

- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)
- [Academic](#)

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

By Joseph D. Sloan

Publisher: O'Reilly

Pub Date: November 2004

ISBN: 0-596-00570-9

Pages: 360

[Copyright](#)

[Preface](#)

[Audience](#)

[Organization](#)

[Conventions](#)

[How to Contact Us](#)

[Using Code Examples](#)

[Acknowledgments](#)

[Part I: An Introduction to Clusters](#)

[Chapter 1. Cluster Architecture](#)

[Section 1.1. Modern Computing and the Role of Clusters](#)

[Section 1.2. Types of Clusters](#)

[Section 1.3. Distributed Computing and Clusters](#)

[Section 1.4. Limitations](#)

[Section 1.5. My Biases](#)

[Chapter 2. Cluster Planning](#)

[Section 2.1. Design Steps](#)

[Section 2.2. Determining Your Cluster's Mission](#)

[Section 2.3. Architecture and Cluster Software](#)

[Section 2.4. Cluster Kits](#)

[Section 2.5. CD-ROM-Based Clusters](#)

[Section 2.6. Benchmarks](#)

[Chapter 3. Cluster Hardware](#)

[Section 3.1. Design Decisions](#)

[Section 3.2. Environment](#)

[Chapter 4. Linux for Clusters](#)

[Section 4.1. Installing Linux](#)

[Section 4.2. Configuring Services](#)

[Section 4.3. Cluster Security](#)

[Part II: Getting Started Quickly](#)

[Chapter 5. openMosix](#)

[Section 5.1. What Is openMosix?](#)

[Section 5.2. How openMosix Works](#)

[Section 5.3. Selecting an Installation Approach](#)

[Section 5.4. Installing a Precompiled Kernel](#)

[Section 5.5. Using openMosix](#)

[Section 5.6. Recompiling the Kernel](#)

[Section 5.7. Is openMosix Right for You?](#)

[Chapter 6. OSCAR](#)

[Section 6.1. Why OSCAR?](#)

[Section 6.2. What's in OSCAR](#)

[Section 6.3. Installing OSCAR](#)

[Section 6.4. Security and OSCAR](#)

[Section 6.5. Using switcher](#)

[Section 6.6. Using LAM/MPI with OSCAR](#)

[Chapter 7. Rocks](#)

[Section 7.1. Installing Rocks](#)

[Section 7.2. Managing Rocks](#)

[Section 7.3. Using MPICH with Rocks](#)

[Part III: Building Custom Clusters](#)

[Chapter 8. Cloning Systems](#)

[Section 8.1. Configuring Systems](#)

[Section 8.2. Automating Installations](#)

[Section 8.3. Notes for OSCAR and Rocks Users](#)

[Chapter 9. Programming Software](#)

[Section 9.1. Programming Languages](#)

[Section 9.2. Selecting a Library](#)

[Section 9.3. LAM/MPI](#)

[Section 9.4. MPICH](#)

[Section 9.5. Other Programming Software](#)

[Section 9.6. Notes for OSCAR Users](#)

[Section 9.7. Notes for Rocks Users](#)

[Chapter 10. Management Software](#)

[Section 10.1. C3](#)

[Section 10.2. Ganglia](#)

[Section 10.3. Notes for OSCAR and Rocks Users](#)

[Chapter 11. Scheduling Software](#)

[Section 11.1. OpenPBS](#)

[Section 11.2. Notes for OSCAR and Rocks Users](#)

[Chapter 12. Parallel Filesystems](#)

[Section 12.1. PVFS](#)

[Section 12.2. Using PVFS](#)

[Section 12.3. Notes for OSCAR and Rocks Users](#)

[Part IV: Cluster Programming](#)

[Chapter 13. Getting Started with MPI](#)

[Section 13.1. MPI](#)

[Section 13.2. A Simple Problem](#)

[Section 13.3. An MPI Solution](#)

[Section 13.4. I/O with MPI](#)

[Section 13.5. Broadcast Communications](#)

[Chapter 14. Additional MPI Features](#)

[Section 14.1. More on Point-to-Point Communication](#)

[Section 14.2. More on Collective Communication](#)

[Section 14.3. Managing Communicators](#)

[Section 14.4. Packaging Data](#)

[Chapter 15. Designing Parallel Programs](#)

[Section 15.1. Overview](#)

[Section 15.2. Problem Decomposition](#)

[Section 15.3. Mapping Tasks to Processors](#)

[Section 15.4. Other Considerations](#)

[Chapter 16. Debugging Parallel Programs](#)

[Section 16.1. Debugging and Parallel Programs](#)

[Section 16.2. Avoiding Problems](#)

[Section 16.3. Programming Tools](#)

[Section 16.4. Rereading Code](#)

[Section 16.5. Tracing with printf](#)

[Section 16.6. Symbolic Debuggers](#)

[Section 16.7. Using gdb and ddd with MPI](#)

[Section 16.8. Notes for OSCAR and Rocks Users](#)

[Chapter 17. Profiling Parallel Programs](#)

[Section 17.1. Why Profile?](#)

[Section 17.2. Writing and Optimizing Code](#)

[Section 17.3. Timing Complete Programs](#)

[Section 17.4. Timing C Code Segments](#)

[Section 17.5. Profilers](#)

[Section 17.6. MPE](#)

[Section 17.7. Customized MPE Logging](#)

[Section 17.8. Notes for OSCAR and Rocks Users](#)

[Part V: Appendix](#)

[Appendix A. References](#)

[Section A.1. Books](#)

[Section A.2. URLs](#)

[Colophon](#)

[Index](#)

Copyright © 2005 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The Linux series designations, High Performance Linux Clusters with OSCAR, Rocks, openMosix, and MPI, images of the American West, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Preface

Clusters built from open source software, particularly based on the GNU/Linux operating system, are increasingly popular. Their success is not hard to explain because they can cheaply solve an ever-widening range of number-crunching applications. A wealth of open source or free software has emerged to make it easy to set up, administer, and program these clusters. Each individual package is accompanied by documentation, sometimes very rich and thorough. But knowing where to start and how to get the different pieces working proves daunting for many programmers and administrators.

This book is an overview of the issues that new cluster administrators have to deal with in making clusters meet their needs, ranging from the initial hardware and software choices through long-term considerations such as performance.

This book is not a substitute for the documentation that accompanies the software that it describes. You should download and read the documentation for the software. Most of the documentation available online is quite good; some is truly excellent.

In writing this book, I have evaluated a large number of programs and selected for inclusion the software I believe is the most useful for someone new to clustering. While writing descriptions of that software, I culled through thousands of pages of documentation to fashion a manageable introduction. This book brings together the information you'll need to get started. After reading it, you should have a clear idea of what is possible, what is available, and where to go to get it. While this book doesn't stand alone, it should reduce the amount of work you'll need to do. I have tried to write the sort of book I would have wanted when I got started with clusters.

The software described in this book is freely available, open source software. All of the software is available for use with Linux; however, much of it should work nicely on other platforms as well. All of the software has been installed and tested as described in this book. However, the behavior or suitability of the software described in this book cannot be guaranteed. While the material in this book is presented in good faith, neither the author nor O'Reilly Media, Inc. makes any explicit or implied warranty as to the behavior or suitability of this software. We strongly urge you to evaluate the software and information provided in this book as appropriate for your own circumstances.

One of the more important developments in the short life of high performance clusters has been the creation of cluster installation kits such as OSCAR and Rocks. With software packages like these, it is possible to install everything you need and very quickly have a fully functional cluster. For this reason, OSCAR and Rocks play a central role in this book.

OSCAR and Rocks are composed of a number of different independent packages, as well as customizations available only with each kit. A fully functional cluster will have a number of software packages each addressing a different need, such as programming, management, and scheduling. OSCAR and Rocks use a best-in-category approach, selecting the best available software for each type of cluster-related task. In addition to the core software, other compatible packages are available as well. Consequently, you will often have several products to choose from for any given need.

Most of the software included in OSCAR or Rocks is significant in its own right. Such software is often nontrivial to install and takes time to learn to use to its full potential. While both OSCAR and Rocks automate the installation process, there is still a lot to learn to effectively use either kit. Installing OSCAR or Rocks is only the beginning.

After some basic background information, this book describes the installation of OSCAR and

then Rocks. The remainder of the book describes in greater detail much of the software found in these packages. In each case, I describe the installation, configuration, and use of the software apart from OSCAR or Rocks. This should provide the reader with the information he will need to customize the software or even build a custom cluster bypassing OSCAR or Rocks completely, if desired.

I have also included a chapter on openMosix in this book, which may seem an odd choice to some. But there are several compelling reasons for including this information. First, not everyone needs a world-class high-performance cluster. If you have several machines and would like to use them together, but don't want the headaches that can come with a full cluster, openMosix is worth investigating. Second, openMosix is a nice addition to some more traditional clusters. Including openMosix also provides an opportunity to review recompiling the Linux kernel and an alternative kernel that can be used to demonstrate OSCAR's *kernel_picker*. Finally, I think openMosix is a really nice piece of software. In a sense, it represents the future, or at least one possible future, for clusters.

I have described in detail (too much, some might say) exactly how I have installed the software. Unquestionably, by the time you read, this some of the information will be dated. I have decided not to follow the practice of many authors in such situations, and offer just vague generalities. I feel that readers benefit from seeing the specific sorts of problems that appear in specific installations and how to think about their solutions.

Audience

This book is an introduction to building high-performance clusters. It is written for the biologist, chemist, or physicist who has just acquired two dozen recycled computers and is wondering how she might combine them to perform that calculation that has always taken too long to complete on her desktop machine. It is written for the computer science student who needs help getting started building his first cluster. It is not meant to be an exhaustive treatment of clusters, but rather attempts to introduce the basics needed to build and begin using a cluster.

In writing this book, I have assumed that the reader is familiar with the basics of setting up and administering a Linux system. At a number of places in this book, I provide a very quick overview of some of the issues. These sections are meant as a review, not an exhaustive introduction. If you need help in this area, several excellent books are available and are listed in the Appendix of this book.

When introducing a topic as extensive as clusters, it is impossible to discuss every relevant topic in detail without losing focus and producing an unmanageable book. Thus, I have had to make a number of hard decisions about what to include. There are many topics that, while of no interest to most readers, are nonetheless important to some. When faced with such topics, I have tried to briefly describe alternatives and provide pointers to additional material. For example, while computational grids are outside the scope of this book, I have tried to provide pointers for those of you who wish to know more about grids.

For the chapters dealing with programming, I have assumed a basic knowledge of C. For high-performance computing, FORTRAN and C are still the most common choices. For Linux-based systems, C seemed a more reasonable choice.

I have limited the programming examples to MPI since I believe this is the most appropriate parallel library for beginners. I have made a particular effort to keep the programming examples as simple as possible. There are a number of excellent books on MPI programming. Unfortunately, the available books on MPI all tend to use fairly complex problems as examples. Consequently, it is all too easy to get lost in the details of an example and miss the point. While you may become annoyed with my simplistic examples, I hope that you won't miss the point. You can always turn to these other books for more complex, real-world examples.

With any introductory book, there are things that must be omitted to keep the book manageable. This problem is further compounded by the time constraints of publication. I did not include a chapter on diskless systems because I believe the complexities introduced by using diskless systems are best avoided by people new to clusters. Because covering computational grids would have considerably lengthened this book, they are not included. There simply wasn't time or space to cover some very worthwhile software, most notably PVM and Condor. These were hard decisions.

Organization

This book is composed of 17 chapters, divided into four parts. The first part addresses background material; the second part deals with getting a cluster running quickly; the third part goes into more depth describing how a custom cluster can be built; and the fourth part introduces cluster programming.

Depending on your background and goals, different parts of this book are likely to be of interest. I have tried to provide information here and at the beginning of each section that should help you in selecting those parts of greatest interest. You should not need to read the entire book for it to be useful.

Part I, *An Introduction to Clusters*

[Chapter 1](#), is a general introduction to high-performance computing from the perspective of clusters. It introduces basic terminology and provides a description of various high-performance technologies. It gives a broad overview of the different cluster architectures and discusses some of the inherent limitations of clusters.

[Chapter 2](#), begins with a discussion of how to determine what you want your cluster to do. It then gives a quick overview of the different types of software you may need in your cluster.

[Chapter 3](#), is a discussion of the hardware that goes into a cluster, including both the individual computers and network equipment.

[Chapter 4](#), begins with a brief discussion of Linux in general. The bulk of the chapter covers the basics of installing and configuring Linux. This chapter assumes you are comfortable using Linux but may need a quick review of some administrative tasks.

Part II, *Getting Started Quickly*

[Chapter 5](#), describes the installation, configuration, and use of openMosix. It also reviews how to recompile a Linux kernel.

[Chapter 6](#), describes installing and setting up OSCAR. It also covers a few of the basics of using OSCAR.

[Chapter 7](#), describes installing Rocks. It also covers a few of the basics of using Rocks.

Part III, *Building Custom Clusters*

[Chapter 8](#), describes tools you can use to replicate the software installed on one machine onto others. Thus, once you have decided how to install and configure the software on an individual node in your cluster, this chapter will show you how to duplicate that installation on a number of machines quickly and efficiently.

[Chapter 9](#), first describes programming software that you may want to consider. Next, it describes the installation and configuration of the software, along with additional utilities you'll need if you plan to write the application programs that will run on your

cluster.

[Chapter 10](#), describes tools you can use to manage your cluster. Once you have a working cluster, you face numerous administrative tasks, not the least of which is insuring that the machines in your cluster are running properly and configured identically. The tools in this chapter can make life much easier.

[Chapter 11](#), describes OpenPBS, open source scheduling software. For heavily loaded clusters, you'll need software to allocate resources, schedule jobs, and enforce priorities. OpenPBS is one solution.

[Chapter 12](#), describes setting up and configuring the Parallel Virtual File System (PVFS) software, a high-performance parallel file system for clusters.

Part IV, *Cluster Programming*

[Chapter 13](#), is a tutorial on how to use the MPI library. It covers the basics. There is a lot more to MPI than what is described in this book, but that's a topic for another book or two. The material in this chapter will get you started.

[Chapter 14](#), describes some of the more advanced features of MPI. The intent is not to make you proficient with any of these features but simply to let you know that they exist and how they might be useful.

[Chapter 15](#), describes some techniques to break a program into pieces that can be run in parallel. There is no silver bullet for parallel programming, but there are several helpful ways to get started. The chapter is a quick overview.

[Chapter 16](#), first reviews the techniques used to debug serial programs and then shows how the more traditional approaches can be extended and used to debug parallel programs. It also discusses a few problems that are unique to parallel programs.

[Chapter 17](#), looks at techniques and tools that can be used to profile parallel programs. If you want to improve the performance of a parallel program, the first step is to find out where the program is spending its time. This chapter shows you how to get started.

Part V, *Appendix*

The [Appendix](#) includes source information and documentation for the software discussed in the book. It also includes pointers to other useful information about clusters.

Conventions

This book uses the following typographical conventions:

Italics

Used for program names, filenames, system names, email addresses, and URLs, and for emphasizing new terms.

`Constant width`

Used in examples showing programs, output from programs, the contents of files, or literal information.

Constant-width italics

Used for general syntax and items that should be replaced in expressions.



Indicates a tip, suggestion, or general note.



Indicates a warning or caution.

How to Contact Us

In a sense, any book is a work in progress. If you have comments, suggestions, or corrections, I would appreciate hearing from you. You can contact me through booktech@oreilly.com.

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
1-800-998-9938 (in the U.S. or Canada)
1-707-829-0515 (international or local)
1-707-829-0104 (fax)

You can send us messages electronically. To be put on the mailing list or to request a catalog, send email to:

info@oreilly.com

To ask technical questions or to comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for the book, where we'll list examples, errata, and any plans for future editions. You can access this page at:

<http://www.oreilly.com/catalog/highperlinuxc/>

For more information about this book and others, see the O'Reilly web site:

<http://www.oreilly.com>

Using Code Examples

The code developed in this book is available for download for free from *the O'Reilly* web site for this book <http://www.oreilly.com/catalog/highperlinuxc>. (Before installing, take a look at *readme.txt* in the download).

This book is here to help you get your job done. In general, you can use the code in this book in your programs and documentation. You don't need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book doesn't require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code doesn't require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but don't require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*High Performance Linux Clusters with OSCAR, Rocks, openMosix, and MPI*, by Joseph Sloan. Copyright 2005 O'Reilly, 0-596-00570-9."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at permissions@oreilly.com.

Acknowledgments

While the cover of this book displays only my name, it is the work of a number of people. First and foremost, credit goes to the people who created the software described in this book. The quality of this software is truly remarkable. Anyone building a cluster owes a considerable debt to these developers.

This book would not exist if not for the students I have worked with both at Lander University and Wofford College. Brian Bell's interest first led me to investigate clusters. Michael Baker, Jonathan DeBusk, Ricaye Harris, Tilisha Haywood, Robert Merting, and Robert Veasey all suffered through courses using clusters. I can only hope they learned as much from me as I learned from them.

Thanks also goes to the computer science department and to the staff of information technology at Wofford College—in particular, to Angela Shiflet for finding the funds and to Dave Whisnant for finding the computers used to build the clusters used in writing this book. Martin Aigner, Joe Burnet, Watts Hudgens, Jim Sawyers, and Scott Sperka, among others, provided support beyond the call of duty. Wofford is a great place to work and to write a book. Thanks to President Bernie Dunlap, Dean Dan Maulsby, and the faculty and staff for making Wofford one of the top liberal arts colleges in the nation.

I was very fortunate to have a number of technical reviewers for this book, including people intimately involved with the creation of the software described here, as well as general reviewers. Thanks goes to Kris Buytaert, a senior consultant with X-Tend and author of the *openMosix HOWTO*, for reviewing the chapter on openMosix. Kris's close involvement with the openMosix project helped provide a perspective not only on openMosix as it is today, but also on the future of the openMosix project.

Thomas Naughton and Stephen L. Scott, both from Oak Ridge National Laboratory and members of the OSCAR work group, reviewed the book. They provided not only many useful corrections, but helpful insight into cluster software as well, particularly OSCAR.

Edmund J. Sutcliffe, a consultant with Thoughtful Solutions, attempted to balance my sometimes myopic approach to clusters, arguing for a much broader perspective on clusters. Several topics were added or discussed in greater detail at his insistence. Had time allowed, more would have been added.

John McKowen Taylor, Jr., of Cadence Design System, Inc., also reviewed the book. In addition to correcting many errors, he provided many kind words and encouragement that I greatly appreciated.

Robert Bruce Thompson, author of two excellent books on PC hardware, corrected a number of leaks in the hardware chapter. Unfortunately, developers for Rocks declined an invitation to review the material, citing the pressures of putting together a new release.

While the reviewers unfailingly pointed out my numerous errors and misconceptions, it didn't follow that I understood everything they said or faithfully amended this manuscript. The blame for any errors that remain rests squarely on my shoulders.

I consider myself fortunate to be able to work with the people in the O'Reilly organization. This is the second book I have written with them and both have gone remarkably smoothly. If you are thinking of writing a technical book, I strongly urge you to consider O'Reilly. Unlike some other publishers, you will be working with technically astute people from the beginning. Particular thanks goes to Andy Oram, the technical editor for this book. Andy was constantly looking for ways to improve this book. Producing any book requires an small army of people, most of whom are hidden in the background and never receive proper recognition. A debt of

gratitude is owed to many others working at O'Reilly.

This book would not have been possible without the support and patience of my family. Thank you.

Part I: An Introduction to Clusters

The first section of this book is a general introduction to clusters. It is largely background material. Readers already familiar with clusters may want to quickly skim this material and then move on to subsequent chapters. This section is divided into four chapters.

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.

1.1 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.

The third approach is parallelism, i.e., executing instructions simultaneously. There are a variety of ways to achieve this. At one end of the spectrum, parallelism can be integrated into the architecture of a single CPU (which brings us back to buying the best computer you can afford). At the other end of the spectrum, you may be able to divide the computation up among different computers on a network, each computer working on a part of the calculation, all working at the same time. This book is about that approach—harnessing a team of horses.

1.1.1 Uniprocessor Computers

The traditional classification of computers based on size and performance, i.e., classifying computers as microcomputers, workstations, minicomputers, mainframes, and supercomputers, has become obsolete. The ever-changing capabilities of computers means that today's microcomputers now outperform the mainframes of the not-too-distant past. Furthermore, this traditional classification scheme does not readily extend to parallel systems and clusters. Nonetheless, it is worth looking briefly at the capabilities and problems associated with more traditional computers, since these will be used to assemble clusters. If you are working with a team of horses, it is helpful to know something about a horse.

Regardless of where we place them in the traditional classification, most computers today are based on an architecture often attributed to the Hungarian mathematician John von Neumann. The basic structure of a *von Neumann computer* is a CPU connected to memory by a communications channel or bus. Instructions and data are stored in memory and are moved to and from the CPU across the bus. The overall speed of a computer depends on both the

speed at which its CPU can execute individual instructions and the overhead involved in moving instructions and data between memory and the CPU.

Several technologies are currently used to speed up the processing speed of CPUs. The development of *reduced instruction set computer (RISC)* architectures and post-RISC architectures has led to more uniform instruction sets. This eliminates cycles from some instructions and allows a higher clock-rate. The use of RISC technology and the steady increase in chip densities provide great benefits in CPU speed.

Superscalar architectures and *pipelining* have also increased processor speeds. Superscalar architectures execute two or more instructions simultaneously. For example, an addition and a multiplication instruction, which use different parts of the CPU, might be executed at the same time. Pipelining overlaps the different phase of instruction execution like an assembly line. For example, while one instruction is executed, the next instruction can be fetched from memory or the results from the previous instructions can be stored.

Memory bandwidth, basically the rate at which bits are transferred from memory over the bus, is a different story. Improvements in memory bandwidth have not kept up with CPU improvements. It doesn't matter how fast the CPU is theoretically capable of running if you can't get instructions and data into or out of the CPU fast enough to keep the CPU busy. Consequently, memory access has created a performance bottleneck for the classical von Neumann architecture: the *von Neumann bottleneck*.

Computer architects and manufacturers have developed a number of techniques to minimize the impact of this bottleneck. Computers use a hierarchy of memory technology to improve overall performance while minimizing cost. Frequently used data is placed in very fast cache memory, while less frequently used data is placed in slower but cheaper memory. Another alternative is to use multiple processors so that memory operations are spread among the processors. If each processor has its own memory and its own bus, all the processors can access their own memory simultaneously.

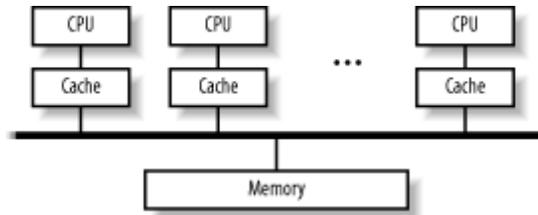
1.1.2 Multiple Processors

Traditionally, supercomputers have been pipelined, superscalar processors with a single CPU. These are the "big iron" of the past, often requiring "forklift upgrades" and multiton air conditioners to prevent them from melting from the heat they generate. In recent years we have come to augment that definition to include parallel computers with hundreds or thousands of CPUs, otherwise known as multiprocessor computers. Multiprocessor computers fall into two basic categories—centralized multiprocessors (or single enclosure multiprocessors) and multicomputers.

1.1.2.1 Centralized multiprocessors

With centralized multiprocessors, there are two architectural approaches based on how memory is managed—*uniform memory access (UMA)* and *nonuniform memory access (NUMA)* machines. With UMA machines, also called *symmetric multiprocessors (SMP)*, there is a common shared memory. Identical memory addresses map, regardless of the CPU, to the same location in physical memory. Main memory is equally accessible to all CPUs, as shown in [Figure 1-1](#). To improve memory performance, each processor has its own cache.

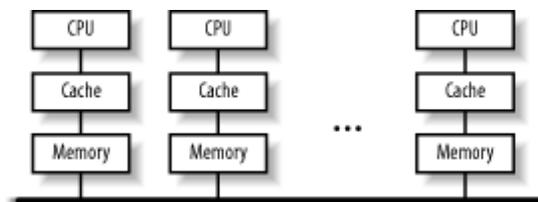
Figure 1-1. UMA architecture



There are two closely related difficulties when designing a UMA machine. The first problem is synchronization. Communications among processes and access to peripherals must be coordinated to avoid conflicts. The second problem is *cache consistency*. If two different CPUs are accessing the same location in memory and one CPU changes the value stored in that location, then how is the cache entry for the other CPU updated? While several techniques are available, the most common is *snooping*. With snooping, each cache listens to all memory accesses. If a cache contains a memory address that is being written to in main memory, the cache updates its copy of the data to remain consistent with main memory.

A closely related architecture is used with NUMA machines. Roughly, with this architecture, each CPU maintains its own piece of memory, as shown in [Figure 1-2](#). Effectively, memory is divided among the processors, but each process has access to all the memory. Each individual memory address, regardless of the processor, still references the same location in memory. Memory access is nonuniform in the sense that some parts of memory will appear to be much slower than other parts of memory since the bank of memory "closest" to a processor can be accessed more quickly by that processor. While this memory arrangement can simplify synchronization, the problem of memory coherency increases.

Figure 1-2. NUMA architecture



Operating system support is required with either multiprocessor scheme. Fortunately, most modern operating systems, including Linux, provide support for SMP systems, and support is improving for NUMA architectures.

When dividing a calculation among processors, an important concern is *granularity*, or the smallest piece that a computation can be broken into for purposes of sharing among different CPUs. Architectures that allow smaller pieces of code to be shared are said to have a *finer* granularity (as opposed to a *coarser* granularity). The granularity of each of these architectures is the thread. That is, the operating system can place different threads from the same process on different processors. Of course, this implies that, if your computation generates only a single thread, then that thread can't be shared between processors but must run on a single CPU. If the operating system has nothing else for the other processors to do, they will remain idle and you will see no benefit from having multiple processors.

A third architecture worth mentioning in passing is *processor array*, which, at one time, generated a lot of interest. A processor array is a type of vector computer built with a collection of identical, synchronized processing elements. Each processor executes the same instruction on a different element in a data array.

Numerous issues have arisen with respect to processor arrays. While some problems map nicely to this architecture, most problems do not. This severely limits the general use of

processor arrays. The overall design doesn't work well for problems with large serial components. Processor arrays are typically designed around custom VLSI processors, resulting in much higher costs when compared to more commodity-oriented multiprocessor designs. Furthermore, processor arrays typically are single user, adding to the inherent cost of the system. For these and other reasons, processor arrays are no longer popular.

1.1.2.2 Multicomputers

A multicomputer configuration, or cluster, is a group of computers that work together. A cluster has three basic elements—a collection of individual computers, a network connecting those computers, and software that enables a computer to share work among the other computers via the network.

For most people, the most likely thing to come to mind when speaking of multicomputers is a *Beowulf cluster*. Thomas Sterling and Don Becker at NASA's Goddard Space Flight Center built a parallel computer out of commodity hardware and freely available software in 1994 and named their system Beowulf.^[1] While this is perhaps the best-known type of multicomputer, a number of variants now exist.

[1] If you think back to English lit, you will recall that the epic hero Beowulf was described as having "the strength of many."

First, both commercial multicomputers and commodity clusters are available. Commodity clusters, including Beowulf clusters, are constructed using *commodity, off-the-shelf (COTS)* computers and hardware. When constructing a commodity cluster, the norm is to use freely available, open source software. This translates into an extremely low cost that allows people to build a cluster when the alternatives are just too expensive. For example, the "Big Mac" cluster built by Virginia Polytechnic Institute and State University was initially built using 1100 dual-processor Macintosh G5 PCs. It achieved speeds on the order of 10 teraflops, making it one of the fastest supercomputers in existence. But while supercomputers in that class usually take a couple of years to construct and cost in the range of \$100 million to \$250 million, Big Mac was put together in about a month and at a cost of just over \$5 million. (A list of the fastest machines can be found at <http://www.top500.org>. The site also maintains a list of the top 500 clusters.)

In commodity clusters, the software is often mix-and-match. It is not unusual for the processors to be significantly faster than the network. The computers within a cluster can be dedicated to that cluster or can be standalone computers that dynamically join and leave the cluster. Typically, the term Beowulf is used to describe a cluster of dedicated computers, often with minimal hardware. If no one is going to use a node as a standalone machine, there is no need for that node to have a dedicated keyboard, mouse, video card, or monitor. Node computers may or may not have individual disk drives. (*Beowulf* is a politically charged term that is avoided in this book.) While a commodity cluster may consist of identical, high-performance computers purchased specifically for the cluster, they are often a collection of recycled cast-off computers, or a *pile-of-PCs (POP)*.

Commercial clusters often use proprietary computers and software. For example, a SUN Ultra is not generally thought of as a COTS computer, so an Ultra cluster would typically be described as a proprietary cluster. With proprietary clusters, the software is often tightly integrated into the system, and the CPU performance and network performance are well matched. The primary disadvantage of commercial clusters is, as you no doubt guessed, their cost. But if money is not a concern, then IBM, Sun Microsystems, or any number of other companies will be happy to put together a cluster for you. (The salesman will probably even take you to lunch.)

A *network of workstations (NOW)*, sometimes called a *cluster of workstations (COW)*, is a cluster composed of computers usable as individual workstations. A computer laboratory at a university might become a NOW on the weekend when the laboratory is closed. Or office machines might join a cluster in the evening after the daytime users leave.

