you may have some idea of what you want, it is still worthwhile to review the implications of your choices. There are several closely related, overlapping key issues to consider when acquiring PCs for the nodes in your cluster:

- Will you have identical systems or a mixture of hardware?
- Will you scrounge for existing computers, buy assembled computers, or buy the parts and assemble your own computers?
- Will you have full systems with monitors, keyboards, and mice, minimal systems, or something in between?
- Will you have dedicated computers, or will you share your computers with other users?
- Do you have a broad or shallow user base?

This is this most important thing I'll say in this chapter—*if at all possible, use identical systems for your nodes*. Life will be much simpler. You'll need to develop and test only one configuration and then you can clone the remaining machines. When programming your cluster, you won't have to consider different hardware capabilities as you attempt to balance the workload among machines. Also, maintenance and repair will be easier since you will have

less to become familiar with and will need to keep fewer parts on hand. You can certainly use heterogeneous hardware, but it will be more work.

In constructing a cluster, you can scrounge for existing computers, buy assembled computers, or buy the parts and assemble your own. Scrounging is the cheapest way to go, but this approach is often the most time consuming. Usually, using scrounged systems means you'll end up with a wide variety of hardware, which creates both hardware and software problems. With older scrounged systems, you are also more likely to have even more hardware problems. If this is your only option, try to standardize hardware as much as possible. Look around for folks doing bulk upgrades when acquiring computers. If you can find someone replacing a number of computers at one time, there is a good chance the computers being replaced will have been a similar bulk purchase and will be very similar or identical. These could come from a computer laboratory at a college or university or from an IT department doing a periodic upgrade.

Environment

You are going to need some place to put your computers. If you are lucky enough to have a dedicated machine room, then you probably have everything you need. Otherwise, select or prepare a location that provides physical security, adequate power, and adequate heating and cooling. While these might not be issues with a small cluster, proper planning and preparation is essential for large clusters. Keep in mind, you are probably going to be so happy with your cluster that you'll want to expand it. Since small clusters have ways of becoming large clusters, plan for growth from the start.

Since the more computers you have, the more space they will need, plan your layout with wiring, cooling, and physical access in mind. Ignore any of these at your peril. While it may be tempting to stack computers or pack them into large shelves, this can create a lot of problems if not handled with care. First, you may find it difficult to physically access individual computers to make repairs. If the computers are packed too tightly, you'll create heat dissipation problems. And while this may appear to make wiring easier, in practice it can lead to a rat's nest of cables, making it difficult to divide your computers among different power circuits.

From the perspective of maintenance, you'll want to have physical access to individual computers without having to move other computers and with a minimum of physical labor. Ideally, you should have easy access to both the front and back of your computers. If your nodes are headless (no monitor, mouse, or keyboard), it is a good idea to assemble a crash cart. So be sure to leave enough space to both wheel and park your crash cart (and a chair) among your machines.

To prevent overheating, leave a small gap between computers and take care not to obstruct any ventilation openings. (These are occasionally seen on the sides of older computers!) An

inch or two usually provides enough space between computers, but watch for signs of overheating.

Chapter 4: **Linux for Clusters**

This chapter reviews some of the issues involved in setting up a Linux system for use in a cluster. While several key services are described in detail, for the most part the focus is more on the issues and rationales than on specifics. Even if you are an old pro at Linux system administration, you may still want to skim this chapter for a quick overview of the issues as they relate to clusters, particularly the section on configuring services. If you are new to Linux system administration, this chapter will probably seem very terse. What's presented here is the bare minimum a novice system administrator will need to get started. The Appendix A lists additional sources.

This chapter covers material you'll need when setting up the head node and a typical cluster node. Depending on the approach you take, much of this may be done for you. If you are building your cluster from the ground up, you'll need to install the head node, configure the individual services on it, and build at least one compute node. Once you have determined how a compute node should be configured, you can turn to Chapter 8 for a discussion of how to duplicate systems in an efficient manner. It is much simpler with kits like OSCAR and Rocks.

With OSCAR, you'll need to install Linux on the head system, but OSCAR will configure the services for you. It will also build the client, i.e., generate a system image and install it on the compute nodes. OSCAR will configure and install most of the packages you'll need. The key to using OSCAR is to use a version of Linux that is known to be compatible with OSCAR. OSCAR is described in Chapter 6. With Rocks, described in Chapter 7, everything will be done for you. Red Hat Linux comes as part of the Rocks distribution.

This chapter begins with a discussion of selecting a Linux distribution. A general discussion of installing Linux follows. Next, the configuration of relevant network services is described. Finally, there is a brief discussion of security. If you are adding clustering software to an existing collection of workstations, presumably Linux is already installed on your machines. If this is the case, you can probably skim the first couple of sections. But while you won't need to install Linux, you will need to ensure that it is configured correctly and all the services you'll need are available.

Installing Linux

If Linux isn't built into your cluster software, the first step is to decide what distribution and version of Linux you want.

This decision will depend on what clustering software you want to use. It doesn't matter what the "best" distribution of Linux (Red Hat, Debian, SUSE, Mandrake, etc.) or version (7.3, 8.0,

9.0, etc.) is in some philosophical sense if the clustering software you want to use isn't available for that choice. This book uses the Red Hat distribution because the clustering software being discussed was known to work with that distribution. This is not an endorsement of Red Hat; it was just a pragmatic decision.

Keep in mind that your users typically won't be logging onto the compute nodes to develop programs, etc., so the version of Linux used there should be largely irrelevant to the users. While users will be logging onto the head node, this is not a general-purpose server. They won't be reading email, writing memos, or playing games on this system (hopefully). Consequently, many of the reasons someone might prefer a particular distribution are irrelevant.

This same pragmatism should extend to selecting the version as well as the distribution you use. In practice, this may mean using an older version of Linux. There are basically three issues involved in using an older version—compatibility with newer hardware; bug fixes, patches, and continued support; and compatibility with clustering software.

If you are using recycled hardware, using an older version shouldn't be a problem since drivers should be readily available for your older equipment. If you are using new equipment, however, you may run into problems with older Linux releases. The best solution, of course, is to avoid this problem by planning ahead if you are buying new hardware. This is something you should be able to work around by putting together a single test system before buying the bulk of the equipment.

With older versions, many of the problems are known. For bugs, this is good news since someone else is likely to have already developed a fix or workaround. With security holes, this is bad news since exploits are probably well circulated. With an older version, you'll need to review and install all appropriate security patches. If you can isolate your cluster, this will be less of an issue.

Configuring Services

Once you have the basic installation completed, you'll need to configure the system. Many of the tasks are no different for machines in a cluster than for any other system. For other tasks, being part of a cluster impacts what needs to be done. The following subsections describe the issues associated with several services that require special considerations. These subsections briefly recap how to configure and use these services. Remember, most of this will be done for you if you are using a package like OSCAR or Rocks. Still, it helps to understand the issues and some of the basics.

Dynamic Host Configuration Protocol (DHCP) is used to supply network configuration parameters, including IP addresses, host names, and other information to clients as they boot. With clusters, the head node is often configured as a DHCP server and the compute nodes as

DHCP clients. There are two reasons to do this. First, it simplifies the installation of compute nodes since the information DHCP can supply is often the only thing that is different among the nodes. Since a DHCP server can handle these differences, the node installation can be standardized and automated. A second advantage of DHCP is that it is much easier to change the configuration of the network. You simply change the configuration file on the DHCP server, restart the server, and reboot each of the compute nodes.

The basic installation is rarely a problem. The DHCP system can be installed as a part of the initial Linux installation or after Linux has been installed. The DHCP server configuration file, typically `/etc/dhcpd.conf`, controls the information distributed to the clients. If you are going to have problems, the configuration file is the most likely source.

The DHCP configuration file may be created or changed automatically when some cluster software is installed. Occasionally, the changes may not be done optimally or even correctly so you should have at least a reading knowledge of DHCP configuration files. Here is a heavily commented sample configuration file that illustrates the basics. (Lines starting with "#" are comments.)

Cluster Security

Security is always a two-edged sword. Adding security always complicates the configuration of your systems and makes using a cluster more difficult. But if you don't have adequate security, you run the risk of losing sensitive data, losing control of your cluster, having it damaged, or even having to completely rebuild it. Security management is a balancing act, one of trying to figure out just how little security you can get by with.

As previously noted, the usual architecture for a cluster is a set of machines on a dedicated subnet. One machine, the head node, connects this network to the outside world, i.e., the organization's network and the Internet. The only access to the cluster's dedicated subnet is through the head node. None of the compute nodes are attached to any other network. With this model, security typically lies with the head node. The subnet is usually a trust-based open network.

There are several reasons for this approach. With most clusters, the communication network is the bottleneck. Adding layers of security to this network will adversely affect performance. By focusing on the head node, security administration is localized and thus simpler. Typically, with most clusters, any sensitive information resides on the head node, so it is the point where the greatest level of protection is needed. If the compute nodes are not isolated, each one will need to be secured from attack.

This approach also simplifies setting up packet filtering, i.e., firewalls. Incorrectly configured, packet filters can create havoc within your cluster. Determining what traffic to allow can be a formidable challenge when using a number of different applications. With the isolated

network approach, you can configure the internal interface to allow all traffic and apply the packet filter only to public interface.

This approach doesn't mean you have a license to be sloppy within the cluster. You should take all reasonable precautions. Remember that you need to protect the cluster not just from external threats but from internal ones as well—whether intentional or otherwise.

Chapter 5: **openMosix**

`openMosix` is software that extends the Linux kernel so that processes can migrate transparently among the different machines within a cluster in order to more evenly distribute the workload. This chapter gives the basics of setting up and using an openMosix cluster. There is a lot more to openMosix than described here, but this should be enough to get you started and keep you running for a while unless you have some very special needs.

Basically, the openMosix software includes both a set of kernel patches and support tools. The patches extend the kernel to provide support for moving processes among machines in the cluster. Typically, process migration is totally transparent to the user. However, by using the tools provided with openMosix, as well as third-party tools, you can control the migration of processes among machines.

Let's look at how openMosix might be used to speed up a set of computationally expensive tasks. Suppose, for example, you have a dozen files to compress using a CPU-intensive program on a machine that isn't part of an openMosix cluster. You could compress each file one at a time, waiting for one to finish before starting the next. Or you could run all the compressions simultaneously by starting each compression in a separate window or by running each compression in the background (ending each command line with an **&**). Of course, either way will take about the same amount of time and will load down your computer while the programs are running.

However, if your computer is part of an openMosix cluster, here's what will happen: First, you will start all of the processes running on your computer. With an openMosix cluster, after a few seconds, processes will start to migrate from your heavily loaded computer to other idle or less loaded computers in the clusters. (As explained later, because some jobs may finish quickly, it can be counterproductive to migrate too quickly.) If you have a dozen idle machines in the cluster, each compression should run on a different machine. Your machine will have only one compression running on it (along with a little added overhead) so you still may be able to use it. And the dozen compressions will take only a little longer than it would normally take to do a single compression.

What Is openMosix?

Basically, the openMosix software includes both a set of kernel patches and support tools. The patches extend the kernel to provide support for moving processes among machines in the cluster. Typically, process migration is totally transparent to the user. However, by using the tools provided with openMosix, as well as third-party tools, you can control the migration of processes among machines.

Let's look at how openMosix might be used to speed up a set of computationally expensive tasks. Suppose, for example, you have a dozen files to compress using a CPU-intensive program on a machine that isn't part of an openMosix cluster. You could compress each file one at a time, waiting for one to finish before starting the next. Or you could run all the compressions simultaneously by starting each compression in a separate window or by running each compression in the background (ending each command line with an **&**). Of course, either way will take about the same amount of time and will load down your computer while the programs are running.

However, if your computer is part of an openMosix cluster, here's what will happen: First, you will start all of the processes running on your computer. With an openMosix cluster, after a few seconds, processes will start to migrate from your heavily loaded computer to other idle or less loaded computers in the clusters. (As explained later, because some jobs may finish quickly, it can be counterproductive to migrate too quickly.) If you have a dozen idle machines in the cluster, each compression should run on a different machine. Your machine will have only one compression running on it (along with a little added overhead) so you still may be able to use it. And the dozen compressions will take only a little longer than it would normally take to do a single compression.

If you don't have a dozen computers, or some of your computers are slower than others, or some are otherwise loaded, openMosix will move the jobs around as best it can to balance the load. Once the cluster is set up, this is all done transparently by the system. Normally, you just start your jobs. openMosix does the rest. On the other hand, if you want to control the migration of jobs from one computer to the next, openMosix

How openMosix Works

openMosix originated as a fork from the earlier `MOSIX` (`Multicomputer Operating System for Unix`) project. The openMosix project began when the licensing structure for MOSIX moved away from a General Public License. Today, it has evolved into a project in its own right. The original MOSIX project is still quite active under the direction of Amnon Barak (**`http://www.mosix.org`**). openMosix is the work of Moshe Bar, originally a member of the MOSIX team, and a number of volunteers. This book focuses on openMosix, but MOSIX is a viable alternative that can be downloaded at no cost.

As noted in Chapter 1, one approach to sharing a computation between processors in a single-enclosure computer with multiple CPUs is `symmetric multiprocessor` (`SMP`) computing. openMosix has been described, accurately, as turning a cluster of computers into a

virtual SMP machine, with each node providing a CPU. openMosix is potentially much cheaper and scales much better than SMPs, but communication overhead is higher. (openMosix will work with both single-processor systems and SMP systems.) openMosix is an example of what is sometimes called `single system image clustering (SSI)` since each node in the cluster has a copy of a single operating system kernel.

The granularity for openMosix is the process. Individual programs, as in the compression example, may create the processes, or the processes may be the result of different forks from a single program. However, if you have a computationally intensive task that does everything in a single process (and even if multiple threads are used), then, since there is only one process, it can't be shared among processors. The best you can hope for is that it will migrate to the fastest available machine in the cluster.

Not all processes migrate. For example, if a process only lasts a few seconds (very roughly, less than 5 seconds depending on a number of factors), it will not have time to migrate. Currently, openMosix does not work with multiple processes using shared writable memory, such as web servers. Similarly, processes doing direct manipulation of I/O devices won't migrate. And processes using real-time scheduling won't migrate. If a process has already migrated to another processor and attempts to do any these things, the process will migrate back to its

Selecting an Installation Approach

Since openMosix is a kernel extension, it won't work with just any kernel. At this time, you are limited to a relatively recent (at least version 2.4.17 or more recent) IA32-compatible Linux kernel. An IA64 port is also available. However, don't expect openMosix to be available for a new kernel the same day a new kernel is released. It takes time to develop patches for a kernel. Fortunately, your choice of Linux distributions is fairly broad. Among others, openMosix has been reported to work on Debian, Gentoo, Red Hat, and SuSe Linux. If you just want to play with it, you might consider Bootable Cluster CD (BCCD), Knoppix, or PlumpOS, three CD-bootable Linux distributions that include openMosix. You'll also need a reasonably fast network and a fair amount of swap space to run openMosix.

To build your openMosix cluster, you need to install an openMosix extended kernel on each of the nodes in the cluster. If you are using a suitable version of Linux and have no other special needs, you may be able to download a precompiled version of the kernel. This will significantly simplify setup. Otherwise, you'll need to obtain a clean copy of the kernel sources, apply the openMosix patches to the kernel source code, recompile the sources, and install the patched kernel. This isn't as difficult as it might sound, but it is certainly more involved than just installing a precompiled kernel. Recompiling the kernel is described in detail later in this chapter. We'll start with precompiled kernels.

While using a precompiled kernel is the easiest way to go, it has a few limitations. The documentation is a little weak with the precompiled kernels, so you won't know exactly what

options have been compiled into the kernel without doing some digging. (However, the `.config` files are available via CVS and the options seem to be reasonable.) If you already have special needs that required recompiling your kernel, e.g., nonstandard hardware, don't expect those needs to go away.

You'll need to use the same version of the patched kernel on all your systems, so choose accordingly. This doesn't mean you must use the same kernel image. For example, you can use different compiles to support different hardware. But all your kernels should have the same version number.

Installing a Precompiled Kernel

The basic steps for installing a precompiled kernel are selecting and downloading the appropriate files and packages, installing those packages, and making a few minor configuration changes.

You'll find links to available packages at **`http://openmosix.sourceforge.net`**. You'll need to select from among several versions and compilations. At the time this was written, there were half a dozen different kernel versions available. For each of these, there were eight possible downloads, including a README file, a kernel patch file, a source file that contains both a clean copy of the kernel and the patches, and five precompiled kernels for different processors. The precompiled versions are for an Intel 386 processor, an Intel 686 processor, an Athlon processor, Intel 686 SMP processors, or Athlon SMP processors. The Intel 386 is said to be the safest version. The Intel 686 version is for Intel Pentium II and later CPUs. With the exception of the text README file and a compressed (`gz`) set of patches, the files are in RPM format.

The example that follows uses the package `openmosix-kernel-2.4.24-openmosix.i686.rpm` for a single processor Pentium II system running Red Hat 9. Be sure you read the README file! While you are at it, you should also download a copy of the latest suitable version of the `openMosix` user tools from the same site. Again, you'll have a number of choices. You can download binaries in RPM or DEB format as well as the sources. For this example, the file `openmosix-tools-0.3.5-1.i386.rpm` was used.

Perhaps the easiest thing to do is to download everything at once and burn it to a CD so you'll have everything handy as you move from machine to machine. But you could use any of the techniques described in Chapter 8, or you could use the C3 tools described in Chapter 10. Whatever your preference, you'll need to get copies of these files on each machine in your cluster.

There is one last thing to do before you install—create an emergency boot disk if you don't have one. While it is unlikely that you'll run into any problems with

Using openMosix

At its simplest, openMosix is transparent to the user. You can sit back and reap the benefits. But at times, you'll want more control. At the very least, you may want to verify that it is really running properly. (You could just time applications with computers turned on and off, but you'll probably want to be a little more sophisticated than that.) Fortunately, openMosix provides some tools that allow you to monitor and control various jobs. If you don't like the tools that come with openMosix, you can always install other tools such as openMosixView.

You should install the openMosix user tools before you start running openMosix. This package includes several useful management tools (`migrate`, `mosctl`, `mosmon`, `mosrun`, and `setpe`), an openMosix aware version of `ps` and `top` called, suitably, `mps` and `mtop`, and a startup script `/etc/init.d/openmosix` . (This is actually a link to the file `/etc/rc.d/init.d/openmosix`.)

Section 5.5.1.1: mps and mtop

Both `mps` and `mtop` will look a lot like their counterparts, `ps` and `top`. The major difference is that each has an additional column that gives the node number on which a process is running. Here is part of the output from `mps`:

```
[root@fanny sloanjd]# mps
  PID TTY NODE STAT TIME COMMAND
...
19766  ?     0 R    2:32 ./loop
19767  ?     2 S    1:45 ./loop
19768  ?     5 S    3:09 ./loop
19769  ?     4 S    2:58 ./loop
19770  ?     2 S    1:47 ./loop
19771  ?     3 S    2:59 ./loop
19772  ?     6 S    1:43 ./loop
19773  ?     0 R    1:59 ./loop
...
```

As you can see from the third column, process 19769 is running on node 4. It is important to note that `mps` must be run on the machine where the process originated. You will not see the process if you run `ps`, `mps`, `top`, or `mtop` on any of the other machines in the cluster even if the process has migrated to that machine. (Arguably, in this respect, openMosix is perhaps a little too transparent. Fortunately, a couple of the other tools help.)

Recompiling the Kernel

First, ask yourself why you would want to recompile the kernel. There are several valid reasons. If you normally have to recompile your kernel, perhaps because you use less-common hardware or need some special compile option, then you'll definitely need to recompile for openMosix. Or maybe you just like tinkering with things. If you have a reason, go for it. Even if you have never done it before, it is not that difficult, but the precompiled kernels do work well. For most readers, recompiling the kernel is optional, not mandatory. (If you are not interested in recompiling the kernel, you can skip the rest of this section.)

Before you start, do you have a recovery disk? Are you sure you can boot from it? If not, go make one right now before you begin.

Let's begin by going over the basic steps of a fairly generic recompilation, and then we'll go through an example. First, you'll need to decide which version of the kernel you want to use. Check to see what is available. (You can use the `uname -r` command to see what you are currently using, but you don't have to feel bound by that.)

You are going to need both a set of patches and a clean set of kernel source files. Accepted wisdom says that you shouldn't use the source files that come with any specific Linux releases because, as a result of customizations, the patches will not apply properly. As noted earlier in this chapter, you can download the kernel sources and patches from **`http://openmosix.sourceforge.net`** or you can just download the patches. If you have downloaded just the patches, you can go to **`http://www.kernel.org`** to get the sources. You'll end up with the same source files either way.

If you download the source file from the openMosix web site, you'll have an RPM package to install. When you install this, it will place compressed copies of the patches and the source tree (in `gzip` or `bzip2` format) as well as several sample kernel configuration files in the directory `/usr/src/redhat/SOURCES`

Is openMosix Right for You?

openMosix has a lot to recommend it. Not having to change your application code is probably the biggest advantage. As a control mechanism, it provides both transparency to the casual user and a high degree of control for the more experienced user. With precompiled kernels, setup is very straightforward and goes quickly.

There is a fair amount of communication overhead with openMosix, so it works best on high-performance networks, but that is true of any cluster. It is also more operating system-specific than most approaches to distributed computing. For a high degree of control for highly parallel code, MPI is probably a better choice. This is particularly true if latency becomes an issue. But you should not overlook the advantages of using both MPI and openMosix. At the very least, openMosix may improve performance by migrating processes to less-loaded nodes.

There are a couple of other limitations to openMosix that are almost unfair to mention since they are really outside the scope of the openMosix project. The first is the inherit granularity attached to process migration. If your calculation doesn't fork off processes, much of the advantage of openMosix is lost. The second limitation is a lack of scheduling control. Basically, openMosix deals with processes as it encounters them. It is up to the user to manage scheduling or just take what comes. Keep in mind that if you are using a scheduling program to get very tight control over your resources, openMosix may compete with your scheduler in unexpected ways.

In looking at openMosix, remember that it is a product of an ongoing and very active research project. Any description of openMosix is likely to become dated very quickly. By the time you have read this, it is likely that openMosix will have evolved beyond what has been described here. This is bad news for writers like me, but great news for users. Be sure to consult the openMosix documentation.

If you need to run a number of similar applications simultaneously and need to balance the load among a group of computers, you should consider openMosix.

Chapter 6: **OSCAR**

Setting up a cluster can involve the installation and configuration of a lot of software as well as reconfiguration of the system and previously installed software. `OSCAR (Open Source Cluster Application Resources)` is a software package that is designed to simplify cluster installation. A collection of open source cluster software, OSCAR includes everything that you are likely to need for a dedicated, high-performance cluster. OSCAR takes you completely through the installation of your cluster. If you download, install, and run OSCAR, you will have a completely functioning cluster when you are done.

This chapter begins with an overview of why you might use OSCAR, followed by a description of what is included in OSCAR. Next, the discussion turns to the installation and configuration of OSCAR. This includes a description of how to customize OSCAR and the changes OSCAR makes to your system. Finally, there are three brief sections, one on cluster security, one on `switcher`, and another on using OSCAR with LAM/MPI.

Because OSCAR is an extensive collection of software, it is beyond the scope of this book to cover every package in detail. Most of the software in OSCAR is available as standalone versions, and many of the key packages included by OSCAR are described in later chapters in this book. Consequently, this chapter focuses on setting up OSCAR and on software unique to OSCAR. By the time you have finished this chapter, you should be able to judge whether OSCAR is appropriate for your needs and know how to get started.

The design goals for OSCAR include using the best-of-class software, eliminating the downloading, installation, and configuration of individual components, and moving toward the standardization of clusters. OSCAR, it is said, reduces the need for expertise in setting up a cluster. In practice, it might be more fitting to say that OSCAR delays the need for expertise and allows you to create a fully functional cluster before mastering all the skills you will eventually need. In the long run, you will want to master those packages in OSCAR that you come to rely on. OSCAR makes it very easy to experiment with packages and dramatically lowers the barrier to getting started.

Why OSCAR?

The design goals for OSCAR include using the best-of-class software, eliminating the downloading, installation, and configuration of individual components, and moving toward the standardization of clusters. OSCAR, it is said, reduces the need for expertise in setting up a cluster. In practice, it might be more fitting to say that OSCAR delays the need for expertise and allows you to create a fully functional cluster before mastering all the skills you will eventually need. In the long run, you will want to master those packages in OSCAR that you come to rely on. OSCAR makes it very easy to experiment with packages and dramatically lowers the barrier to getting started.

OSCAR was created and is maintained by the Open Cluster Group (**http://www.openclustergroup.org**), an informal group dedicated to simplifying the installation and use of clusters and broadening their use. Over the years, a number of organizations and companies have supported the Open Cluster Group, including Dell, IBM, Intel, NCSA, and ORNL, to mention only a few.

OSCAR is designed with high-performance computing in mind. Basically, it is designed to be used with an asymmetric cluster (see Chapter 1). Unless you customize the installation, the computer nodes are meant to be dedicated to the cluster. Typically, you do not log directly onto the client nodes but rather work from the head node. (Although OSCAR sets up SSH so that you can log onto clients without a password, this is done primarily to simplify using the cluster software.)

While identical hardware isn't an absolute requirement, installing and managing an OSCAR cluster is much simpler when identical hardware is used.

Actually, OSCAR could be used for any cluster application—not just high-performance computing. (A recently created subgroup, *HA-OSCAR*, is starting to look into high-availability clusters.) While OSCAR installs a number of packages specific to high-performance computing by default which would be of little use for some other cluster uses, e.g., MPI and PVM, it is easy to skip the installation of these packages. It is very easy to include additional RPM packages to an OSCAR installation. Although OSCAR does not provide a simple mechanism to do a post-installation configuration for such packages, you can certainly include configuration scripts if you create your own packages. There is a HOWTO on the OSCAR web site that describes how to create custom packages. Generally, this will be easier than manually configuring added packages after the installation. (However, by using the C3 tool set included in OSCAR, many post-install configuration tasks shouldn't be too difficult.)

What's in OSCAR

OSCAR brings together a number of software packages for clustering. Most of the packages listed in this section are available as standalone packages and have been briefly described in Chapter 2. Some of the more important packages are described in detail in later chapters as

well. However, there are several scripts unique to OSCAR. Most are briefly described in this chapter.

It is likely that everything you really need to get started with a high-performance cluster is included either in the OSCAR tar-ball or as part of the base operating system OSCAR is installed under. Nonetheless, OSCAR provides a script, the `Oscar Package Downloader (opd)` that simplifies the download and installation of additional packages that are available from OSCAR repositories in an OSCAR-compatible format. `opd` is so easy to use that for practical purposes any package available through `opd` can be considered part of OSCAR. `opd` can be invoked as a standalone program or from the OSCAR `installation wizard`, the GUI-based OSCAR installer. Additional packages available using `opd` include things like Myrinet drivers and support for thin OSCAR clients, as well as management packages like Ganglia. Use of `opd` is described later in this chapter.

OSCAR packages fall into three categories. Core packages must be installed. Included packages are distributed as part of OSCAR, but you can opt out on installing these packages. Third-party packages are additional packages that are available for download and are compatible with OSCAR, but aren't required. There are six core packages at the heart of OSCAR that you must install:

Core

This is the core OSCAR package.

C3

The Cluster, Command, and Control tool suite provides a command-line administration interface (described in Chapter 10).

Installing OSCAR

This section should provide you with a fairly complete overview of the installation process. The goal here is to take you through a typical installation and to clarify a few potential problems you might encounter. Some customizations you might want to consider are described briefly at the end of this section. The OSCAR project provides a very detailed set of installation instructions running over 60 pages, which includes a full screen-by-screen walkthrough. If you decide OSCAR is right for you, you should download the latest version and read it very carefully before you begin. It will be more current and complete than the overview provided here. Go to **`http://oscar.openclustergroup.org`** and follow the documentation link.

Because OSCAR is a complex set of software that includes a large number of programs and services, it can be very unforgiving if you make mistakes when setting it up. For some errors, you may be able to restart the installation process. For others, you will be better served by starting again from scratch. A standard installation, however, should not be a problem. If you

have a small cluster and the hardware is ready to go, with a little practice you can be up and running in less than a day.

The installation described here is typical. Keep in mind, however, that your installation may not go exactly like the one described here. It will depend on some of the decisions you make. For example, if you select to install PVFS, you'll see an additional console window early in the installation specific to that software.

There are several things you need to do before you install OSCAR. First, you need to plan your system. Figure 6-1 shows the basic architecture of an OSCAR cluster. You first install OSCAR on the cluster's head node or server, and then OSCAR installs the remaining machines, or clients, from the server. The client image is a disk image for the client that includes the boot sector, operating system, and other software for the client. Since the head node is used to build the client image, is the home for most user services, and is used to administer the cluster, you'll need a well-provisioned machine. In particular, don't try to skimp on disk space—OSCAR uses a lot. The installation guide states that after you have installed the system, you will need at least 2 GB (each) of free space under both the

Security and OSCAR

OSCAR uses a layered approach to security. The architecture used in this chapter, a single-server node as the only connection to the external network, implies that everything must go through the server. If you can control the placement of the server on the external network, e.g., behind a corporate firewall, you can minimize the threat to the cluster. While outside the scope of this discussion, this is something you should definitely investigate.

The usual advice for securing a server applies to an OSCAR server. For example, you should disable unneeded services and delete unused accounts. With a Red Hat installation, `TCP wrappers` is compiled into `xinetd` and available by default. You'll need to edit the `/etc/hosts.allow` and `/etc/hosts.deny` files to configure this correctly. There are a number of good books (and web pages) on security. Get one and read it!

In an OSCAR cluster, access to the cluster is controlled through `pfilter`, a package included in the OSCAR distribution. `pfilter` is both a firewall and a compiler for firewall rulesets. (The `pfilter` software can be downloaded separately from **`http://pfilter.sourceforge.net/`**.)

`pfilter` is run as a service, which makes it easy to start it, stop it, or check its status.

```
[root@amy root]# service pfilter stop
Stopping pfilter:                                          [
OK  ]
[root@amy root]# service pfilter start
```

```
Starting pfilter:                                          [
OK  ]
[root@amy root]# service pfilter status
pfilter is running
```

If you are having communications problems between nodes, you may want to temporarily disable `pfilter`. Just don't forget to restart it when you are done!

You can request a list of the chains or rules used by `pfilter` with the `service` command.

```
[root@amy root]# service pfilter chains

table filter:
...
```

This produces a lot of output that is not included here.

Using switcher

`switcher` is a script that simplifies changes to a user's environment. It allows the user to make, with a single command, all the changes to paths and environmental variables needed to run an application. `switcher` is a script that uses the `modules` package.

The `modules` package is an interesting package in its own right. It is a general utility that allows users to dynamically modify their environment using `modulefiles`. Each `modulefile` contains the information required to configure a shell for a specific application. A user can easily switch to another application, making required environmental changes with a single command. While it is not necessary to know anything about `modules` to use `switcher`, OSCAR installs the modules system and, it is available should you need or wish to use it. `modules` can be downloaded from **`http://modules.sourceforge.net/`**.

`switcher` is designed so that changes take effect on future shells, not the current one. This was a conscious design decision. The disadvantage is that you will need to start a new shell to see the benefits of your change. On the positive side, you will not need to run `switcher` each time you log in. Nor will you need to edit your "dot" files such as *.bashrc*. You can make your changes once and forget about them. While `switcher` is currently used to change between the two MPI environments provided with OSCAR, it provides a general mechanism that can be used for other tasks. When experimenting with `switcher`, it is a good idea to create a new shell and test changes before closing the old shell. If you have problems, you can go back to the old shell and correct them.

With *switcher*, tags are used to group similar software packages. For example, OSCAR uses the tag **`mpi`** for the included MPI systems. (You can list all available tags by invoking `switcher` with just the **`--list`** option.) You can easily list the attributes associated with a tag.

```
[sloanjd@amy sloanjd]$ switcher mpi --list
```

Using LAM/MPI with OSCAR

Before we leave OSCAR, let's look at a programming example. You can use this to convince yourself that everything is really working. You can find several LAM/MPI examples in `/usr/share/doc/lam-oscar-7.0/examples` and the documentation in `/opt/lam-7.0/share/lam/doc`. (For MPICH, look in `/opt/mpich-1.2.5.10-ch_p4-gcc/examples` for code and `/opt/mpich-1.2.5.10-ch_p4-gcc/doc` for documentation.**)**

Log on as a user other than root and verify that LAM/MPI is selected using switcher.

```
[sloanjd@amy doc]$ switcher mpi --show
user:default=lam-7.0
system:exists=true
```

If necessary, change this and log off and back on.