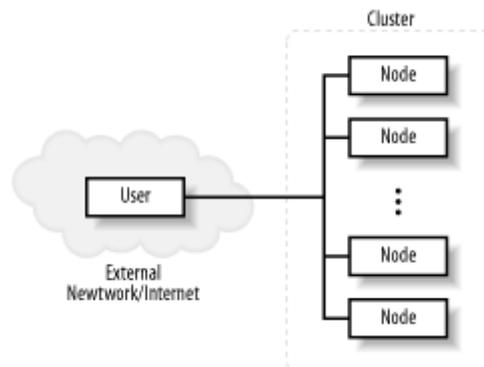
Software is an integral part of any cluster. A discussion of cluster software will constitute the bulk of this book. Support for clustering can be built directly into the operating system or may sit above the operating system at the application level, often in user space. Typically, when clustering support is part of the operating system, all nodes in the cluster need to have identical or nearly identical kernels; this is called a *single system image (SSI)*. At best, the granularity is the process. With some software, you may need to run distinct programs on each node, resulting in even coarser granularity. Since each computer in a cluster has its own memory (unlike a UMA or NUMA computer), identical addresses on individual CPUs map different physical memory locations. Communication is more involved and costly.

1.1.2.3 Cluster structure

It's tempting to think of a cluster as just a bunch of interconnected machines, but when you begin constructing a cluster, you'll need to give some thought to the internal structure of the cluster. This will involve deciding what roles the individual machines will play and what the interconnecting network will look like.

The simplest approach is a *symmetric cluster*. With a symmetric cluster ([Figure 1-3](#)) each node can function as an individual computer. This is extremely straightforward to set up. You just create a subnetwork with the individual machines (or simply add the computers to an existing network) and add any cluster-specific software you'll need. You may want to add a server or two depending on your specific needs, but this usually entails little more than adding some additional software to one or two of the nodes. This is the architecture you would typically expect to see in a NOW, where each machine must be independently usable.

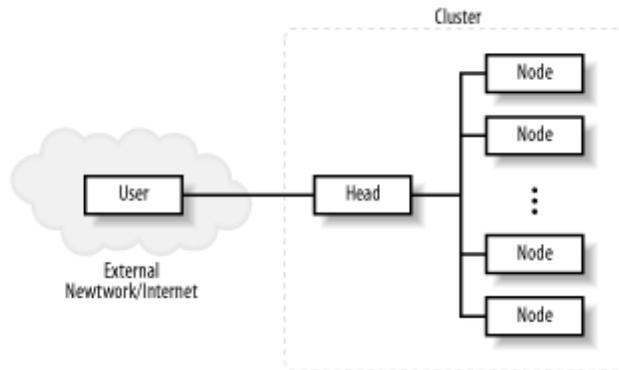
Figure 1-3. Symmetric clusters



There are several disadvantages to a symmetric cluster. Cluster management and security can be more difficult. Workload distribution can become a problem, making it more difficult to achieve optimal performance.

For dedicated clusters, an asymmetric architecture is more common. With *asymmetric clusters* ([Figure 1-4](#)) one computer is the *head node* or *frontend*. It serves as a gateway between the remaining nodes and the users. The remaining nodes often have very minimal operating systems and are dedicated exclusively to the cluster. Since all traffic must pass through the head, asymmetric clusters tend to provide a high level of security. If the remaining nodes are physically secure and your users are trusted, you'll only need to harden the head node.

Figure 1-4. Asymmetric clusters

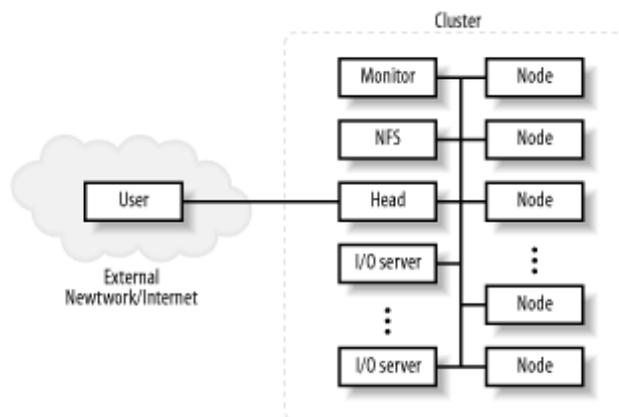


The head often acts as a primary server for the remainder of the clusters. Since, as a dual-homed machine, it will be configured differently from the remaining nodes, it may be easier to keep all customizations on that single machine. This simplifies the installation of the remaining machines. In this book, as with most descriptions of clusters, we will use the term *public interface* to refer to the network interface directly connected to the external network and the term *private interface* to refer to the network interface directly connected to the internal network.

The primary disadvantage of this architecture comes from the performance limitations imposed by the cluster head. For this reason, a more powerful computer may be used for the head. While beefing up the head may be adequate for small clusters, its limitations will become apparent as the size of the cluster grows. An alternative is to incorporate additional servers within the cluster. For example, one of the nodes might function as an NFS server, a second as a management station that monitors the health of the clusters, and so on.

I/O represents a particular challenge. It is often desirable to distribute a shared filesystem across a number of machines within the cluster to allow parallel access. [Figure 1-5](#) shows a more fully specified cluster.

Figure 1-5. Expanded cluster



Network design is another key issue. With small clusters, a simple switched network may be adequate. With larger clusters, a fully connected network may be prohibitively expensive. Numerous topologies have been studied to minimize connections (costs) while maintaining viable levels of performance. Examples include hyper-tree, hyper-cube, butterfly, and shuffle-exchange networks. While a discussion of network topology is outside the scope of this book, you should be aware of the issue.

Heterogeneous networks are not uncommon. Although not shown in the figure, it may be desirable to locate the I/O servers on a separate parallel network. For example, some clusters

have parallel networks allowing administration and user access through a slower network, while communications for processing and access to the I/O servers is done over a high-speed network.



1.2 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.

To some extent, any cluster can provide redundancy, scalability, and improved performance, regardless of its classification. Since load-balancing provides greater availability, it is not unusual to see both load-balancing and high-availability in the same cluster. The *Linux Virtual Server Project (LVS)* is an example of combining these two approaches. An LVS server is a high-availability server implemented by distributing tasks among a number of real servers. Interested readers are encouraged to visit the web pages for the Linux Virtual Server Project (<http://www.linux-vs.org>) and the High-Availability Linux Project (<http://www.linux-ha.org>) and to read the relevant HOWTOs. OSCAR users will want to visit the High-Availability OSCAR web site <http://www.openclustergroup.org/HA-OSCAR/>.

1.3 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.

Peer-to-peer computing provides yet another approach to distributed computing. Again this is an ambiguous term. Peer-to-peer may refer to sharing cycles, to the communications infrastructure, or to the actual data distributed across a WAN or the Internet. Peer-to-peer cycle sharing is best exemplified by *SETI@Home*, a project to analyze radio telescope data for signs of extraterrestrial intelligence. Volunteers load software onto their Internet-connected computers. To the casual PC or Mac user, the software looks like a screensaver. When a computer becomes idle, the screensaver comes on and the computer begins analyzing the data. If the user begins using the computer again, the screensaver closes and the data analysis is suspended. This approach has served as a model for other research, including the analysis of cancer and AIDS data.

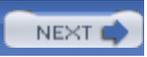
Data or file-sharing peer-to-peer networks are best exemplified by Napster, Gnutella, or Kazaa technologies. With some peer-to-peer file-sharing schemes, cycles may also be provided for distributed computations. That is, by signing up and installing the software for some services, you may be providing idle cycles to the service for other uses beyond file sharing. Be sure you read the license before you install the software if you don't want your computers used in this way.

Other entries in the distributed computing taxonomy include *federated clusters* and *constellations*. Federated clusters are clusters of clusters, while constellations are clusters where the number of CPUs is greater than the number of nodes. A four-node cluster of SGI

Altrix computers with 128 CPUs per node is a constellation. Peer-to-peer, grids, federated clusters, and constellations are outside the scope of this book.



< Day Day Up >



1.4 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.

1.4.1 Amdahl's Law

As just noted, the amount of code that must be executed serially limits how much of a speedup you can expect from parallel execution. This idea has been formalized by what is known as *Amdahl's Law*, named after Gene Amdahl, who first stated the law in the late sixties. In a nutshell, Amdahl's Law states that the serial portion of a program will be the limiting factor in how much you can speed up the execution of the program using multiple processors.^[2]

^[2] While Amdahl's Law is the most widely known and most useful metric for describing parallel performance, there are others. These include Gustafson-Barsus's, Sun's, and Ni's Laws and the Karp-Flat and the Isoefficiency Metrics.

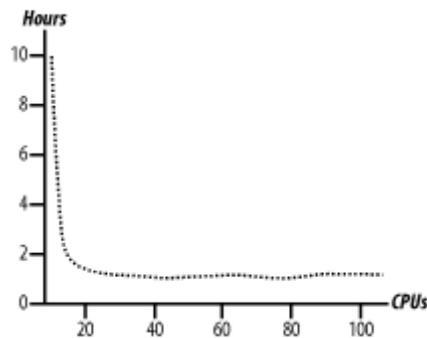
An example should help clarify Amdahl's Law. Let's assume you have a computation that takes 10 hours to complete on a currently available computer and that 90 percent of your code can be parallelized. In other words, you are spending one hour doing instructions that must be done serially and nine hours doing instructions that can be done in parallel. Amdahl's Law states that you'll never be able to run this code on this class of computers in less than one hour, regardless of how many additional computers you have available. To see this, imagine that you had so many computers that you could execute all the parallel code instantaneously. You would still have the serial code to execute, which has to be done on a

single computer, and it would still take an hour.^[3]

^[3] For those of you who love algebra, the speedup factor is equal to $1/(s + p/N)$, where s is the fraction of the code that is inherently serial, p is the fraction of the code that can be parallelized, and N is the number of processors available. Clearly, $p + s = 1$. As the number of processors becomes very large, p/N becomes very small, and the speedup becomes essentially $1/s$. So if s is 0.1, the largest speedup you can expect is a factor of 10, no matter how many processors you have available.

In practice, you won't have an unlimited number of processors, so your total time will always be longer. [Figure 1-6](#) shows the amount of time needed for this example, depending on the number of processors you have available.

Figure 1-6. Execution time vs. number of processors



You should also remember that Amdahl's law is an ideal. In practice, there is the issue of the overhead introduced by parallelizing the code. For example, coordinating communications among the various processes will require additional code. This adds to the overall execution time. And if there is contention for the network, this can stall processes, further slowing the calculation. In other words, Amdahl's Law is the best speedup you can hope for, but not the actual speedup you'll see.

What can you do if you need to do this calculation in less than one hour? As I noted earlier, you have three choices when you want to speed up a calculation—better algorithms, faster computers, or more computers. If more computers won't take you all the way, your remaining choices are better algorithms and faster computers. If you can rework your code so that a larger fraction can be done in parallel, you'll see an increased benefit from a parallel approach. Otherwise, you'll need to dig deep into your pocket for faster computers.

Surprisingly, a fair amount of controversy still surrounds what should be obvious once you think about it. This stems in large part from the misapplication of Amdahl's Law over the years. For example, Amdahl's Law has been misused as an argument favoring faster computers over parallel computing.

The most common misuse is based on the assumption that the amount of speedup is independent of the size of the problem. Amdahl's Law simply does not address how problems scale. The fraction of the code that must be executed serially usually changes as the size of the problem changes. So, it is a mistake to assume that a problem's speedup factor will be the same when the scale of the problem changes. For instance, if you double the length of a simulation, you may find that the serial portions of the simulation, such as the initialization and report phases, are basically unchanged, while the parallelizable portion of the code is what doubles. Hence, the fraction of the time spent in the serial code will decrease and Amdahl's Law will specify a greater speedup. This is good news! After all, it's when problems get bigger that we most need the speedup. For most problems, the speedup factor will depend upon the problem size. As the problem size changes, so does the speedup factor. The amount will depend on the nature of the individual problem, but typically, the speedup will increase as the size of the problem increases. As the problem size grows, it is not unusual to see a linear increase in the amount of time spent in the serial portion of the code and a

quadratic increase in the amount of time spent in the parallelizable portion of the code. Unfortunately, if you only apply Amdahl's Law to the smaller problem size, you'll underestimate the benefit of a parallel approach.

Having said this, it is important to remember that Amdahl's Law does clearly state a limitation of parallel computing. But this limitation varies not only from problem to problem, but with the size of the problem as well.

One last word about the limitations of clusters—the limitations are often tied to a particular approach. It is often possible to mix approaches and avoid limitations. For example, in constructing your clusters, you'll want to use the best computers you can afford. This will lessen the impact of inherently serial code. And don't forget to look at your algorithms!



1.5 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.

One last comment—if you keep reading, I promise not to mention horses again.

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.

2.1 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 in greater detail.

2.2 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.

2.2.1 What Is Your User Base?

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.

2.2.2 How Heavily Will the Cluster Be Used?

Will the cluster be in constant use, with users fighting over it, or will it be used on an occasional basis as large problems arise? Will some of your jobs have higher priorities than others? Will you have a mix of jobs, some requiring the full capabilities of the cluster while others will need only a subset?

If you have a large user base with lots of potential conflicts, you will need some form of scheduling software. If your cluster will be lightly used or have very few users who are willing to work around each other, you may be able to postpone installing scheduling software.

2.2.3 What Kinds of Software Will You Run on the Cluster?

There are several levels at which this question can be asked. At a cluster management level, you'll need to decide which systems software you want, e.g., BSD, Linux, or Windows, and you'll need to decide what clustering software you'll need. Both of these choices will be addressed later in this chapter.

From a user perspective, you'll need to determine what application-level software to use. Will your users be using canned applications? If so, what are these applications and what are their requirements? Will your users be developing software? If so, what tools will they need? What is the nature of the software they will write and what demands will this make on your cluster? For example, if your users will be developing massive databases, will you have adequate storage? Will the I/O subsystem be adequate? If your users will carry out massive calculations, do you have adequate computational resources?

2.2.4 How Much Control Do You Need?

Closely related to the types of code you will be running is the question of how much control you will need over the code. There are a range of possible answers. If you need tight control over resources, you will probably have to write your own applications. User-developed code can make explicit use of the available resources.

For some uses, explicit control isn't necessary. If you have calculations that split nicely into separate processes and you'd just like them to run faster, software that provides transparent control may be the best solution. For example, suppose you have a script that invokes a file compression utility on a large number of files. It would be convenient if you could divide these file compression tasks among a number of processes, but you don't care about the details of how this is done.

openMosix, code that extends the Linux kernel, provides this type of transparent support. Processes automatically migrate among cluster computers. The advantage is that you may need to rewrite user code. However, the transparent control provided by openMosix will not work if the application uses shared memory or runs as a single process.

2.2.5 Will This Be a Dedicated or Shared Cluster?

Will the machines that comprise the cluster be dedicated to the cluster, or will they be used for other tasks? For example, a number of clusters have been built from office machines. During the day, the administrative staff uses the machines. In the evening and over the weekend, they are elements of a cluster. University computing laboratories have been used in the same way.

Obviously, if you have a dedicated cluster, you are free to configure the nodes as you see fit. With a shared cluster, you'll be limited by the requirements of the computers' day jobs. If this is the case, you may want to consider whether a dual-boot approach is feasible.

2.2.6 What Resources Do You Have?

Will you be buying equipment or using existing equipment? Will you be using recycled equipment? Recycled equipment can certainly reduce your costs, but it will severely constrain what you can do. At the very least, you'll need a small budget to adapt and maintain the equipment you have. You may need to purchase networking equipment such as a switch and cables, or you may need to replace failing parts such as disk drives and network cards. (See [Chapter 3](#) for more information about hardware.)

2.2.7 How Will Cluster Access Be Managed?

Will you need local or remote access or both? Will you need to provide Internet access, or can you limit it to the local or campus network? Can you isolate the cluster? If you must provide remote access, what will be the nature of that access? For example, will you need to install software to provide a graphical interface for remote users? If you can isolate your network,

security becomes less of an issue. If you must provide remote access, you'll need to consider tools like SSH and VNC. Or is serial port access by a terminal server sufficient?

2.2.8 What Is the Extent of Your Cluster?

The term *cluster* usually applies to computers that are all on the same subnet. If you will be using computers on different networks, you are building a *grid*. With a grid you'll face greater communications overhead and more security issues. Maintaining the grid will also be more involved and should be addressed early in the design process. This book doesn't cover the special considerations needed for grids.

2.2.9 What Security Concerns Do You Have?

Can you trust your users? If the answer is yes, this greatly simplifies cluster design. You can focus on controlling access to the cluster. If you can't trust your users, you'll need to harden each machine and develop secure communications. A closely related question is whether you can control physical access to your computers. Again, controlling physical access will simplify securing your cluster since you can focus on access points, e.g., the head node rather than the cluster as a whole. Finally, do you deal with sensitive data? Often the value of the data you work with determines the security measures you must take.

2.3 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.

2.3.1 System Software

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](#).)

In addition to the operating system itself, you may need additional utilities or extensions to the basic services provided by the operating system. For example, to create a cluster you'll need to install the operating system and software on a large number of machines. While you could do this manually with a small cluster, it's an error-prone and tedious task. Fortunately, you can automate the process with cloning software. Cloning is described in detail in [Chapter 8](#).

High-performance systems frequently require extensive I/O. To optimize performance, parallel file systems may be used. [Chapter 12](#) looks at the *Parallel Virtual File System (PVFS)*, an open source high-performance file system.

2.3.2 Programming Software

There are two basic decisions you'll need to make with respect to programming software—the programming languages you want to support and the libraries you want to use. If you have a small user base, you may be able to standardize on a single language and a single library. If you can pull this off, go for it; life will be much simpler. However, if you need to support a number of different users and applications, you may be forced to support a wider variety of

programming software.

The parallel programming libraries provide a mechanism that allows you to easily coordinate computing and exchange data among programs running on the cluster. Without this software, you'll be forced to rely on operating system primitives to program your cluster. While it is certainly possible to use sockets to build parallel programs, it is a lot more work and more error prone. The most common libraries are the *Message Passing Interface (MPI)* and *Parallel Virtual Machine (PVM)* libraries.

The choice of program languages depends on the parallel libraries you want to use. Typically, the libraries provide bindings for only a small number of programming languages. There is no point in installing Ada if you can't link it to the parallel library you want to use. Traditionally, parallel programming libraries support C and FORTRAN, and C++ is growing in popularity. Libraries and languages are discussed in greater detail in [Chapter 9](#).

2.3.3 Control and Management

In addition to the programming software, you'll need to keep your cluster running. This includes scheduling and management software.

Cluster management includes both routine system administration tasks and monitoring the health of your cluster. With a cluster, even a simple task can become cumbersome if it has to be replicated over a large number of systems. Just checking which systems are available can be a considerable time sink if done on a regular basis. Fortunately, there are several packages that can be used to simplify these tasks. *Cluster Command and Control (C3)* provides a command-line interface that extends across a cluster, allowing easy replication of tasks on each machine in a cluster or on a subset of the cluster. *Ganglia* provides web-based monitoring in a single interface. Both C3 and Ganglia can be used with federated clusters as well as simple clusters. C3 and Ganglia are described in [Chapter 10](#).

Scheduling software determines when your users' jobs will be executed. Typically, scheduling software can allocate resources, establish priorities, and do basic accounting. For Linux clusters there are two likely choices—*Condor* and *Portable Batch System (PBS)*. If you have needs for an advanced scheduler, you might also consider *Maui*. PBS is available as a commercial product, *PBSPro*, and as open source software, *OpenPBS*. OpenPBS is described in [Chapter 11](#).

2.4 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.

A second problem is that, in making everything work together, kit builders occasionally have to do things a little differently. So when you look at the original documentation for the individual components in a kit, you may find that the software hasn't been installed as described. When you learn more about the software, you'll come to understand and appreciate why the changes were made. But in the short term, these changes can add to the confusion.

So while a cluster kit can get you up and running quickly, you will still need to learn the details of the individual software. You should follow up the installation with a thorough study of how the individual pieces in the kit work. For most beginners, the single advantage of being able to get a cluster up and running quickly probably outweighs all of the disadvantages.

While other cluster kits are available, the three most common kits for Linux clusters are NPACI Rocks, OSCAR, and Scyld Beowulf.^[1] While Scyld Beowulf is a commercial product available from Penguin Computing, an earlier, unsupported version is available for a very nominal cost from <http://www.linuxcentral.com/>. Donald Becker, one of the original Beowulf developers, founded Scyld Computing, which was subsequently acquired by Penguin Computing. Scyld is built on top of Red Hat Linux and includes an enhanced kernel, tools, and utilities. While Scyld Beowulf is a solid system, you face the choice of using an expensive commercial product or a somewhat dated, unsupported product. Furthermore, variants of both Rocks and OSCAR are available. For example, BioBrew (<http://bioinformatics.org/biobrew/>) is a Rocks-based system that contains a number of packages for analyzing bioinformatics information. For these reasons, either Rocks or OSCAR is arguably a better choice than Scyld Beowulf.

[1] For grid computing, which is outside the scope of this book, the Globus Toolkit is a likely choice.

NPACI (National Partnership for Advanced Computational Infrastructure) Rocks is a collection of open source software for creating a cluster built on top of Red Hat Linux. Rocks takes a cookie-cutter approach. To install Rocks, begin by downloading a set of ISO images from <http://rocks.npaci.edu/Rocks/> and use them to create installation CD-ROMs. Next, boot to the first CD-ROM and answer a few questions as the cluster is built. Both Linux and the clustering software are installed. (This is a mixed blessing—it simplifies the installation but you won't have any control over how Linux is installed.) The installation should go very quickly. In fact, part of the Rocks' management strategy is that, if you have problems with a node, the best solution is to reinstall the node rather than try to diagnose and fix the problem. Depending on hardware, it may be possible to reinstall a node in under 10 minutes. When a Rocks installation goes as expected, you can be up and running in a very short amount of time. However, because the installation of the cluster software is tied to the installation of the operating system, if the installation fails, you can be left staring at a dead system and little idea of what to do. Fortunately, this rarely happens.

OSCAR, from the Open Cluster Group, uses a different installation strategy. With OSCAR, you first install Linux (but only on the head node) and then install OSCAR—the installations of the two are separate. This makes the installation more involved, but it gives you more control over the configuration of your system, and it is somewhat easier (that's easier, not easy) to recover when you encounter installation problems. And because the OSCAR installation is separate from the Linux installation, you are not tied to a single Linux distribution.

Rocks uses a variant of Red Hat's Anaconda and Kickstart programs to install the compute nodes. Thus, Rocks is able to probe the system to see what hardware is present. To be included in Rocks, software must be available as an RPM and configuration must be entirely automatic. As a result, with Rocks it is very straightforward to set up a cluster using heterogeneous hardware. OSCAR, in contrast, uses a system image cloning strategy to distribute the disk image to the compute nodes. With OSCAR it is best to use the same hardware throughout your cluster. Rocks requires systems with hard disks. Although not discussed in this book, OSCAR's thin client model is designed for diskless systems.

Both Rocks and OSCAR include a variety of software and build complete clusters. In fact, most of the core software is the same for both OSCAR and Rocks. However, there are a few packages that are available for one but not the other. For example, Condor is readily available for Rocks while LAM/MPI is included in OSCAR.

Clearly, Rocks and OSCAR take orthogonal approaches to building clusters. Cluster kits are difficult to build. OSCAR scales well over Linux distributions. Rocks scales well with heterogeneous hardware. No one approach is better in every situation.

Rocks and OSCAR are at the core of this book. The installation, configuration, and use of OSCAR are described in detail in [Chapter 6](#). The installation, configuration, and use of Rocks is described in [Chapter 7](#). Rocks and OSCAR heavily influenced the selection of the individual tools described in this book. Most of the software described in this book is included in Rocks and OSCAR or is compatible with them. However, to keep the discussions of different software clean, the book includes separate chapters for the various software packages included in Rocks and OSCAR.

This book also describes many of the customizations made by these kits. At the end of many of the chapters, there is a brief section for Rocks and OSCAR users summarizing the difference between the default, standalone installation of the software and how these kits install it. Hopefully, therefore, this book addresses both of the potential difficulties you might encounter with a cluster—learning the details of the software and discovering the differences that cluster kits introduce.

Putting aside other constraints such as the need for diskless systems or heterogeneous hardware, if all goes well, a novice can probably build a Rocks cluster a little faster than an OSCAR cluster. But if you want greater control over how your cluster is configured, you may be happier with OSCAR in the long run. Typically, OSCAR provides better documentation, although Rocks documentation has been improving. You shouldn't go far wrong with either.

2.5 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.

2.5.1 BCCD

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, *bccd*. Next, you'll answer some questions about your network. The system should autodetect your network card. Then it will prompt you for the appropriate driver. If you know the driver, select it from the list BCCD displays. Otherwise, select "auto" from the menu to have the system load drivers until a match is found. If you have a DHCP and DNS server available on your network, this will go much faster. Otherwise, you'll need to enter the usual network configuration information—IP address, netmask, gateway, etc.

Once the system boots, log in to complete the configuration process. When prompted, start the BCCD heartbeat process. Next, run the utilities *bccd-allowall* and *bccd-snarfhosts*. The first of these collects hosts' keys used by SSH and the second creates the *machines* file used by MPI. You are now ready to use the system.

Admittedly, this is a pretty brief description, but it should give you some idea as to what's involved in using BCCD. The boot process is described in greater detail at the project's web site. To perform this on a regular basis with a number of machines would be an annoying process. But for a few machines on an occasional basis, it is very straightforward.

2.6 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.

There are several things to keep in mind when selecting benchmarks. A variety of different things affect the overall performance of a cluster, including the configuration of the individual systems and the network, the job mix on the cluster, and the instruction mix in the cluster applications. Benchmarks attempt to characterize performance by measuring, in some sense, the performance of CPU, memory, or communications. Thus, there is no exact correspondence between what may affect a cluster's performance and what a benchmark actually measures.

Furthermore, since several factors are involved, different benchmarks may weight different factors. Thus, it is generally meaningless to compare the results of one benchmark on one system with a different set of benchmarks on a different system, even when the benchmarks reputedly measure the same thing.

When you select a benchmark, first decide why you need it and how it will be used. For many purposes, the best benchmark is the actual applications that you will run on your cluster. It doesn't matter how well your cluster does with memory benchmarks if your applications are constantly thrashing. The primary difficulty in using actual applications is running them in a consistent manner so that you have repeatable results. This can be a real bear! Even small changes in data can produce significant changes in performance. If you do decide to use your applications, be consistent.

If you don't want to use your applications, there are a number of cluster benchmarks available. Here are a few that you might consider:

Hierarchical Integration (HINT)

The HINT benchmark, developed at the U.S. Department of Energy's Ames Research Laboratory, is used to test subsystem performance. It can be used to compare both processor performance and memory subsystem performance. It is now supported by Brigham Young University. (<http://hint.byu.edu>)

High Performance Linpack

Linpack was written by Jack Dongarra and is probably the best known and most widely used benchmark in high-performance computing. The HPL version of Linpack is used to rank computers on the TOP500 Supercomputer Site. HPL differs from its predecessor in that the user can specify the problem size. (<http://www.netlib.org/benchmark/hpl/>)

Iozone

Iozone is an I/O and filesystem benchmark tool. It generates and performs a variety of file operations and can be used to access filesystem performance. (<http://www.iozone.org>)

Iperf

Iperf was developed to measure network performance. It measures TCP and UDP bandwidth performance, reporting delay jitter and datagram loss as well as bandwidth. (<http://dast.nlanr.net/Projects/Iperf/>)

NAS Parallel Benchmarks

The Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks (NPB) are application-centric benchmarks that have been widely used to compare the performance of parallel computers. NPB is actually a suite of eight programs. (<http://science.nas.nasa.gov/Software/NPB/>)

There are many other benchmarks available. The *Netlib Repository* is a good place to start if you need additional benchmarks, <http://www.netlib.org>.

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).

3.1 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.

Buying new, preassembled computers may be the simplest approach if money isn't the primary concern. This is often the best approach for mission-critical applications or when time is a critical factor. Buying new is also the safest way to go if you are uncomfortable assembling computers. Most system integrators will allow considerable latitude over what to include with your systems, particularly if you are buying in bulk. If you are using a system integrator, try to have the integrator provide a list of MAC addresses and label each machine.

Building your own system is cheaper, provides higher performance and reliability, and allows for customization. Assembling your own computers may seem daunting, but it isn't that difficult. You'll need time, personnel, space, and a few tools. It's a good idea to build a single system and test it for hardware and software compatibility before you commit to a large bulk order. Even if you do buy preassembled computers, you will still need to do some testing and maintenance. Unfortunately, even new computers are occasionally DOA.^[1] So the extra time may be less than you'd think. And by building your own, you'll probably be able to afford more computers.

[1] Dead on arrival: nonfunctional when first installed.

If you are constructing a dedicated cluster, you will not need full systems. The more you can leave out of each computer, the more computers you will be able to afford, and the less you will need to maintain on individual computers. For example, with dedicated clusters you can probably do without monitors, keyboards, and mice for each individual compute node. Minimal machines have the smallest footprint, allowing larger clusters when space is limited and have smaller power and air conditioning requirements. With a minimal configuration, wiring is usually significantly easier, particularly if you use rack-mounted equipment. (However, heat dissipation can be a serious problem with rack-mounted systems.) Minimal machines also have the advantage of being less likely to be reallocated by middle management.