If you haven't logged onto the individual machines, you need to do so now using `ssh` to register each machine with `ssh`. You could do this with a separate command for each machine.

```
[sloanjd@amy sloanjd]$ ssh node1
...
```

Using a shell looping command is probably better since it will ensure that you don't skip any machines and can reduce typing. With the Bash shell, the following command will initiate your logon to the machines `node1` through `node99`, each in turn.

```
[sloanjd@amy sloanjd]$ for ((i=1; i<100; i++))
> do
>   ssh node${i}
> done
```

Just adjust the loop for a different number of machines. You will need to adjust the syntax accordingly for other shells. This goes fairly quickly and you'll need to do this only once.

Create a file that lists the individual machines in the cluster by IP address. For example, you might create a file called `myhosts` like the following:

```
[sloanjd@amy sloanjd]$ cat myhosts
172.16.1.1
172.16.1.2
172.16.1.3
172.16.1.4
172.16.1.5
```

This should contain the server as well as the clients.

Next, run `lamboot` with the file's name as an argument.

```
[sloanjd@amy sloanjd]$ lamboot myhosts

LAM 7.0/MPI 2 C++/ROMIO - Indiana University
```

You now have a LAM/MPI daemon running on each machine in your cluster.

Chapter 7: **Rocks**

The previous chapter showed the use of OSCAR to coordinate the many activities that go into setting up and administering a cluster. This chapter discusses another popular kit for accomplishing roughly the same tasks.

NPACI Rocks is a collection of open source software for building a high-performance cluster. The primary design goal for Rocks is to make cluster installation as easy as possible. Unquestionably, they have gone a long way toward meeting this goal. To accomplish this, the default installation makes a number of reasonable assumptions about what software should be included and how the cluster should be configured. Nonetheless, with a little more work, it is possible to customize many aspects of Rocks.

When you install Rocks, you will install both the clustering software and a current version of Red Hat Linux updated to include security patches. The Rocks installation will correctly configure various services, so this is one less thing to worry about. Installing Rocks installs Red Hat Linux, so you won't be able to add Rocks to an existing server or use it with some other Linux distribution.

Default installations tend to go very quickly and very smoothly. In fact, Rocks' management strategy assumes that you will deal with software problems on a node by reinstalling the system on that node rather than trying to diagnose and fix the problem. Depending on hardware, it may be possible to reinstall a node in under 10 minutes. Even if your systems

take longer, after you start the reinstall, everything is automatic, so you don't need to hang around.

In this chapter, we'll look briefly at how to build and use a Rocks cluster. This coverage should provide you with enough information to decide whether Rocks is right for you. If you decide to install Rocks, be sure you download and read the current documentation. You might also want to visit Steven Baum's site, **`http://stommel.tamu.edu/~baum/npaci.html`**.

In this section we'll look at a default Rocks installation. We won't go into the same level of detail as we did with OSCAR, in part because Rocks offers a simpler installation. This section should give you the basics.

Installing Rocks

In this section we'll look at a default Rocks installation. We won't go into the same level of detail as we did with OSCAR, in part because Rocks offers a simpler installation. This section should give you the basics.

There are several things you need to do before you begin your installation. First, you need to plan your system. A Rocks cluster has the same basic architecture as an OSCAR cluster (see Figure 6-1). The head node or `frontend` is a server with two network interfaces. The *public* interface is attached to the campus network or the Internet while the *private* interface is attached to the cluster. With Rocks, the first interface (e.g., `eth0`) is the private interface and the second (e.g., `eth1`) is the public interface. (This is the opposite of what was described for OSCAR.)

You'll install the frontend first and then use it to install the compute nodes. The compute nodes use HTTP to pull the Red Hat and cluster packages from the front-end. Because Rocks uses Kickstart and Anaconda (described in Chapter 8), heterogeneous hardware is supported.

Diskless clusters are not an option with Rocks. It assumes you will have hard disks in all your nodes. For a default installation, you'll want at least an 8 GB disk on the frontend. For compute nodes, by altering the defaults, you can get by with smaller drives. It is probably easier to install the software on the compute nodes by booting from a CD-ROM, but if your systems don't have CD-ROM drives, you can install the software by booting from a floppy or by doing a network boot. Compute nodes should be configured to boot without an attached keyboard or should have a keyboard or KVM switch attached.

Rocks supports both Ethernet and Myrinet. For the cluster's private network, use a private address space distinct from the external address space per RFC 1918. It's OK to let an external DHCP server configure the public interface, but you should let Rocks configure the private interface.

Managing Rocks

One of Rocks' strengths is the web-based management tools it provides. Initially, these are available only from within the clusters since the default firewall configuration blocks HTTP connections to the frontend's public interface. If you want to allow external access, you'll need to change the firewall configuration. To allow access over the public interface, edit the file `/etc/sysconfig/iptables` and uncomment the line:

```
-A INPUT -i eth1 -p tcp -m tcp --dport www -j ACCEPT
```

Then restart the `iptables` service.

```
[root@frontend sysconfig]# service iptables restart
```

Some pages, for security reasons, will still be unreachable.

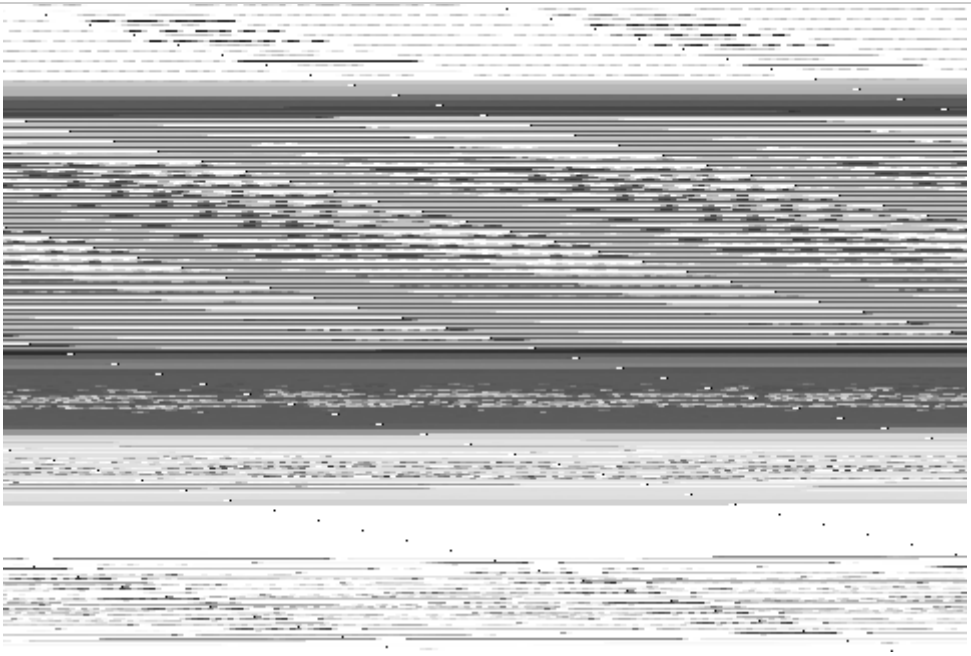
To view the management page locally, log onto the frontend, start the X Window System, start your browser, and go to *http://localhost*. You should get a screen that looks something like Figure 7-1.

Figure 7-1: Rocks' web interface

The links on the page will vary depending on the software or rolls you chose to install. For example, if you didn't install PBS, you won't see a link to the PBS Job Queue. Here is a brief description of the links shown on this page.

Cluster Database (SSL)

Rocks maintains a MySQL database for the server. The database is used to generate service-specific configuration files such as `/etc/hosts` and `/etc/dhcpd.conf`. This `phpMyAdmin` web interface to the database can be accessed through the first link. This page will not be accessible over the public interface even if you've changed the firewall. Figure 7-2 shows the first screen into the database. You can follow the links on the left side of the page to view information about the cluster.



Using MPICH with Rocks

Before we leave Rocks, let's look at a programming example you can use to convince yourself that everything is really working.

While Rocks doesn't include MPI/LAM, it gives you your choice of several MPICH distributions. The `/opt` directory contains subdirectories for MPICH, MPICH-MPD, and MPICH2-MPD. Under MPICH, there is also a version of MPICH for Myrnet users. The distinctions are described briefly in Chapter 9. We'll stick to MPICH for now.

You can begin by copying one of the examples to your home directory.

```
[sloanjd@frontend sloanjd]$ cd /opt/mpich/gnu/examples
[sloanjd@frontend examples]$ cp cpi.c ~
[sloanjd@frontend examples]$ cd
```

```
```

Next, compile the program.

```
[sloanjd@frontend sloanjd]$ /opt/mpich/gnu/bin/mpicc cpi.c -o
cpi
```

(Clearly, you'll want to add this directory to your path once you decide which version of MPICH to use.)

Before you can run the program, you'll want to make sure SSH is running and that no error or warning messages are generated when you log onto the remote machines. (SSH is discussed in Chapter 4.)

Now you can run the program. (Rocks automatically creates the `machines` file used by the system, so that's one less thing to worry about. But you can use the **`-machinefile`** *filename* option if you wish.)

```
[sloanjd@frontend sloanjd]$ /opt/mpich/gnu/bin/mpirun -np 4
cpi
Process 0 on frontend.public
Process 2 on compute-0-1.local
Process 1 on compute-0-0.local
Process 3 on compute-0-0.local
pi is approximately 3.1416009869231245, Error is
0.0000083333333314
wall clock time = 0.010533
```

That's all there is to it.

Since Rocks also includes the *High-Performance Linpack (HPL)* benchmark, so you might want to run it. You'll need the `HPL.dat` file. With Rocks 3.2.0, you can copy it to your directory from `/var/www/html/rocks-documentation/3.2.0/`. To run the benchmark, use the command

```
[sloanjd@frontend sloanjd]$
```


Setting up a cluster means setting up machines—hopefully, lots of machines. While you should begin with a very small number of machines as you figure out what you want, eventually you'll get to the point where you are mindlessly installing system after system.

Fortunately, most of those machines will have identical setups. You could simply repeat the process for each machine, but this will be both error prone and immensely boring. You need a way to automate the process.

The approach you need depends on the number of machines to be set up and configured, the variety of machines, how mission critical the cluster is, and your level of patience. For three or four machines, a manual install and configuration of each machine is a reasonable approach, particularly if you are working with an odd mix of different machines so that each setup is different. But even with a very small number of machines, the process will go more smoothly if you can automate some of the post-installation tasks such as copying configuration files.

Unless you have the patience of Job, with more than eight or ten machines in your cluster, you'll want to automate as much of the process as possible. And as your cluster's continuous operation becomes more crucial, the need for an automated approach becomes even more important.

This chapter begins with a quick look at simple approaches to ease configuring multiple systems after the operating system has been installed. These techniques are useful for any size cluster. Even if you are clearly in the fully automated camp, you should still skim this section since these techniques apply to maintaining clusters as well as setting up clusters.

Next, three tools that are useful when building larger clusters are described—`Kickstart`, `g4u` (ghost for Unix), and `SystemImager` (part of the `Systems Installation Suite`). These tools are representative of three different approaches that can be used. Kickstart is a package-based installation program that allows you to automate the installation of the operating system. g4u is a simple image-based program that allows you to copy and distribute disk images. SystemImager is a more versatile set of tools with capabilities that extend beyond installing systems. The tools in SystemImager allow you to build, clone, and configure a system. While these tools vary in scope, each does what it was designed to do quite well. There are many other tools not discussed here.

Configuring Systems

Cloning refers to creating a number of identical systems. In practice, you may not always want systems that are exactly alike. If you have several different physical configurations, you'll need to adapt to match the hardware you have. It would be pointless to use the identical partitioning schemes on hard disks with different capacities. Furthermore, each system will have different parameters, e.g., an IP address or host name that must be unique to the system.

Setting up a system can be divided roughly into two stages—installing the operating system and then customizing it to fit your needs. This division is hazy at best. Configuration changes to the operating system could easily fall into either category. Nonetheless, many tools and techniques fall, primarily, into one of these stages so the distinction is helpful. We'll start with

the second task first since you'll want to keep this ongoing process in mind when looking at tools designed for installing systems.

The major part of the post-install configuration is getting the right files onto your system and keeping those files synchronized. This applies both to configuring the machine for the first time and to maintaining existing systems. For example, when you add a new user to your cluster, you won't want to log onto every machine in the cluster and repeat the process. It is much simpler if you can push the relevant accounting files to each machine in the cluster from your head node.

What you will want to copy will vary with your objectives, but Table 8-1 lists a few likely categories.

Types of Files

Accounting files, e.g., `/etc/passwd`, `/etc/shadow`, `/etc/group`, `/etc/gshadow`

There are two real benefits from an automated installation—it should save you work, and it will ensure the consistency of your installation, which will ultimately save you a lot more work. There are several approaches you can take, but the key to any approach is documentation. You'll first want to work through one or more manual installations to become clear on the details. You need to determine how you want your system configured and in what order the configuration steps must be done. Create an install and a post-install checklist.

If you are only doing a few machines, you can do the installations manually from the checklist if you are very careful. But this can be an error-prone activity, so even small clusters can benefit from automated installs. If you are building a large cluster, you'll definitely need some tools. There are many. This chapter focuses on three fairly representative approaches—Red Hat's Kickstart, g4u, and SystemImager.

Each of the tools described in this chapter has it place. Kickstart does a nice job for repetitive installations. It is the best approach if you have different hardware. You just create and edit a copy of the configuration file for each machine type. However, Kickstart may not be the best tool for post-installation customizations.

With image software like g4u or SystemImager, you can install software and reconfigure systems to your heart's delight before cloning. If you prepare your disk before using it, g4u images use less space than SystemImager, and it is definitely faster. g4u is the simplest tool to learn to use and is largely operating system independent. SystemInstaller is the more versatile tool, but comes with a significant learning curve. Used in combination with rsync, it provides

a mechanism to maintain your systems as well as install them. In the long run, this combination may be your best choice.

Red Hat's Kickstart is a system designed to automate the installation of a large number of identical Linux systems. Similar programs exist for other releases, such as

Notes for OSCAR and Rocks Users

Since OSCAR installs and uses SIS, much of this material probably seemed vaguely familiar to you. OSCAR uses SystemInstaller to build the image directly on the server rather than capture the image from a golden client. However, once you have installed OSCAR, you can use the SIS scripts as you see fit.

The configuration file for *rsync* is in `/etc/systemimager/rsync`. OSCAR stores the SystemImager files in `/var/lib/systemimager`. For example, the image files it creates are in `/var/lib/systemimager/images`.

Rocks uses Kickstart. It uses XML files to record configuration information, dynamically generating the Kickstart configuration file. Changing these XML files is described in Chapter 7. You can interactively re-Kickstart a compute node with the `shoot-node` command. See the manpage `shoot-node(8)` for more details.

Chapter 9: **Programming Software**

After the operating system and other basic system software, you'll want to install the core software as determined by the cluster's mission. If you are planning to develop applications, you'll need software development tools, including libraries that support parallel processing. If you plan to run a set of existing cluster-ready applications, you'll need to select and install those applications as part of the image you will clone.

This chapter presupposes you'll want to develop cluster software and will need the tools to do so. For many clusters this may not be the case. For example, if you are setting up a cluster to process bioinformatics data, your needs may be met with the installation of applications such as BLAST, ClustalW, FASTA, etc. If this is the path you are taking, then identifying, installing, and learning to use these applications are the next steps you need to take. For now, you can safely skip this chapter. But don't forget that it is here. Even if you are using canned applications, at some point you may want to go beyond what is available and you'll need the tools in this chapter.

This chapter describes the installation and basic use of the software development tools used to develop and run cluster applications. It also briefly mentions some tools that you are likely to need that should already be part of your system. For clusters where you develop the application software, the software described in this chapter is essential. In contrast, you may

be able to get by without management and scheduling software. You won't get far without the software described here.

If you've installed OSCAR or Rocks, you will have pretty much everything you need. Nonetheless, you'll still want to skim this chapter to learn more about how to use that software. For cluster application developers, this is the first software you need to learn how to use.

While there are hundreds of programming languages available, when it comes to writing code for high-performance clusters, there are only a couple of realistic choices. For pragmatic reasons, your choices are basically FORTRAN or C/C++.

Programming Languages

While there are hundreds of programming languages available, when it comes to writing code for high-performance clusters, there are only a couple of realistic choices. For pragmatic reasons, your choices are basically FORTRAN or C/C++.

Like it or not, FORTRAN has always been the lingua franca of high-performance computing. Because of the installed base of software, this isn't likely to change soon. This doesn't mean that you need to use FORTRAN for new projects, but if you have an existing project using FORTRAN, then you'll need to support it. This comes down to knowing how your cluster will be used and knowing your users' needs.

FORTRAN has changed considerably over the years, so the term can mean different things to different people. While there are more recent versions of FORTRAN, your choice will likely be between FORTRAN 77 and FORTRAN 90. For a variety of reasons, FORTRAN 77 is likely to get the nod over FORTRAN 90 despite the greater functionality of FORTRAN 90. First, the GNU implementation of FORTRAN 77 is likely to already be on your machine. If it isn't, it is freely available and easily obtainable. If you really want FORTRAN 90, don't forget to budget for it. But you should also realize that you may face compatibility issues. When selecting parallel programming libraries to use with your compiler, your choices will be more limited with FORTRAN 90.

C and C++ are the obvious alternatives to FORTRAN. For new applications that don't depend on compatibility with legacy FORTRAN applications, C is probably the best choice. In general, you have greater compatibility with libraries. And at this point in time, you are likely to find more programmers trained in C than FORTRAN. So when you need help, you are more likely to find a helpful C than FORTRAN programmer. For this and other reasons, the examples in this book will stick to C.

With most other languages you are out of luck. With very few exceptions, the parallel programming libraries simply don't have binding for other languages. This is changing. While

bindings for Python and Java are being developed, it is probably best to think of these as works in progress. If you want to play it safe, you'll stick to C or FORTRAN.

Selecting a Library

Those of you who do your own dentistry will probably want to program your parallel applications from scratch. It is certainly possible to develop your code with little more than a good compiler. You could manually set up communication channels among processes using standard systems calls.

The rest of you will probably prefer to use libraries designed to simplify parallel programming. This really comes down to two choices—the `Parallel Virtual Machine (PVM)` library or the `Message Passing Interface (MPI)` library. Work was begun on PVM in 1989 and continued into the early '90s as a joint effort among Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie-Mellon University. An implementation of PVM is available from **`http://www.netlib.org/pvm3/`**. This PVM implementation provides both libraries and tools based on a message-passing model.

Without getting into a philosophical discussion, MPI is a newer standard that seems to be generally preferred over PVM by many users. For this reason, this book will focus on MPI. However, both PVM and MPI are solid, robust approaches that will potentially meet most users' needs. You won't go too far wrong with either. OSCAR, you will recall, installs both PVM and MPI.

MPI is an API for parallel programming based on a message-passing model for parallel computing. MPI processes execute in parallel. Each process has a separate address space. Sending processes specify data to be sent and a destination process. The receiving process specifies an area in memory for the message, the identity of the source, etc.

Primarily, MPI can be thought of as a standard that specifies a library. Users can write code in C, C++, or FORTRAN using a standard compiler and then link to the MPI library. The library implements a predefined set of function calls to send and receive messages among collaborating processes on the different machines in the cluster. You write your code using these functions and link the completed code to the library.

LAM/MPI

The `Local Area Multicomputer/Message Passing Interface (LAM/MPI)` was originally developed by the Ohio Supercomputing Center. It is now maintained by the Open Systems Laboratory at Indiana University. As previously noted, LAM/MPI (or LAM for short) is both an MPI library and an execution environment. Although beyond the scope of this book, LAM was designed to include an extensible component framework known as `System Service Interface (SSI)`, one of its major

strengths. It works well in a wide variety of environments and supports several methods of inter-process communications using TCP/IP. LAM will run on most Unix machines (but not Windows). New releases are tested with both Red Hat and Mandrake Linux.

Documentation can be downloaded from the LAM site, **http://www.lam-mpi.org/**. There are also tutorials, a FAQ, and archived mailing lists. This chapter provides an overview of the installation process and a description of how to use LAM. For more up-to-date and detailed information, you should consult the *LAM/MPI Installation Guide* and the *LAM/MPI User's Guide*.

You have two basic choices when installing LAM. You can download and install a Red Hat package, or you can download the source and recompile it. The package approach is very quick, easy to automate, and uses somewhat less space. If you have a small cluster and are manually installing the software, it will be a lot easier to use packages. Installing from the source will allow you to customize the installation, i.e., select which features are enabled and determine where the software is installed. It is probably a bad idea to mix installations since you could easily end up with different versions of the software, something you'll definitely want to avoid.

Installing from a package is done just as you'd expect. Download the package from **http://www.lam-mpi.org/** and install it just as you would any Red Hat package.

MPICH

`Message Passing Interface Chameleon (MPICH)` was developed by William Gropp and Ewing Lusk and is freely available from Argonne National Laboratory (**http://www-unix.mcs.anl.gov/mpi/mpich/**). Like LAM, it is both a library and an execution environment. It runs on a wide variety of Unix platforms and is even available for Windows NT.

Documentation can be downloaded from the web site. There are separate manuals for each of the communication models. This chapter provides an overview of the installation process and a description of how to use MPICH. For more up-to-date and detailed information, you should consult the appropriate manual for the communications model you are using.

There are five different "flavors" of MPICH reflecting the type of machine it will run on and how interprocess communication has been implemented:

ch_p4

This is probably the most common version. The "ch" is for channel and the "p4" for portable programs for parallel processors.

ch_p4mpd

This extends ch_p4 mode by including a set of daemons built to support parallel processing. The MPD is for multipurpose daemon. MPD is a new high-performance job launcher designed as a replacement for `mpirun`.

ch_shmem

This is a version for shared memory or SMP systems.

globus2

This is a version for computational grids. (See **http://www.globus.org** for more on the *Globus* project.)

Other Programming Software

Keeping in mind that your head node will also serve as a software development platform, there are other software packages that you'll want to install. One obvious utility is the ubiquitous text editor. Fortunately, most likely choices are readily available and will be part of your basic installation. Just don't forget them when you install the system. Because personal preferences vary so widely, you'll want to include the full complement.

Another essential tool is a software debugger. Let's face it, using `printf` to debug parallel code is usually a hopeless task. With multiple processes and buffered output, it is unlikely you'll know where the program was executing when you actually see the output. The best solution is a debugger designed specifically for parallel code. While commercial products such as `TotalView` are available and work well with MPI, free software is wanting. At the very least, you will want a good traditional debugger such as `gdb`. Programs that extend `gdb`, such as `ddd` (the *Data Display Debugger*), are a nice addition. (Debugging is discussed in greater detail in Chapter 16.) Since it is difficult to tell when they will be needed and just how essential they will be, try to be as inclusive as possible when installing these tools. As part of the `gcc` development package, `gdb` is pretty standard fare and should already be on your system. However, `ddd` may not be installed by default.

Since `ddd` provides a GUI for other debuggers such as `gdb`, there is no point installing it on a system that doesn't have X Windows and `gdb` a or similar debugger. `ddd` is often included as part of a Linux distribution; for instance, Red Hat includes it. If not, you can download it from **http://www.gnu.org/software/ddd**. The easiest way to install it is from an RPM.

```
[root@fanny root]# rpm -vih ddd-3.3.1-23.i386.rpm
warning: ddd-3.3.1-23.i386.rpm: V3 DSA signature: NOKEY, key
ID db42a60e
Preparing...
########################################### [100%]
   1:ddd
########################################### [100%]
```

Notes for OSCAR Users

LAM/MPI, MPICH, and HDF5 are installed as part of a standard OSCAR installation under the `/opt` directory to conform to the File System Hierarchy (FSH) standard (**`http://www.pathname.com/fhs/`**). Both MPICH and HDF5 have documentation subdirectories `doc` with additional information. OSCAR does not install MPE as part of the MPICH installation. If you want to use MPE, you'll need to go back and do a manual installation. Fortunately, this is not particularly difficult, but it can be a bit confusing.

First, use switcher to select your preferred version of MPI . Since you can't run LAM/MPI as root, MPICH is probably a better choice. For example,

```
[root@amy root]# switcher mpi --list
```

```
lam-7.0
```

```
lam-with-gm-7.0
mpich-ch_p4-gcc-1.2.5.10
[root@amy root]# switcher mpi = mpich-ch_p4-gcc-1.2.5.10
Attribute successfully set; new attribute setting will be
effective for
future shells
```

If you had to change MPI, log out and back onto the system.

Next, you'll need to retrieve and unpack a copy of MPICH.

```
[root@amy root]# cp mpich.tar.gz /usr/local/src
[root@amy root]# cd /usr/local/src
[root@amy src]# gunzip mpich.tar.gz
[root@amy src]# tar -xvf mpich.tar
...
```

`/usr/local/src` is a reasonable location.

If you don't have it on your system, you'll need to install Java to build the `jumpshot`.

```
[root@amy src]# bunzip2 j2sdk-1.3.1-FCS-linux-i386.tar.bz2
[root@amy src]# tar -xvf j2sdk-1.3.1-FCS-linux-i386.tar
...
```

Again, `/usr/local/src` is a reasonable choice.

Next, you need to set your PATH to include Java and set environmental variables for MPICH.

```
[root@amy src]# export
PATH=/usr/local/src/j2sdk1.3.1/bin:$PATH
[root@amy src]# export MPI_INC="-I/opt/mpich-1.2.5.10-ch_p4-
gcc/include"
[root@amy src]# export MPI_LIBS="-L/opt/mpich-1.2.5.10-ch_p4-
gcc/lib"
[root@amy src]# export MPI_CC=mpicc
[root@amy src]# export MPI_F77=mpif77
```

Notes for Rocks Users

Rocks does not include LAM/MPI or HDF5 but does include several different MPICH releases, located in `/opt`. MPE is included as part of Rocks with each release. The MPE libraries are included with the MPICH libraries, e.g., `/opt/mpich/gnu/lib`. Rocks includes the `jumpshot3` script as well, e.g., `/opt/mpich/gnu/share/jumpshot-3/bin` for MPICH. (Rocks also includes `upshot`.)

By default, Rocks does not include Java. There is, however, a Java roll for Rocks. To use `jumpshot3`, you'll need to install the appropriate version of Java. You can look in the `jumpshot3` script to see what it expects. You should see something like the following near the top of the file:

```
...
JAVA_HOME=/usr/java/j2sdk1.4.2_02
...
JVM=/usr/java/j2sdk1.4.2_02/bin/java
...
```

You can either install `j2sdk1.4.2-02` in `/usr/java` or you can edit these lines to match your Java installation. For example, if you install the Java package described in the last section, you might change these lines to

```
JAVA_HOME=/usr/local/src/j2sdk1.3.1
JVM=/usr/local/src/j2sdk1.3.1/bin/java
```

Adjust the path according to your needs.

Chapter 10: **Management Software**

Now that you have a cluster, you are going to want to keep it running, which will involve a number of routine system administration tasks. If you have done system administration before, then for the most part you won't be doing anything new. The administrative tasks you'll face are largely the same tasks you would face with any multiuser system. It is just that these tasks will be multiplied by the number of machines in your cluster. While creating 25

new accounts on a server may not sound too hard, when you have to duplicate those accounts on each node in a 200-node cluster, you'll probably want some help.

For a small cluster with only a few users, you may be able to get by doing things the way you are used to doing them. But why bother? The tools in this chapter are easy to install and use. Mastering them, which won't take long, will lighten your workload.

While there are a number of tools available, two representative tools (or tool sets) are described in this chapter—the *Cluster Command and Control (C3)* tools set and *Ganglia*. C3 is a set of utilities that can be used to automate a number of tasks across a cluster or multiple clusters, such as executing the same command on every machine or distributing files to every machine. Ganglia is used to monitor the health of your cluster from a single node using a web-based interface.

Cluster Command and Control is a set of about a dozen command-line utilities used to execute common management tasks. These commands were designed to provide a look and feel similar to that of issuing commands on a single machine. The commands are both secure and scale reliably. Each command is actually a Python script. C3 was developed at Oak Ridge National Laboratory and is freely available.

There are two ways C3 can be installed. With the basic install, you'll do a full C3 installation on a single machine, typically the head node, and issue commands on that machine. With large clusters, this can be inefficient because that single machine must communicate with each of the other machines in the cluster. The alternate approach is referred to as a scalable installation. With this method, C3 is installed on all the machines and the configuration is changed so that a tree structure is used to distribute commands. That is, commands fan out through intermediate machines and are relayed across the cluster more efficiently. Both installations begin the same way; you'll just need to repeat the installation with the scalable install to alter the configuration file. This description will stick to the simple install. The simple installation includes a file

C3

Cluster Command and Control is a set of about a dozen command-line utilities used to execute common management tasks. These commands were designed to provide a look and feel similar to that of issuing commands on a single machine. The commands are both secure and scale reliably. Each command is actually a Python script. C3 was developed at Oak Ridge National Laboratory and is freely available.

There are two ways C3 can be installed. With the basic install, you'll do a full C3 installation on a single machine, typically the head node, and issue commands on that machine. With large clusters, this can be inefficient because that single machine must communicate with each of the other machines in the cluster. The alternate approach is referred to as a scalable

installation. With this method, C3 is installed on all the machines and the configuration is changed so that a tree structure is used to distribute commands. That is, commands fan out through intermediate machines and are relayed across the cluster more efficiently. Both installations begin the same way; you'll just need to repeat the installation with the scalable install to alter the configuration file. This description will stick to the simple install. The simple installation includes a file *README.scale* that describes the scalable installation.

Since the C3 tools are scripts, there is very little to do to install them. However, since they rely on several other common packages and services, you will need to be sure that all the prerequisites are met. On most systems this won't be a problem; everything you'll need will already be in place.

Before you can install C3, make sure that rsync, Perl, SSH, and Python are installed on your system and available. Name resolution, either through DNS or a host file, must be available as well. Additionally, if you want to use the C3 command `pushimage`, SystemImager must be installed. Installing SystemImager is discussed in Chapter 8.

Ganglia

With a large cluster, it can be a daunting task just to ensure that every machine is up and running every day if you try to do it manually. Fortunately, there are several tools that you can use to monitor the state of your cluster. In clustering circles, the better known of these include `Ganglia`, `Clumon`, and `Performance Co-Pilot (CPC)`. While this section will describe Ganglia, you might reasonably consider any of these.

Ganglia is a real-time performance monitor for clusters and grids. If you are familiar with MRTG, Ganglia uses the same round-robin database package that was developed for MRTG. Memory efficient and robust, Ganglia scales well and has been used with clusters with hundreds of machines. It is also straightforward to configure for use with multiple clusters so that a single management station can monitor all the nodes within multiple clusters. It was developed at UCB, is freely available (via a BSD license), and has been ported to a number of different architectures.

Ganglia uses a client-server model and is composed of four parts. The monitor daemon `gmond` needs to be installed on every machine in the cluster. The backend for data collection, the daemon `gmetad`, and the web interface frontend are installed on a single management station. (There is also a Python class for sorting and classifying data from large clusters.) Data are transmitted using XML and XDR via both TCP and multicasting.

In addition to these core components, there are two command-line tools. The cluster status tool `gstat` provides a way to query `gmond`, allowing you to create a status report for your cluster. The metric tool `gmetric` allows you to easily monitor additional host metrics in addition to Ganglia's predefined metrics. For instance, suppose you have a program (and

interface) that measures a computer's temperature on each node. `gmetric` can be used to request that `gmond` run this program. By running the

Notes for OSCAR and Rocks Users

C3 is a core OSCAR package that is installed in `/opt/c3-4` and can be used as shown in this chapter. Both Ganglia and Clumon (which uses Performance Co-Pilot) may be available as additional packages for OSCAR. As add-ons, these may not always be available immediately when new versions of OSCAR are released. For example, there was a delay with both when OSCAR 3.0 was released. When installing Ganglia using the package add option with OSCAR, you may want to tweak the configuration files, etc.

Although not as versatile as the C3 command set, Rocks supplies the command `cluster-fork` for executing commands across a cluster.

For OSCAR, the web-accessible reports for Clumon and Ganglia are installed in `/var/www/html/clumon` and `/var/www/html/ganglia`, respectively. Thus, to access the Ganglia web report on `amy.wofford.int`, the URL is **`http://amy.wofford.int/ganglia/`**. The page format used by OSCAR is a little different, but you would use Ganglia in much the same way.

Ganglia is fully integrated into Rocks and is available as a link from the administrative home for the frontend.