The size of your user base will also affect your cluster design. With a broad user base, you'll need to prepare for a wider range of potential uses—more applications software and more systems tools. This implies more secondary storage and, perhaps, more memory. There is also the increased likelihood that your users will need direct access to individual nodes.

Shared machines, i.e., computers that have other uses in addition to their role as a cluster node, may be a way of constructing a part-time cluster that would not be possible otherwise. If your cluster is shared, then you will need complete, fully functioning machines. While this book won't focus on such clusters, it is certainly possible to have a setup that is a computer lab on work days and a cluster on the weekend, or office machines by day and cluster nodes at night.

3.1.1 Node Hardware

Obviously, your computers need adequate hardware for all intended uses. If your cluster includes workstations that are also used for other purposes, you'll need to consider those other uses as well. This probably means acquiring a fairly standard workstation. For a dedicated cluster, you determine your needs and there may be a lot you won't need—audio cards and speakers, video capture cards, etc. Beyond these obvious expendables, there are other additional parts you might want to consider omitting such as disk drives, keyboards, mice, and displays. However, you should be aware of some of the potential problems you'll face with a truly minimalist approach. This subsection is a quick review of the design decisions you'll need to make.

3.1.1.1 CPUs and motherboards

While you can certainly purchase CPUs and motherboards from different sources, you need to select each with the other in mind. These two items are the heart of your system. For optimal performance, you'll need total compatibility between these. If you are buying your systems piece by piece, consider buying an Intel- or ADM-compatible motherboard with an installed CPU. However, you should be aware that some motherboards with permanently affixed CPUs are poor performers, so choose with care.

You should also buy your equipment from a known, trusted source with a reputable warranty. For example, in recent years a number of boards have been released with low-grade electrolytic capacitors. While these capacitors work fine initially, the board life is disappointingly brief. People who bought these boards from fly-by-night companies were out of luck.

In determining the performance of a node, the most important factors are processor clock rate, cache size, bus speed, memory capacity, disk access speed, and network latency. The first four are determined by your selection of CPU and motherboard. And if you are using integrated EIDE interfaces and network adapters, all six are at least influenced by your choice of CPU and motherboard.

Clock speed can be misleading. It is best used to compare processors within the same family since comparing processors from different families is an unreliable way to measure performance. For example, an AMD Athlon 64 may outperform an Intel Pentium 4 when

running at the same clock rate. Processor speed is also very application dependent. If your data set fits within the large cache in a Prescott-core Pentium 4 but won't fit in the smaller cache in an Athlon, you may see much better performance with the Pentium.

Selecting a processor is a balancing act. Your choice will be constrained by cost, performance, and compatibility. Remember, the rationale behind a commodity off-the-shelf (COTS) cluster is buying machines that have the most favorable price to performance ratio, not pricey individual machines. Typically you'll get the best ratio by purchasing a CPU that is a generation behind the current cutting edge. This means comparing the numbers. When comparing CPUs, you should look at the increase in performance versus the increase in the *total cost* of a node. When the cost starts rising significantly faster than the performance, it's time to back off. When a 20 percent increase in performance raises your cost by 40 percent, you've gone too far.

Since Linux works with most major chip families, stay mainstream and you shouldn't have any software compatibility problems. Nonetheless, it is a good idea to test a system before committing to a bulk purchase. Since a primary rationale for building your own cluster is the economic advantage, you'll probably want to stay away from the less common chips. While clusters built with UltraSPARC systems may be wonderful performers, few people would describe these as commodity systems. So unless you just happen to have a number of these systems that you aren't otherwise using, you'll probably want to avoid them.^[2]

^[2] Radajewski and Eadline's *Beowulf HOWTO* refers to "*Computer Shopper*"-certified equipment. That is, if equipment isn't advertised in *Computer Shopper*, it isn't commodity equipment.

With standalone workstations, the overall benefit of multiple processors (i.e., SMP systems) is debatable since a second processor can remain idle much of the time. A much stronger argument can be made for the use of multiple processor systems in clusters where heavy utilization is assured. They add additional CPUs without requiring additional motherboards, disk drives, power supplies, cases, etc.

When comparing motherboards, look to see what is integrated into the board. There are some significant differences. Serial, parallel, and USB ports along with EIDE disk adapters are fairly standard. You may also find motherboards with integrated FireWire ports, a network interface, or even a video interface. While you may be able to save money with built-in network or display interfaces (provided they actually meet your needs), make sure they can be disabled should you want to install your own adapter in the future. If you are really certain that some fully integrated motherboard meets your needs, eliminating the need for daughter cards may allow you to go with a small case. On the other hand, expandability is a valuable hedge against the future. In particular, having free memory slots or adapter slots can be crucial at times.

Finally, make sure the BIOS Setup options are compatible with your intended configuration. If you are building a minimal system without a keyboard or display, make sure the BIOS will allow you to boot without them attached. That's not true for some BIOSs.

3.1.1.2 Memory and disks

Subject to your budget, the more cache and RAM in your system, the better. Typically, the faster the processor, the more RAM you will need. A very crude rule of thumb is one byte of RAM for every floating-point operation per second. So a processor capable of 100 MFLOPs would need around 100 MB of RAM. But don't take this rule too literally.

Ultimately, what you will need depends on your applications. Paging creates a severe performance penalty and should be avoided whenever possible. If you are paging frequently, then you should consider adding more memory. It comes down to matching the memory size to the cluster application. While you may be able to get some idea of what you will need by profiling your application, if you are creating a new cluster for as yet unwritten applications, you will have little choice but to guess what you'll need as you build the cluster and then evaluate its performance after the fact. Having free memory slots can be essential under

these circumstances.

Which disks to include, if any, is perhaps the most controversial decision you will make in designing your cluster. Opinions vary widely. The cases both for and against diskless systems have been grossly overstated. This decision is one of balancing various tradeoffs. Different contexts tip the balance in different directions. Keep in mind, diskless systems were once much more popular than they are now. They disappeared for a reason. Despite a lot of hype a few years ago about thin clients, the reemergence of these diskless systems was a spectacular flop. Clusters are, however, a notable exception. Diskless clusters are a widely used, viable approach that may be the best solution in some circumstances.

There are a number of obvious advantages to diskless systems. There is a lower cost per machine, which means you may be able to buy a bigger cluster with better performance. With rapidly declining disk prices, this is becoming less of an issue. A small footprint translates into lowered power and HVAC needs. And once the initial configuration has stabilized, software maintenance is simpler.

But the real advantage of diskless systems, at least with large clusters, is reduced maintenance. With diskless systems, you eliminate all moving parts aside from fans. For example, the average life (often known as mean time between failures, mean time before failure, or mean time to failure) of one manufacturer's disks is reported to be 300,000 hours or 34 years of continuous operation. If you have a cluster of 100 machines, you'll replace about three of these drives a year. This is a nuisance, but doable. If you have a cluster with 12,000 nodes, then you are looking at a failure, on average, every 25 hours—roughly once a day.

There is also a downside to consider. Diskless systems are much harder for inexperienced administrators to configure, particularly with heterogeneous hardware. The network is often the weak link in a cluster. In diskless systems the network will see more traffic from the network file system, compounding the problem. Paging across a network can be devastating to performance, so it is critical that you have adequate local memory. But while local disks can reduce network traffic, they don't eliminate it. There will still be a need for network-accessible file systems.

Simply put, disk-based systems are more versatile and more forgiving. If you are building a dedicated cluster with new equipment and have experience with diskless systems, you should definitely consider diskless systems. If you are new to clusters, a disk-based cluster is a safer approach. (Since this book's focus is getting started with clusters, it does not describe setting up diskless clusters.)

If you are buying hard disks, there are three issues: interface type (EIDE vs. SCSI), disk latency (a function of rotational speed), and disk capacity. From a price-performance perspective, EIDE is probably a better choice than SCSI since virtually all motherboards include a built-in EIDE interface. And unless you are willing to pay a premium, you won't have much choice with respect to disk latency. Almost all current drives rotate at 7,200 RPM. While a few 10,000 RPM drives are available, their performance, unlike their price, is typically not all that much higher. With respect to disk capacity, you'll need enough space for the operating system, local paging, and the data sets you will be manipulating. Unless you have extremely large data sets, when recycling older computers a 10 GB disk should be adequate for most uses. Often smaller disks can be used. For new systems, you'll be hard pressed to find anything smaller than 20 GB, which should satisfy most uses. Of course, other non-cluster needs may dictate larger disks.

You'll probably want to include either a floppy drive or CD-ROM drive in each system. Since CD-ROM drives can be bought for under \$15 and floppy drives for under \$5, you won't save much by leaving these out. For disk-based systems, CD-ROMs or floppies can be used to initiate and customize network installs. For example, when installing the software on compute nodes, you'll typically use a boot floppy for OSCAR systems and a CD-ROM on Rocks systems. For diskless systems, CD-ROMs or floppies can be used to boot systems over the network without special BOOT ROMs on your network adapters. The only compelling reason to not include a CD-ROM or floppy is a lack of space in a truly minimal system.

When buying any disks, don't forget the cables.

3.1.1.3 Monitors, keyboards, and mice

Many minimal systems elect not to include monitors, keyboards, or mice but rely on the network to provide local connectivity as needed. While this approach is viable only with a dedicated cluster, its advantages include lower cost, less equipment to maintain, and a smaller equipment footprint. There are also several problems you may encounter with these *headless* systems. Depending on the system BIOS, you may not be able to boot a system without a display card or keyboard attached. When such systems boot, they probe for an attached keyboard and monitor and halt if none are found. Often, there will be a CMOS option that will allow you to override the test, but this isn't always the case.

Another problem comes when you need to configure or test equipment. A lack of monitor and keyboard can complicate such tasks, particularly if you have network problems. One possible solution is the use of a crash cart—a cart with keyboard, mouse, and display that can be wheeled to individual machines and connected temporarily. Provided the network is up and the system is booting properly, X Windows or VNC provide a software solution.

Yet another alternative, particularly for small clusters, is the use of a *keyboard-video-mouse (KVM) switch*. With these switches, you can attach a single keyboard, mouse, and monitor to a number of different machines. The switch allows you to determine which computer is currently connected. You'll be able to access only one of the machines at a time, but you can easily cycle among the machines at the touch of a button. It is not too difficult to jump between machines and perform several tasks at once. However, it is fairly easy to get confused about which system you are logged on to. If you use a KVM switch, it is a good idea to configure the individual systems so that each displays its name, either as part of the prompt for command-line systems or as part of the background image for GUI-based systems.

There are a number of different switches available. Avocet even sells a KVM switch that operates over IP and can be used with remote clusters. Some KVM switches can be very pricey so be sure to shop around. Don't forget to include the cost of cables when pricing KVM switches. Frequently, these are not included with the switch and are usually overpriced. You'll need a set for every machine you want to leave connected, but not necessarily every machine.

The interaction between the system and the switch may provide a surprise or two. As previously noted, some systems don't allow booting without a keyboard, i.e., there is no CMOS override for booting without a keyboard. A KVM switch may be able to fool these systems. Such systems may detect a keyboard when connected to a KVM switch even when the switch is set to a different system. On the other hand, if you are installing Linux on a computer and it probes for a monitor, unless the switch is set to that system, the monitor won't be found.



Keep in mind, both the crash cart and the KVM switch approaches assume that individual machines have display adapters.

For this reason, you should seriously consider including a video card even when you are going with a headless systems. Very inexpensive cards or integrated adapters can be used since you won't need anything fancy. Typically, embedded video will only add a few dollars to the price of a motherboard.

One other possibility is to use serial consoles. Basically, the idea is to replace the attached monitor and keyboard with a serial connection to a remote system. With a fair amount of work, most Linux systems can be reconfigured to work in this manner. If you are using rack-

mount machines, many of them support serial console redirection out of the box. With this approach, the systems use a connection to a serial port to eliminate the need for a KVM switch. Additional hardware is available that will allow you to multiplex serial connections from a number of machines. If this approach is of interest, consult the Remote Serial Console HOWTO at <http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/>.

3.1.1.4 Adapters, power supplies, and cases

As just noted, you should include a video adapter. The network adapter is also a key component. You must buy an adapter that is compatible with the cluster network. If you are planning to boot a diskless system over the network, you'll need an adapter that supports it. This translates into an adapter with an appropriate network BOOT ROM, i.e., one with *pre-execution environment (PXE)* support. Many adapters come with a built-in (but empty) BOOT ROM socket so that the ROM can be added. You can purchase BOOT ROMs for these cards or burn your own. However, it may be cheaper to buy a new card with an installed BOOT ROM than to add the BOOT ROMs. And unless you are already set up to burn ROMs, you'll need to be using several machines before it becomes cost effective to buy an EPROM burner.

To round things out, you'll need something to put everything in and a way to supply power, i.e., a case and power supply. With the case, you'll have to balance keeping the footprint small and having room to expand your system. If you buy too small a power supply, it won't meet your needs or allow you to expand your system. If you buy too large a power supply, you waste money and space. If you add up the power requirements for your individual components and add in another 50 percent as a fudge factor, you should be safe.

One last word about node selection—while we have considered components individually, you should also think about the system collectively before you make a final decision. If collectively the individual systems generate more heat that you can manage, you may need to reconsider how you configure individual machines. For example, Google is said to use less-powerful machines in its clusters in order to balance computation needs with total operational costs, a judgment that includes the impact of cooling needs.

3.1.2 Cluster Head and Servers

Thus far, we have been looking at the compute nodes within the cluster. Depending on your configuration, you will need a head node and possibly additional servers. Ideally, the head node and most servers should be complete systems since it will add little to your overall cost and can simplify customizing and maintaining these systems. Typically, there is no need for these systems to use the same hardware that your compute nodes use. Go for enhancements that will improve performance that you might not be able to afford on every node. These machines are the place for large, fast disks and lots of fast memory. A faster processor is also in order.

On smaller clusters, you can usually use one machine as both the head and as the network file server. This will be a dual-homed machine (two network interfaces) that serves as an access point for the cluster. As such, it will be configured to limit and control access as well as provide it. When the services required by the network file systems put too great a strain on the head node, the network file system can be moved to a separate server to improve performance.

If you are setting up systems as I/O servers for a parallel file system, it is likely that you'll want larger and faster drives on these systems. Since you may have a number of I/O servers in a larger cluster, you may need to look more closely at cost and performance trade-offs.

3.1.3 Cluster Network

By definition, a cluster is a networked collection of computers. For commodity clusters, networking is often the weak link. The two key factors to consider when designing your network are bandwidth and latency. Your application or application mix will determine just how important these two factors are. If you need to move large blocks of data, bandwidth will be critical. For real-time applications or applications that have lots of interaction among nodes, minimizing latency is critical. If you have a mix of applications, both can be critical.

It should come as no surprise that a number of approaches and products have been developed. High-end Ethernet is probably the most common choice for clusters. But for some low-latency applications, including many real-time applications, you may need to consider specialized low-latency hardware. There are a number of choices. The most common alternative to Ethernet is *Myrinet* from Myricom, Inc. Myrinet is a proprietary solution providing high-speed bidirectional connectivity (currently about 2 Gbps in each direction) and low latencies (currently under 4 microseconds). Myrinet uses a source-routing strategy and allows arbitrary length packets.

Other competitive technologies that are emerging or are available include *cLAN* from Emulex, *QsNet* from Quadrics, and *Infiniband* from the Infiniband consortium. These are high-performance solutions and this technology is rapidly changing.

The problem with these alternative technologies is their extremely high cost. Adapters can cost more than the combined cost of all the other hardware in a node. And once you add in the per node cost of the switch, you can easily triple the cost of a node. Clearly, these approaches are for the high-end systems.

Fortunately, most clusters will not need this extreme level of performance. Continuing gains in speed and rapidly declining costs make Ethernet the network of choice for most clusters. Now that Gigabit Ethernet is well established and 10 Gigabit Ethernet has entered the marketplace, the highly expensive proprietary products are no longer essential for most needs.

For Gigabit Ethernet, you will be better served with an embedded adapter rather than an add-on PCI board since Gigabit can swamp the PCI bus. Embedded adapters use workarounds that take the traffic off the PCI bus. Conversely, with 100BaseT, you may prefer a separate adapter rather than an embedded one since an embedded adapter may steal clock cycles from your applications.

Unless you are just playing around, you'll probably want, at minimum, switched Fast Ethernet. If your goal is just to experiment with clusters, almost any level of networking can be used. For example, clusters have been created using FireWire ports. For two (or even three) machines, you can create a cluster using crossover cables.

Very high-performance clusters may have two parallel networks. One is used for messages passing among the nodes, while the second is used for the network file system. In the past, elaborate technology, architectures, and topologies have been developed to optimize communications. For example, channel bonding uses multiple interfaces to multiplex channels for higher bandwidth. Hypercube topologies have been used to minimize communication path length. These approaches are beyond the scope of this book. Fortunately, declining networking prices and faster networking equipment have lessened the need for these approaches.

3.2 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.

3.2.1 Cluster Layout

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.

Cable management is also a concern. For the well-heeled, there are a number of cable management systems on the market. Ideally, you want to keep power cables and data cables separated. The traditional rule of thumb was that there should be at least a foot of separation between parallel data cables and power cables runs, and that data cables and power cables should cross at right angles. In practice, the 60Hz analog power signal doesn't affect high-speed digital signals. Still, separating cables can make your cluster more manageable.

Standard equipment racks are very nice if you can afford them. Cabling is greatly simplified. But keep in mind that equipment racks pack things very closely and heat can be a problem. One rule of thumb is to stay under 100 W per square foot. That is about 1000 W for a 6-foot, 19-inch rack.

Otherwise, you'll probably be using standard shelving. My personal preference is metal shelves that are open on all sides. When buying shelves, take into consideration both the size and the weight of all the equipment you will have. Don't forget any displays, keyboards, mice, KVM switches, network switches, or uninterruptible power supplies that you plan to use. And leave yourself some working room.

3.2.2 Power and Air Conditioning

You'll need to make sure you have adequate power for your cluster, and to remove all the heat generated by that power, you'll need adequate air conditioning. For small clusters, power and air conditioning may not be immediate concerns (for now!), but it doesn't hurt to estimate your needs. If you are building a large cluster, take these needs into account from the beginning. Your best bet is to seek professional advice if it is readily available. Most large organizations have heating, ventilation, and air conditioning (HVAC) personnel and electricians on staff. While you can certainly estimate your needs yourself, if you have any problems you will need to turn to these folks for help, so you might want to include them from the beginning. Also, a second set of eyes can help prevent a costly mistake.

3.2.2.1 Power

In an ideal universe, you would simply know the power requirements of your cluster. But if you haven't built it yet, this knowledge can be a little hard to come by. The only alternative is to estimate your needs. A rough estimate is fairly straightforward: just inventory all your equipment and then add up all the wattages. Divide the total wattage by the voltage to get the amperage for the circuit, and then figure in an additional 50 percent or so as a safety factor.

For a more careful analysis, you should take into account the *power factor*. A switching power supply can draw more current than reported by their wattage ratings. For example, a fully loaded 350 W power supply may draw 500 W for 70 percent of the time and be off the other 30 percent of the time. And since a power supply may be 70 percent efficient, delivering those 500 W may require around 715 W. In practice, your equipment will rarely operate at maximum-rated capacity. Some power supplies are *power-factor corrected* (PFC). These power supplies will have power factors closer to 95 percent than 70 percent.

As you can see, this can get complicated very quickly. Hopefully, you won't be working with fully loaded systems. On the other hand, if you expect your cluster to grow, plan for more. Having said all this, for small clusters a 20-amp circuit should be adequate, but there are no guarantees.

When doing your inventory, the trick is remembering to include everything that enters the environment. It is not just the computers, network equipment, monitors, etc., that make up a cluster. It includes everything—equipment that is only used occasionally such as vacuum cleaners, personal items such as the refrigerator under your desk, and fixtures such as lights. (Ideally, you should keep the items that potentially draw a lot of current, such as vacuum cleaners, floor polishers, refrigerators, and laser printers, off the circuits your cluster is on.) Also, be careful to ensure you aren't sharing a circuit unknowingly—a potential problem in an older building, particularly if you have remodeled and added partitions.

The quality of your power can be an issue. If in doubt, put a line monitor on your circuit to see how it behaves. You might consider an uninterruptible power supply (UPS), particularly for your servers or head nodes. However, the cost can be daunting when trying to provide UPSs for an entire cluster. Moreover, UPSs should not be seen as an alternative to adequate wiring. If you are interested in learning more about or sizing a UPS, see the UPS FAQ at the site of the Linux Documentation Project (<http://www.tldp.org/>).

While you are buying UPSs, you may also want to consider buying other power management equipment. There are several vendors that supply managed power distribution systems. These often allow management over the Internet, through a serial connection, or via SNMP. With this equipment, you'll be able to monitor your cluster and remotely power-down or reboot equipment.

And one last question to the wise:



Do you know how to kill the power to your system?

This is more than idle curiosity. There may come a time when you don't want power to your cluster. And you may be in a big hurry when the time comes.

Knowing where the breakers are is a good start. Unfortunately, these may not be close at hand. They may even be locked away in a utility closet. One alternative is a *scram* switch. A scram switch should be installed between the UPS and your equipment. You should take care to ensure the switch is accessible but will not inadvertently be thrown.

You should also ensure that your maintenance staff knows what a UPS is. I once had a server/UPS setup in an office that flooded. When I came in, the UPS had been unplugged from the wall, but the computer was still plugged into the UPS. Both computer and UPS were drenched—a potentially deadly situation. Make sure your maintenance staff knows what they are dealing with.

3.2.2.2 HVAC

As with most everything else, when it comes to electronics, heat kills. There is no magical temperature or temperature range that if you just keep your computers and other equipment within that range, everything will be OK. Unfortunately, it just isn't that simple.

Failure rate is usually a nonlinear function of temperature. As the temperature rises, the probability of failure also increases. For small changes in temperature, a rough rule of thumb is that you can expect the failure rate to double with an 18F (10C) increase in temperature. For larger changes, the rate of failure typically increases more rapidly than the rise in temperature. Basically, you are playing the odds. If you operate your machine room at a higher than average temperature, you'll probably see more failures. It is up to you to decide if the failure rate is unacceptable.

Microenvironments also matter. It doesn't matter if it is nice and cool in your corner of the room if your equipment rack is sitting in a corner in direct sunlight where the temperature is 15F (8C) warmer. If the individual pieces of equipment don't have adequate cooling, you'll have problems. This means that computers that are spread out in a room with good ventilation may be better off at a higher room temperature than those in a tightly packed cluster that lacks ventilation, even when the room temperature is lower.

Finally, the failure rate will also depend on the actual equipment you are using. Some equipment is designed and constructed to be more heat tolerant, e.g., military grade equipment. Consult the specifications if in doubt.

While occasionally you'll see recommended temperature ranges for equipment or equipment rooms, these should be taken with a grain of salt. Usually, recommended temperatures are a little below 70F (21C). So if you are a little chilly, your machines are probably comfortable.

Maintaining a consistent temperature can be a problem, particularly if you leave your cluster up and running at night, over the weekend, and over holidays. Heating and air conditioning are often turned off or scaled back when people aren't around. Ordinarily, this makes good economic sense. But when the air conditioning is cut off for a long Fourth of July weekend, equipment can suffer. Make sure you discuss this with your HVAC folks before it becomes a problem. Again, occasional warm spells *probably* won't be a problem, but you are pushing your luck.

Humidity is also an issue. At a high humidity, condensation can become a problem; at a low humidity, static electricity is a problem. The optimal range is somewhere in between. Recommended ranges are typically around 40 percent to 60 percent.

Estimating your air conditioning needs is straightforward but may require information you don't have. Among other things, proper cooling depends on the number and area of external walls, the number of windows and their exposure to the sun, the external temperature, and insulation. Your maintenance folks may have already calculated all this or may be able to estimate some of it.

What you are adding is heat contributed by your equipment and staff, something that your maintenance folks may not have been able to accurately predict. Once again, you'll start with an inventory of your equipment. You'll want the total wattage. You can convert this to British Thermal Units per hour by multiplying the wattage by 3.412. Add in another 300 BTU/H for each person working in the area. Add in the load from the lights, walls, windows, etc., and then figure in another 50 percent as a safety factor. Since air conditioning is usually expressed in tonnage, you may need to divide the BTU/H total by 12,000 to get the tonnage you need. (Or, just let the HVAC folks do all this for you.)

3.2.3 Physical Security

Physical security includes both controlling access to computers and protecting computers from physical threats such as flooding. If you are concerned about someone trying to break into your computers, the best solution is to take whatever steps you can to ensure that they don't have physical access to the computers. If you can't limit access to the individual computers, then you should password protect the CMOS, set the boot order so the system only boots from the hard drive, and put a lock on each case. Otherwise, someone can open the case and remove the battery briefly (roughly 15 to 20 minutes) to erase the information in CMOS including the password.^[3] With the password erased, the boot order can be changed. Once this is done, it is a simple matter to boot to a floppy or CD-ROM, mount the hard drive, and edit the password files, etc. (Even if you've removed both floppy and CD-ROM drives, an intruder could bring one with them.) Obviously, this solution is only as good as the locks you can put on the computers and does very little to protect you from vandals.

^[3] Also, there is usually a jumper that will immediately discharge the CMOS.

Broken pipes and similar disasters can be devastating. Unfortunately, it can be difficult to access these potential threats. Computers can be damaged when a pipe breaks on another floor. Just because there is no pipe immediately overhead doesn't mean that you won't be rained on as water from higher floors makes its way to the basement. Keeping equipment off the floor and off the top of shelves can provide some protection. It is also a good idea to keep equipment away from windows.

There are several web sites and books that deal with disaster preparedness. As the importance of your cluster grows, disaster preparedness will become more important.

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.

4.1 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.

4.1.1 Selecting a Distribution

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.

Unfortunately, at some point you can expect support for older systems to be discontinued. However, a system will not stop working just because it isn't supported. While not desirable, this is also something you can live with.

The final and key issue is software compatibility. Keep in mind that it takes time to develop software for use with a new release, particularly if you are customizing the kernel. As a result, the clustering software you want to use may not be available for the latest version of your favorite Linux distribution. In general, software distributed as libraries (e.g., MPI) are more forgiving than software requiring kernel patches (e.g., openMosix) or software that builds kernel modules (e.g., PVFS). These latter categories, by their very nature, must be system specific. Remember that using clustering software is the *raison d'être* for your cluster. If you can't run it, you are out of business. Unless you are willing to port the software or compromise your standards, you may be forced to use an older version of Linux. While you may want the latest and greatest version of your favorite flavor of Linux, you need to get over it.

If at all feasible, it is best to start your cluster installation with a clean install of Linux. Of course, if you are adding clustering software to existing systems, this may not be feasible, particularly if the machines are not dedicated to the cluster. If that is the case, you'll need to tread lightly. You'll almost certainly need to make changes to these systems, changes that may not go as smoothly as you'd like. Begin by backing up and carefully documenting these systems.

4.1.2 Downloading Linux

With most flavors of Linux, there are several ways you can do the installation. Typically you can install from a set of CD-ROMs, from a hard disk partition, or over a network using NFS, FTP, or HTTP. The decision will depend in part on the hardware you have available, but for initial experimentation it is probably easiest to use CD-ROMs. Buying a boxed set can be a real convenience, particularly if it comes with a printed set of manuals. But if you are using an older version of Linux, finding a set of CD-ROMs to buy can be difficult. Fortunately, you should have no trouble finding what you need on the Internet.

Downloading is the cheapest and easiest way to go if you have a fast Internet connection and a CD-ROM burner. Typically, you download ISO images—disk images for CD-ROMs. These are basically single-file archives of everything on a CD-ROM. Since ISO images are frequently over 600 MB each and since you'll need several of them, downloading can take hours even if you have a fast connection and days if you're using a slow modem.

If you decide to go this route, follow the installation directions from your download site. These should help clarify exactly what you need and don't need and explain any other special considerations. For example, for Red Hat Linux the place to start is <http://www.redhat.com/apps/download/>. This will give you a link to a set of directions with links to download sites. Don't overlook the mirror sites; your download may go faster with them than with Red Hat's official download site.

For Red Hat Linux 9.0, there are seven disks. (Earlier versions of Red Hat have fewer disks.) Three of these are the installation disks and are essential. Three disks contain the source files for the packages. It is very unlikely you'll ever need these. If you do, you can download them later. The last disk is a documentation disk. You'd be foolish to skip this disk. Since the files only fill a small part of a CD, the ISO image is relatively small and the download doesn't take very long.

It is a good idea to check the MD5SUM for each ISO you download. Run the *md5sum* program and compare the results to published checksums.

```
[root@cs sloanjd]# md5sum FC2-i386-rescuecd.iso
22f4bfca5baefe89f0e04166e738639f  FC2-i386-rescuecd.iso
```

This will ensure both that the disk image hasn't been tampered with and that your download wasn't corrupted.

Once you have downloaded the ISO images, you'll need to burn your CD-ROMs. If you downloaded the ISO images to a Windows computer, you could use something like Roxio Easy Creator.^[1] If you already have a running Linux system, you might use X-CD-Roast.