Chapter 11: **Scheduling Software**

Basically, scheduling software lets you run your cluster like a batch system, allowing you to allocate cluster resources, such as CPU time and memory, on a job-by-job basis. Jobs are queued and run as resources become available, subject to the priorities you establish. Your users will be able to add and remove jobs from the job queue as well as track the progress of their jobs. As the administrator, you will be able to establish priorities and manage the queue.

Scheduling software is not a high priority for everyone. If the cluster is under the control of a single user, then scheduling software probably isn't needed. Similarly, if you have a small cluster with very few users or if your cluster is very lightly used, you may not need scheduling software. As long as you have more resources than you need, manual scheduling may be a viable alternative—at least initially. If you have a small cluster and only occasionally wish you had scheduling software, it may be easier to add a few more computers or build a second cluster than deal with the problems that scheduling software introduces.

But if you have a large cluster with a growing user base, at some point you'll want to install scheduling software. At a minimum, scheduling software helps you effectively use your hardware and provides a more equitable sharing of resources. Scheduling software has other

uses as well, including accounting and monitoring. The information provided by good scheduling software can be a huge help when planning for the future of your cluster.

There are several freely available scheduling systems from which you can select, including `Portable Batch System (PBS)`, `Maui`, `Torque`, and `Condor`. OSCAR includes Portable Batch System (PBS) along with Maui. Torque is also available for OSCAR via `opd`. Rocks provides a PBS roll that includes Maui and Torque and a second roll that includes Condor. Since PBS is available for both OSCAR and Rocks, that's what's described in this chapter. (For more information on the alternatives, visit the web sites listed in the Appendix A.)

OpenPBS

Before the emergence of clusters, the Unix-based *Network Queuing System (NQS)* from NASA Ames Research Center was a commonly used batch-queuing system. With the emergence of parallel distributed system, NQS began to show its limitations. Consequently, Ames led an effort to develop requirements and specifications for a newer, cluster-compatible system. These requirements and specifications later became the basis for the IEEE 1003.2d POSIX standard. With NASA funding, PBS, a system conforming to those standards, was developed by Veridian in the early 1990s.

PBS is available in two forms—OpenPBS or PBSPro. OpenPBS is the unsupported original open source version of PBS, while PBSPro is a newer commercial product. In 2003, PBSPro was acquired by Altair Engineering and is now marketed by Altair Grid Technologies, a subsidiary of Altair Engineering. The web site for OpenPBS is **`http://www.openpbs.org`**; the web site for PBSPro is **`http://www.pbspro.com`**. Although much of the following will also apply to PBSPro, the remainder of this chapter describes OpenPBS, which is often referred to simply as PBS. However, if you have the resources to purchase software, it is well worth looking into PBSPro. Academic grants have been available in the past, so if you are eligible, this is worth looking into as well.

As an unsupported product, OpenPBS has its problems. Of the software described in this book, it was, for me, the most difficult to install. In my opinion, it is easier to install OSCAR, which has OpenPBS as a component, or Rocks along with the PBS roll than it is to install just OpenPBS. With this warning in mind, we'll look at a typical installation later in this chapter.

Before we install PBS, it is helpful to describe its architecture. PBS uses a client-server model and is organized as a set of user-level commands that interact with three system-level daemons. Jobs are submitted using the user-level commands and managed by the daemons. PBS also includes an API.

Notes for OSCAR and Rocks Users

As previously noted, both OpenPBS and Maui are installed as part of the OSCAR setup. The installation directory for OpenPBS is `/opt/pbs`. You'll find the various commands in subdirectories under this directory. The working directory for OpenPBS is `/var/spool/pbs`, where you'll find the configuration and log files. The default queue, as you may have noticed from previous examples, is `workq`. Under OSCAR, Maui is installed in the directory `/opt/maui`. By default, the OpenPBS FIFO scheduler is disabled.

OpenPBS and Maui are available for Rocks as a separate roll. If you need OpenPBS, be sure you include the roll when you build your cluster as it is not currently possible to add the roll once the cluster has been installed. Once installed, the system is ready to use. The default queue is `default`.

Rocks also provides a web-based interface for viewing the job queue that is available from the frontend's home page. Using the web interface, you can view both the job queue and the physical job assignments. PBS configuration files are located in `/opt/torque`. Manpages are in `/opt/torque/man`. Maui is installed under `/opt/maui`.

Chapter 12: **Parallel Filesystems**

If you are certain that your cluster will only be used for computationally intensive tasks that involve very little interaction with the filesystem, you can safely skip this chapter. But increasingly, tasks that are computationally expensive also involve a large amount of I/O, frequently accessing either large data sets or large databases. If this is true for at least some of your cluster's applications, you need to ensure that the I/O subsystem you are using can keep up. For these applications to perform well, you will need a high-performance filesystem.

Selecting a filesystem for a cluster is a balancing act. There are a number of different characteristics that can be used to compare filesystems, including robustness, failure recovery, journaling, enhanced security, and reduced latency. With clusters, however, it often comes down to a trade-off between convenience and performance. From the perspective of convenience, the filesystem should be transparent to users, with files readily available across the cluster. From the perspective of performance, data should be available to the processor that needs it as quickly as possible. Getting the most from a high-performance filesystem often means programming with the filesystem in mind—typically a very "inconvenient" task. The good news is that you are not limited to a single filesystem.

The Network File System (NFS) was introduced in Chapter 4. NFS is strong on convenience. With NFS, you will recall, files reside in a directory on a single disk drive that is shared across the network. The centralized availability provided by NFS makes it an important part of any cluster. For example, it provides a transparent mechanism to ensure that binaries of freshly compiled parallel programs are available on all the machines in the cluster. Unfortunately, NFS is not very efficient. In particular, it has not been optimized for the types of I/O often needed with many high-performance cluster applications.

High-performance filesystems for clusters are designed using different criteria, primarily to optimize performance when accessing large data sets from parallel applications. With parallel filesystems, files may be distributed across a cluster with different pieces of the file on different machines allowing parallel access.

PVFS

PVFS is a freely available, software-based solution jointly developed by Argonne National Laboratory and Clemson University. PVFS is designed to distribute data among the disks throughout the cluster and will work with both serial and parallel programs. In programming, it works with traditional Unix file I/O semantics, with the MPI-2 ROMIO semantics, or with the native PVFS semantics. It provides a consistent namespace and transparent access using existing utilities along with a mechanism for programming application-specific access. Although PVFS is developed using X-86-based Linux platforms, it runs on some other platforms. It is available for both OSCAR and Rocks. PVFS2, a second generation PVFS, is in the works.

On the downside, PVFS does not provide redundancy, does not support symbolic or hard links, and it does not provide a `fsck`-like utility.

Figure 12-1 shows the overall architecture for a cluster using PVFS. Machines in a cluster using PVFS fall into three possibly overlapping categories based on functionality. Each PVFS has one metadata server. This is a filesystem management node that maintains or tracks information about the filesystem such as file ownership, access privileges, and locations, i.e., the filesystem's metadata.

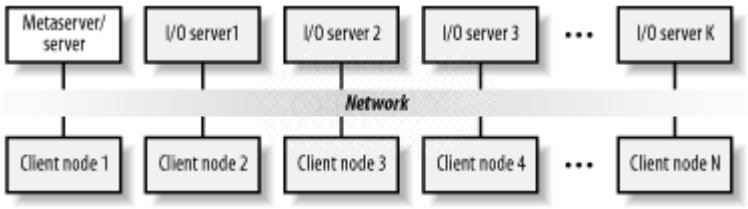


Figure 12-1: Internal cluster architecture

Because PVFS distributes files across the cluster nodes, the actual files are located on the disks on I/O servers. I/O servers store the data using the existing hardware and filesystem on that node. By spreading or striping a file across multiple nodes, applications have multiple paths to data. A compute node may access a portion of the file on one machine while another node accesses a different portion of the file located on a different I/O server. This eliminates the bottleneck inherent in a single file server approach such as NFS.

Using PVFS

To make effective use of PVFS, you need to understand how PVFS distributes files across the cluster. PVFS uses a simple striping scheme with three striping parameters.

base

The cluster node where the file starts, given as an index where the first I/O server is 0. Typically, this defaults to 0.

pcount

The number of I/O servers among which the file is partitioned. Typically, this defaults to the total number of I/O servers.

ssize

The size of each strip, i.e., contiguous blocks of data. Typically, this defaults to 64 KB.

Figure 12-2 should help clarify how files are distributed. In the figure, the file is broken into eight pieces and distributed among four I/O servers. `base` is the index of the first I/O server. `pcount` is the number of servers used, i.e., four in this case. `ssize` is the size of each of the eight blocks. Of course, the idea is to select a block size that will optimize parallel access to the file.

Figure 12-2: Overlap within files

You can examine the distribution of a file using the `pvstat` utility. For example,

```
[root@fanny pvfs]# pvstat data
data: base = 0, pcount = 5, ssize = 65536
[root@fanny pvfs]# ls -l data
-rw-r--r--    1 root     root     10485760 Jun 21 12:49 data
```

A little arithmetic shows this file is broken into 160 pieces with 32 blocks on each I/O server.

If you copy a file to a PVFS filesystem using `cp`, it will be partitioned automatically for you using what should be reasonable defaults. For more control, you can use the

Both OSCAR and Rocks use NFS. Rocks uses `autofs` to mount home directories; OSCAR doesn't. (Automounting and `autofs` is discussed briefly in Chapter 4.)

PVFS is available as an add-on package for OSCAR. By default, it installs across the first eight available nodes using the OSCAR server as the metadata server and mount point. The OSCAR server is not configured as an I/O server. OSCAR configures PVFS to start automatically when the system is rebooted.

With OSCAR, PVFS is installed in the directory `/opt/pvfs`, e.g., the libraries are in `/opt/pvfs/lib` and the manpages are in `/opt/pvfs/man`. The manpages are not placed on the user's path but can be with the **`-M`** option to man. For example,

```
[root@amy /]# man -M /opt/pvfs/man/ pvfs_chmod
```

The PVFS utilities are in `/opt/pvfs/bin` and the daemons are in `/opt/pvfs/sbin`. The mount point for PVFS is `/mnt/pvfs`. Everything else is pretty much where you would expect it to be.

PVFS is fully integrated into Rocks on all nodes. However, you will need to do several configuration tasks. Basically, this means following the steps outlined in this chapter. However, you'll find that some of the steps have been done for you.

On the meta-server, the directory `/pvfs-meta` is already in place; run `/usr/bin/mkmgrconf` to create the configuration files. For the I/O servers, you'll need to create the data directory `/pvfs-data` but the configuration file is already in place. The kernel modules are currently in `/lib/modules/2.4.21-15.EL/fs/` and are preloaded. You'll need to start the I/O daemon `/usr/sbin/iod`, and you'll need to mount each client using `/sbin/mount.pvfs`. All in all it goes quickly. Just be sure to note locations for the various commands.

Chapter 13: **Getting Started with MPI**

This chapter takes you through the creation of a simple program that uses the MPI libraries. It begins with a few brief comments about using MPI. Next, it looks at a program that can be run on a single processor without MPI, i.e., a serial solution to the problem. This is followed by an explanation of how the program can be rewritten using MPI to create a parallel program that divides the task among the machines in a cluster. Finally, some simple ways the solution can be extended are examined. By the time you finish this chapter, you'll know the basics of using MPI.

Three versions of the initial solution to this problem are included in this chapter. The first version, using C, is presented in detail. This is followed by briefer presentations showing how the code can be rewritten, first using FORTRAN, and then using C++. While the rest of this book sticks to C, these last two versions should give you the basic idea of what's involved if you would rather use FORTRAN or C++. In general, it is very straightforward to switch between C and FORTRAN. It is a little more difficult to translate code into C++, particularly if you want to make heavy use of objects in your code. You can safely skip either or both the FORTRAN and C++ solutions if you won't be using these languages.

The major difficulty in parallel programming is subdividing problems so that different parts can be executed simultaneously on different machines. MPI is a library of routines that provides the functionality needed to allow those parts to communicate. But it will be up to you to determine how a problem can be broken into pieces so that it can run on different machines.

The simplest approach is to have the number of processes match the number of machines or processors that are available. However, this is not required. If you have a small problem that can be easily run on a subset of your cluster, or if your problem logically decomposes in such a way that you don't need the entire cluster, then you can (and should) execute the program on fewer machines. It is also possible to have multiple processes running on the same machine. This is particularly common when developing code. In this case, the operating system will switch between processes as needed. You won't benefit from the parallelization of the code, but the job will still complete correctly.

The major difficulty in parallel programming is subdividing problems so that different parts can be executed simultaneously on different machines. MPI is a library of routines that provides the functionality needed to allow those parts to communicate. But it will be up to you to determine how a problem can be broken into pieces so that it can run on different machines.

The simplest approach is to have the number of processes match the number of machines or processors that are available. However, this is not required. If you have a small problem that can be easily run on a subset of your cluster, or if your problem logically decomposes in such a way that you don't need the entire cluster, then you can (and should) execute the program on fewer machines. It is also possible to have multiple processes running on the same machine. This is particularly common when developing code. In this case, the operating system will switch between processes as needed. You won't benefit from the parallelization of the code, but the job will still complete correctly.

With most parallelizable problems, programs running on multiple computers do the bulk of the work and then communicate their individual results to a single computer that collects these intermediate results, combines them, and reports the final results. It is certainly possible to write a different program for each machine in the cluster, but from a software management perspective, it is much easier if we can write just one program. As the program executes on each machine, it will first determine which computer it is running on and, based on that information, tackle the appropriate part of the original problem. When the computation is complete, one machine will act as a receiver and all the other machines will send their results to it.

For this approach to work, each executing program or process must be able to differentiate itself from other processes. Let's look at a very basic example that demonstrates how processes, i.e., the program in execution on different computers, are able to differentiate themselves. While this example doesn't accomplish anything particularly useful, it shows how the pieces fit together. It introduces four key functions and one other useful function. And with a few minor changes, this program will serve as a template for future programs.

A Simple Problem

Before we can continue examining MPI, we need a more interesting problem to investigate. We will begin by looking at how you might write a program to calculate the area under a curve, i.e., a numerical integration. This is a fairly standard problem for introducing parallel calculations because it can be easily decomposed into parts that can be shared among the computers in a cluster. Although in most cases it can be solved quickly on a single processor, the parallel solution illustrates all the basics you need to get started writing MPI code. We'll keep coming back to this problem in later chapters so you'll probably grow tired of it. But sticking to the same problem will make it easy for us to focus on programming constructs without getting bogged down with the details of different problems.

If you are familiar with numerical integration, you can skim this section quickly and move on to the next. Although this problem is a bit mathematical, it is straightforward and the mathematics shouldn't create much of a problem. Each step in the problem in this section is carefully explained, and you don't need to worry about every detail to get the basic idea.

Let's get started. Suppose you are driving a car whose clock and speedometer work, but whose odometer doesn't work. How do you determine how far you have driven? If you are traveling at a constant speed, the distance traveled is the speed that you are traveling multiplied by the amount of time you travel. If you go 60 miles an hour for two hours, you travel 120 miles. If your speed is changing, you'll need to do a lot of little calculations and add up the results. For example, if you go 60 for 30 minutes, slow down to 40 for construction for the next 30 minutes, and then hotfoot it at 70 for the next hour to make up time, your total distance is 30

plus 20 plus 70 or 120 miles. You just calculate the distance traveled at each speed and add up the results.

If we plot speed against time, we can see that what we are calculating is the area under the curve. Basically, we are dividing the area into rectangles, calculating the area of each rectangle, and then adding up the results. In our example, the first rectangle has a width of one half (half an hour) and a height of 60, the second a width of one half and a height of 40, and the third a width of 1 and a height of 70. If your speed changes a lot, you will just have more rectangles. Figure 13-1 gives the basic idea.

An MPI Solution

Now that we've seen how to create a serial solution, let's look at a parallel solution. We'll look at the solution first in C and then in FORTRAN and C++.

The reason this area problem is both interesting and commonly used is that it is very straightforward to subdivide this problem. We can let different computers calculate the areas for different rectangles. Along the way, we'll introduce two new functions, **MPI_Send** and **MPI_Receive**, used to exchange information among processes.

Basically, **MPI_Comm_size** and **MPI_Comm_rank** are used to divide the problem among processors. **MPI_Send** is used to send the intermediate results back to the process with rank 0, which collects the results with **MPI_Recv** and prints the final answer. Here is the program:

```
#include "mpi.h"
#include <stdio.h>

/* problem parameters */
#define f(x)            ((x) * (x))
#define numberRects     50
#define lowerLimit      2.0
#define upperLimit      5.0

int main( int argc, char * argv[  ] )
{
   /* MPI variables */
   int dest, noProcesses, processId, src, tag;
   MPI_Status status;

   /* problem variables */
   int         i;
   double      area, at, height, lower, width, total, range;

   /* MPI setup */
   MPI_Init(&argc, &argv);
```

```
   MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
   MPI_Comm_rank(MPI_COMM_WORLD, &processId);

   /* adjust problem size for subproblem*/
   range = (upperLimit - lowerLimit) / noProcesses;
   width = range / numberRects;
   lower = lowerLimit + range * processId;

   /* calculate area for subproblem */
   area = 0.0;
   for (i = 0; i < numberRects; i++)
   {  at = lower + i * width + width / 2.0;
      height = f(at);
      area = area + width * height;
   }

   /* collect information and print results */
   tag = 0;
   if (processId = = 0)         /* if rank is 0, collect
results */
   {  total = area;
      for (src=1; src < noProcesses; src++)
      {  MPI_Recv(&area, 1, MPI_DOUBLE, src, tag,
MPI_COMM_WORLD, &status);
         total = total + area;
      }
      fprintf(stderr, "The area from %f to %f is: %f\n",
              lowerLimit, upperLimit, total );
   }
   else                         /* all other processes only
send */
   {  dest = 0;
      MPI_Send(&area, 1, MPI_DOUBLE, dest, tag,
MPI_COMM_WORLD);
   };

   /* finish */
   MPI_Finalize( );
   return 0;
}
```

I/O with MPI

One severe limitation to our solution is that all of the parameters are hardwired into the program. If we want to change anything, we need to recompile the program. It would be much more useful if we read parameters from standard input.

Thus far, we have glossed over the potential difficulties that arise with I/O and MPI. In general, I/O can get very messy with parallel programs. With our very first program, we saw messages from each processor on our screen. Stop and think about it—how did the messages from the other remote processes get to our screen? That bit of magic was handled by `mpirun`. The MPI standard does not fully specify how I/O should be handled. Details are left to the implementer. In general, you can usually expect the rank 0 process to be able to both read from standard input and write to standard output. Output from other processes is usually mapped back to the home node and displayed. Input calls by other processes are usually mapped to `/dev/zero`, i.e., they are ignored. If in doubt, consult the documentation for your particular implementation. If you can't find the answer in the documentation, it is fairly straightforward to write a simple test program.

In practice, this strategy doesn't cause too many problems. It is certainly adequate for our modest goals. Our strategy is to have the rank 0 process read the parameters from standard input and then distribute them to the remaining processes. With that in mind, here is a solution. New code appears in boldface.

```
#include "mpi.h"
#include <stdio.h>

/* problem parameters */
#define f(x)            ((x) * (x))

int main( int argc, char * argv[  ] )
{
   /* MPI variables */
   int dest, noProcesses, processId, src, tag;
   MPI_Status status;

   /* problem variables */
   int         i, numberRects;
   double      area, at, height, lower, width, total, range;
   double      lowerLimit, upperLimit;

   /* MPI setup */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
   MPI_Comm_rank(MPI_COMM_WORLD, &processId);
```

Broadcast Communications

In this subsection, we will further improve the efficiency of our code by introducing two new MPI functions. In the process, we'll reduce the amount of code we have to work with.

If you look back to the last solution, you'll notice that the parameters are sent individually to each process one at a time even though each process is receiving the same information. For example, if you are using 10 processes, while process 0 communicates with process 1, processes 2 through 10 are idle. While process 0 communicates with process 2, processes 3 through 10 are sill idle. And so on. This may not be a big problem with a half dozen processes, but if you are running on 1,000 machines, this can result in a lot of wasted time. Fortunately, MPI provides an alternative, **`MPI_Bcast`**.

Section 13.5.1.1: MPI_Bcast

`MPI_Bcast` provides a mechanism to distribute the same information among a communication group or communicator. **`MPI_Bcast`** takes five arguments. The first three define the data to be transmitted. The first argument is the buffer that contains the data; the second argument is the number of items in the buffer; and the third argument, the data type. (The supported data types are the same as with **`MPI_Send`**, etc.)

The next argument is the rank of the process that is generating the broadcast, sometimes called the root of the broadcast. In our example, this is 0, but this isn't a requirement. All processes use identical calls to **`MPI_Bcast`**. By comparing their rank to the rank specified in the can, a process can determine whether it is sending or receiving data. Consequently, there is no need for any additional control structures with **`MPI_Bcast`**. The final argument is the communicator, which effectively defines which processes will participate in the broadcast. When the call returns, the data in the root's communications buffer will have been copied to each of the remaining processes in the communicator.

Chapter 14: **Additional MPI Features**

This chapter is an overview of a few of the more advanced features found in MPI. The goal of this chapter is not to make you an expert on any of these features but simply to make you aware that they exist. You should come away with a basic understanding of what they are and how they might be used. The four sections in this chapter describe additional MPI features that provide greater control for some common parallel programming tasks.

- If you want more control when exchanging messages, the first section describes MPI commands that provide non-blocking and bidirectional communications.
- If you want to investigate other collective communication strategies, the second section describes MPI commands for distributing data across the cluster or collecting data from all the nodes in a cluster.
- If you want to create custom communication groups, the third section describes how it is done.

- If you want to group data to minimize communication overhead, the last section describes two alternatives—packed data and user-defined types.

While you may not need these features for simple programs, as your projects become more ambitious, these features can make life easier.

In Chapter 13, you were introduced to point-to-point communication, the communication between a pair of cooperating processes. The two most basic commands used for point-to-point communication are **`MPI_Send`** and **`MPI_Recv`**. Several variations on these commands that can be helpful in some contexts are described in this section.

One major difference among point-to-point commands is how they handle buffering and the potential for blocking. **`MPI_Send`** is said to be a blocking command since it will wait to return until the send buffer can be reclaimed. At a minimum, the message has to be copied into a system buffer before

More on Point-to-Point Communication

In Chapter 13, you were introduced to point-to-point communication, the communication between a pair of cooperating processes. The two most basic commands used for point-to-point communication are **`MPI_Send`** and **`MPI_Recv`**. Several variations on these commands that can be helpful in some contexts are described in this section.

One major difference among point-to-point commands is how they handle buffering and the potential for blocking. **`MPI_Send`** is said to be a blocking command since it will wait to return until the send buffer can be reclaimed. At a minimum, the message has to be copied into a system buffer before **`MPI_Send`** will return. Similarly, **`MPI_Recv`** blocks until the receive buffer actually contains the contents of the message.

Section 14.1.1.1: MPI_Isend and MPI_Irecv

Although more complicated to use, non-blocking versions of **`MPI_Send`** and **`MPI_Recv`** are included in MPI. These are **`MPI_Isend`** and **`MPI_Irecv`**. (The "I" denotes an immediate return.) With the non-blocking versions, the communication operation is begun or, in the parlance, a message is *posted*. At some later point, the program must explicitly complete the operation. Several functions are provided to complete the operation, the simplest being **`MPI_Wait`** and **`MPI_Test`**.

`MPI_Isend` takes the same arguments as **`MPI_Send`** with one exception. **`MPI_Isend`** has had one additional parameter at the end of its parameter list. This is a request handle, an opaque object that is used in future references to this message exchange. That is, the handle identifies the pending operation. (Handles are of type **`MPI_Request`**.) In **`MPI_Irecv`** the status parameter, which is now found in **`MPI_Wait`**, has been replaced by a request handle. Otherwise, the parameters to **`MPI_Irecv`** are the same as **`MPI_Recv`**.

Section 14.1.1.2: MPI_Wait

MPI_Wait takes two arguments. The first is the request handle just described; the second is a status variable, which contains the same information and is used in exactly the same way as in

More on Collective Communication

Unlike point-to-point communication, collective communication involves every process in a communication group. In Chapter 13, you saw two examples of collective communication functions, **MPI_Bcast** and **MPI_Reduce** (along with **MPI_Allreduce**). There are two advantages to collective communication functions. First, they allow you to express a complex operation using simpler semantics. Second, the implementation may be able to optimize the operations in ways not available with simple point-to-point operations.

Collective operations fall into three categories: a barrier synchronization function, global communication or data movement functions (e.g., **MPI_Bcast**), and global reduction or collective computation functions (e.g., **MPI_Reduce**). There is only one barrier synchronization function, **MPI_Barrier**. It serves to synchronize the processes. No data is exchanged. This function is described in Chapter 17. All of the other collective functions are nonsynchronous. That is, a collective function can return as soon as its role in the communication process is complete. Unlike point-to-point operations, nonsynchronous mode is the only mode supported by collective functions.

While collective functions don't have to wait for the corresponding functions to execute on other nodes, they may block while waiting for space in system buffers. Thus, collective functions come only in blocking versions.

The requirements that all collective functions be blocking, nonsynchronous, and support only one communication mode simplify the semantics of collective operations. There are other features in the same vein: no tag argument is used, the amount of data sent must exactly match the amount of data received, and every process must call the function with the same arguments.

After **MPI_Bcast** and **MPI_Reduce**, the two most useful collective operations are **MPI_Gather** and **MPI_Scatter**.

Section 14.2.1.1: MPI_Gather

Managing Communicators

Collective communication simplifies the communication process but has the limitation that you must communicate with every process in the communicator or communication group. There are times when you may want to communicate with only a subset of available processes.

For example, you may want to divide your processes so that different groups of processes work on different tasks. Fortunately, the designers of MPI foresaw that possibility and included functions that allow you to define and manipulate new communicators. By creating new communicators that are subsets of your original communicator, you'll still be able to use collective communication. This ability to create and manipulate communicators has been described as MPI's key distinguishing feature, i.e., what distinguishes MPI from other message passing systems.

Communicators are composed of two parts: a group of processes and a context. New communicators can be built by manipulating an existing communicator or by taking the group from an existing communicator and, after modifying that group, building a new communicator based on that group. The default communicator **`MPI_COMM_WORLD`** is usually the starting point, but once you have other communicators, you can use them as well.

MPI provides a number of functions for manipulating groups. The simplest way to create a new group is to select processes from an existing group, either by explicitly including or excluding processes. In the following example, process 0 is excluded from the group associated with **`MPI_COMM_WORLD`** to create a new group. You might want to do this if you are organizing your program using one process as a master, typically process 0, and all remaining processes as workers. This is often called a master/slave algorithm. At times, the slave processes may need to communicate with each other without including process 0. By creating a new communicator (**`newComm`** in the following example), you can then carry out the communication using collective functions.

Packaging Data

Since communication is expensive, the fewer messages sent, the better your program performance will be. With this in mind, MPI provides several ways of packaging data. This allows you to maximize the amount of information exchanged in each message. There are three basic strategies.

Although we glossed over it, you've already seen one technique. You'll recall that the messaged package in **`MPI_Send`** consists of a buffer address, a count, and a data type. Clearly, this mechanism can be used to send multiple pieces of information as a single message, provided they are of the same type. For example, in our first interactive version of the numerical integration program, three calls to **`MPI_Send`** were used to distribute the values of **`numberRects`**, **`lowerLimit`**, **`upperLimit`** to all the processes.

```
MPI_Send(&numberRects, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
MPI_Send(&lowerLimit, 1, MPI_DOUBLE, dest, 1, MPI_COMM_WORLD);
MPI_Send(&upperLimit, 1, MPI_DOUBLE, dest, 2, MPI_COMM_WORLD);
```

We could have eliminated one of these calls by putting **lowerLimit** and **upperLimit** in an array and sending it in a single call.

```
        params[0] = lowerLimit;
        params[1] = upperLimit;
        MPI_Send(params, 2, MPI_DOUBLE, dest, 1,
MPI_COMM_WORLD);
```

If you do this, don't forget to declare the array **params** and to make corresponding changes to call to **MPI_Recv** to retrieve the data from the array.

For this to work, items must be in contiguous locations in memory. While this is true for arrays, there are no guarantees for variables in general. Hence, using an array was necessary. This is certainly a legitimate way to write code and, when sending blocks of data, is very reasonable and efficient. In this case we've removed only one call so its value is somewhat dubious. Furthermore, we weren't able to include **numberRects** since it is an integer rather than a double.

It might seem that a structure would be a logical way around this last problem since the elements in a structure are guaranteed to be in contiguous memory. Before a structure can be used in an MPI function, however, it is necessary to define a new MPI type. Fortunately, MPI provides a mechanism to do just that.

Chapter 15: **Designing Parallel Programs**

There are no silver bullets for parallel program design. While many parallel programs may appear to match one of several standard parallel program designs, every significant program will have its own quirks that make it unique. Nevertheless, parallel program design is the essential first step in writing parallel programs. This chapter will introduce you to some of the basics. This should provide help in getting started. Just remember there is a lot more to learn.

We are going to look at a couple of different ways of classifying or approaching problems in this chapter. While there is considerable overlap, these various schemes will provide you with different perspectives in the hope that they at least will suggest a solution or approach that may fit your individual needs.

Algorithm design is a crucial part of the development process for parallel programs. In many cases, the best serial algorithm can be easily parallelized, while in other cases a fundamentally different algorithm will be needed. In this chapter, we'll focus on parallelizing a serial algorithm. Keep in mind that this may not provide the best solution to your problem. There are a number of very detailed books on parallel algorithm design, parallel programming in general, and on MPI programming in particular. Most have extensive examples. Whenever possible, you should look for an existing, optimized solution rather than trying to develop

your own. This is particularly true when faced with a problem that requires an algorithm that is fundamentally different from the serial algorithm you might use. Don't reinvent the wheel.

The optimal algorithm will depend on the underlying architecture that is used. For parallel programming, most algorithms will be optimized for either a shared memory architecture—a scheme where global memory is shared among all processes—or a message passing architecture. If you are looking at existing algorithms, be sure to take this into account.

Since this is a book about clusters, we will be looking at parallel program design from the perspective of message passing. This isn't always the best approach for every problem, but it is the most common for use with a cluster.

Overview

Algorithm design is a crucial part of the development process for parallel programs. In many cases, the best serial algorithm can be easily parallelized, while in other cases a fundamentally different algorithm will be needed. In this chapter, we'll focus on parallelizing a serial algorithm. Keep in mind that this may not provide the best solution to your problem. There are a number of very detailed books on parallel algorithm design, parallel programming in general, and on MPI programming in particular. Most have extensive examples. Whenever possible, you should look for an existing, optimized solution rather than trying to develop your own. This is particularly true when faced with a problem that requires an algorithm that is fundamentally different from the serial algorithm you might use. Don't reinvent the wheel.

The optimal algorithm will depend on the underlying architecture that is used. For parallel programming, most algorithms will be optimized for either a shared memory architecture—a scheme where global memory is shared among all processes—or a message passing architecture. If you are looking at existing algorithms, be sure to take this into account.

Since this is a book about clusters, we will be looking at parallel program design from the perspective of message passing. This isn't always the best approach for every problem, but it is the most common for use with a cluster.

Parallel algorithms are more complicated than serial algorithms. While a serial algorithm is just a sequence of steps, a parallel algorithm must also specify which steps can be executed in parallel and provide adequate control mechanisms to describe the concurrency.

The process of parallel algorithm design can be broken into several steps. First, we must identify the portions of the code that can, at least potentially, be executed safely in parallel. Next, we must devise a plan for mapping those parallel portions into individual processes (or onto individual processors). After that, we need to address the distribution of data as well as the collection and consolidation of results. This step also includes addressing any

synchronization issues that might arise, which must be done so that we can, finally, synchronize the execution of the processes.

Problem Decomposition

When decomposing a program, we will talk in terms of `tasks` . The meaning of this word may vary slightly depending upon context. Typically, a task is a portion of a program that can be executed as a unit. It may be used to mean that part of a program that can become an independent process, or it may be used to mean a piece of the work that that process will execute. It should be clear from context which meaning is intended.

Let's begin by looking at some of the issues involved in decomposing a problem into parallelizable parts. The first issue we must face is `task granularity`. Depending on the problem, a task may be broken into very small pieces (fine granularity), into relatively large pieces (coarse granularity), or into a mixture of pieces of varying sizes.