^[1] There is an appealing irony to using Windows to download Linux.

Once you have the CD-ROMs, you can do an installation by following the appropriate directions for your software and system. Usually, this means booting to the first CD-ROM, which, in turn, runs an installation script. If you can't boot from the CD-ROM, you'll need to create a boot floppy using the directions supplied with the software. For Red Hat Linux, see the *README* file on the first installation disk.

4.1.3 What to Install?

What you install will depend on how you plan to use the machine. Is this a dedicated cluster? If so, users probably won't log onto individual machines, so you can get by with installing the minimal software required to run applications on each compute node. Is it a cluster of workstations that will be used in other ways? If that is the case, be sure to install X and any other appropriate applications. Will you be writing code? Don't forget the software development package and editors. Will you be recompiling the kernel? If so, you'll need the kernel sources.^[2] If you are building kernel modules, you'll need the kernel header files. (In particular, these are needed if you install PVFS. PVFS is described in [Chapter 12](#).) A custom installation will give you the most control over what is installed, i.e., the greatest opportunity to install software that you don't need and omit that which you do need.

^[2] In general, you should avoid recompiling the kernel unless it is absolutely necessary. While you may be able to eke out some modest performance gains, they are rarely worth the effort.

Keep in mind that you can go back and add software. You aren't trapped by what you include at this point. At this stage, the important thing is to remember what you actually did. Take careful notes and create a checklist as you proceed. The quickest way to get started is to take a minimalist approach and add anything you need later, but some people find it very annoying to have to go back and add software. If you have the extra disk space (2 GB or so), then you may want to copy all the packages to a directory on your server. Not having to mount disks and search for packages greatly simplifies adding packages as needed. You only need to do this with one system and it really doesn't take that long. Once you have worked out the details, you can create a Kickstart configuration file to automate all this. Kickstart is described in more detail in [Chapter 8](#).

4.2 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.

4.2.1 DHCP

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.)

```
# A sample DHCP configuration file.

# The first commands in this file are global,
# i.e., they apply to all clients.

# Only answer requests from known machines,
# i.e., machines whose hardware addresses are given.
deny unknown-clients;

# Set the subnet mask, broadcast address, and router address.
```

```
option subnet-mask 255.255.255.0;

option broadcast-address 172.16.1.255;

option routers 172.16.1.254;

# This section defines individual cluster nodes.
# Each subnet in the network has its own section.

subnet 172.16.1.0 netmask 255.255.255.0 {
    group {
        # The first host, identified by the given MAC address,
        # will be named node1.cluster.int, will be given the
        # IP address 172.16.1.1, and will use the default router
        # 172.16.1.254 (the head node in this case).

        host node1{
            hardware ethernet 00:08:c7:07:68:48;

            fixed-address 172.16.1.1;

            option routers 172.16.1.254;

            option domain-name "cluster.int";
        }

        host node2{
            hardware ethernet 00:08:c7:07:c1:73;

            fixed-address 172.16.1.2;

            option routers 172.16.1.254;

            option domain-name "cluster.int";
        }

        # Additional node definitions go here.
    }
}

# For servers with multiple interfaces, this entry says to ignore requests
# on specified subnets.
```

```
subnet 10.0.32.0 netmask 255.255.248.0 { not authoritative; }
```

As shown in this example, you should include a subnet section for each subnet on your network. If the head node has an interface for the cluster and a second interface connected to the Internet or your organization's network, the configuration file will have a group for each interface or subnet. Since the head node should answer DHCP requests for the cluster but not for the organization, DHCP should be configured so that it will respond only to DHCP requests from the compute nodes.

4.2.2 NFS

A network filesystem is a filesystem that physically resides on one computer (the file server), which in turn shares its files over the network with other computers on the network (the clients). The best-known and most common network filesystem is *Network File System (NFS)*. In setting up a cluster, designate one computer as your NFS server. This is often the head node for the cluster, but there is no reason it has to be. In fact, under some circumstances, you may get slightly better performance if you use different machines for the NFS server and head node. Since the server is where your user files will reside, make sure you have enough storage. This machine is a likely candidate for a second disk drive or raid array and a fast I/O subsystem. You may even want to consider mirroring the filesystem using a small high-availability cluster.

Why use an NFS? It should come as no surprise that for parallel programming you'll need a copy of the compiled code or executable on each machine on which it will run. You could, of course, copy the executable over to the individual machines, but this quickly becomes tiresome. A shared filesystem solves this problem. Another advantage to an NFS is that all the files you will be working on will be on the same system. This greatly simplifies backups. (You do backups, don't you?) A shared filesystem also simplifies setting up SSH, as it eliminates the need to distribute keys. (SSH is described later in this chapter.) For this reason, you may want to set up NFS before setting up SSH. NFS can also play an essential role in some installation strategies.

If you have never used NFS before, setting up the client and the server are slightly different, but neither is particularly difficult. Most Linux distributions come with most of the work already done for you.

4.2.2.1 Running NFS

Begin with the server; you won't get anywhere with the client if the server isn't already running. Two things need to be done to get the server running. The file */etc/exports* must be edited to specify which machines can mount which directories, and then the server software must be started. Here is a single line from the file */etc/exports* on the server *amy*.

```
/home    basil(rw) clara(rw) desmond(rw) ernest(rw) george(rw)
```

This line gives the clients *basil*, *clara*, *desmond*, *ernest*, and *george* read/write access to the directory */home* on the server. Read access is the default. A number of other options are available and could be included. For example, the *no_root_squash* option could be added if you want to edit root permission files from the nodes.



Pay particular attention to the use of spaces in this file.

Had a space been inadvertently included between `basil` and `(rw)`, read access would have been granted to `basil` and read/write access would have been granted to all other systems. (Once you have the systems set up, it is a good idea to use the command `showmount -a` to see who is mounting what.)

Once `/etc/exports` has been edited, you'll need to start NFS. For testing, you can use the `service` command as shown here

```
[root@fanny init.d]# /sbin/service nfs start

Starting NFS services:                [ OK ]

Starting NFS quotas:                  [ OK ]

Starting NFS mountd:                  [ OK ]

Starting NFS daemon:                  [ OK ]

[root@fanny init.d]# /sbin/service nfs status

rpc.mountd (pid 1652) is running...

nfsd (pid 1666 1665 1664 1663 1662 1661 1660 1657) is running...

rpc.rquotad (pid 1647) is running...
```

(With some Linux distributions, when restarting NFS, you may find it necessary to explicitly stop and restart both `nfslock` and `portmap` as well.) You'll want to change the system configuration so that this starts automatically when the system is rebooted. For example, with Red Hat, you could use the `serviceconf` or `chkconfig` commands.

For the client, the software is probably already running on your system. You just need to tell the client to mount the remote filesystem. You can do this several ways, but in the long run, the easiest approach is to edit the file `/etc/fstab`, adding an entry for the server. Basically, you'll add a line to the file that looks something like this:

```
amy:/home /home nfs rw,soft 0 0
```

In this example, the local system mounts the `/home` filesystem located on `amy` as the `/home` directory on the local machine. The filesystems may have different names. You can now manually mount the filesystem with the `mount` command

```
[root@ida /]# mount /home
```

When the system reboots, this will be done automatically.

When using NFS, you should keep a couple of things in mind. The mount point, `/home`, must exist on the client prior to mounting. While the remote directory is mounted, any files that were stored on the local system in the `/home` directory will be inaccessible. They are still there; you just can't get to them while the remote directory is mounted. Next, if you are running a firewall, it will probably block NFS traffic. If you are having problems with NFS, this is one of the first things you should check.

File ownership can also create some surprises. User and group IDs should be consistent among systems using NFS, i.e., each user will have identical IDs on all systems. Finally, be aware that root privileges don't extend across NFS shared systems (if you have configured your systems correctly). So if, as root, you change the directory (`cd`) to a remotely mounted filesystem, don't expect to be able to look at every file. (Of course, as root you can always

use *su* to become the owner and do all the snooping you want.) Details for the syntax and options can be found in the *nfs(5)*, *exports(5)*, *fstab(5)*, and *mount(8)* manpages. Additional references can be found in the [Appendix A](#).

4.2.2 Automount

The preceding discussion of NFS describes editing the */etc/fstab* to mount filesystems. There's another alternative—using an automount program such as *autofs* or *amd*. An automount daemon mounts a remote filesystem when an attempt is made to access the filesystem and unmounts the filesystem when it is no longer needed. This is all transparent to the user.

While the most common use of automounting is to automatically mount floppy disks and CD-ROMs on local machines, there are several advantages to automounting across a network in a cluster. You can avoid the problem of maintaining consistent */etc/fstab* files on dozens of machines. Automounting can also lessen the impact of a server crash. It is even possible to replicate a filesystem on different servers for redundancy. And since a filesystem is mounted only when needed, automounting can reduce network traffic. We'll look at a very simple example here. There are at least two different HOWTOs (<http://www.tldp.org/>) for automounting should you need more information.

Automounting originated at Sun Microsystems, Inc. The Linux automounter *autofs*, which mimics Sun's automounter, is readily available on most Linux systems. While other automount programs are available, most notably *amd*, this discussion will be limited to using *autofs*.

Support for *autofs* must be compiled into the kernel before it can be used. With most Linux releases, this has already been done. If in doubt, use the following to see if it is installed:

```
[root@fanny root]# cat /proc/filesystems
...
```

Somewhere in the output, you should see the line

```
nodev    autofs
```

If you do, you are in business. Otherwise, you'll need a new kernel.

Next, you need to configure your systems. *autofs* uses the file */etc/auto.master* to determine mount points. Each line in the file specifies a mount point and a map file that defines which filesystems will be mounted to the mount point. For example, in Rocks the *auto.master* file contains the single line:

```
/home auto.home --timeout 600
```

In this example, */home* is the mount point, i.e., where the remote filesystem will be mounted. The file *auto.home* specifies what will be mounted.

In Rocks, the file */etc/auto.home* will have multiple entries such as:

```
sloanjd  frontend.local:/export/home/sloanjd
```

The first field is the name of the subdirectory that will be created under the original mount point. In this example, the directory *sloanjd* will be mounted as a subdirectory of */home* on the client system. The subdirectories are created dynamically by automount and should not exist on the client. The second field is the hostname (or server) and directory that is exported. (Although not shown in this example, it is possible to specify mount parameters for each

directory in */etc/auto.home*.) NFS should be running and you may need to update your */etc/exports* file.

Once you have the configuration files copied to each system, you need to start *autofs* on each system. *autofs* is usually located in */etc/init.d* and accepts the commands `start`, `restart`, `status`, and `reload`. With Red Hat, it is available through the */sbin/service* command. After reading the file, *autofs* starts an automount process with appropriate parameters for each mount point and mounts filesystems as needed. For more information see the *autofs(8)* and *auto.master(5)* manpages.

4.2.3 Other Cluster File System

NFS has its limitations. First, there are potential security issues. Since the idea behind NFS is sharing, it should come as no surprise that over the years crackers have found ways to exploit NFS. If you are going to use NFS, it is important that you use a current version, apply any needed patches, and configure it correctly.

Also, NFS does not scale well, although there seems to be some disagreement about its limitations. For clusters, with fewer than 100 nodes, NFS is probably a reasonable choice. For clusters with more than 1,000 nodes, NFS is generally thought to be inadequate. Between 100 and 1,000 nodes, opinions seem to vary. This will depend in part on your hardware. It will also depend on how your applications use NFS. For a bioinformatics clusters, many of the applications will be read intensive. For a graphics processing cluster, rendering applications will be write intensive. You may find that NFS works better with the former than the latter. Other applications will have different characteristics, each stressing the filesystem in a different way. Ultimately, it comes down to what works best for you and your applications, so you'll probably want to do some experimenting.

Keep in mind that NFS is not meant to be a high-performance, parallel filesystem. Parallel filesystems are designed for a different purpose. There are other filesystems you could consider, each with its own set of characteristics. Some of these are described briefly in [Chapter 12](#). Additionally, there are other storage technologies such as storage area network (SAN) technology. SANs offer greatly improve filesystem failover capabilities and are ideal for use with high-availability clusters. Unfortunately, SANs are both expensive and difficult to set up. iSCSI (SCSI over IP) is an emerging technology to watch.

If you need a high-performance, parallel filesystems, PVFS is a reasonable place to start, as it is readily available for both Rocks and OSCAR. PVFS is discussed in [Chapter 12](#).

4.2.4 SSH

To run software across a cluster, you'll need some mechanism to start processes on each machine. In practice, a prerequisite is the ability to log onto each machine within the cluster. If you need to enter a password for each machine each time you run a program, you won't get very much done. What is needed is a mechanism that allows logins without passwords.

This boils down to two choices—you can use *remote shell (RSH)* or *secure shell (SSH)*. If you are a trusting soul, you may want to use RSH. It is simpler to set up with less overhead. On the other hand, SSH network traffic is encrypted, so it is safe from snooping. Since SSH provides greater security, it is generally the preferred approach.

SSH provides mechanisms to log onto remote machines, run programs on remote machines, and copy files among machines. SSH is a replacement for *ftp*, *telnet*, *rlogin*, *rsh*, and *rcp*. A commercial version of SSH is available from SSH Communications Security (<http://www.ssh.com>), a company founded by Tatu Ylönen, an original developer of SSH. Or you can go with OpenSSH, an open source version from <http://www.openssh.org>.

OpenSSH is the easiest since it is already included with most Linux distributions. It has other

advantages as well. By default, OpenSSH automatically forwards the `DISPLAY` variable. This greatly simplifies using the X Window System across the cluster. If you are running an SSH connection under X on your local machine and execute an X program on the remote machine, the X window will automatically open on the local machine. This can be disabled on the server side, so if it isn't working, that is the first place to look.

There are two sets of SSH protocols, SSH-1 and SSH-2. Unfortunately, SSH-1 has a serious security vulnerability. SSH-2 is now the protocol of choice. This discussion will focus on using OpenSSH with SSH-2.

Before setting up SSH, check to see if it is already installed and running on your system. With Red Hat, you can check to see what packages are installed using the package manager.

```
[root@fanny root]# rpm -q -a | grep ssh
```

```
openssh-3.5p1-6
```

```
openssh-server-3.5p1-6
```

```
openssh-clients-3.5p1-6
```

```
openssh-askpass-gnome-3.5p1-6
```

```
openssh-askpass-3.5p1-6
```

This particular system has the SSH core package, both server and client software as well as additional utilities. The SSH daemon is usually started as a service. As you can see, it is already running on this machine.

```
[root@fanny root]# /sbin/service sshd status
```

```
sshd (pid 28190 1658) is running...
```

Of course, it is possible that it wasn't started as a service but is still installed and running. You can use `ps` to double check.

```
[root@fanny root]# ps -aux | grep ssh
```

```
root      29133  0.0  0.2 3520  328 ?        S    Dec09   0:02 /usr/sbin/sshd
```

```
...
```

Again, this shows the server is running.

With some older Red Hat installations, e.g., the 7.3 workstation, only the client software is installed by default. You'll need to manually install the server software. If using Red Hat 7.3, go to the second install disk and copy over the file `RedHat/RPMS/openssh-server-3.1p1-3.i386.rpm`. (Better yet, download the latest version of this software.) Install it with the package manager and then start the service.

```
[root@james root]# rpm -vih openssh-server-3.1p1-3.i386.rpm
```

```
Preparing...                               ##### [100%]
```

```
 1:openssh-server                           ##### [100%]
```

```
[root@james root]# /sbin/service sshd start
```

```
Generating SSH1 RSA host key:           [ OK ]
Generating SSH2 RSA host key:           [ OK ]
Generating SSH2 DSA host key:           [ OK ]
Starting sshd:                           [ OK ]
```

When SSH is started for the first time, encryption keys for the system are generated. Be sure to set this up so that it is done automatically when the system reboots.

Configuration files for both the server, *sshd_config*, and client, *ssh_config*, can be found in */etc/ssh*, but the default settings are usually quite reasonable. You shouldn't need to change these files.

4.2.4.1 Using SSH

To log onto a remote machine, use the command `ssh` with the name or IP address of the remote machine as an argument. The first time you connect to a remote machine, you will receive a message with the remote machines' *fingerprint*, a string that identifies the machine. You'll be asked whether to proceed or not. This is normal.

```
[root@fanny root]# ssh amy

The authenticity of host 'amy (10.0.32.139)' can't be established.

RSA key fingerprint is 98:42:51:3e:90:43:1c:32:e6:c4:cc:8f:4a:ee:cd:86.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'amy,10.0.32.139' (RSA) to the list of known hosts.

root@amy's password:

Last login: Tue Dec  9 11:24:09 2003

[root@amy root]#
```

The fingerprint will be recorded in a list of known hosts on the local machine. SSH will compare fingerprints on subsequent logins to ensure that nothing has changed. You won't see anything else about the fingerprint unless it changes. Then SSH will warn you and query whether you should continue. If the remote system has changed, e.g., if it has been rebuilt or if SSH has been reinstalled, it's OK to proceed. But if you think the remote system hasn't changed, you should investigate further before logging in.

Notice in the last example that SSH automatically uses the same identity when logging into a remote machine. If you want to log on as a different user, use the `-l` option with the appropriate account name.

You can also use SSH to execute commands on remote systems. Here is an example of using *date* remotely.

```
[root@fanny root]# ssh -l sloanjd hector date

sloanjd@hector's password:

Mon Dec 22 09:28:46 EST 2003
```

Notice that a different account, *sloanjd*, was used in this example.

To copy files, you use the `scp` command. For example,

```
[root@fanny root]# scp /etc/motd george:/root/
root@george's password:
motd                               100% |*****|           0          00:00
```

Here file */etc/motd* was copied from *fanny* to the */root* directory on *george*.

In the examples thus far, the system has asked for a password each time a command was run. If you want to avoid this, you'll need to do some extra work. You'll need to generate a pair of authorization keys that will be used to control access and then store these in the directory *~/.ssh*. The *ssh-keygen* command is used to generate keys.

```
[sloanjd@fanny sloanjd]$ ssh-keygen -b1024 -trsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/sloanjd/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/sloanjd/.ssh/id_rsa.
Your public key has been saved in /home/sloanjd/.ssh/id_rsa.pub.
The key fingerprint is:
2d:c8:d1:e1:bc:90:b2:f6:6d:2e:a5:7f:db:26:60:3f sloanjd@fanny
[sloanjd@fanny sloanjd]$ cd .ssh
[sloanjd@fanny .ssh]$ ls -a
.  ..  id_rsa  id_rsa.pub  known_hosts
```

The options in this example are used to specify a 1,024-bit key and the RSA algorithm. (You can use DSA instead of RSA if you prefer.) Notice that SSH will prompt you for a passphrase, basically a multi-word password.

Two keys are generated, a public and a private key. The private key should never be shared and resides only on the client machine. The public key is distributed to remote machines. Copy the public key to each system you'll want to log onto, renaming it *authorized_keys2*.

```
[sloanjd@fanny .ssh]$ cp id_rsa.pub authorized_keys2
[sloanjd@fanny .ssh]$ chmod go-rwx authorized_keys2
[sloanjd@fanny .ssh]$ chmod 755 ~/.ssh
```

If you are using NFS, as shown here, all you need to do is copy and rename the file in the

current directory. Since that directory is mounted on each system in the cluster, it is automatically available.



If you used the NFS setup described earlier, root's home directory */root*, is not shared. If you want to log in as root without a password, manually copy the public keys to the target machines. You'll need to decide whether you feel secure setting up the root account like this.

You will use two utilities supplied with SSH to manage the login process. The first is an SSH agent program that caches private keys, *ssh-agent*. This program stores the keys locally and uses them to respond to authentication queries from SSH clients. The second utility, *ssh-add*, is used to manage the local key cache. Among other things, it can be used to add, list, or remove keys.

```
[sloanjd@fanny .ssh]$ ssh-agent $SHELL
```

```
[sloanjd@fanny .ssh]$ ssh-add
```

```
Enter passphrase for /home/sloanjd/.ssh/id_rsa:
```

```
Identity added: /home/sloanjd/.ssh/id_rsa (/home/sloanjd/.ssh/id_rsa)
```

(While this example uses the `$SHELL` variable, you can substitute the actual name of the shell you want to run if you wish.) Once this is done, you can log in to remote machines without a password.

This process can be automated to varying degrees. For example, you can add the call to *ssh-agent* as the last line of your login script so that it will be run before you make any changes to your shell's environment. Once you have done this, you'll need to run *ssh-add* only when you log in. But you should be aware that Red Hat console logins don't like this change.

You can find more information by looking at the *ssh(1)*, *ssh-agent(1)*, and *ssh-add(1)* manpages. If you want more details on how to set up *ssh-agent*, you might look at *SSH, The Secure Shell* by Barrett and Silverman, O'Reilly, 2001. You can also find scripts on the Internet that will set up a persistent agent so that you won't need to rerun *ssh-add* each time.



One last word of warning: If you are using *ssh-agent*, it becomes very important that you log off whenever you leave your machine. Otherwise, you'll be leaving not just one system wide open, but all of your systems.

4.2.5 Other Services and Configuration Tasks

Thus far, we have taken a minimalist approach. To make life easier, there are several other services that you'll want to install and configure. There really isn't anything special that you'll need to do—just don't overlook these.

4.2.5.1 Apache

While an HTTP server may seem unnecessary on a cluster, several cluster management tools such as Clumon and Ganglia use HTTP to display results. If you will monitor your cluster only from the head node, you may be able to get by without installing a server. But if you want to do remote monitoring, you'll need to install an HTTP server. Since most management

packages like these assume Apache will be installed, it is easiest if you just go ahead and set it up when you install your cluster.

4.2.5.2 Network Time Protocol (NTP)

It is important to have synchronized clocks on your cluster, particularly if you want to do performance monitoring or profiling. Of course, you don't have to synchronize your system to the rest of the world; you just need to be internally consistent. Typically, you'll want to set up the head node as an NTP server and the compute nodes as NTP clients. If you can, you should sync the head node to an external timeserver. The easiest way to handle this is to select the appropriate option when you install Linux. Then make sure that the NTP daemon is running:

```
[root@fanny root]# /sbin/service ntpd status
```

```
ntpd (pid 1689) is running...
```

Start the daemon if necessary.

4.2.5.3 Virtual Network Computing (VNC)

This is a very nice package that allows remote graphical logins to your system. It is available as a Red Hat package or from <http://www.realvnc.com/>. VNC can be tunneled using SSH for greater security.

4.2.5.4 Multicasting

Several clustering utilities use multicasting to distribute data among nodes within a cluster, either for cloning systems or when monitoring systems. In some instances, multicasting can greatly increase performance. If you are using a utility that relies on multicasting, you'll need to ensure that multicasting is supported. With Linux, multicasting must be enabled when the kernel is built. With most distributions, this is not a problem. Additionally, you will need to ensure that an appropriate multicast entry is included in your route tables. You will also need to ensure that your networking equipment supports multicast. This won't be a problem with hubs; this may be a problem with switches; and, should your cluster span multiple networks, this will definitely be an issue with routers. Since networking equipment varies significantly from device to device, you need to consult the documentation for your specific hardware. For more general information on multicasting, you should consult the multicasting HOWTOs.

4.2.5.5 Hosts file and name services

Life will be much simpler in the long run if you provide appropriate name services. NIS is certainly one possibility. At a minimum, don't forget to edit */etc/hosts* for your cluster. At the very least, this will reduce network traffic and speed up some software. And some packages assume it is correctly installed. Here are a few lines from the host file for *amy*.

```
127.0.0.1          localhost.localdomain localhost
```

```
10.0.32.139       amy.wofford.int    amy
```

```
10.0.32.140       basil.wofford.int  basil
```

```
...
```

Notice that `amy` is not included on the line with `localhost`. Specifying the host name as an alias for `localhost` can break some software.

4.3 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.

Since a thorough discussion of security could easily add a few hundred pages to this book, it is necessary to assume that you know the basics of security. If you are a novice system administrator, this is almost certainly not the case, and you'll need to become proficient as quickly as possible. To get started, you should:

- Be sure to apply all appropriate security patches, at least to the head node, and preferably to all nodes. This is a task you will need to do routinely, not just when you set up the cluster.
- Know what is installed on your system. This can be a particular problem with cluster kits. Audit your systems regularly.
- Differentiate between what's available inside the cluster and what is available outside the cluster. For example, don't run NFS outside the cluster. Block *portmapper* on the public interface of the head node.
- Don't put too much faith in firewalls, but use one, at least on the head node's public interface, and ensure that it is configured correctly.
- Don't run services that you don't need. Routinely check which services are running, both with *netstat* and with a port scanner like *nmap*.
- Your head node should be dedicated to the cluster, if at all possible. Don't set it up as a

general server.

- Use the root account only when necessary. Don't run programs as root unless it is absolutely necessary.

There is no easy solution to the security dilemma. While you may be able to learn enough, you'll never be able to learn it all.



Part II: Getting Started Quickly

This section describes the installation of three software packages that, when installed, will provide you with a complete working cluster. These packages differ radically. openMosix provides Linux kernel extensions that transparently move processes among machines to balance loads and optimize performance. While a truly remarkable package, it is not what people typically think about when they hear the word "cluster." OSCAR and Rocks are collections of software packages that can be installed at once, providing a more traditional Beowulf-style cluster. Whichever way you decide to go, you will be up and running in short order.

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.

5.1 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 supplies you with the tools to do just that.

(Currently, openMosix also includes a distributed filesystem. However, this is slated for removal in future releases. The new goal is to integrate support for a clustering filesystem such as Intermezzo.)

5.2 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.^[1] 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 *unique home node (UHN)*, the node where the process was initially created, before continuing.

[1] Actually, the *migration of shared memory (MigSHM)* patch is an openMosix patch that implements shared memory migration. At the time this was written, it was not part of the main openMosix tree. (Visit <http://mcaserta.com/maask/>.)

To support process migration, openMosix divides processes into two parts or *contexts*. The *user context* contains the program code, stack, data, etc., and is the part that can migrate. The *system context*, which contains a description of the resources the process is attached to and the kernel stack, does not migrate but remains on the UHN.

openMosix uses an adaptive resource allocation policy. That is, each node monitors and compares its own load with the loads on a portion of the other computers within the cluster. When a computer finds a more lightly loaded computer (based on the overall capacity of the computer), it will attempt to migrate a process to the more lightly loaded computer, thereby creating a more balanced load between the two. As the loads on individual computers change, e.g., when jobs start or finish, processes will migrate among the computers to rebalance loads across the cluster, adapting dynamically to the changes in loads.

Individual nodes, acting as autonomous systems, decide which processes migrate. The communications among small sets of nodes within the cluster used to compare loads is randomized. Consequently, clusters scale well because of this random element. Since communications is within subsets in the cluster, nodes have limited but recent information

about the state of the whole cluster. This approach reduces overhead and communication.

While load comparison and process migration are generally automatic within a cluster, openMosix provides tools to control migration. It is possible to alter the cluster's perception of how heavily an individual computer is loaded, to tie processes to a specific computer, or to block the migration of processes to a computer. However, precise control for the migration of a group of processes is not practical with openMosix at this time.^[2]

^[2] This issue is addressed by a patch that allows the creation of process groups, available at <http://www.openmosixview.com/miggroup/>.

The openMosix API uses the values in the flat files in */proc/hpc* to record and control the state of the cluster. If you need information about the current configuration, want to do really low-level management, or write management scripts, you can look at or write to these files.



5.3 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.

The openMosix user tools should be downloaded when you download the openMosix kernel or kernel patches. Additionally, you will also want to download and install *openMosixView*, third-party tools for openMosix.

5.4 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.

5.4.1 Downloading

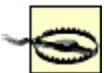
You'll find links to available packages at <http://openmosix.sourceforge.net>.^[3] You'll need to select from among several versions and compilations. At the time this was written, there were half a dozen different kernel versions available. For each of these, there were eight possible downloads, including 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.

^[3] And while you are at it, you should also download a copy of Kris Buytaert's *openMosix HOWTO* from <http://www.tldp.org/HOWTO/openMosix-HOWTO/>.

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 *openMosix*, you are adding a new kernel.



Don't delete the old kernel. As long as you keep it and leave it in your boot configuration file, you should still be able to go back to it. If you do delete it, an emergency boot disk will be your only hope.

To create a boot disk, you use the *mkbootdisk* command as shown here:

```
[root@fanny root]# uname -r
```

```
2.4.20-6
```

```
[root@fanny root]# mkbootdisk \
```

```
>--device /dev/fd0 2.4.20-6
```

Insert a disk in /dev/fd0. Any information on the disk will be lost.

Press <Enter> to continue or ^C to abort:

(The last argument to *mkbootdisk* is the kernel version. If you can't remember this, use the command *uname -r* first to refresh your memory.)

5.4.2 Installing

Since we are working with RPM packages, installation is a breeze. Just change to the directory where you have the files and, as root, run *rpm*.

```
[root@fanny root]# rpm -vih openmosix-kernel-2.4.24-openmosix1.i686.rpm
```

```
Preparing...                               ##### [100%]
```

```
 1:openmosix-kernel                         ##### [100%]
```

```
[root@fanny root]# rpm -vih openmosix-tools-0.3.5-1.i386.rpm
```

```
Preparing...                               ##### [100%]
```

```
 1:openmosix-tools                          ##### [100%]
```

Edit /etc/openmosix.map if you don't want to use the autodiscovery daemon.

That's it! The kernel has been installed for you in the */boot* directory.

This example uses the *2.4.24-om1* release. *2.4.24-om2* should be available by the time you read this. This newer release corrects several bugs and should be used.

You should also take care to use an openMosix tool set that is in sync with the kernel you are using, i.e., one that has been compiled with the same kernel header files. If you are compiling both, this shouldn't be a problem. Otherwise, you should consult the release notes for the tools.

5.4.3 Configuration Changes

While the installation will take care of the stuff that can be automated, there are a few changes you'll have to do manually to get openMosix running. These are very straightforward.

As currently installed, the next time you reboot your systems, your loader will give you the option of starting openMosix but it won't be your default kernel. To boot to the new openMosix kernel, you'll just need to select it from the menu. However, unless you set openMosix as the default kernel, you'll need to manually select it every time you reboot a system.

If you want openMosix as the default kernel, you'll need to reconfigure your boot loader. For example, if you are using *grub*, then you'll need to edit */etc/grub.conf* to select the openMosix kernel. The installation will have added openMosix to this file, but will not have set it as the default kernel. You should see two sets of entries in this file. (You'll see more than two if you already have other additional kernels). Change the variable `default` to select which kernel

you want as the default. The variable is indexed from 0. If openMosix is the first entry in the file, change the line to setting `default` so that it reads `default=0`.

If you are using LILO, the procedure is pretty much the same except that you will need to manually create the entry in the configuration file and rerun the loader. Edit the file `/etc/lilo.conf`. You can use a current entry as a template. Just copy the entry, edit it to use the new kernel, and give it a new label. Change `default` so that it matches your new label, e.g., `default=openMosix`. Save the file and run the command `/sbin/lilo -v`.

Another issue is whether your firewall will block openMosix traffic. The *openMosix FAQ* reports that openMosix uses UDP ports in the 5000-5700 range, UDP port 5428, and TCP ports 723 and 4660. (You can easily confirm this by monitoring network traffic, if in doubt.) You will also need to allow any other related traffic such as NFS or SSH traffic. Address this before you proceed with the configuration of openMosix.

In general, security has not been a driving issue with the development of openMosix. Consequently, it is probably best to use openMosix in a restrictive environment. You should either locate your firewall between your openMosix cluster and all external networks, or you should completely eliminate the external connection.

openMosix needs to know about the other machines in your cluster. You can either use the autodiscovery tool *omdiscd* to dynamically create a map, or you can create a static map by editing the file `/etc/openmosix.map` (or `/etc/mosix.map` or `/etc/hpc.map` on earlier versions of openMosix). *omdiscd* can be run as a foreground command or as a daemon in the background. Routing must be correctly configured for *omdiscd* to run correctly. For small, static clusters, it is probably easier to edit `/etc/openmosix.map` once and be done with it.

For a simple cluster, this file can be very short. Its simplest form has one entry for each machine. In this format, each entry consists of three fields—a unique device node number (starting at 1) for each machine, the machine's IP address, and a 1 indicating that it is a single machine. It is also possible to have a single entry for a range of machines that have contiguous IP addresses. In that case, the first two fields are the same—the node number for the first machine and the IP address of the first machine. The third field is the number of machines in the range. The address can be an IP number or a device name from your `/etc/hosts` file. For example, consider the following entry:

```
1      fanny.wofford.int      5
```

This says that *fanny.wofford.int* is the first of five nodes in a cluster. Since fanny's IP address is 10.0.32.144, the cluster consists of the following five machines: 10.0.32.144, 10.0.32.145, 10.0.32.146, 10.0.32.147, and 10.0.32.148. Their node numbers are 1 through 5. You could use separate entries for each machine. For example,

```
1      fanny.wofford.int      1
2      george.wofford.int     1
3      hector.wofford.int     1
4      ida.wofford.int        1
5      james.wofford.int      1
```

or, equivalently

```
1      10.0.32.144           1
2      10.0.32.145           1
```

```

3      10.0.32.146      1
4      10.0.32.147      1
5      10.0.32.148      1

```

Again, you can use the first of these two formats only if you have entries for each machine in */etc/hosts*. If you have multiple blocks of noncontiguous machines, you will need an entry for each contiguous block. If you use host names, be sure you have an entry in your host table for your node that has its actual IP address, not just the local host address. That is, you need lines that look like

```

127.0.0.1      localhost
172.16.1.1     amy

```

not

```

127.0.0.1      localhost  amy

```

You can list the map that openMosix is using with the *showmap* command. (This is nice to know if you are using autodiscovery.)

```
[root@fanny etc]# showmap
```

```
My Node-Id: 0x0001
```

```

Base Node-Id Address      Count
-----
0x0001      10.0.32.144      1
0x0002      10.0.32.145      1
0x0003      10.0.32.146      1
0x0004      10.0.32.147      1
0x0005      10.0.32.148      1

```

Keep in mind that the format depends on the map file format. If you use the range format for your map file, you will see something like this instead:

```
[root@fanny etc]# showmap
```

```
My Node-Id: 0x0001
```

```

Base Node-Id Address      Count
-----
0x0001      10.0.32.144      5

```

While the difference is insignificant, it can be confusing if you aren't expecting it.

There is also a configuration file `/etc/openmosix/openmosix.config`. If you are using autodiscovery, you can edit this to start the discovery daemon whenever openMosix is started. This file is heavily commented, so it should be clear what you might need to change, if anything. It can be ignored for most small clusters using a map file.

Of course, you will need to duplicate this configuration on each node on your cluster. You'll also need to reboot each machine so that the openMosix kernel is loaded. As root, you can turn openMosix on or off as needed. When you install the user tools package, a script called `openmosix` is copied to `/etc/init.d` so that openMosix will be started automatically. (If you are manually compiling the tools, you'll need to copy this script over.) The script takes the arguments `start`, `stop`, `status`, `restart`, and `reload`, as you might have guessed. For example,

```
[root@james root]# /etc/init.d/openmosix status

This is OpenMosix node #5

Network protocol: 2 (AF_INET)

OpenMosix range      1-5      begins at fanny.wofford.int

Total configured: 5
```

Use this script to control openMosix as needed. You can also use the `setpe` command, briefly described later in this chapter, to control openMosix.

Congratulations, you are up and running.

5.5 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.

5.5.1 User Tools

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*.)

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.)

5.5.1.2 migrate

The tool *migrate* explicitly moves a process from one node to another. Since there are circumstances under which some processes can't migrate, the system may be forced to ignore this command. You'll need the PID and the node number of the destination machine. Here is an example:

```
[sloanjd@fanny sloanjd]$ migrate 19769 5
```

This command will move process 19769 to node number 5. (You can use `home` in place of the node number to send a process back to the CPU where it was started.) It might be tempting to think you are reducing the load on node number 4, the node where the process was running, but in a balanced system with no other action, another process will likely migrate to node 4.

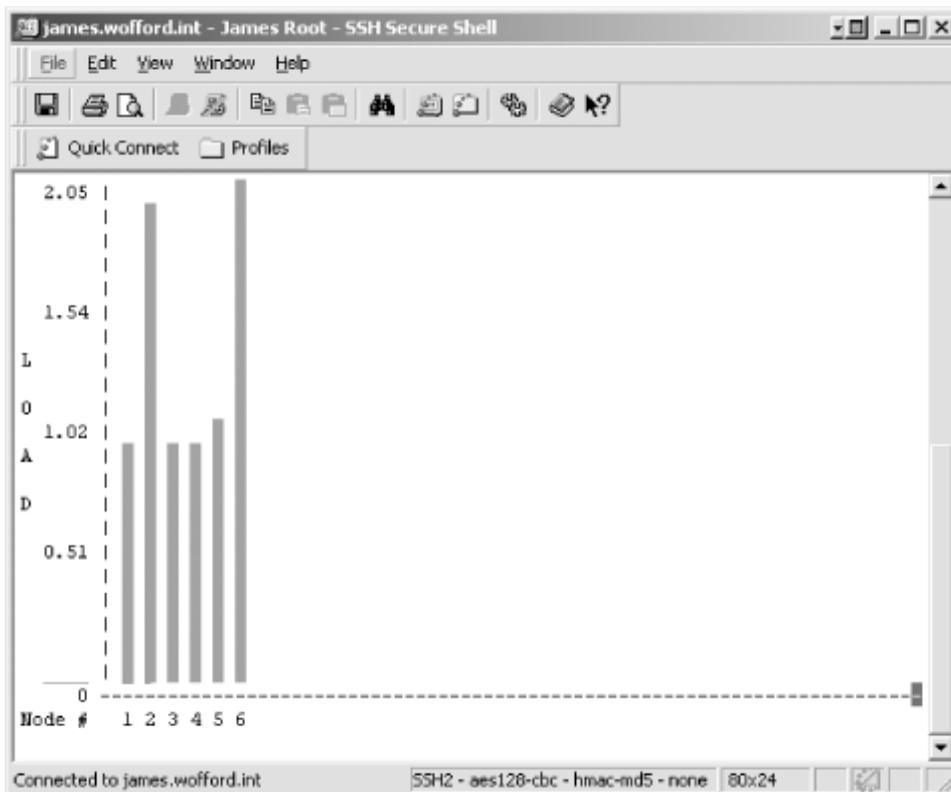
5.5.1.3 mosctl

With *mosctl*, you have greater control over how processes are run on individual machines. For example, you can block the arrival of guest processes to lighten the load on a machine. You can use *mosctl* with the *setspeed* option to override a node's idea of its own speed. This can be used to attract or discourage process migration to the machine. *mosctl* can also be used to display utilization or tune openMosix performance parameters. There are too many arguments to go into here, but they are described in the manpage.

5.5.1.4 mosmon

While *mps* won't tell you if a process has migrated to your machine, you can get a good idea of what is going across the cluster with the *mosmon* utility. *mosmon* is an *ncurses*-based utility that will display a simple bar graph showing the loads on the nodes in your cluster. This can give you a pretty good idea of what is going on. [Figure 5-1](#) shows *mosmon* in action.

Figure 5-1. mosmon



In this example, eight identical processes are running on a six-node cluster. Obviously, the second and sixth nodes have two processes each while the remaining four machines are each running a single process. Of course, other processes could be mixed into this, affecting an individual machine's load. You can change the view to display memory, speed, and utilization as well as change the layout of the graph. Press **h** while the program is running to display the various options. Press **q** to quit the program.

Incidentally, *mosmon* goes by several different names, including *mon* and, less commonly, *mmon*. The original name was *mon*, and it is often referred to by that name in openMosix documentation. The shift to *mosmon* was made to eliminate a naming conflict with the network-monitoring tool *mon*. The local name is actually set by a compile-time variable.

5.5.1.5 mosrun

The *mosrun* command can also be used to advise the system to run a specific program on a specified node. You'll need the program name and the destination node number (or use **-h** for the home node). Actually, *mosrun* is one of a family of commands used to control node allocation preferences. These are listed and described on the manpage for *mosrun*.

5.5.1.6 setpe

The *setpe* command can be used to manually configure a node. (In practice, *setpe* is usually called from the script `/etc/init.d/openmosix` rather than used directly.) As root, you can use *setpe* to start or stop openMosix. For example, you could start openMosix with a specific configuration file with a command like

```
[root@ida sloanjd]# /sbin/setpe -w -f /etc/openmosix.map
```

setpe takes several options including **-r** to read the configuration file, **-c** to check the map's

consistency, and `-off` to shut down openMosix. Consult the manpage for more information.

5.5.2 openMosixView

openMosixView extends the basic functionality of the user tools while providing a spiffy X-based GUI. However, the basic user tools must be installed for openMosixView to work. openMosixView is actually seven applications that can be invoked from the main administration application.

If you want to install openMosixView, which is strongly recommended, download the package from <http://www.openmosixview.com>. Look over the documentation for any dependencies that might apply. Depending on what you have already installed on your system, you may need to install additional packages. For example, GLUT is one of more than two dozen dependences. Fortunately (or annoyingly), `rpm` will point out to you what needs to be added.

Then, as root, install the appropriate packages.

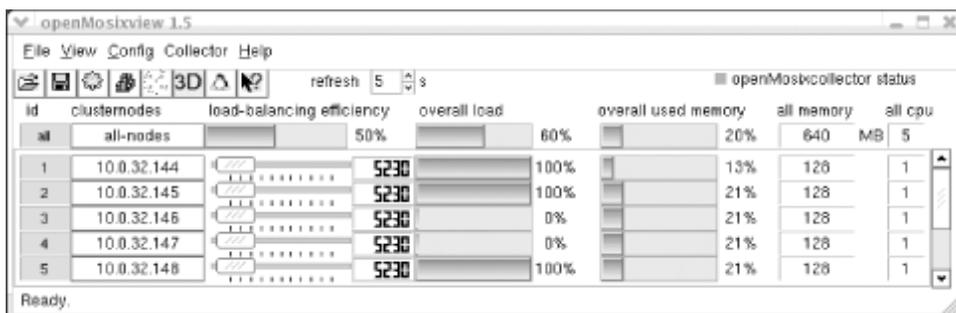
```
[root@fanny root]# rpm -vih glut-3.7-12.i386.rpm
warning: glut-3.7-12.i386.rpm: V3 DSA signature: NOKEY, key ID db42a60e
Preparing...                               ##### [100%]
 1:glut                                     ##### [100%]
[root@fanny root]# rpm -vih openmosixview-1.5-redhat90.i386.rpm
Preparing...                               ##### [100%]
 1:openmosixview                           ##### [100%]
```

As with the kernel, you'll want to repeat this on every node. This installation will install documentation in `/usr/local`.

Once installed, you are basically ready to run. However, by default, openMosixView uses RSH. It is strongly recommended that you change this to SSH. Make sure you have SSH set up on your system. (See [Chapter 4](#) for more information on SSH.) Then, from the main application, select the `Config` menu.

The main applications window is shown in [Figure 5-2](#). You get this by running the command `openmosixview` in an X window environment.

Figure 5-2. openMosixView



This view displays information for each of the five nodes in this cluster. The first column displays the node's status by node number. The background color is green if the node is available or red if it is unavailable. The second column, buttons with IP numbers, allows you to configure individual systems. If you click on one of these buttons, a pop-up window will appear for that node, as shown in [Figure 5-3](#). You'll notice that the configuration options are very similar to those provided by the *mosctl* command.

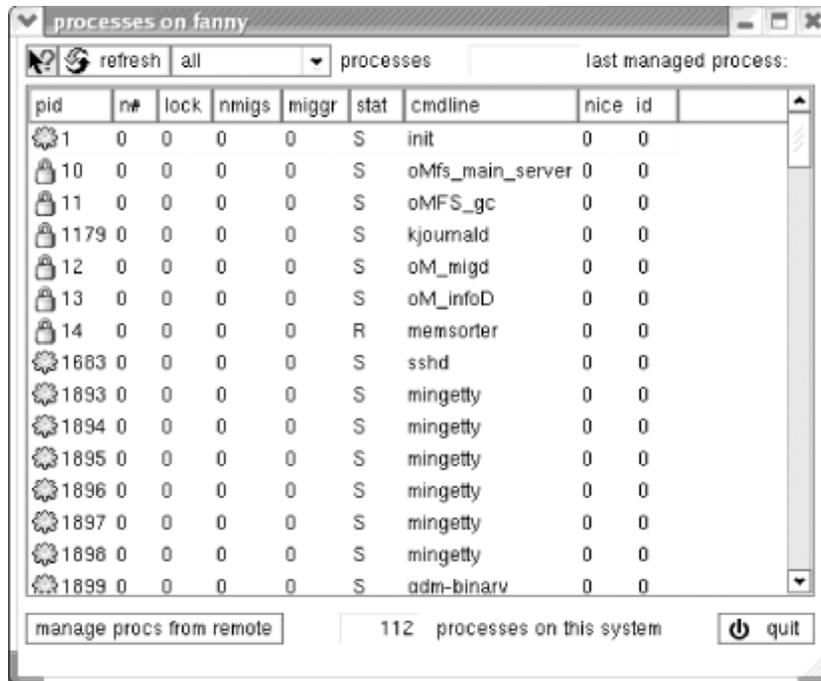
Figure 5-3. openMosix configuration window



As you can see from the figure, you can control process migration, etc., with this window. The third column in [Figure 5-2](#), the sliders, controls the node efficiencies used by openMosix when load balancing. By changing these, you alter openMosix's idea of the relative efficiencies of the nodes in the cluster. This in turn influences how jobs migrate. Note that the slider settings do not change the efficiency of the node, just openMosix's perception of the node's capabilities. The remaining columns provide general information about the nodes. These should be self-explanatory.

The buttons along the top provide access to additional applications. For example, the third button, which looks like a gear, launches the process viewer *openMosixprocs*. This is shown in [Figure 5-4](#).

Figure 5-4. openMosixprocs



openMosixprocs allows you to view and manage individual processes started on the node from which *openMosixprocs* is run. (Since it won't show you processes migrated from other systems, you'll need *openMosixprocs* on each node.) You can select a user in the first entry field at the top of the window and click on **refresh** to focus in on a single user's processes. By double-clicking on an individual process, you can call up the *openMosixprocs-Migrator*, which will provide additional statistics and allow some control of a process.

openMosixView provides a number of additional tools that aren't described here. These include a 3D process viewer (*3dmosmar*), a data collection daemon (*openMosixcollector*), an analyzer (*openMosixanalyzer*), an application for viewing process history (*openMosixHistory*), and a migration monitor and controller (*openMosixmigmar*) that supports drag-and-drop control on process migration.

5.5.3 Testing openMosix

It is unlikely that you will have any serious problems setting up *openMosix*. But you may want to confirm that it is working. You could just start a few processes and time them with *openMosix* turned on and off. Here is the simple C program that can be used to generate some activity.

```
#include <stdio.h>

int foo(int,int);

int main( void )
{
    int i,j;

    for (i=1; i<100000; i++)
```

```

        for (j=1; j<100000; j++)
            foo(i,j);

    return 0;
}

int foo(int x, int y)
{
    return(x+y);
}

```

This program does nothing useful, but it will take several minutes to complete on most machines. (You can adjust the loop count if it doesn't run long enough to suit you.) By compiling this (without optimizations) and then starting several copies running in the background, you'll have a number of processes you can watch.

While timing will confirm that you are actually getting a speedup, you'll get a better idea of what is going on if you run *mosmon*. With *mosmon*, you can watch process migration and load balancing as it happens.

If you are running a firewall on your machines, the most likely problem you will have is getting connection privileges correct. You may want to start by disconnecting your cluster from the Internet and disabling the firewall. This will allow you to confirm that openMosix is correctly installed and that the firewall is the problem. You can use the command *netstat -a* to identify which connections you are using. This should give you some guidance in reconfiguring your firewall.

Finally, an openMosix stress test is available for the truly adventurous. It can be downloaded from <http://www.openmosixview.com/omtest/>. This web page also describes the test (actually a test suite) and has a link to a sample report. You can download sources or an RPM. You'll need to install *expect* before installing the stress test. To run the test, you should first change to the */usr/local/omtest* directory and then run the script *./openmosix_stress_test.sh*. A report is saved in the */tmp* directory.

The test takes a while to run and produces a very long report. For example, it took over an hour and a half on an otherwise idle five-node cluster of Pentium II's and produced an 18,224-line report. While most users will find this a bit of overkill for their needs, it is nice to know it is available. Interpretation of the results is beyond the scope of this book.

5.6 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.)

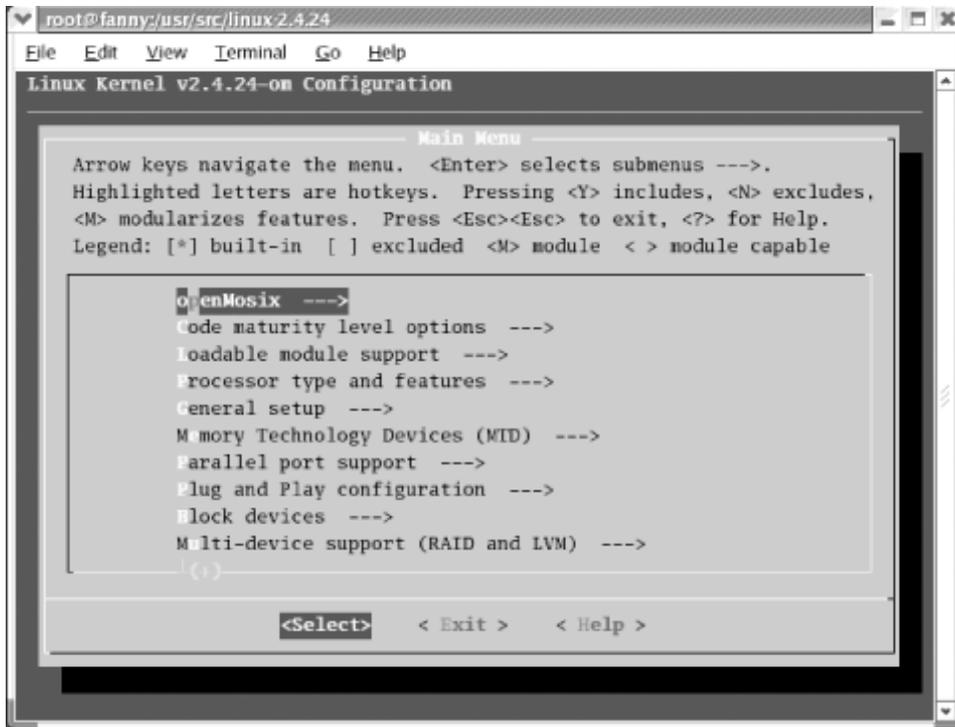
You are going to need both a set of patches and a clean set of kernel source files. Accepted wisdom says that you shouldn't use the source files that come with any specific Linux releases because, as a result of customizations, the patches will not apply properly. As noted earlier in this chapter, you 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`. The next step is to unpack the sources and apply the patches.

Using *gunzip* or *bzip2* and then *tar*, unpack the files in the appropriate directory. Where you put things is largely up to you, but it is a good idea to try to be consistent with the default layout of your system. Move the patch files into the root directory of your source tree. Once you have all the files in place, you can use the *patch* command to patch the kernel sources.

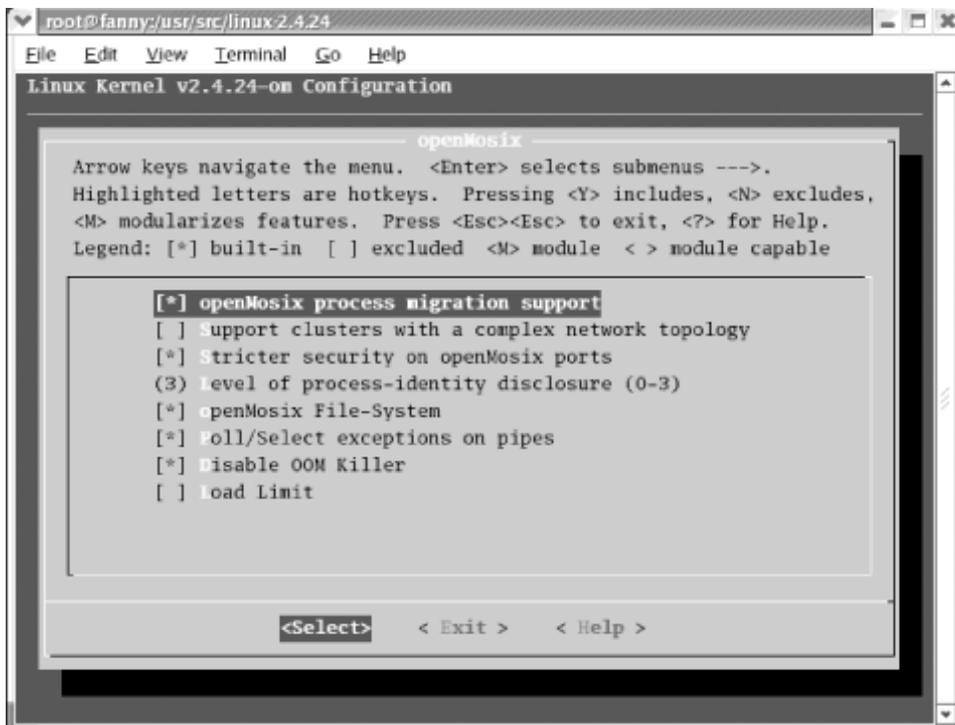
The next step is to create the appropriate configuration file. In theory, there are four ways you can do this. You could directly edit the default configuration file, typically `/usr/src/linux/.config`, or you can run one of the commands *make config*, *make menuconfig*, or *make xconfig*. In practice, you should limit yourself to the last two choices. Direct editing of the configuration file for anything other than minor changes is for fools, experts, or foolish experts. And while *config* is the most universal approach, it is also the most unforgiving and should be used only as a last resort. It streams the configuration decisions past you and there is no going back once you have made a decision. The remaining choices are *menuconfig*, which requires the ncurses library, and *xconfig*, which requires X windows and TCL/TK libraries. Both work nicely. [Figure 5-5](#) shows the basic layout with *menuconfig*.

Figure 5-5. Main menuconfig menu



Configuration parameters are arranged in groups by functionality. The first group is for openMosix. You can easily move through this menu and select the appropriate actions. You will be given a submenu for each group. [Figure 5-6](#) shows the openMosix submenu.

Figure 5-6. openMosix system submenu



xconfig is very similar but has a fancy GUI.

Because there are so many decisions, this is the part of the process where you are most apt to make a mistake. This isn't meant to discourage you, but don't be surprised if you have to go through this process several times. For the most part, the defaults are reasonable. Be sure you select the right processor type and all appropriate file systems. (Look at `/etc/fstab`, run the `mount` command, or examine `/proc/filesystems` to get an idea of what file systems you are currently using.) If you downloaded the sources from the openMosix web page, you have several sample configuration files. You can copy one of these over and use it as your starting point. This will give you some reasonable defaults. You can also get a description of various options (including openMosix options!) by looking in the `Documentation/Configure.help` file in your source tree. As a general rule of thumb, if you don't need something, don't include it.

Once you have the configuration file, you are ready to build the image. You'll use the commands `make dep`, `make clean`, `make bzImage`, `make modules`, and `make modules_install`. (You'll need modules enabled, since `openMosix` uses them.) If all goes well, you'll be left with a file `bzImage` in the directory `arch/i386/boot/` under your source tree.

The next to last step is to install the kernel, i.e., arrange for the system to boot from this new kernel. You'll probably want to move it to the `/boot` directory and rename it. Since you are likely to make several kernels once you get started, be sure to use a meaningful name. You may need to create a ram-disk. You also need to configure your boot loader to find the file as described earlier in this chapter. When copying over the new kernel, don't delete the original kernel!

Now you are ready to reboot and test your new kernel. Pay close attention to the system messages when you reboot. This will be your first indication of any configuration errors you may have made. You'll need to go back to the configuration step to address these.

Of course, this is just the kernel you've installed. You'll still need to go back and install the user tools and configure openMosix for your system. But even if you are compiling the kernel, there is no reason you can't use the package to install the user tools.

Here is an example using Red Hat 9. Although Red Hat 9 comes with the 2.4.20 version of the kernel, this example uses a later version of the kernel, `openmosix-kernel-2.4.24-openmosix1.src.rpm`. The first step is installing this package.

```
[root@fanny root]# rpm -vih openmosix-kernel-2.4.24-openmosix1.src.rpm

 1:openmosix-kernel      ##### [100%]

[root@fanny root]# cd /usr/src/redhat/SOURCES

[root@fanny SOURCES]# ls

kernel-2.4.20-athlon.config      kernel-2.4.24-athlon-smp.config
kernel-2.4.20-athlon-smp.config  kernel-2.4.24-i386.config
kernel-2.4.20-i386.config        kernel-2.4.24-i686.config
kernel-2.4.20-i686.config        kernel-2.4.24-i686-smp.config
kernel-2.4.20-i686-smp.config    linux-2.4.24.tar.bz2
kernel-2.4.24-athlon.config      openMosix-2.4.24-1.bz2
```

As you can see, the package includes the source files, patches, and sample configuration files.

Next, unpack the files. (With some versions, you may need to use `gunzip` instead of `bunzip2`.)

```
[root@fanny SOURCES]# bunzip2 linux-2.4.24.tar.bz2
```

```
[root@fanny SOURCES]# bunzip2 openMosix-2.4.24-1.bz2
```

```
[root@fanny SOURCES]# mv linux-2.4.24.tar /usr/src
```

```
[root@fanny SOURCES]# cd /usr/src
```

```
[root@fanny src]# tar -xvf linux-2.4.24.tar
```

```
...
```

The last command creates the directory *linux-2.4.24* under */usr/src*. If you are working with different versions of the kernel, you probably want to give this directory a more meaningful name.

The next step is to copy over the patch file and, if you desire, one of the sample configuration files. Then, you can apply the patches.

```
[root@fanny src]# cd /usr/src/redhat/SOURCES
```

```
[root@fanny SOURCES]# cp openMosix-2.4.24-1 /usr/src/linux-2.4.24/
```

```
[root@fanny SOURCES]# cp kernel-2.4.24-i686.config \
```

```
>/usr/src/linux-2.4.24/.config
```

```
[root@fanny SOURCES]# cd /usr/src/linux-2.4.24
```

```
[root@fanny linux-2.4.24]# cat openMosix-2.4.24-1 | patch -Np1
```

```
...
```

You should see a list of the patched files stream by as the last command runs.

Next, you'll need to create or edit a configuration file. This example uses the supplied configuration file that was copied over as a starting point.

```
[root@fanny linux-2.4.24]# make menuconfig
```

Make whatever changes you need and then save your new configuration.

Once configured, it is time to make the kernel.

```
[root@fanny linux-2.4.24]# make dep
```

```
...
```

```
[root@fanny linux-2.4.24]# make clean
```

```
...
```

```
[root@fanny linux-2.4.24]# make bzImage
```

```
...
```

```
[root@fanny linux-2.4.24]# make modules
```

```
...
```

```
[root@fanny linux-2.4.24]# make modules_install
```

```
...
```

These commands can take a while and produce a lot of output, which has been omitted here.

The worst is over now. You need to copy your kernel to */boot*, create a ram-disk, and configure your boot loader.

```
[root@fanny linux-2.4.24]# cd /usr/src/linux-2.4.24/arch/i386/boot/
```

```
[root@fanny boot]# cp bzImage /boot/vmlinuz-8jul04
```

If you haven't changed kernels, you may be able to use the existing ram-disk. Otherwise, use the *mkinitrd* script to create a new one.

```
[root@fanny boot]# cd /boot
```

```
[root@fanny boot]# mkinitrd /boot/initrd-2.4.24.img 2.4.24-om
```

The first argument is the name for the ram-disk and the second argument is the appropriate module directory under */lib/modules*. See the manpage for details.

The last step is to change the boot loader. This system uses *grub*, so the file */etc/grub.conf* needs to be edited. You might add something like the following:

```
title My New openMosix Kernel

    root (hd0,0)

    kernel /vmlinuz-8jul04 ro root=LABEL=/

    initrd /initrd-2.4.24.img
```

When the system reboots, the boot menu now has *My New openMosix Kernel* as an entry. Select that entry to boot to the new kernel.

While these steps should be adequate for most readers, it is important to note that, depending on your hardware, etc., additional steps may be required. Fortunately, there has been a lot written on the general process of recompiling Linux kernels. See the [Appendix A](#) for pointers to more information.

5.7 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.

6.1 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.)

Because of the difficulty in bringing together a wide variety of software and because the individual software packages are constantly being updated, some of the software included in OSCAR has not always been the most current versions available. In practice, this is not a problem. The software OSCAR includes is stable and should meet most of your needs.

While OSCAR was originally created using Red Hat Linux, a goal of the project is to move beyond support for a single distribution and Mandrake Linux is now also supported. The OSCAR project has shifted to SIS in order to eventually support most RPM-based versions of Linux. But don't expect support for the latest Linux versions to be immediately available as the new versions are released.

6.2 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](#)).

Environmental Switcher

This is based on Modules, a Perl script that allows the user to make changes to the environment of future shells. For example, Switcher allows a user to change between MPICH and LAM/MPI.

oda

The OSCAR database application provides a central database for OSCAR.

perl-qt

This is the Perl object-oriented interface to the Qt GUI toolkit.

SIS

The System Installation Suite is used to install the operating systems on the clients (described in [Chapter 8](#)).

OSCAR includes a number of packages and scripts that are used to build your cluster. The installation wizard will give you the option of deciding which to include:

disable-services

This script disables unneeded services on the clients, such as *kudzu*, *slocate*, and mail services such as *sendmail*.

networking

This script configures the cluster server as a caching nameserver for the clients.

ntpconfig

This script configures NTP. OSCAR uses NTP to synchronize clocks within the cluster.

kernel_picker

This is used to change the kernel used in your SIS image before building the cluster nodes.

loghost

This configures *syslog* settings, e.g., it configures nodes to forward *syslog* messages to the head node.

OSCAR provides additional system tools, either as part of the OSCAR distribution or through *opd*, used to manage your cluster:

Autoupdate

This is a Perl script used to update clients and the server (similar to *up2date* or *autorpm*).

clumon (by *opd*)

Clumon is a web-based performance-monitoring system from NCSA.

Ganglia (by opa)

Ganglia is a real-time monitoring system and execution environment (described in [Chapter 10](#)).

MAUI

This job scheduler is used with *openPBS*.

Myrnet drivers (by opa)

If you have Myrnet hardware, you need to load drivers for it.

openPBS

The portable batch system is a workload management system (described in [Chapter 11](#)).

Pfilter

This package is used to generate sets of rules used for packet filtering.

PVFS (by opa)

Parallel Virtual File System is a high-performance, scalable, parallel virtual file system (described in [Chapter 12](#)).

OPIUM

This is the OSCAR password installer and user management toolset.

thin client (by opa)

This package provides support for diskless OSCAR nodes.

Torque (by opa)

The Tera-scale Open-source Resource and QUEue manager resource manager is based on *openPBS*.

VMI (by opa)

The Virtual Machine Interface provides a middleware communications layer for SAN over grids.

Of course, any high-performance cluster would be incomplete without programming tools. The OSCAR distribution includes four packages, while two more (as noted) are available through *opd*:

HDF5

This is a hierarchical data format library for maintaining scientific data.

LAM/MPI

This is one implementation of the message passing interface (MPI) libraries (described in [Chapter 9](#)).

MPICH

This is another implementation of the message passing interface (MPI) libraries (also described in [Chapter 9](#)).

MPICH-GM (by *opd*)

This package provided MPICH with support for low-level message passing for Myrnet networks.

MPICH-VMI (by *opd*)

This version of MPICH uses VMI.

PVM

This package provides the parallel virtual machine system, another message passing library.

If you install the four included packages, the default, they should cover all your programming needs.

Additionally, OSCAR will install and configure (or reconfigure) a number of services and packages supplied as part of your Linux release.^[1] These potentially include *Apache*, *DHCP*, *NFS*, *mySQL*, *openSSL*, *openSSH*, *rrdtool*, *pcp*, *php*, *python*, *rsync*, *ftp*, etc. Exactly which of these is actually installed or configured will depend on what other software you elect to install. In the unlikely event that you are unhappy with the way OSCAR sets up any of these, you'll need to go back and reconfigure them after the installation is completed.

^[1] Sometimes OSCAR needs to make slight changes to packages. By convention, the replacement packages that OSCAR uses have **oscar** as part of their names, e.g., *lam-oscar-7.0-2.1586.rpm*.

6.3 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.

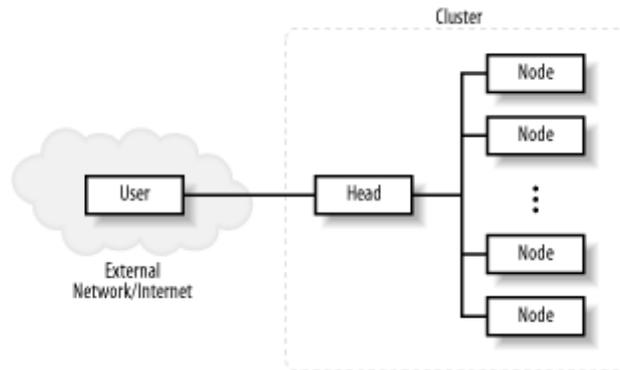
Because OSCAR is a complex set of software that includes a large number of programs and services, it can be very unforgiving if you make mistakes when setting it up. For some errors, you may be able to restart the installation process. For others, you will be better served by starting again from scratch. 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.

6.3.1 Prerequisites

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 `/` and `/var` directories while 4 GB for each is recommended. Since the head is also the home for your users' files, you'll need to keep this in mind as well. It is a good idea to put the `/`, `/var`, and `/home` directories on separate disk partitions. This will simplify reinstalls and provide a more robust server.

Figure 6-1. OSCAR architecture



As you can see from the figure, the server or head is dual homed; that is, it has two network interfaces. The interface attached to the external network is called the *public* interface. The *private* interface attaches to the cluster's network. While you don't have to use this configuration, be aware that OSCAR will set up a DHCP server on the private interface. If you put everything on a public network with an existing DHCP server, you may have a war between the two DHCP servers. The remainder of this chapter assumes you'll be using a configuration like the one shown in [Figure 6-1](#).

It is strongly recommended that you begin with a clean install of your operating system and that you customize your OSCAR installation as little as possible the first time you install it. OSCAR is a complex collection of software. With a vanilla installation, all should work well. This isn't to say you can't do customizations, just do so with discretion. Don't be surprised if a custom install takes a few tries to get right.

The installation documentation lists a few supported versions of Linux. It is strongly recommend that you stick to the list. For Red Hat, a workstation install that includes the *Software Development* group and an X Windows environment should work nicely for the server. (You may also want to add some network utilities such as *VNC-server* and *Etherreal* to make life easier, and you may want to remove *openOffice* to discourage that kind of activity on the cluster. That's your call; it won't affect your OSCAR installation either way.) You should also do manual disk partitioning to ensure that you meet the space requirements and to control the disk layout. (It is possible to work around some allocation problems using links, but this is a nuisance best avoided.) Don't install any updates to your system at this point. Doing so may break the OSCAR installation, and you can always add these after you install OSCAR.

6.3.2 Network Configuration

Since you have two interfaces, you need to make sure that your network configuration is correct. The configuration of the public interface, of course, will be determined by the configuration of the external network. For example, an external DHCP server might be used to configure the public interface when booting the server. For the cluster's network, use a private address space distinct from the external address space. [Table 6-1](#) lists reserved address spaces that you might use per RFC 1918.

Table 6-1. Private IP address spaces

Address Spaces
10.0.0.0 to 10.255.255.255
172.16.0.0 to 172.31.255.255
192.168.0.0 to 192.168.255.255

By way of example, assume you have fewer than 255 computers and your organization's internal network is already using the first address range (10.x.x.x). You might select one of the class C ranges from the third address range, e.g., 192.168.1.0 through 192.168.1.255. The usual IP configuration constraints apply, e.g., don't assign the broadcast address to a machine. In this example, you would want to avoid 192.168.1.0 (and, possibly, 192.168.1.255). Once you have selected the address space, you can configure the private interface using the tool of your choice, e.g., *neat*, *ifconfig*, or *netcfg*. You will need to set the IP address, subnet mask, and default gateway. And don't forget to configure the interface to be active on startup. In this example, you might use an IP address of 192.168.1.1 with a mask of 255.255.255.0 for the private interface.^[2] The public interface will be the gateway for the private network. This will leave 192.168.1.2 through 192.168.1.254 as addresses for your compute nodes when you set up DHCP. Of course, if you plan ahead, you can also configure the interface during the Linux installation.

^[2] While this is the simplest choice, a better choice is to use 192.168.1.254 for the server and starting at 192.168.1.1 for the clients. The advantage is that the low-order portion of the IP addresses will match the node numbers, at least for your first 253 machines.

Once you have the interfaces configured, reboot the server and verify that everything works. You can use *ifconfig -a* to quickly confirm that both interfaces are up. If it is possible to put a live machine on the internal network, you can confirm that routing works correctly by pinging the machine. Do as much checking as you can at this point. Once the cluster is installed, testing can be more difficult. You don't want to waste a lot of time trying to figure out what went wrong with the OSCAR installation when the network was broken before you began.

Another pre-installation consideration is the security settings for the server you are building. If you have the security set too tightly on the server, it will interfere with the client installation. If you have customized the security settings on a system, you need to pay particular attention. For example, if you have already installed SSH, be sure that you permit root logins to your server (or plan to spend a lot of time at the server). If you can isolate the cluster from the external network, you can just turn off the firewall.

Even if the installation goes well, you still may encounter problems later. For example, with Red Hat 9, the default firewall settings may cause problems for services like Ganglia. Since OSCAR includes *pfilter*, it is usually OK to just turn off Red Hat's firewall. However, this is a call you will have to make based on your local security policies.

You should also ensure that the head node's host name is correctly set. Make sure that the `hostname` command returns something other than `localhost` and that the returned name resolves to the internal interface. For example,

```
[root@amy root]# /bin/hostname
amy
[root@amy root]# ping -c1 amy
PING amy (172.16.1.254) 56(84) bytes of data.
64 bytes from amy (172.16.1.254): icmp_seq=1 ttl=64 time=0.166 ms
```

```
--- amy ping statistics ---
```

```
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

```
rtt min/avg/max/mdev = 0.166/0.166/0.166/0.000 ms
```

Notice that *hostname* returns *amy* and that when *amy* is pinged, the name resolves to the address *172.16.1.254*.

It is also a good idea to make sure you have enough disk space before going on. You can use the *df -h* command. This is also a good point to do other basic configuration tasks, such as setting up printers, setting the message of the day, etc.

6.3.3 Loading Software on Your Server

The next step is to get the software you will need onto the server. This consists of the OSCAR distribution and the Linux packages you need to build the image for the client machines. For the Linux packages, first create the directory */tftpboot/rpm* and then copy over the packages. It will be a lot simpler if you just copy everything over rather than try to figure out exactly what is needed. For Red Hat 9, mount each of the three distribution disks and copy over the all the RPM files from *../cdrom/RedHat/RPMS*. The sequence looks like this:

```
[root@amy /root]# mkdir -p /tftpboot/rpm

[root@amy /root]# mount /mnt/cdrom

[root@amy /root]# cd /mnt/cdrom/RedHat/RPMS

[root@amy RPMS]# cp *.rpm /tftpboot/rpm/

[root@amy RPMS]# cd /

[root@amy /]# eject cdrom
```

You repeat the last five steps two more times, once for each of the remaining CD-ROMs. If your system automounts CD-ROMs, you'll skip the manual mounts. You'll copy more than 1,400 packages, so this can take a while with slower systems. (OSCAR will subsequently add additional packages to this directory.)

If you are tight on disk space, you can install the packages on a different partition and link to them. For example, if you've installed the packages in */var/tftpboot/rpm*, you could do the following:

```
[root@amy root]# mkdir /tftpboot/

[root@amy root]# ln -s /var/tftpboot/rpm /tftpboot/rpm
```

Note that the directory, not the individual packages, is linked.

You can download the OSCAR package from <http://oscar.sourceforge.net>. You'll have the option of downloading OSCAR with or without the sources (SRPMs) for most of the packages in OSCAR. Since it is unlikely you'll need the sources and since you can download them separately later should you need them, it is OK to skip them and go with the standard download. We'll download to the */root* directory, a safe place to install from.

Next, you will unpack the code

```
[root@amy root]# gunzip oscar-3.0.tar.gz
[root@amy root]# tar -xvf oscar-3.0.tar
...
```

This creates a directory, `/root/oscar-3.0`, which you should `cd` to for the next phase of the installation process. You may also want to browse the subdirectories that are created.

6.3.4 A Basic OSCAR Installation

Before the installation wizard can be run the first time, it must be configured and installed. Log in as root or use `su -` to become root. Change to the installation directory and run `configure` and `make install`.

```
[root@amy root]# cd /root/oscar-3.0
[root@amy oscar-3.0]# ./configure
...
[root@amy oscar-3.0]# make install
...
```

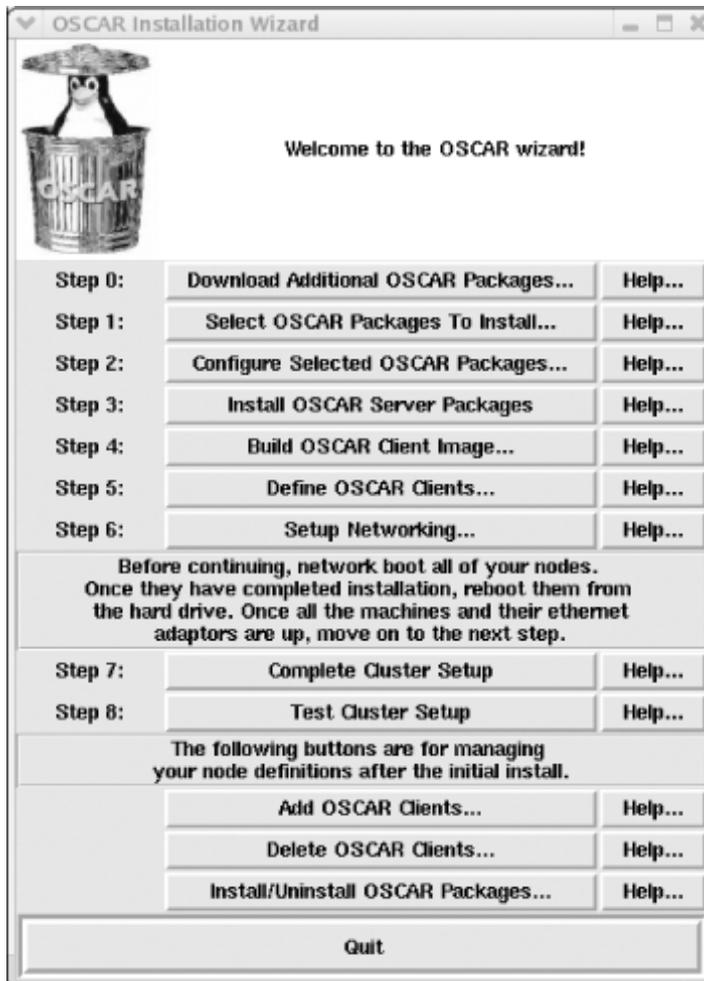
Now you are ready to run the wizard.

At this point, it is generally a good idea to start another shell so the environment variables are sourced from `/etc/profile.d`. To start the installation, change to the installation directory and run the `install_cluster` script from a terminal window under X. The `install_cluster` script expects the private interface as an argument. Be sure to adjust this parameter as needed. Here is an example of starting the script:

```
[root@amy oscar-3.0]# cd $OSCAR_HOME && pwd
/opt/oscar
[root@amy oscar]# ./install_cluster eth1
```

The first time you run the wizard, you will be prompted for a password for the MySQL database. Then, after a bit (depending on dependencies that need to be addressed), the OSCAR GUI-style installation wizard will appear. It may take several minutes for the wizard to appear. The console window from which the script was run will provide additional output, so keep it visible. This information is also written to an install log in the OSCAR installation directory. [Figure 6-2](#) shows the wizard.

Figure 6-2. OSCAR Installation Wizard



The Installation Wizard shows the basic steps that you will be going through to install your cluster. You can get a helpful explanation for any step by using the adjacent *Help...* button.

6.3.4.1 Step 0: Downloading additional packages

Before the installation can proceed, you should download any third-party packages you'll want using *opd*. Since *opd* downloads packages over the Internet, you'll need a working Internet connection to use it. Of course, if you are not interested in any of the third-party packages, you can skip this step. Also, it is possible to add packages later. But it is generally simpler if you do everything at once. You'll miss out on some very nice software if you skip this step.

Standalone *opd*

If you decide to run *opd* from the command line, you can find the command in the *scripts* subdirectory.

```
[root@amy oscar]# scripts/opd
```

Running *opd* as a standalone program the first time may prove tricky since, with a fresh install, several Perl modules that *opd* needs may not be installed. If this is the case, you'll get an error message. While you could manually install these modules, the

OSCAR installation script will also install them. If you run the wizard but stop it when the wizard window opens, you'll get around this problem and you'll be able to run the *opd* script.

When *opd* runs, after some initial output, it gives you a list of repositories for OSCAR packages to select from. Enter the number for the repository of interest.

Please select a default repository:

1. NCSA OSCAR package repository
= => <http://sponge.ncsa.uiuc.edu/ftp/oscar/repository/>
2. thin-OSCAR package repository
= => <http://thin-oscar.ccs.usherbrooke.ca/oscar-package/>
3. GSC OSCAR package repository
= => <http://www.bcgsc.ca/downloads/oscar/repository/>
4. Open Systems Lab, Indiana University
= => http://www.osl.iu.edu/~jsquyres/opd_repository/
5. Network & Cluster Computing Group, Oak Ridge National Laboratory
= => <http://www.csm.ornl.gov/oscar/repository/>

Selection (1-5): **1**

Next, *opd* takes you to that repository. You should see some output as the connection is made and then an *opd*> prompt. You can list the available packages with the **list** command.

...

```
= => NCSA OSCAR package repository
= => http://sponge.ncsa.uiuc.edu/ftp/oscar/repository/
= => 8 packages available
```

```
opd>list
```

1. clumon 1.2.1-6 (5.2MB)
2. Myrinet Driver (GM) 2.0.9-1 (15.4kB)
3. Maui 3.2.5p7-2 (18.5MB)
4. mpich-gm 1.2.5-4 (15.4MB)
5. MPICH-VMI 2.0.b3p1-1 (15.7MB)
6. PVFS 1.6-3 (707.9kB)
7. Torque 1.0.1p5-3 (5.5MB)
8. VMI 2.0.b3p1-1 (6.6MB)

To download a package (or packages), select the package by giving its number (or numbers separated by commas), and then use the `download` command to retrieve it (or them).

```
opd>8
```

```
Package "VMI" is selected
```

```
opd>download
```

You see a fair number of messages as the package(s) are downloaded.

```
...
```

```
10:15:40 (157.47 KB/s) - `/var/cache/oscar/downloads/vmi20b3p1-1.tgz.opd'
```

```
saved [6992096]
```

```
Successful!
```

```
- Checking size... OK
```

```
- Checking MD5 sum... OK
```

```
- Checking SHA1 sum... OK
```

```
- Saving to /var/cache/oscar/downloads/vmi20b3p1-1.tgz... OK
```

```
- Unpacking into /var/lib/oscar/packages/... OK
```

```
opd>quit
```

```
Goodbye.
```

You can quit `opd` with the `quit` command. Other commands are listed with the `help` command. Much of the output has been omitted in this example.

`opd` can be run as a separate program outside of the wizard or you can run it from the wizard by clicking on the first button, *Downloading Additional OSCAR Packages...* Generally, it is easier to run `opd` from the wizard, so that's what's described here. But there are some rare circumstances where you might want use the command-line version of `opd`, so there is a very brief description in the accompanying sidebar.

When you open `opd` from the wizard, a window will appear as shown in [Figure 6-3](#). Another pop up will appear briefly displaying the message `Downloading PackageInformation...` as the OSCAR repositories on the Internet are visited to see what packages are available. (Keep in mind that packages are added over time, so you may see additional packages not shown or discussed here.)

Using the downloader is straightforward. If you click on an item, it will display information about the package in the lower pane, including a description, prerequisite packages, and conflict. Just select the appropriate tab. In the upper pane, put a checkmark next to the packages you want. Then click on the *Download Selected Packages* button. A new pop up will appear with the message `Downloading PackageFile` with a file name and a percentage. Be patient; it may look like nothing is happening although the download is proceeding

normally.^[3] If you have a reasonable connection to the Internet, the download should go quickly. The packages are downloaded to the directory `/var/cache/oscar/downloads` and are unpacked in separate directories under `/var/lib/oscar/packages/`.

^[3] The percentage refers not to an individual package download but to the percentage of the total number of packages that have been downloaded. So if you are downloading five packages, the percentages will jump by 20 percent as each package is retrieved.

6.3.4.2 Step 1: Package selection

The next step is to select the packages you want to install. When you click on the *Select OSCAR Packages to Install...* button, the Oscar Package Selection window will

be displayed as shown in [Figure 6-4](#). This displays the packages that are available (but not the individual RPMs).

Figure 6-3. OSCAR's GUI for opd

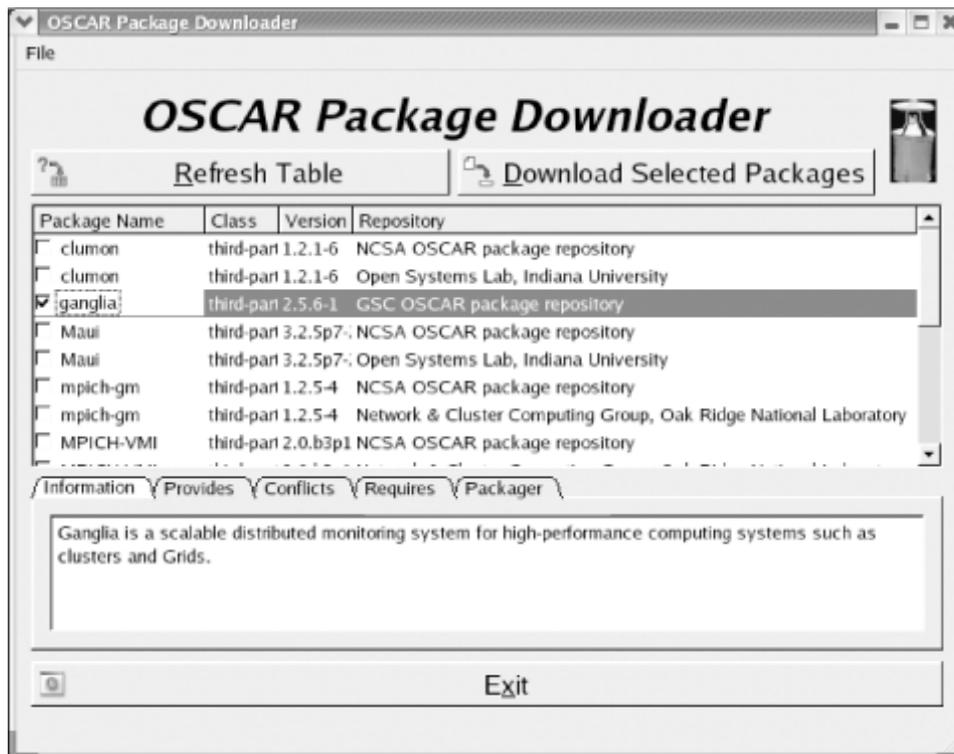
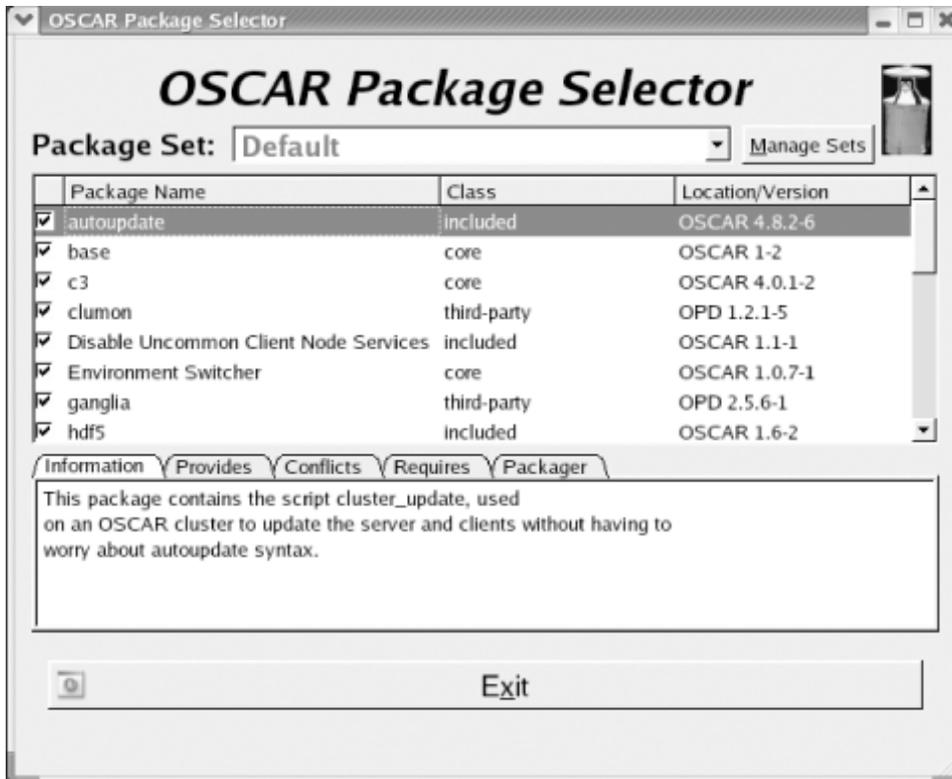


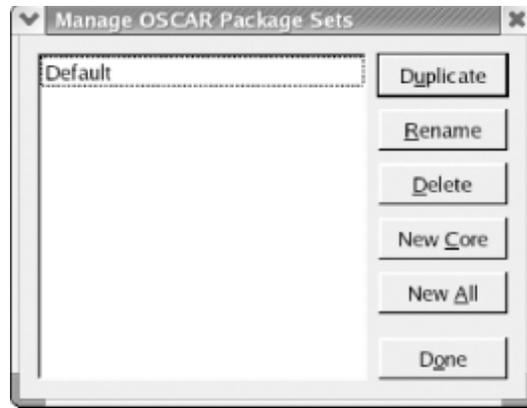
Figure 6-4. OSCAR's package selector



The information provided in the lower pane is basically the same as that provided by the OSCAR Package Downloader window, except the information is available for all the packages. The check boxes in the upper pane determine whether the packages are to be installed. Any package that you added with *opd* will also be included in the list, but by default, will not be selected. Don't forget to select these. If you haven't downloaded any packages, you probably won't need to change anything here, but scroll down the list and carefully look it over. If there is something you don't need or want, disable it. But keep in mind that it is generally easier to include something now than to go back and add it later. Don't bother trying to remove any of OSCAR's core packages; OSCAR won't let you. And it is strongly recommended that you don't remove *pfilter*. (If you have a compelling reason not to include *pfilter*, be sure to consult the installation manual for additional details explaining how to do this correctly.)

OSCAR constructs an *image* for client nodes, i.e., a copy of the operating system files and software that will be installed on the client. With OSCAR, you can build multiple images. If you are going to build multiple images, it is possible to define different sets of installation packages. The drop-down box at the top of the window allows you to select among the sets you've defined. You can define and manipulate sets by clicking on the *Manage Sets* button at the top of the window. A pop-up window, shown in [Figure 6-5](#), allows you to manipulate sets, etc. The easiest way to create a new set is to duplicate an existing set, rename the set, and then edit it.

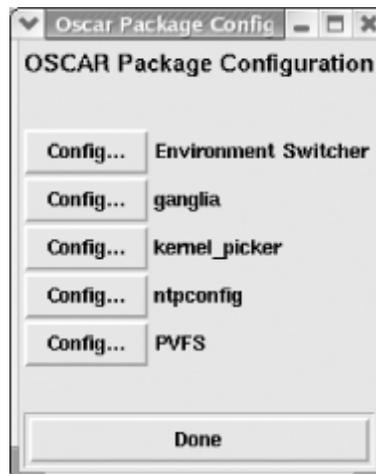
Figure 6-5. Managing package sets



6.3.4.3 Step 2: Configuring packages

Step 2 is the configuration of selected OSCAR packages. All in all, the default configurations should meet most users' needs, so you can probably skip this step. [Figure 6-6](#) shows the configuration menu. Most packages do not require configuration at this point and are not included in the menu.

Figure 6-6. Package configuration



In this example, only five of the packages need or permit additional configuration. Each of these, if selected, will generate a window that is self-explanatory. The *Environment Switcher* allows you to select either LAM/MPI or MPICH as the default. Since a user can change the default setting, your selection isn't crucial. The *switcher* script can be run on the command line and is described later in the chapter.

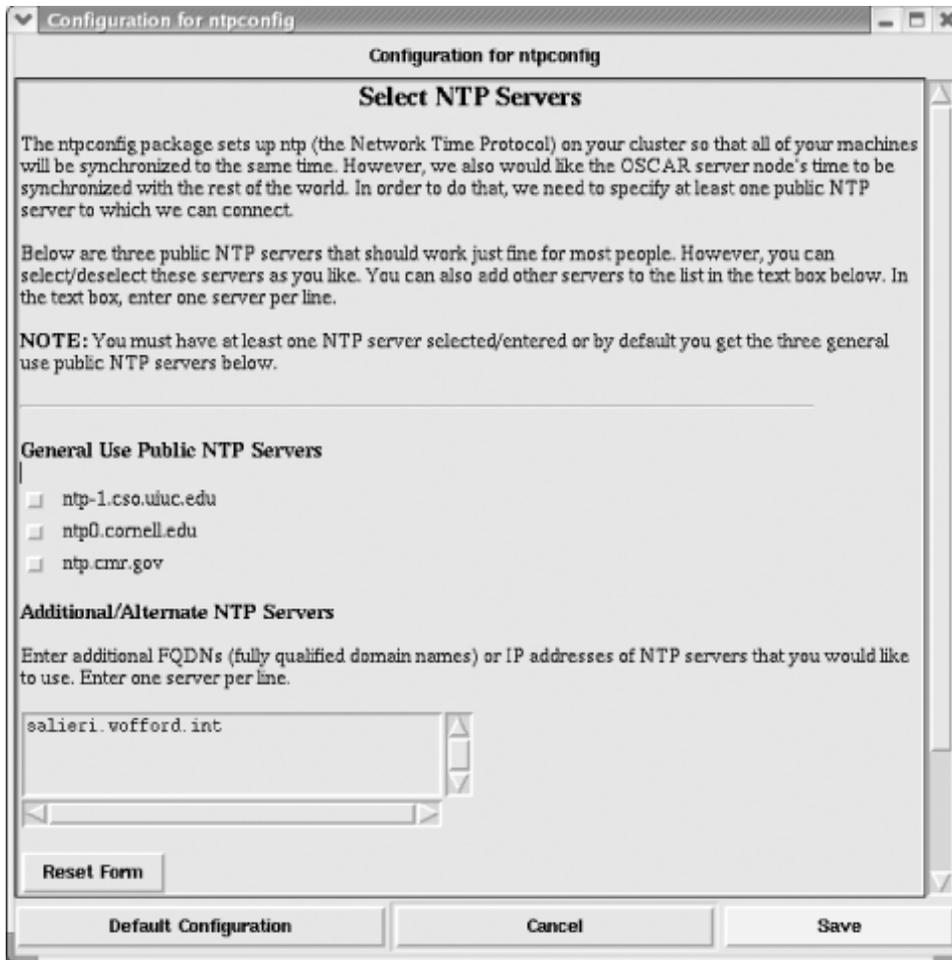
The *kernel_picker* is potentially a complicated option. Fortunately, if you are using the default kernel, you can ignore it completely. Basically, the *kernel_picker* allows you to change kernels used when building system images. You could use it to install a previously built kernel such as one configured with the openMosix extensions. The *kernel_picker* window is shown in [Figure 6-7](#). (See the *kernel_picker(1)* manpage for more information.)

Figure 6-7. GUI for kernel_picker



[Figure 6-8](#) shows the *ntpconfig* window. The *ntpconfig* option allows you to specify the address of NTP servers used by the cluster server. While the server synchronizes to an external source, the clients synchronize to the cluster server. There are several default NTP servers listed with check boxes, and you can enter your own choices. In this example, *saleri.wofford.int* has been added. If you have a local timeserver, you'll certainly want to use that instead of the defaults, or if you know of a "closer" timeserver, you may prefer to use it. But if in doubt, the defaults will work.

Figure 6-8. Configuring NTP



Pretty much everyone can expect to see the three choices just described. If you have added additional packages, you may have other choices. In this example, the packages for Ganglia and PVFS were both added, so there are configuration windows for each of these. (With Ganglia you can change the naming information and the network interface used to reach the client nodes. With PVFS you can change the number of I/O servers you are using.)

When you complete a step successfully, you should see a message to that effect in the console window, as shown in [Figure 6-9](#). For some steps, there is also a pop-up window that tells you when the step is finished. While the first two steps are optional, in general be very careful not to go to the next step until you are told to do so. The console window also displays error messages. Unfortunately, the console can be a little misleading. You may see some benign error messages, particularly from *rpm* and *rsync*, and occasionally real error messages may get lost in the output. Nonetheless, the console is worth watching and will give you an idea of what is going on.

Figure 6-9. Console window during installation

```
root@amy:~/oscar.3.0
File Edit View Terminal Go Help
=====
== Running step 2 of the OSCAR wizard: Configure selected OSCAR packages
=====

--> About to run /opt/oscar/packages/kernel_picker/scripts/pre_configure for ker
nel_picker
warning: /tftpboot/rpm/redhat-release-9-3.i386.rpm: V3 DSA signature: NOKEY, key
ID db42a60e
[OSCAR::PackageBest :: Line 407] Reading package directory
[OSCAR::PackageBest :: Line 419] Reading cache file.
[OSCAR::PackageBest :: Line 432] Comparing cache to directory.
[OSCAR::PackageBest :: Line 457] Writing new cache file.
62750 blocks
--> About to run /opt/oscar/packages/switcher/scripts/pre_configure for switcher
--> About to run /var/lib/oscar/packages/pvfs/scripts/post_configure for pvfs
Building PVFS rpm...
pvfs          #####
--> About to run /opt/oscar/packages/switcher/scripts/post_configure for switche
r
Setting default for tag mpi ("lan-7.0")
Tag "mpi" does not seem to exist yet. Skipping.
--> Step 2: Completed successfully
```

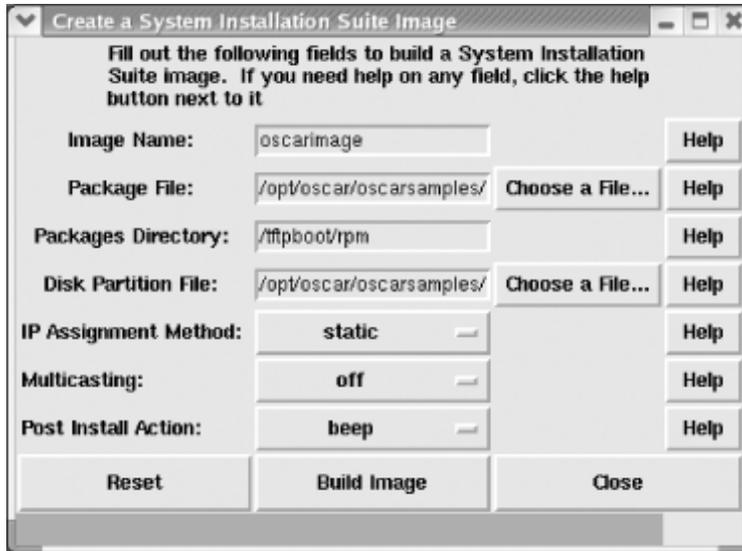
6.3.4.4 Step 3: Installing server software

In Step 3, you will install all the packages that the server needs and configure them. There are no fancy graphics here, but you will see a lot of activity in the console window. It will take several minutes to set up everything. A pop-up window will appear, telling you that you were successful or that there was an error, when this step completes. If all is well, you can close the popup window and move on to the next step. If not, you'll need to go to the console window and try to puzzle out the error messages, correct the problem, and begin again. You should need to run this step only once.

6.3.4.5 Step 4: Building a client image

In Step 4, you build the client image. The client image is all the software that will be installed on a client, including the operating system. Since it is possible to create multiple client images, you are given the option to specify a few details as shown in [Figure 6-10](#). You can specify image names if you have multiple images, the location of the packages used to build the image, and the names of the package list and disk partition files. These last two files are described later in this chapter. The defaults are shown in the figure. If you aren't building multiple images, you can probably stick with the defaults. You can also determine how the IP addresses of the clients are set and the behavior of the clients once the installation completes. Your choices are *dhcp*, *static*, and *replicant*. With *static*, the IP addresses will be assigned to the clients once and for all at the time of the installation. This is the most reasonable choice. *dhcp* used DHCP to set IP addresses, while *replicant* doesn't mess with addresses. The next button allows you to turn multicasting on or off. The possible post-install actions are *beep*, *reboot*, or *shutdown*. With *beep*, the clients will unmount the file system and beep at you until rebooted. *reboot* and *shutdown* are just what you would expect. All in all, OSCAR's defaults are reasonable. When you have made your selection, click on *Build Image*.

Figure 6-10. Creating client images



OSCAR uses SIS to create the image. Unlike our example in [Chapter 8](#), you do not need to create a sample system. Image creation is done on the server.

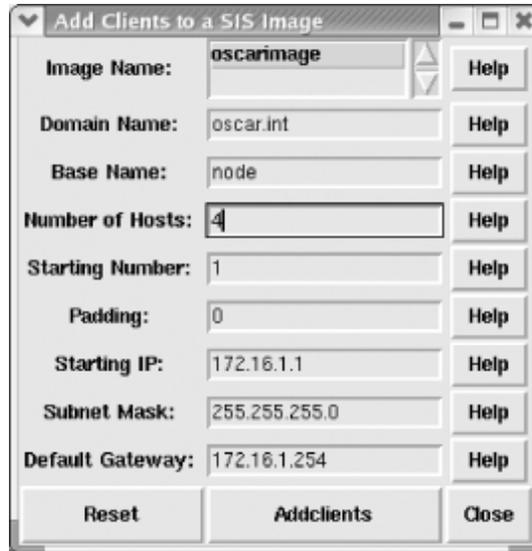
This step takes a while to complete. There is a red bar that grows from left to right at the bottom of the window that will give you some idea of your progress. However, you will be done before the bar is complete. Another pop-up window will appear when you are done. You'll run this step once for each *different* image you want to create. For most clusters, that's one image. Keep in mind that images take a lot of space. Images are stored in the directory `/var/lib/systemimager/images`.

6.3.4.6 Step 5: Defining clients

Once you have built the image, things should start going a lot faster. Step 5 defines the scope of your network. This is done using the window shown in [Figure 6-11](#). If you have multiple images, you can select the image you want to use in the first field. The next five fields are used to specify how node names will be constructed. The host name is constructed by appending a number to the base name. That number begins at the start value and is padded with leading zeros, if needed, as specified by the padding field. The domain name is then appended to the node name to form the fully qualified domain name or FQDN. The number of hosts you create is specified in the fourth field. In this example, four nodes are created with the names `node1.oscar.int`, `node2.oscar.int`, `node3.oscar.int`, and `node4.oscar.int`. (With padding set to 3, you would get `node001.oscar.int`, etc.) OSCAR assumes that hosts are numbered sequentially. If for some reason you aren't building a single block of sequential hosts, you can rerun this step to build the block's hosts as needed.

The last three fields are used to set IP parameters. In this example, the four hosts will have IP addresses from `172.16.1.1` through `172.16.1.4` inclusive.

Figure 6-11. Defining OSCAR clients

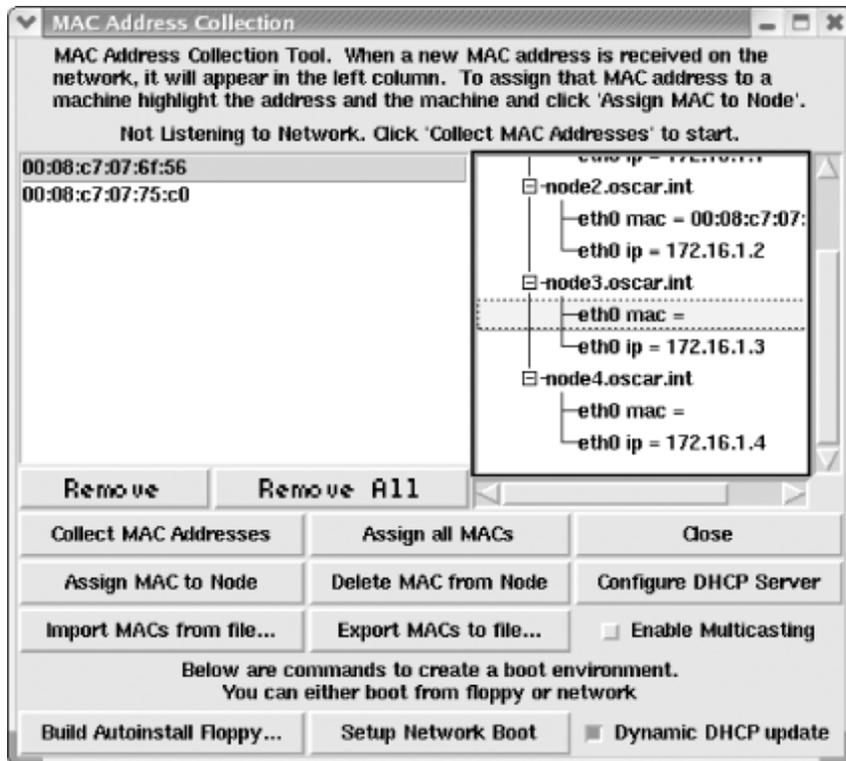


Once you have the fields the way you want them, click on the *Addclients* button. You should see a small pop-up window indicating that you were successful. If so, you can close the pop-up window and the client definition window and go on to the next step.

6.3.4.7 Step 6: Setting up the network

Step 6, shown in [Figure 6-12](#), sets up the DHCP server and maps IP addresses to MAC addresses. (It is possible to run OSCAR without configuring the head as a DHCP server, but that isn't described here.) This step requires several substeps. First, you will need to collect the MAC or Ethernet addresses from the adapters in each of the client machines. You can do this manually or use OSCAR to do it. If you select the *Collect MAC Addresses* button and then power on each client, OSCAR will listen to the network, capture MAC addresses from DHCP requests, and display the captured addresses in the upper left pane. However, if no DHCP requests are generated, the machines won't be discovered. (Be sure to turn this option off when you have collected your addresses.) Under some circumstances, it is possible to collect MAC addresses from machines not in your cluster. If this happens, you can use the *Remove* button to get rid of the addresses you don't want. If you collect the MAC addresses, be sure to save them to a file using the *Export MACs to file...* button.

Figure 6-12. Setting up networking



Alternately, if you know the MAC addresses, you can enter them into a file and read the file with the *Import MACs from file...* button. To create the file, just put one MAC address on a line with the fields separated by colons. Here is part of a MAC file:

```
00:08:c7:07:6e:57
00:08:c7:07:68:48
00:08:c7:07:c1:73
00:08:c7:07:6f:56
```

OSCAR can be picky about the format of these addresses. (If you are collecting MAC addresses rather than importing them from a file, it is a good idea to export the collected MAC addresses. In the event you want to reinstall your clusters, this can save some work.)

Once you have the MACs, you'll need to assign them to the clients displayed in the top right pane. You can do this all at once with the *Assign all MACs* button, or you can do it individually with the *Assign MAC to Node* button. While the first method is quicker, you may prefer the second method to better control which machine gets which address. With the second method, click on a MAC address to select it, click on a client's interface, and then click the *Assign MAC to Node* button. Repeat this step for each client.

If the *Dynamic DHCP update* checkbox is selected, then each time you assign an MAC address, the DHCP server is refreshed. If not selected, then once you have configured your nodes you can click on *Configure DHCP Server*. OSCAR creates the DHCP configuration file */etc/dhcpd.conf* and starts DHCP. If you already have a DHCP configuration file, OSCAR will save it as *dhcpd.conf.oscarbak* before creating the new file.

SIS is used to push files to the nodes. By default, images are transferred using *rsync*. It is also possible to distribute images using *flamethrower*, a multicast-based program. Because the multicast facilities are still somewhat experimental, *rsync* is the recommended method for new users. If you elect to use *flamethrower*, you'll need to ensure that your network is properly configured to support multicasting. If the *Enable Multicasting* checkbox is selected,

flamethrower is used to push files. If it is unselected, *rsync* is used. Chapter 8 provides a detailed description of SIS and *rsync*.

Next, you'll need to create an autoinstall diskette. When the potential client machines are booted with this diskette, the process of downloading their image begins. Click on the button in the lower left of the window and a new window will take you through the creation of the floppy. Use the default **standard** when prompted for a flavor. If you have a large cluster, you should create several diskettes so you can install several systems at once.



The next step installs the software on the individual machines. This step will overwrite the existing system! Are you sure you are ready to do this?

You are through with the **Mac Address Collection** window but there is one more thing you must do before going to the next step—install the image on your clients. While this sounds formidable, it is very straightforward with OSCAR. Just insert the floppy you just created and reboot each system.

You should see a "SYSLINUX 2.0 Screen" with a boot prompt. You can hit return at the prompt or just wait a few seconds. The system will go to the OSCAR server and download and install the client operating system. Repeat this process with each system. You can do all your clients at the same time if you wish. The boot floppy is only used for a couple of minutes so once the install is on its way, you can remove the floppy and move on to another machine. If you have several floppies, you can get a number of installations going very quickly. The installation will depend on how many clients you have, how fast your network is, and how many packages went into your cluster image, but it should go fairly quickly.



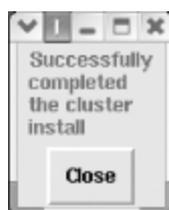
You may need to go into the ROM startup menu and change the client's boot configuration so it will boot from a diskette. If you do, don't forget to change it back when you are done.

When a client's image is installed, the machine will start beeping. If you haven't already removed the floppy, do so now and reboot the system. The filesystems on the clients will not be mounted at this point so it is safe to just cycle the power. (Actually, you could have set the system to automatically reboot back in Step 4, but you'll need to make sure the floppy has been removed in a timely manner if you do so.)

6.3.4.8 Step 7: Completing the setup

Once all the clients have booted, there are a few post-install scripts that need to be run. Just click on the button. After a few minutes, you should get the popup window shown in [Figure 6-13](#). Well done! But just to be on the safe side, you should test your cluster.

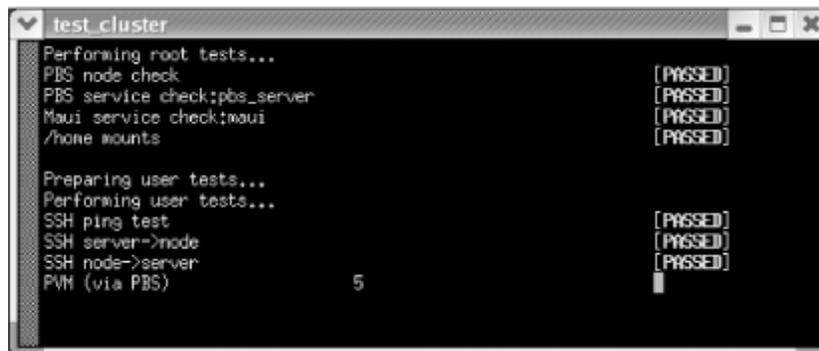
Figure 6-13. Success!



6.3.4.9 Step 8: Testing

Step 8 tests your cluster. Another console window opens and you see the results from a variety of tests. [Figure 6-14](#) shows what the output looks like early in the process. There is a lot more output that will vary depending on what you've installed. (Note that you may see some PBS errors because the PBS server is initially shutdown. It's OK to ignore these.)

Figure 6-14. Testing the cluster



```
test_cluster
Performing root tests...
PBS node check [PASSED]
PBS service check:pbs_server [PASSED]
Maui service check:maui [PASSED]
/home mounts [PASSED]

Preparing user tests...
Performing user tests...
SSH ping test [PASSED]
SSH server->node [PASSED]
SSH node->server [PASSED]
PVM (via PBS) 5
```

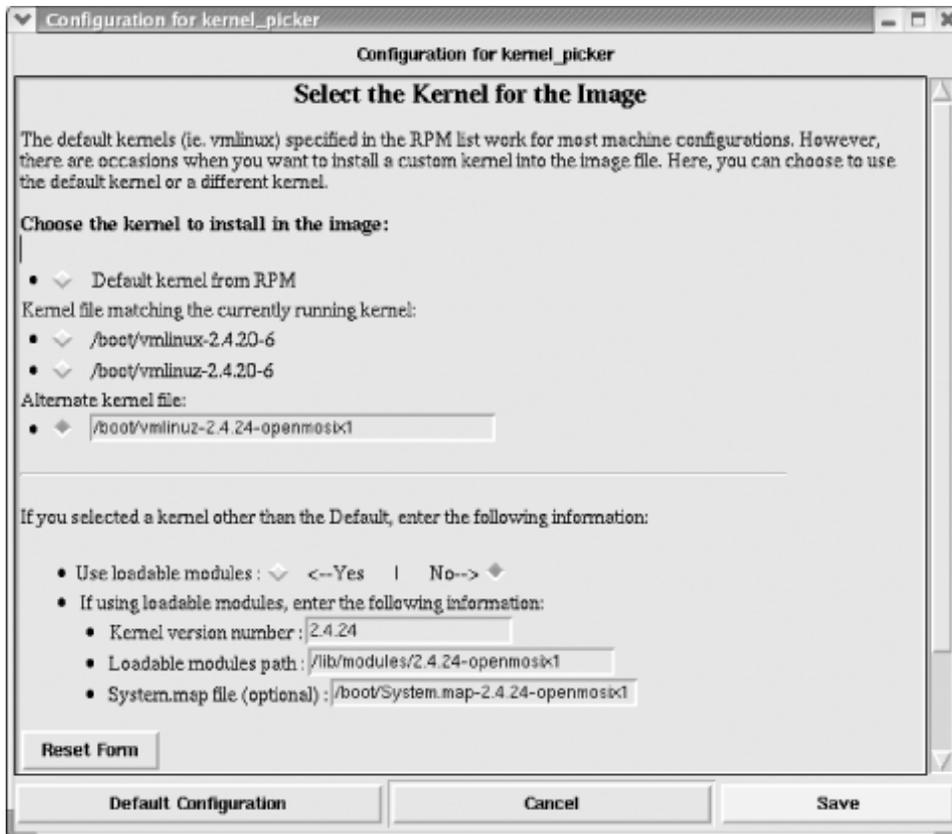
Congratulations! You have an OSCAR cluster up and running! This probably seems like a complicated process when you read about it here, but it all goes fairly quickly. And think for a moment how much you have accomplished.

If something may goes wrong with your installation, OSCAR provides a *start_over* script that can be used to clean up from the installation and give you another shot at installing OSCAR. This is not an uninstaller. It will not return your machine to the pristine state it was in before the installation but should clean things up enough so that you'll be able to reinstall OSCAR. If you use this script, be sure to log out and back onto the system before you reinstall OSCAR. On the other hand, you may just want to go back and do a clean install.

6.3.5 Custom Installations

As should be apparent from the installation you just went through, there are several things you can do to customize your installation. First, you can alter the kernel using *kernel_picker*. For example, if you want to install the openMosix kernel on each system, you would begin by installing the openMosix kernel on the head node. Then, when installing OSCAR, you would use *kernel_picker* to select the openMosix kernel. This is shown in [Figure 6-15](#).

Figure 6-15. Using the openMosix kernel



Of course, for a new kernel to boot properly, you'll need to ensure that the appropriate kernel load modules are available on each machine. For openMosix, you can do this by installing the openMosix package.

Fortunately, it is straightforward to change the packages that OSCAR installs. For example, if you are installing the openMosix kernel, you'll want the openMosix tools as well. If you look back at [Figure 6-10](#), one of the fields was *Package File*. In the directory */opt/oscar/oscarsamples* there are several files, one for each supported Linux distribution. These files contain the packages that will be installed by OSCAR. For example, for Red Hat 9 the file is *redhat-9-i386.rpm/ist*. If there are some additional packages that you would like to install on the cluster nodes, you can make a backup copy of the desired lists and then add those packages to the list. You should put one package per line. You need to include only the package name, not its version number. For example, to install the openMosix tools package, you could add a line with `openmosix-tools` (rather than `openmosix-tools-0.3.5-1.i386.rpm`). The package list is pretty basic, which leads to a quick install but a minimal client. Of course, you'll need to make sure the packages are in (or linked to) the */tftpboot/rpm* directory and that you include all dependencies in the package list.

While you are in the */opt/oscar/oscarsamples* directory, you can also alter the disk setup by editing either the *sample.disk.ide* or *sample.disk.scsi* file. For example, if you have an IDE drive and you want to use the ext3 file system rather than ext2, just change all the `ext2` entries to `ext3` in the file *sample.disk.ide*. Of course, unless you have a compelling reason, you should probably skip these changes.

6.3.6 Changes OSCAR Makes

It is pretty obvious that OSCAR has just installed a number of applications on your system. As you might expect, OSCAR made a number of additional, mostly minor, changes. It will probably take you a while to discover everything that has changed, but these changes shouldn't cause any problems.

While OSCAR tries to conform to standard installation practices, you won't get exactly the same installation and file layout that you might have gotten had you installed each application individually. The changes are really minimal, however. If you've never done individual installations, the whole issue is probably irrelevant unless you are looking at the original documentation that comes with the application.

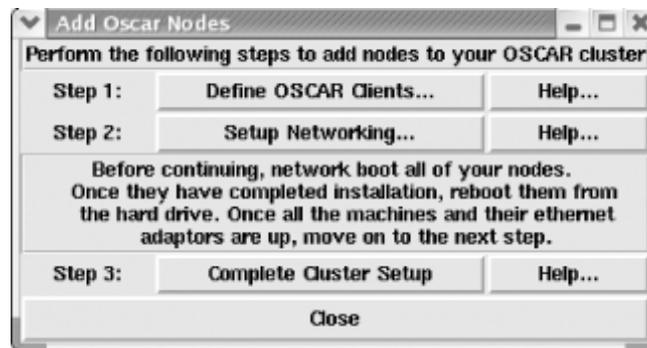
You can expect to find most configuration files in the usual places—typically but not always under the `/etc` directory. Configuration files that OSCAR creates or changes include `c3.conf`, `crontab`, `dhcpd.conf`, `gmetad.conf`, `gmond.conf`, `ntp.conf`, `ntp/step-tickers`, `pcp.conf`, `pfilter.conf`, `ssh/ssh_config`, and files in `xinetd.d`. OSCAR will also update `/etc/hosts`, `/etc/exports`, and `/etc/fstab` as needed.

Several of the packages that are installed require accounts, which are created during the install. Take a look at `/etc/passwd` to see which accounts have been added to your system. For the global user profiles, OSCAR includes a link to a script to set up SSH keys and adds some paths. You might want to look at `/etc/profile.d/ssh-oscar.sh` and `/etc/profile.d/ssh-oscar.csh`. OSCAR restarts all affected services.

6.3.7 Making Changes

There are three more buttons above the *Quit* button on the wizard. Each does exactly what you would expect. The *Add OSCAR Clients...* adds additional nodes. Adding a node involves three, now familiar steps. When you select *Add OSCAR Clients...* you'll get the menu shown in [Figure 6-16](#).

Figure 6-16. Adding nodes



The first step defines the client or range of clients. You'll get the same menu ([Figure 6-11](#)) you used when you originally set up clients. Be sure you set every field as appropriate. OSCAR doesn't remember what you used in the past, so it is possible to end up with inconsistent host names and domains. (If this happens, you can just delete the new nodes and add them again, correcting the problem, but be sure to exit and restart OSCAR after deleting and before adding a node back.) Of course, you'll also need to set the starting node and number of nodes you are adding. In the second step, you map the MAC address to a machine just as you've done before (see [Figure 6-12](#)). Finally, with the last step you run the scripts to complete the setup.

Deleting a node is even easier. Just select the *Delete OSCAR Clients...* button on the wizard. You'll see a window like the one shown in [Figure 6-17](#) listing the nodes on your cluster. Select the nodes you want to delete and click on the *Delete clients* button. OSCAR will take care of the rest. (Deleting a node only removes it from the cluster. The data on the node's hard disk is unaffected as are services running on the node.)

Figure 6-17. Deleting nodes

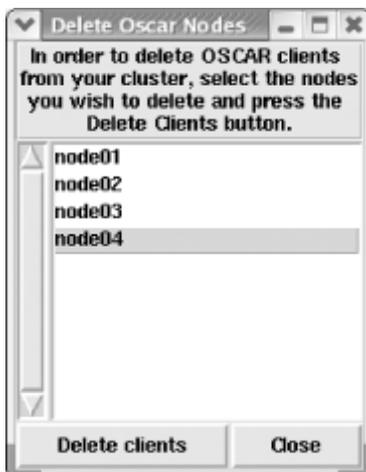
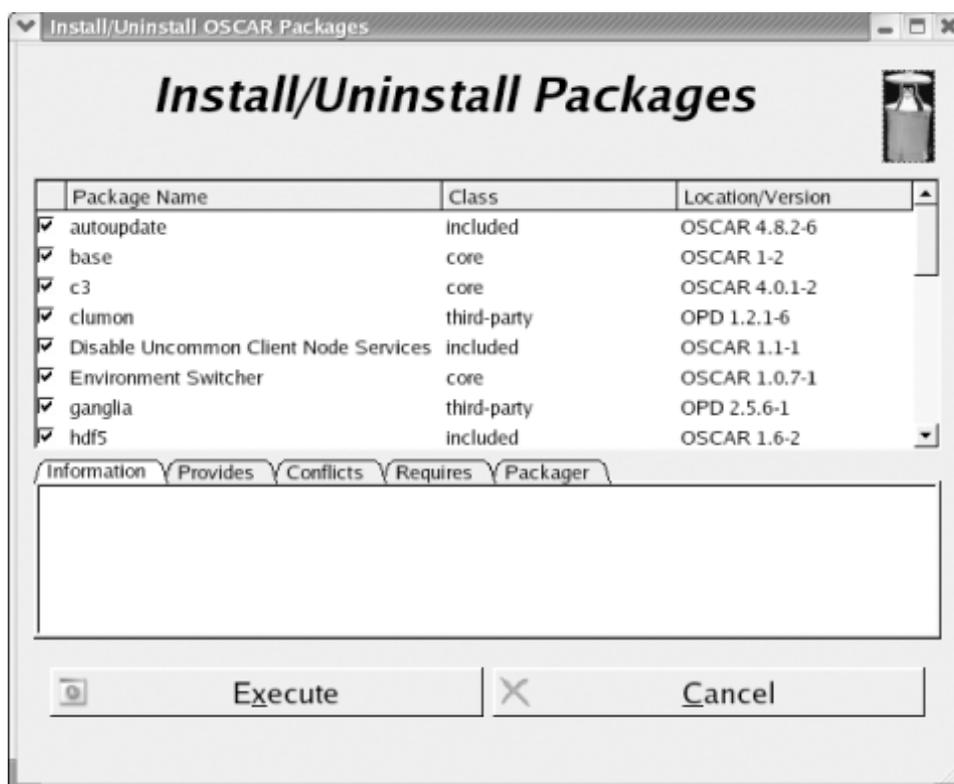


Figure 6-18. Adding and removing packages



Finally, you can install and uninstall packages using the *Install/Uninstall OSCAR Packages...* button. This opens the window shown in [Figure 6-18](#). Set the checkbox and click on the *Execute* button. Any new packages you've checked will be installed, while old packages you've unchecked will be uninstalled. This is a new feature in OSCAR and should be used with caution.

6.4 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!

6.4.1 pfilter

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.

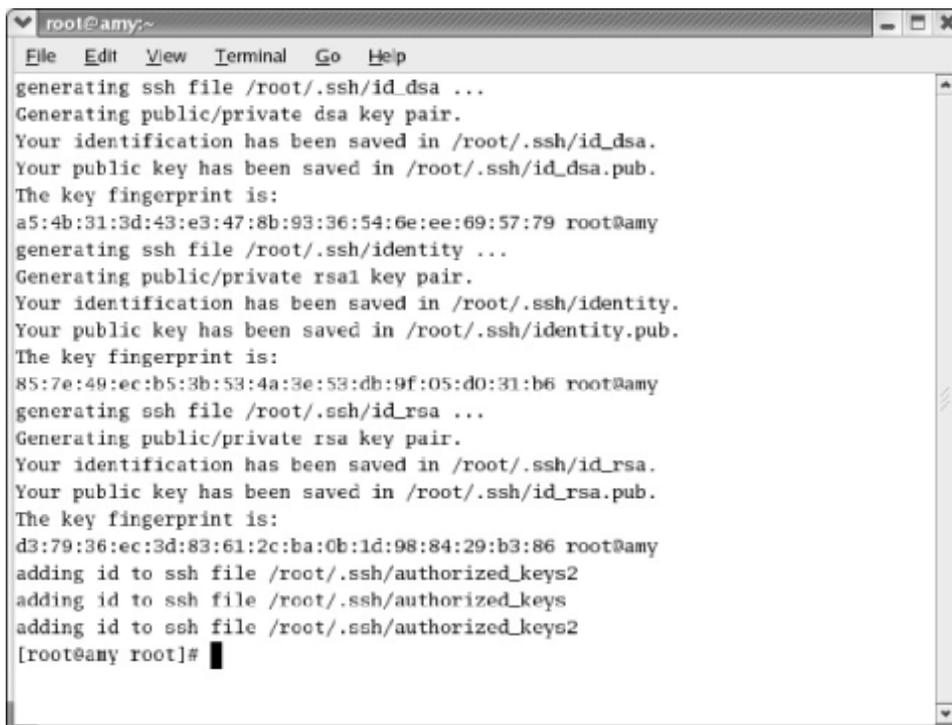
The configuration file for *pfilter*, */etc/pfilter.conf*, contains the rules used by *pfilter* and can be edited if you need to change them. The OSCAR installation adds some rules to the default configuration. These appear to be quite reasonable, so it is unlikely that you'll need to make

any changes. The manpages for *pfilter.conf(5)* and *pfilter.rulesets(5)* provide detailed instructions should you wish to make changes. While the rules use a very simple and readable syntax, instruction in firewall rulesets is outside the scope of this book.

6.4.2 SSH and OPIUM

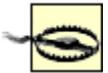
Within the cluster, OSCAR is designed to use the SSH protocol for communications. Use of older protocols such as TELNET or RSH is strongly discouraged and really isn't needed. *openSSH* is set up for you as part of the installation. *OPIUM*, the *OSCAR Password Installer and User Manager* tool, handles this. OPIUM installs scripts that will automatically generate SSH keys for users. Once OSCAR is installed, the next time a user logs in or starts a new shell, she will see the output from the key generation script. (Actually, at any point after Step 3 in the installation of OSCAR, key generation is enabled.) [Figure 6-19](#) shows such a login. Note that no action is required on the part of the user. Apart from the display of a few messages, the process is transparent to users.

Figure 6-19. Key setup upon login



```
root@amy:~  
File Edit View Terminal Go Help  
generating ssh file /root/.ssh/id_dsa ...  
Generating public/private dsa key pair.  
Your identification has been saved in /root/.ssh/id_dsa.  
Your public key has been saved in /root/.ssh/id_dsa.pub.  
The key fingerprint is:  
a5:4b:31:3d:43:e3:47:8b:93:36:54:6e:ee:69:57:79 root@amy  
generating ssh file /root/.ssh/identity ...  
Generating public/private rsal key pair.  
Your identification has been saved in /root/.ssh/identity.  
Your public key has been saved in /root/.ssh/identity.pub.  
The key fingerprint is:  
85:7e:49:ec:b5:3b:53:4a:3e:53:db:9f:05:d0:31:b6 root@amy  
generating ssh file /root/.ssh/id_rsa ...  
Generating public/private rsa key pair.  
Your identification has been saved in /root/.ssh/id_rsa.  
Your public key has been saved in /root/.ssh/id_rsa.pub.  
The key fingerprint is:  
d3:79:36:ec:3d:83:61:2c:ba:0b:1d:98:84:29:b3:86 root@amy  
adding id to ssh file /root/.ssh/authorized_keys2  
adding id to ssh file /root/.ssh/authorized_keys  
adding id to ssh file /root/.ssh/authorized_keys2  
[root@amy root]#
```

Once you set up the cluster, you should be able to use the *ssh* command to log onto any node from any other node, including the server, without using a password. On first use, you will see a warning that the host has been added to the list of known hosts. All this is normal. (The changes are saved to the directory */etc/profile.d*.)

 The *openSSH* configuration was not designed to work with other systems such as *Kerberos* or *NIS*.

In addition to setting up *openSSH* on the cluster, OPIUM includes a *sync_users* script that synchronizes password and group files among the cluster using C3 as a transport mechanism. By default, this is run every 15 minutes by *cron*. It can also be run by root with the *--force*

option if you don't want to wait for *cron*. It cannot be run by other users. OPIUM is installed in */opt/opium* with *sync_users* in the subdirectory *bin*. The configuration file for *sync_users*, *sync_user.conf*, is in the *etc* subdirectory. You can edit the configuration file to change how often *cron* runs *sync_user* or which files are updated, among other things. (*sync_users* is something of a misnomer since it can be used to update any file.)



Because the synchronization is done from the server to the clients, it is important that passwords always be changed on the server and never on the clients. The next time *sync_user* runs, password changes on client will be lost as the password changes on the server propagate to the clients.

6.5 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
```

```
lam-7.0
```

```
lam-with-gm-7.0
```

```
mpich-ch_p4-gcc-1.2.5.10
```

In this example, we see the attributes are the two available MPI implementations.

You use the `--show` option to use *switcher* to determine the default MPI environment.

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

```
system:default=lam-7.0
```

```
system:exists=true
```

Alternately, you can use the *which* command:

```
[sloanjd@amy sloanjd]$ which mpicc
```

```
/opt/lam-7.0/bin/mpicc
```

From the path, we can see that we are set up to use LAM/MPI rather than MPICH.

To change the default to MPICH, simply assign the desired attribute value to the tag.

```
[sloanjd@amy sloanjd]$ switcher mpi = mpich-ch_p4-gcc-1.2.5.10
```

```
Attribute successfully set; new attribute setting will be effective for  
future shells
```

The change will not take effect immediately, but you will be using MPICH the next time you log in (and every time you log in until you run *switcher* again.) After the first time you make a change, *switcher* will ask you to confirm tag changes. (Also, the very first time you use *switcher* to change a tag, you'll receive a tag "does not exist" error message that can be safely ignored.)

As root, you can change the default tag for everyone using the `--system` flag.

```
[root@amy root]# switcher mpi = lam-7.0 --system
```

One last word of warning! If you make a typo when entering the value for the attribute, *switcher* will not catch your mistake.

6.6 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.

Copy over the example you want to run, compile it with `mpicc`, and then run it with `mpirun`.

```
[sloanjd@amy sloanjd]$ cp /usr/share/doc/lam-oscar-7.0/examples/
```

```
alltoall/alltoall.c $HOME
```

```
[sloanjd@amy sloanjd]$ mpicc -o alltoall alltoall.c
```

```
[sloanjd@amy sloanjd]$ mpirun -np 4 alltoall
```

```
Rank 0 not sending to myself
```

```
Rank 1 sending message "1" to rank 0
```

```
Rank 2 sending message "2" to rank 0
```

```
...
```

You should see additional output. The amount will depend on the number of machines in *myhosts*. Happy coding, everyone!

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>.

7.1 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.

7.1.1 Prerequisites

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.

7.1.2 Downloading Rocks

To install Rocks, you'll first need the appropriate CD-ROMs. Typically, you'll go to the Rocks web site <http://rocks.npaci.edu/Rocks/>, follow the link to the download page, download the ISO images you want, and burn CD-ROMs from these images. (This is also a good time to download the user manuals if you haven't already done so.) Rocks currently supports x86 (Pentium and Athlon), x86_64 (AMD Opteron), and IA-64 (Itanium) architectures.

Be sure to download the software that is appropriate for your systems. You'll need at least two ISO images, maybe more depending upon the software you want. Every installation will require the Rocks Base and HPC Roll. The core install provides several flavors of MPICH, Ganglia, and PVFS. If you want additional software that is not part of the core Rocks installation, you'll need to download additional rolls. For example, if you want *tripwire* and *chkrootkit*, two common security enhancements, you could download the Area 51 roll. If you are interested in moving on to grid computing, Rocks provides rolls that ease that process (see the sidebar, "Rocks and Grids").

Currently available rolls include the following:

Sun Grid Engine (SGE) roll

This roll includes the Sun Grid Engine, a job queuing system for grids. Think of this as a grid-aware alternative to openPBS. This is open source distributed management software. For more information on SGE, visit <http://gridengine.sunsource.net>.

Grid roll

The NSF Middleware Initiative (NMI) grid roll contains a full complement of grid software, including the Globus toolkit, Condor-G, Network Weather Service, and MPICH-G2, to name only a few. For more information on the NMI project, visit <http://www.nsf-middleware.org>.

Intel roll

This roll installs and configures the Intel C compiler and the Intel FORTRAN compiler. (You'll still need licenses from Intel.) It also includes the MPICH environments built for these compilers. For more information on the Intel compilers and their use with Rocks, visit http://www.intel.com/software/products/distributors/rock_cluster.htm.

Area 51 roll

This roll currently includes *tripwire* and *chkrootkit*. *tripwire* is a security auditing package. *chkrootkit* examines a system for any indication that a root kit has been installed. For more information on these tools, visit the sites <http://www.tripwire.org> and <http://www.chkrootkit.org>.

Scalable Cluster Environment (SCE) roll

This roll includes the OpenSCE software that originated at Kasetsart University, Thailand. For more information on OpenSCE, visit <http://www.opensce.org>.

Java roll

The Java roll contains the Java Virtual Machine. For more information on Java, visit <http://java.sun.com>.

PBS roll

The Portable Batch System roll includes the OpenPBS and Maui queuing and scheduling software. For more information on these packages, see [Chapter 11](#) or visit <http://www.openpbs.org>.

Condor roll

This roll includes the Condor workload management software. Condor provides job

queuing, scheduling, and priority management along with resource monitoring and management. For more information on Condor, visit <http://www.cs.wisc.edu/condor/>.

Some rolls are not available for all architectures. It's OK to install more than one roll, so get what you think you may need now. Generally, you won't be able to add a roll once the cluster is installed. (This should change in the future.)

Once you've burned CD-ROMs from the ISO images, you are ready to start the installation. You'll start with the frontend.

Rocks and Grids

While grids are beyond the scope of this book, it is worth mentioning that, through its rolls mechanism, Rocks makes it particularly easy to move into grid computing. The grid roll is particularly complete, providing pretty much everything you'll need to get started—literally dozens of software tools and packages. Software includes:

- Globus Toolkit—a collection of modular technologies, including tools for authentication, scheduling and file transfer that simplifies collaboration among sites.
- Condor-G—the Condor software with grid and Globus compatibility.
- Network Weather Service—a monitoring service that dynamically forecasts network and resource performance.
- MPICH-G2—a grid-enabled implementation of MPICH.
- Grid Packaging Tools—a collection of packaging tools built around XML. This is a package management system.
- KX.509/KCA—technology that provides a bridge between Kerberos and PKI infrastructure.
- GSI OpenSSH—a modified version of SSH that supports GSI authentication (Grid Security Infrastructure).
- MyProxy—a credential repository for grids.
- Gridconfig Tools—a set of tools to configure and tune grid technologies.

These are just the core. If you are new to grids and want to get started, this is the way to go. (The [Appendix A](#) includes the URLs for these tools.)

7.1.3 Installing the Frontend

The frontend installation should go very smoothly. After the initial boot screens, you'll see a half dozen or so screens asking for additional information along with other screens giving status information for the installation. If you've installed Red Hat Linux before, these screens will look very familiar. On a blue background, you'll see the Rocks version information at the very top of the screen and interface directions at the bottom of the screen. In the center of the screen, you'll see a gray window with fields for user supplied information or status information. Although you can probably ignore them, as with any Red Hat installation, the Linux virtual consoles are available as shown in [Table 7-1](#). If you have problems, don't forget these.

Table 7-1. Virtual consoles

Console	Use	Keystroke
1	Installation	Cntl-Alt-F1
2	Shell prompt	Cntl-Alt-F2
3	Installation log	Cntl-Alt-F3
4	System messages	Cntl-Alt-F4
5	Other messages	Cntl-Alt-F5

Boot the frontend with the Rocks Base CD and stay with the machine. After a moment, you will see a boot screen giving you several options. Type `frontend` at the `boot:` prompt and press Enter. You need to do this quickly because the system will default to a compute node installation after a few seconds and the prompt will disappear. If you miss the prompt, just reboot the system and pay closer attention.

After a brief pause, the system prompts you to register your roll CDs. When it asks whether you have any roll CDs, click on Yes. When the CD drive opens, replace the Rocks Base CD with the HPC Roll CD. After a moment the system will ask if you have another roll CD. Repeat this process until you have added all the roll CDs you have. Once you are done, click on No and the system will prompt you for the original Rocks Base CD. Registration is now done, but at the end of the installation you'll be prompted for these disks again for the purpose of actual software installation.

The next screen prompts you for information that will be included in the web reports that Ganglia creates. This includes the cluster name, the cluster owner, a contact, a URL, and the latitude and longitude for the cluster location. You can skip any or all of this information, but it only takes a moment to enter. You can change all this later, but it can be annoying trying to find the right files. By default, the web interface is not accessible over the public interface, so you don't have to worry about others outside your organization seeing this information.

The next step is partitioning the disk drive. You can select Autopartition and let Rocks partition the disk using default values or you can manually partition the disk using Disk Druid. The current defaults are 6 GB for `/` and 1 GB for swap space. `/export` gets the remaining space. If you manually partition the drive, you need at least 6 GB for `/` and you must have a `/export` partition.

The next few screens are used to configure the network. Rocks begins with the private interface. You can choose to have DHCP configure this interface, but since this is on the internal network, it isn't likely that you want to do this. For the internal network, use a private address range that doesn't conflict with the external address range. For example, if your campus LAN uses `10.x.x.x`, you might use `172.16.1.x` for your internal network. When setting up clients, Rocks numbers machines from the highest number downward, e.g., `172.16.1.254, 172.16.1.253, ...`

For the public interface, you can manually enter an IP address and mask or you can rely on DHCP. If you are manually entering the information, you'll be prompted for a routing gateway and DNS servers. If you are using DHCP, you shouldn't be asked for this information.

The last network setup screen asks for a node name. While it is possible to retrieve this information by DHCP, it is better to set it manually. Otherwise, you'll need to edit `/etc/resolv.conf` after the installation to add the frontend to the name resolution path. Choose the frontend name carefully. It will be written to a number of files, so it is very difficult to

change. It is a very bad idea to try to change hostnames after installing Rocks.

Once you have the network parameters set, you'll be prompted for a root password. Then Rocks will format the filesystem and begin installing the packages. As the installation proceeds, Rocks provides a status report showing each package as it is installed, time used, time remaining, etc. This step will take a while.

Once the Rocks Base CD has been installed, you'll be prompted for each of the roll CDs once again. Just swap CDs when prompted to do so. When the last roll CD has been installed, the frontend will reboot.

Your frontend is now installed. You can move onto the compute nodes or you can stop and poke around on the frontend first. The first time you log onto the frontend, you will be prompted for a file and passphrase for SSH.

```
Rocks Frontend Node - Wofford Rocks Cluster
```

```
Rocks 3.2.0 (Shasta)
```

```
Profile built 17:10 29-Jul-2004
```

```
Kickstarted 17:12 29-Jul-2004
```

```
It doesn't appear that you have set up your ssh key.
```

```
This process will make the files:
```

```
  /root/.ssh/identity.pub
```

```
  /root/.ssh/identity
```

```
  /root/.ssh/authorized_keys
```

```
Generating public/private rsa1 key pair.
```

```
Enter file in which to save the key (/root/.ssh/identity):
```

```
Enter passphrase (empty for no passphrase):
```

```
Enter same passphrase again:
```

```
Your identification has been saved in /root/.ssh/identity.
```

```
Your public key has been saved in /root/.ssh/identity.pub.
```

```
The key fingerprint is:
```

```
86:ad:c4:e3:a4:3a:90:bd:7f:f1:bd:7a:df:f7:a0:1c root@frontend.public
```

The default file name is reasonable, but you really should enter a passphrase—one you can remember.

7.1.4 Install Compute Nodes

The next step is to install the compute nodes. Before you do this, you may want to make a few changes to the defaults. For example, you might want to change how the disks will be partitioned, what packages will be installed, or even which kernel will be used. For now, we'll stick with the defaults. Customizations are described in the next two sections, so you may want to read ahead before going on. But it's really easy to reinstall the compute nodes, so don't feel you have to master everything at once.

To install the compute nodes, you'll begin by running the program *insert-ethers* as root on the frontend. Next, you'll boot a compute node using the Rocks Base CD. Since the Rocks Base CD defaults to compute node install, you won't need to type anything on the cluster node. The *insert-ethers* program listens for a DHCP query from the booting compute node, assigns it a name and IP address, records information in its database, and begins the installation of the client.

Let's look at the process in a little more detail. *insert-ethers* collects MAC address information and enters it into the Rocks cluster database. It can also be used to replace (`--replace`), update (`--update`), and remove (`--remove`) information in the database. This information is used to generate the DHCP configuration file and the host file.

There is one potential problem you might face when using *insert-ethers*. If you have a managed Ethernet switch, when booted it will issue a DHCP request. You don't want to treat it like a compute node. Fortunately, the Rocks implementers foresaw this problem. When you start *insert-ethers*, you are given a choice of the type of appliance to install. You can select *Ethernet Switch* as an option and configure your switch. When you are done, quit and restart *insert-ethers*. This time select *Compute*. Now you are ready to boot your compute nodes. If you aren't setting up an Ethernet switch, you can just select *Compute* the first time you run *insert-ethers*.

The next step is to boot your compute nodes. As previously noted, you can use the Rocks Base CD to do this. If your compute nodes don't have CD-ROM drives, you have two other options. You can use a network boot if your network adapters support a PXE boot, or you can create a PXE boot floppy. Consult your hardware documentation to determine how to do a PXE boot using a network adapter. The Rocks FAQ, included in *NPSCI Rocks Cluster Distribution: Users Guide*, has the details for creating a PXE boot floppy.

When *insert-ethers* runs, it displays a window labeled Inserted Appliances. As each compute node is booted, it displays the node's MAC address and assigned name. Typically, *insert-ethers* will name the systems *compute-0-0*, *compute-0-1*, etc. (The file */etc/host* defines aliases for these, *c0-0*, *c0-1*, etc., for those of us who don't type well.) If you start *insert-ethers* with the command-line option `--cabinet=1`, it will generate the names *compute-1-0*, *compute-1-1*, etc. This allows you to create a two-tier naming system, if you want. You can change the starting point for the second number with the `--rank`. See the *insert-ethers(8)* manpage for more details.

A couple of minutes after you reboot your compute node, it will eject the CD-ROM. You can take the CD-ROM and move on to your next machine. If you have a terminal connected to the system, you'll get a status report as the installation proceeds.

If you need to reinstall a node, you can use the *shoot-node* command. This is useful when changing the configuration of a node, e.g., adding a new package. This command takes the name of the machine or machines as an argument.

```
[root@frontend root]# shoot-node compute-0-0
```

Since this is run on the frontend, it can be used to remotely reinstall a system. This command is described in the *shoot-node(8)* manpage.

7.1.5 Customizing the Frontend

Since Rocks installs Linux for you, you will need to do a little digging to see how things are set up. Among other services, Rocks installs and configures 411 (an NIS replacement), Apache, DHCP, MySQL, NFS, NTP, Postfix, and SSH, as well as cluster-specific software such as Ganglia and PVFS. Configuration files are generally where you would expect them. You'll probably want to browse the files in */etc/*, */etc/init.d/*, */etc/ssh/*, and */etc/xinetd.d/*. Other likely files include *crontab*, *dhcpd.conf*, *exports*, *fstab*, *gmetad.conf*, *gmond.conf*, *hosts*, *ntp.conf*, and *ntp/step-tickers*. You might also run the commands

```
[root@frontend etc]# ps -aux | more
```

...

```
[root@frontend etc]# /sbin/service --status-all | more
```

...

```
[root@frontend etc]# netstat -a | more
```

...

The cluster software that Rocks installs is in */opt* or */usr/share*.

If you have been using Red Hat for a while, you probably have some favorite packages that Rocks may not have installed. Probably the best way to learn what you have is to just poke around and try things.

7.1.5.1 User management with 411

Starting with Rocks 3.1.0, 411 now replaces NIS. 411 automatically synchronizes the files listed in */var/411/Files.mk*. The password and group files are among these. When you add users, you'll want to use *useradd*.

```
[root@frontend 411]# useradd -p xyzzy -c "Joe Sloan" \
```

```
>-d /export/home/sloanjd sloanjd
```

...

This automatically invokes 411. When a user changes a password, you'll need to sync the changes with the compute nodes. You can do this with the command

```
[root@frontend root]# make -C /var/411
```

A more complete discussion of 411 can be found in the Rocks user's guide. At this time, there isn't a 411 man page. To remove users, use *userdel*.

7.1.5.2 X Window System

You'll probably want to start the X Window System so you can run useful graphical tools such as Ganglia. Before you can run X the first time, you'll need to run *redhat-config-xfree86*. If you are comfortable setting options, go for it. If you are new to the X Window System, you'll probably be OK just accepting the defaults. You can then start X with the *xstart* command. (If

you get a warning message about no screen savers, just ignore it.)

Once X is working, you'll need to do the usual local customizations such as setting up printers, creating a message of the day, etc.

7.1.6 Customizing Compute Nodes

Rocks uses Kickstart and Anaconda to install the individual compute nodes. However, rather than use the usual flat, text-based configuration file for Kickstart, Rocks decomposes the Kickstart file into a set of XML files for the configuration information. The Kickstart configuration is generated dynamically from these. These files are located in the `/export/home/install/rocks-dist/enterprise/3/en/os/i386/build/nodes/` directory. Don't change these. If you need to create customization files, you can put them in the directory `/home/install/site-profiles/3.2.0/nodes/` for Rocks Version 3.2.0. There is a sample file `skeleton.xml` that you can use as a template when creating new configuration files. When you make these changes, you'll need to apply the configuration change to the distribution using the `rocks-dist` command. The following subsections give examples. (For more information on `rocks-dist`, see the `rocks-dist(1)` manpage.)

7.1.6.1 Adding packages

If you want to install additional RPM packages, first copy those packages to the directory `/home/install/contrib/enterprise/3/public/arch/RPMS`, where `arch` is the architecture you are using, e.g., `i386`.

```
[root@frontend root]# mv ethereal-0.9.8-6.i386.rpm \  
>/home/install/contrib/enterprise/3/public/i386/RPMS/  
  
[root@frontend root]# mv ethereal-gnome-0.9.8-6.i386.rpm \  
>/home/install/contrib/enterprise/3/public/i386/RPMS/
```

Next, create a configuration file `extend-compute.xml`. Change to the profile directory, copy `skeleton.xml`, and edit it with your favorite text editor such as `vi`.

```
[root@frontend root]# cd /home/install/site-profiles/3.2.0/nodes  
  
[root@frontend nodes]# cp skeleton.xml extend-compute.xml  
  
[root@frontend nodes]# vi extend-compute.xml  
  
...
```

Next, add a line to `extend-compute.xml` for each package.

```
<package> ethereal </package>  
  
<package> ethereal-gnome </package>
```

Notice that only the base name for a package is used; omit the version number and `.rpm` suffix.

Finally, apply the configuration change to the distribution.

```
[root@frontend nodes]# cd /home/install
[root@frontend install]# rocks-dist dist
...
```

You can now install the compute nodes and the desired packages will be included.

7.1.6.2 Changing disk partitions

In general, it is probably a good idea to stick to one disk-partitioning scheme. Unless you turn the feature off as described in the next subsection, compute nodes will automatically be reinstalled after a power outage. If you are using multiple partitioning schemes, the automatic reinstallation could result in some drives with undesirable partitioning. Of course, the downside of a single-partitioning scheme is that it may limit the diversity of hardware you can use.

To change the default disk partitioning scheme used by Rocks to install compute nodes, first create a replacement partition configuration file. Begin by changing to the directory where the site profiles are stored. Create a configuration file *replace-auto-partition.xml*. Change to the profile directory, copy *skeleton.xml*, and edit it.

```
[root@frontend root]# cd /home/install/site-profiles/3.2.0/nodes
[root@frontend nodes]# cp skeleton.xml replace-auto-partition.xml
[root@frontend nodes]# vi replace-auto-partition.xml
...
```

Under the main section, you'll add something like the following:

```
<main>
    <part> / --size 2048 --ondisk hda </part>
    <part> swap --size 500 --ondisk hda </part>
    <part> /mydata --size 1 --grow --ondisk hda </part>
</main>
```

Apart from the XML tags, this is standard Kickstart syntax. This example, a partitioning scheme for an older machine, uses 2 GB for the root partition, 500 MB for a swap partition, and the rest of the disk for the */mydata* partition.

The