Granularity, in one sense, establishes a limit on how many compute nodes or processors you may be able to use effectively. For example, if you are multiplying two 10 by 10 matrices, then you will need to do 100 multiplications. Since you won't be able to subdivide a multiplication, you won't be able to divide this problem into more than 100 pieces. Consequently, having more than 100 processors won't allow you to do the multiplications any faster. In practice, the number of processors you can effectively use will be lower. It is essential to realize that there are a number of trade-offs that must be balanced when dividing a problem. In particular, coarse granularity tends to limit communication overhead but may result in increased idle time and poor processor utilization. We will discuss each of these concerns in detail in this chapter.

We can also speak of the `degree of concurrency` , i.e., the number of tasks that can execute at the same time. Realize that this will vary during programming execution depending on the point you are at in the program. Thus, it is often more meaningful to talk about the maximum or the average degree of concurrency of a program. Generally, both the maximum and average concurrency are larger with fine-grained than coarse-grained problems.

Mapping Tasks to Processors

Being able to decompose a problem is only the first step. You'll also need to be able to map the individual tasks to different processors in your cluster. This is largely a matter of developing appropriate control structures and communication strategies. Since the ultimate goal is to reduce the time to completion, task mapping is largely a balancing act between two conflicting subgoals—the need to maximize concurrency and the need to minimize the overhead introduced with concurrency. This overhead arises primarily from interprocess communications, from process idle time, and to a lesser extent, from redundant calculations.

Consider redundant calculations first. When we separate a program into multiple tasks, the separation may not always go cleanly. Consequently, it may be necessary for each process to do redundant calculations, calculations that could have been done once by a single process. Usually, this doesn't add to the program's overall time to completion since the rest of the processes would have been idle while a single process did the calculation. In fact, having the individual processors each do the calculation may be more efficient since it eliminated the communication overhead that would be required to distribute the results of the calculation. However, this is not always the case, particularly with asymmetric processes. You should be aware of this possibility.

Communication overhead is a more severe problem. Returning to the matrix multiplication example, while we might obtain maximum concurrency by having a different processor for each of the 100 multiplications, the overhead of distributing the matrix elements and collecting the results would more than eliminate any savings garnered from distributing the multiplications. On the other hand, if we want to minimize communication overhead, we could package everything in one process. While this would eliminate any need for communication, it would also eliminate all concurrency. With most problems, the best solution usually (but not always) lies somewhere between maximizing concurrency and minimizing communication.

Other Considerations

The issues we have examined up to this point are fairly generic. There are other programming-specific issues that may need to be addressed as well. In this section, we will look very briefly at two of the more common of these—parallel I/O and random numbers. These are both programming tasks that can cause particular problems with parallel programs. You'll need to take care whenever your programs use either of these. In some instances, dealing with these issues may drive program design.

Large, computationally expensive problems that require clusters often involve large data sets. Since I/O is always much more costly than computing, dealing with large data sets can severely diminish performance and must be addressed.

There are several things you can do to improve I/O performance even before you start programming. First, you should buy adequate I/O hardware. If your cluster will be used for I/O-intensive tasks, you need to pay particular attention when setting up your cluster to ensure you are using fast disks and adequate memory. Next, use a fast filesystem. While NFS may be an easy way to get started with clusters, it is very slow. Other parallel filesystems optimized for parallel performance should be considered, such as PVFS, which is described in Chapter 12.

When programming, if memory isn't a problem, it is generally better to make a few large requests rather than a larger number of smaller requests. Design your programs so that I/O is

distributed across your processes. Because of historical limitations in parallel I/O systems, it is typical for parallel programs to do I/O from a single process. Ideally, you should use an interface, such as MPI-IO, that spreads I/O across the cluster and has been optimized for parallel I/O.

The standard Unix or POSIX filesystem interface for I/O provides relatively poor performance when used in a parallel context, since it does not support collective operations and does not provide noncontiguous access to files. While the original MPI specification avoided the complexities of I/O, the MPI-2 specification dealt with this issue. The MPI-2 specification for parallel I/O (Chapter 9 of the specification) is often known as the MPI-IO. This standard was the joint work of the Scalable I/O Initiative and the MPI-IO Committee through the MPI Forum.

Chapter 16: **Debugging Parallel Programs**

If you are using a cluster, you are probably dealing with large, relatively complicated problems. As problem complexity grows, the likelihood of errors grows as well. In these circumstances, debugging becomes an increasingly important skill. It is a simple fact of life—if you write code, you are going to have to debug it.

In this chapter, we'll begin by looking at why debugging parallel programs can be challenging. Next, we'll review debugging in general. Finally, we'll look at how the traditional serial debugging approaches can be extended to parallel problems. Parallel debugging is an active research area, so there is a lot to learn. We'll stick to the basics here.

Parallel code presents new difficulties, and the task of coordinating processes can result in some novel errors not seen in serial code. While elaborate classification schemes for parallel problems exist, there are two broad categories of errors in parallel code that you are likely to come up against. These are synchronization problems that stem from inherent nondeterminism found in parallel code and deadlock. While we can further subclassify problems, you shouldn't be too concerned about finer distinctions. If you can determine the source of error and how to correct it, you can leave the classification to the more academically inclined.

Synchronization problems result from variations in the order that instructions may be executed when spread among multiple processes. By contrast, serial programs are deterministic, executing each line of code in the order it was written. Once you start forking off processes, all bets are off. Moreover, since the loads on machines fluctuate, as does the competition for communications resources, the timing among processes can vary radically from run to run. One process may run before another process one day and lag behind it the next. If the order of execution among cooperating processes is important, this can lead to problems. For example, the multiplication of matrices is not commutative. If you are multiplying a chain of matrices, you'll need to explicitly control the order in which the

multiplications occur when dividing the problem among the processes. Otherwise, a race condition may exist among processes.

Debugging and Parallel Programs

Parallel code presents new difficulties, and the task of coordinating processes can result in some novel errors not seen in serial code. While elaborate classification schemes for parallel problems exist, there are two broad categories of errors in parallel code that you are likely to come up against. These are synchronization problems that stem from inherent nondeterminism found in parallel code and deadlock. While we can further subclassify problems, you shouldn't be too concerned about finer distinctions. If you can determine the source of error and how to correct it, you can leave the classification to the more academically inclined.

Synchronization problems result from variations in the order that instructions may be executed when spread among multiple processes. By contrast, serial programs are deterministic, executing each line of code in the order it was written. Once you start forking off processes, all bets are off. Moreover, since the loads on machines fluctuate, as does the competition for communications resources, the timing among processes can vary radically from run to run. One process may run before another process one day and lag behind it the next. If the order of execution among cooperating processes is important, this can lead to problems. For example, the multiplication of matrices is not commutative. If you are multiplying a chain of matrices, you'll need to explicitly control the order in which the multiplications occur when dividing the problem among the processes. Otherwise, a race condition may exist among processes.

Deadlock occurs when two or more processes are waiting on each other for something. For example, if process A is waiting for process B to send it information before it can proceed, and if process B is waiting for information from process A before it can proceed, then neither process will be able to advance and send the other process what it needs. Both will wait, very patiently, for the other to act first. While this may seem an obvious sort of problem that should be easy to spot, deadlock can involve a chain of different processes and may depend on a convoluted path through conditional statement in code. As such, it can occur in very nonobvious ways. A variant of deadlock is livelock, where the process is still busy computing but can't proceed beyond some point.

Avoiding Problems

I would be remiss if I didn't begin with the usual obligatory comments about avoiding bugs in the first place. Life will be much simpler if you can avoid debugging. While this is not always possible, there are several things you can do to minimize the amount of debugging you'll need.

- Carefully design your program before you begin coding.

- Be willing to scrap what you've done and start over.
- Comment your code and use reasonable naming conventions.
- Don't try to get too clever.
- Develop and test your code incrementally.
- Never try to write code when you are fatigued or distracted.
- Master all the programming tools that are available to you.

Of course, you already knew all of this. But sometimes it doesn't hurt to badger someone just a little.

Programming Tools

On most systems, a number of debugging tools are readily available. Others can be easily added. While most are designed to work with serial code, they are still worth mastering, since most of your errors will be serial in nature.

First, you should learn to use the features built into your programming language. For example, in C you might use asserts to verify the correct operation of your code. You should also learn how to write error handlers. This advice extends beyond the language to any libraries you are using. For example, MPI provides two error handlers, **`MPI_ERROR_ARE_FATAL`** and **`MPI_ERRORS_RETURN`**. And the MPICH implementation defines additional error handlers. While we have been ignoring them in our programming examples in order to keep the code as simple as possible, almost all MPI functions return error codes.

Next, learn to use the features provided by your compiler. Most compilers provide a wealth of support that is only a compile option or two away. Since the added checking increases compile time, these are generally disabled by default. But if you take the time to read the documentation, you'll find a lot of useful features. For example, with `gcc` you can use the options **`-Wall`** to turn on a number of (but not all) warnings, **`-ansi`** to specify the language standard to use, and **`-pedantic`** to issue all mandatory diagnostics, including those frequently omitted. `mpicc` will pass options like these on to the underlying compiler, so you can use them when compiling MPI programs. When using these, you'll likely see a number of warning messages that you can safely ignore, but you may find a pearl or two as well. Keep in mind that there are a large number of additional options available with `gcc`, so be sure to read the documentation.

Additionally, many systems have other utilities that can be helpful. The granddaddy of them all is `lint` . This is a program that analyzes code for potential errors with which most older compilers didn't bother. Most of the problems that

Rereading Code

There are three basic escalating strategies for locating errors—rereading code, printing information at key points, and using a symbolic debugger. There is an interesting correspondence between these debugging strategies and search strategies, i.e., linear search, binary search, and indexed search. When reading code we are searching linearly for the error. Printing works best when we take a binary approach. Through the breakpoints a symbolic debugger provides, we are often able to move directly to a questionable line of code.

Rereading (or reading for the first time in some cases) means looking at the code really hard with the hope the error will jump out at you. This is the best approach for new code since you are likely to find a number of errors as well as other opportunities to improve the code. It also works well when you have a pretty good idea of where the problem is. If it is a familiar error, if you have just changed a small segment of code, or if the error could only have come from one small segment of code, rereading is a viable approach.

Rereading relies on your repeatedly asking the question, "If I were a computer, what would I do?" You can still play this game with a cluster, you just have to pretend to be several computers at once and keep everything straight. With a cluster, the order of operations is crucial. If you take this approach, you'll need to take extra care to ensure that you don't jump beyond a point in one process that relies on another process without ensuring the other process will do its part. An example may help explain what I mean.

As previously noted, one problem you may encounter with a parallel program is deadlock. For example, if two processes are waiting to receive from each other before sending to each other, both will be stalled. It is very easy when manually tracing a process to skim right over the receive call, assuming the other process has sent the necessary information. Making that type of assumption is what you must guard against when pretending to be a cluster of computers. Here is an example:

Tracing with printf

Printing information at key points is a way of tracing or following the execution of the code. With C code, you stick **`printf`**'s throughout the code that let you know you've reached a particular point in the code or tell you what the value of a variable is. By using this approach, you can zero in on a crucial point in the program and see the value of parameters that may affect the execution of the code. This quick and dirty approach works best when you already have an idea of what might be going wrong. But if you are clueless as to where the problem is, you may need a lot of print statements to zero in on the problem, particularly with large programs. Moreover, it is very easy for the truly useful information to get lost in the deluge of output you create.

On the other hand, there is certainly nothing wrong with printing information that provides the user with some sense of progress and an indication of how the program is working. We did this in our numerical integration program when we printed the process number and the individual areas calculated by each process.

It can be particularly helpful to echo values that are read into the program to ensure that they didn't get garbled in the process. For example, if you've inadvertently coerced a floating point number into an integer, the truncation that occurs will likely cause problems. By printing the value, you may be alerted to the problem.

Including print statements can also be helpful when you are working with complicated data structures since you will be able to format the data in meaningful ways. Examining a large array with a symbolic debugger can be challenging. Since it is straightforward to conditionally print information, print statements can be helpful when the data you are interested in is embedded within a large loop and you want to examine it only under selective conditions.

In developing code, programmers will frequently write large blocks of diagnostic code that they will discard once the code seems to be working. When the code has to be changed at a later date, they will often find themselves rewriting similar code as new problems arise. A better solution is to consider the diagnostic code a key part of the development process and keep it in your program. By using conditional compile directives, the code can be disabled in production versions so that program efficiency isn't compromised, but can be enabled easily should the need arise.

Symbolic Debuggers

If these first two approaches don't seem to be working for you, it's time to turn to a symbolic debugger. (Arguably, the sooner you switch to a symbolic debugger, the better.) Symbolic debuggers will allow you to trace the execution of your program, stop and examine variables, make changes, and resume execution. While you'll need to learn how to use them, most are fairly intuitive and don't take long to master. All you really need to do is learn a few basic commands to get started. You can learn more commands as the need arises.

There are a number of symbolic debuggers available, including debuggers that are specifically designed to work with parallel programs such as commercial products like `TotalView`. With a little extra effort, you'll probably be able to get by with some more common debuggers. In this chapter we'll look at `gdb` and `ddd`, first with serial programs and then with parallel programs.

`gdb` is a command-line symbolic debugger from the GNU project. As such, it is freely available. You probably already have it installed on your Linux system. `ddd` is a GUI frontend that can be used with `gdb` (or other debuggers) in an X Window System environment. You may need to install `ddd`, but the process is straightforward and is described in Chapter 9.

To demonstrate gdb, we'll use the program area.c from Chapter 13 with one slight added error. (Also, the macro for **f** has been replaced with a function.) Here is the now buggy code:

```
#include <stdio.h>

/* problem parameters */

#define numberSteps     50
#define lowerLimit      2.0
#define upperLimit      5.0

double f(double x)
{
   return x*x;
}

int main ( int argc, char * argv[  ] )
{
   int  i;
   double       area = 0.0;
   double       step = (upperLimit - lowerLimit) /
numberSteps;
   double       at, height;

   for (i = 0; i <= numberSteps; i--)
   {    at = lowerLimit + i * step + step / 2.0;
           height = f(at);
           area = area + step * height;
   }

   printf ("The area from %f to %f is: %f\n",
        lowerLimit, upperLimit, area );

   return 0;
}
```

Using gdb and ddd with MPI

Thus far we have used the debugger to start the program we want to debug. But with MPI programs, we have used `mpirun` or `mpiexec` to start programs, which would seem to present a problem. Fortunately, there is a second way to start `gdb` or `ddd` that hasn't been described yet. If a process is already in execution, you can specify its process number and attach `gdb` or `ddd` to it. This is the key to using these debuggers with MPI.

With this approach you'll start a parallel application the way you normally do and then attach to it. This means the program is already in execution before you start the debugger. If it is a

very short program, then it may finish before you can start the debugger. The easiest way around this is to include an input statement near the beginning. When the program starts, it will pause at the input statement waiting for your reply. You can easily start the debugger before you supply the required input. This will allow you to debug the program from that point. Of course, if the program is hanging at some point, you won't have to be in such a hurry.

Seemingly, a second issue is which cluster node to run the debugger on. The answer is "take your pick." You can run the debugger on each machine if you want. You can even run different copies on different machines simultaneously.

This should all be clearer with a couple of examples. We'll look at a serial program first—the flawed area program discussed earlier in this chapter. We'll start it running in one window.

```
[sloanjd@amy DEBUG]$ ./area
```

Then, in a second widow, we'll look to see what its process number is.

```
[sloanjd@amy DEBUG]$ ps -aux | grep area
sloanjd  19338 82.5  0.1  1340  228 pts/4    R    09:57   0:32
./area
sloanjd  19342  0.0  0.5  3576  632 pts/3    S    09:58   0:00
grep area
```

If it takes you several tries to debug your program, watch out for zombie processes and be sure to kill any extraneous or hung processes when you are done.

Notes for OSCAR and Rocks Users

gdb is part of the default Linux installation and should be available on your system. You will need to add ddd to your system if you wish to use it. Since OSCAR installs X Window System only on the head node, you will not be able to run ddd on your compute nodes. Rather, you will need to run gdb on your compute node as shown in the last example in this chapter.

gdb and ddd are included with Rocks on the frontend and compute nodes. However, you'll need to forward ddd sessions to the frontend using the DISPLAY environment variable since the X Window System is not set up to run locally on compute nodes.

Chapter 17: **Profiling Parallel Programs**

Since the raison d'être for a cluster is higher performance, it stands to reason that if you really need a cluster, writing efficient code should be important to you. The key to improving the

efficiency of your code is knowing where your code spends its time. Thus, the astute cluster user will want to master code profiling. This chapter provides an introduction to profiling in general, to the problems you'll face with parallel programs, and to some of the tools you can use.

We'll begin by looking briefly at issues that impact program efficiency. Next, we'll look at ways you can time programs (and parts of programs) using readily available tools and the special features of MPI. Finally, we'll look at the MPE library, a library that extends MPI and is particularly useful for profiling program performance. Where appropriate, we'll look first at techniques typically used with serial programs to put the techniques in context, and then at extending them to parallel programs.

You have probably heard it before—the typical program will spend over 90% of its execution time in less that 10% of the actual code. This is just a rule of thumb or heuristic, and as such, will be wildly inaccurate or totally irrelevant for some programs. But for many, if not most, programs, it is a reasonable observation. The actual numbers don't matter since they will change from program to program. It is the idea that is important—for most programs, most of the execution time spent is in a very small portion of the code.

This is extremely important to keep in mind in this critical portion of code. If your application spends 95% of its time in 5% of the code, there is little to be gained by optimizing the other 95% of the code. Even if you could completely eliminate it, you'd only see a 5% improvement. But if you can manage a 10% improvement in the critical 5% of your code, for example, you'll see a 9.5% overall improvement in your program. Thus, the key to improving your code's performance is to identify that crucial 5%. That's where you should spend your time optimizing code.

Why Profile?

You have probably heard it before—the typical program will spend over 90% of its execution time in less that 10% of the actual code. This is just a rule of thumb or heuristic, and as such, will be wildly inaccurate or totally irrelevant for some programs. But for many, if not most, programs, it is a reasonable observation. The actual numbers don't matter since they will change from program to program. It is the idea that is important—for most programs, most of the execution time spent is in a very small portion of the code.

This is extremely important to keep in mind in this critical portion of code. If your application spends 95% of its time in 5% of the code, there is little to be gained by optimizing the other 95% of the code. Even if you could completely eliminate it, you'd only see a 5% improvement. But if you can manage a 10% improvement in the critical 5% of your code, for example, you'll see a 9.5% overall improvement in your program. Thus, the key to improving your code's performance is to identify that crucial 5%. That's where you should spend your time optimizing code.

Keep in mind that there is a point of diminishing returns when optimizing code. You'll need to balance the amount of time you spend optimizing code with the amount of improvement you actually get. There is a point where your code is good enough. The goals of profiling are two-fold—to decide how much optimization is worth doing and to identify which parts of code should be optimized.

The first step to optimizing code begins before you start writing it. To write the most efficient code, you should begin by selecting the most appropriate or efficient algorithm. As the program size grows, an unoptimized $O(n \log_2 n)$ algorithm will often outperform an optimized $O(n2)$ algorithm. Of course, algorithm selection will depend on your specific application. Unfortunately, it can be problematic for parallel applications.

For serial algorithms, you can often make reasonable estimates on how time is being spent by simply examining and analyzing the algorithm. The standard approach characterizes performance using some measurement of the problem size. For example, when sorting an array of numbers, the problem size would be the number of elements in the array. Some problems are easily characterized by a single number while others may be more difficult to characterize or may depend on several parameters. Since the problem size often provides a bound for algorithmic performance, this approach is sometimes called

Writing and Optimizing Code

Code optimization can be done by hand or by the compiler. While you should avoid writing obviously inefficient code, you shouldn't get carried away doing hand optimizations until you've let your compiler have a try at optimizing your code. You are usually much better off writing clean, clear, maintainable code than writing baroque code that saves a few cycles here or there. Most modern compilers, when used with the appropriate compiler options, are very good at optimizing code. It is often possible to have the best of both worlds—code that can be read by mere mortals but that compiles to a fully optimized executable.

With this in mind, take the time to learn what optimization options are available with your compiler. Because it takes longer to compile code when optimizing, because time-optimized code can be larger than unoptimized code, and because compiler optimizations may reorder instructions, making code more difficult to debug and profile, compilers typically will not optimize code unless specifically directed to do so.

With `gcc`, the optimization level is set with the **`-O`** compiler flag. (That's the letter O.) With the flag **`-O1`**, most basic optimizations are done. More optimizations are done when the **`-O2`** flag is used and still more with the **`-O3`** flag. (**`-O0`** is used to suppress optimization and **`-Os`** is used to optimize for size.) In addition to these collective optimizations, `gcc` provides additional flags for other types of optimizations, such as loop unrolling, that might be useful in some situations. Consult your compiler's documentation for particulars.

If you have selected your algorithm carefully and your compiler has done all it can for you, the next step in optimizing code is to locate what portions of the code may benefit from further attention. But locating the hot spots in your code doesn't mean that you'll be able to eliminate them or lessen their impact. You may be working with an inherently time-consuming problem. On the other hand, if you don't look, you'll never know.

Timing Complete Programs

With many programs, the first and most logical step is simply to time how long the program takes to execute from beginning to end. The total elapsed time is usually called the program's *wall-clock time* . While the wall-clock time reflects a number of peripheral concerns such as system loads caused by other users, it really is the bottom line. Ultimately, what you are really interested in is how long you are going to have to wait for your answers, and this is just what the wall-clock time measures.

Linux shells typically provide an internal timing command, usually called `time` . This command measures the total execution time for a program when executed by the shell. Here is an example with the `bash` shell:

```
[sloanjd@amy PROFILE]$ time ./demo

real    0m6.377s
user    0m5.350s
sys     0m0.010s
```

In this example, the program `demo` ran for a total of 6.377 seconds. This number is the total elapsed time or wall-clock time. Of that time, it spent 5.350 seconds executing the user or non-kernel mode and another 0.010 seconds for system calls or in kernel mode. The difference between the elapsed or real time and the sum of the **`user`** and **`sys`** times is time spent by the system doing computing for other tasks.

While most Unix shells provide a timing command, different shells provide different levels of information. Here is the same program timed under the C shell.

```
[sloanjd@amy PROFILE]$ csh
[sloanjd@amy ~/PROFILE]$ time ./demo
5.340u 0.000s 0:06.37 83.8%     0+0k 0+0io 65pf+0w
[sloanjd@amy ~/PROFILE]$ exit
exit
```

In addition to user, system, and wall-clock times, with the C shell you also get percent of CPU time (83.8% in this example), shared and unshared memory usage (0 and 0), block input and output operations (0 and 0), number of page faults (65), and number of swaps (0).

With some shells such as the Korn shell, there is another timer, `timex`. `timex`, when used with the **`-s`** option, provides still more information. See the appropriate manpage for more details.

Timing C Code Segments

The primary limitation to the various versions of `time` is that they don't tell you what part of your code is running slowly. To know more, you'll need to delve into your code. There are a couple of ways this can be done. The most straightforward way is to "instrument" the code—that is, to embed commands directly into the code that record the system time at key points and then to use these individual times to calculate elapsed times.

The primary advantage to manual instrumentation of code is total control. You determine exactly what you want or need. This control doesn't come cheap. There are several difficulties with manual instrumentation. First and foremost, it is a lot of work. You'll need to add variables, determine collection points, calculate elapsed times, and format and display the results. Typically, it will take several passes to locate the portion of code that is of interest. For a large program, you may have a number of small, critical sections that you need to look at. Once you have these timing values, you'll need to figure out how to interpret them. You'll also need to guard against altering the performance of your program. This can be a result of over-instrumenting your code, particularly at critical points. Of course, these problems are not specific to manual instrumentation and will exist to some extent with whatever approach you take.

The traditional way of instrumenting C code is with the **`time`** system call, provided by the `time.h` library. Here is a code fragment that demonstrates its use:

```
...
#include <sys/time.h>

int main(void)
{
   time_t start, finish;
   ...
   time(&start);
   /* section to be timed */
   ...
   time(&finish);
   printf("Elapsed time: %d\n", finish - start);
   ...
}
```

The time function returns the number of seconds since midnight (GMT) January 1, 1970. Since this is a very large integer, the type **`time_t`** (defined in `<sys/times.h>`) can be

used to ensure that time variables have adequate storage. While easy to use if it meets your needs, the primary limitation for

Profilers

Thus far we have been looking at timing code manually. While this provides a lot of control, it is labor intensive. The alternative to manual timing is to use a profiler. A profiler attempts to capture the profile of a program in execution; that is, a set of timings for an application that maps where time is spent within the program. While with manual timing you'll want to focus in on part of a program, a profiler typically provides information for the entire application in one fell swoop. While a profiler may give more results than you actually need, you are less likely to overlook a hotspot in your code using a profiler, particularly when working with very complicated programs. Most profilers are easy to use and may give you some control over how much information is collected. And most profilers not only collect information, but provide a mechanism for analyzing the results. Graphical output is common, a big help with large, complicated programs.

There are a number of profilers available, particularly if you include commercial products. They differ in several ways, but it usually comes down to a question of how fine a granularity you want and how much detail you need. Choices include information on a line-by-line basis, information based on the basic blocks, or information based on function calls or modules. Profilers may provide timing information or simply count the number of times a statement is executed.

There are two basic categories for profiles—active and passive. (Some profilers, such as `gprof`, have features that span both categories.) A passive profiler gathers information without modifying the code. For example, a passive profiler might collect information by repeatedly sampling the program counter while the program is running. By installing an interrupt service routine that wakes up periodically and examines the program counter, the profiler can construct a statistical profile for the program.

While passive profiles are less intrusive, they have a couple of problems. First, they tend to provide a flat view of functions within an application. For example, if you have a function that is called by several different functions, a passive profiler will give you an idea of how often the function is called, but no information about what functions are making the call. Second, passive profilers are inherently statistical. Key performance issues are how often you sample and how many samples are taken. The quality of your results will depend on getting these parameters right.

MPE

If `gprof` and `gcov` seem too complicated for routine use, or if you just want to investigate all your possibilities, there is another alternative you can consider—*Multi-Processing*

Environment (MPE). If you built MPICH manually on your cluster, you already have MPE. If you installed MPICH as part of OSCAR, you'll need to add MPE. Fortunately, this is straightforward and is described in Chapter 9. Although MPE is supplied with MPICH, it can be used with other versions of MPI.

MPE provides several useful resources. First and foremost, it includes several libraries useful to MPI programmers. These include a library of routines that create logfiles for profiling MPI programs. It also has a tracing library and a real-time animation library that are useful when analyzing code. MPE also provides a parallel X graphics library. There are routines than can be used to ensure that a section of code is run sequentially. There are also debugger setup routines. While this section will focus on using logfiles to profile MPI program performance, remember that this other functionality is available should you need it.

MPE's logging capabilities can generate three different logfile formats—`ALOG`, `CLOG`, and `SLOG`. ALOG is an older ASCII-based format that is now deprecated. CLOG is the current default format, while SLOG is an emerging standard. Unlike SLOG, CLOG does not scale well and should be avoided for large files.

MPE includes four graphical visualization tools that allow you to examine the logfiles that MPE creates, `upshot`, `nupshot`, `jumpshot-2`, and `jumpshot-3`. The primary differences between these four tools are the file formats they read and their implementation languages.

upshot

This tool reads and displays ALOG files and is implemented in Tcl/Tk.

Customized MPE Logging

If you want more control over the information that MPE supplies, you can manually instrument your code. This can be done in combination with MPE default logging or independently. Here is an example of adding MPE command to `rect2.c`, a program you are already familiar with. The new MPE commands are in boldface. (You'll notice a few other minor differences as well if you look closely at the code.)

```
#include "mpi.h"
#include "mpe.h"
#include <stdio.h>

/* problem parameters */
#define f(x)            ((x) * (x))
#define numberRects     50
#define lowerLimit      2.0
#define upperLimit      5.0
```

```
int main( int argc, char * argv[  ] )
{
   /* MPI variables */
   int dest, noProcesses, processId, src, tag;
   int evnt1a, evnt1b, evnt2a, evnt2b, evnt3a, evnt3b, evnt4a,
evnt4b;
   double start, finish;
   MPI_Status status;

   /* problem variables */
   int         i;
   double      area, at, height, lower, width, total, range;

   /* MPI setup */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
   MPI_Comm_rank(MPI_COMM_WORLD, &processId);

   if (processId = = 0) start = MPI_Wtime( );

   MPE_Init_log( );

               /*  Get event ID from MPE  */
                 evnt1a = MPE_Log_get_event_number( );
                 evnt1b = MPE_Log_get_event_number( );
                 evnt2a = MPE_Log_get_event_number( );
                 evnt2b = MPE_Log_get_event_number( );
                 evnt3a = MPE_Log_get_event_number( );
                 evnt3b = MPE_Log_get_event_number( );
                 evnt4a = MPE_Log_get_event_number( );
                 evnt4b = MPE_Log_get_event_number( );

                 if (processId = = 0) {
                 MPE_Describe_state(evnt1a, evnt1b, "Setup",
"yellow");
                 MPE_Describe_state(evnt2a, evnt2b,
"Receive", "red");
                 MPE_Describe_state(evnt3a, evnt3b,
"Display", "blue");
                 MPE_Describe_state(evnt4a, evnt4b, "Send",
"green");
```

Notes for OSCAR and Rocks Users

With OSCAR, you should have all of the basic commands described earlier in this chapter including `gprof` and `gcov`. Both MPICH and LAM/MPI are installed under the `/opt` directory with OSCAR. The MPI commands are readily available, but you'll need to install

MPE if you wish to use it. Rocks also includes `gprof` and `gcov`. Several different MPICH releases are included under `/opt`. MPE is installed but you will need to configure the viewers. More information on setting up MPE is included in Chapter 9.

Appendix A: **References**

While these listings are far from complete, they are the sources that I found the most useful and should certainly keep you busy for a long time.

Books

If you are a new Linux user, the books by Powers or Siever are both good general references. If you want to know more about Linux system administration, my favorite is Nemeth. Frisch, a quicker read but less detailed book, is also a good place to begin. If you need more information on the Linux kernel, Bovet is a reasonable book to look at. For fine-tuning your system, Musumeci is a good resource. For a detailed overview of Unix security issues, you might look at Garfinkel. Limoncelli provides a general overview of system administration practices.

A robust network is a crucial part of any cluster. While general Linux books will take you a long way, at some point you'll need more specialized information than a general administration book can provide. If you want a broad overview of networking, Tanenbaum is very readable. For Ethernet, Spurgeon is a great place to start. If you want more information on TCP/IP, Comer, Hall, and Stevens are all good starting points. For setting up a TCP/IP network, you should consider Hunt. For more information on firewalls, look at Cheswick or Sonnenreich.

Of course, setting up a system will require configuring a number of network services. Hunt provides a very good overview. If you need to delve deeper, there are a number of books dedicated to individual network services, particularly from O'Reilly. For Apache, consider Laurie. For DNS, you won't do better than Albitz. For NFS, look at Callaghan or Stern. For SSH, you might consult Barrett.

For general information on parallel computing, good choices include Culler, Dongarra, and Dowd. Culler is more architecture and performance oriented. Dongarra is a very good source for information on how parallel computing is used. Dowd provides a wealth of information on parallel programming techniques.

For additional information on clusters, the best place to start is Sterling's book. Many of the tools described in this text are discussed by their creators in the book edited by Sterling, listed below. Although uneven at times, parts of Bookman are very helpful.

URLs

These URLs offering software and documentation were current when this book was written. They are grouped roughly by category. Within a category, they are organized roughly in alphabetic order. However, closely related items are grouped together. Most categories are short, so you shouldn't have too much trouble locating an item even if you need to skim the entire category.

http://www.beowulf.org. This site has general information on Beowulf clusters, including tutorials.

http://clustering.foundries.sourceforge.net. Clustering Foundry is a source for cluster software.

http://www.clusterworld.com. This is the web site for *ClusterWorld* magazine.

http://www.dell.com/powersolutions. *Dell Power Solutions Magazine* has frequent articles or special issues devoted to clustering.

http://www.linux-ha.org. This is the home for the Linux High-Availability Project. It provides many links to information useful in setting up an HA cluster.

http://www.lcic.org. Linux Clustering Information Center is a great source of information and links.

http://www.tldp.org. Linux Documentation Project is the home to a vast store of Linux documentation, including FAQs, HOWTOs, and other guides.

http://www.linux-vs.org. This is the home to the Linux Virtual Server Project, another site of interest if you want high availability or load balancing.

http://www.linuxhpc.org. This is the home to LinuxHPC.org, another site to visit for high-performance cluster information.

http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/. This is the Remote Serial Console HOWTO.

http://setiathome.ssl.berkeley.edu. This is the home for the SETI@Home project.

http://www.top500.org

Return to High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI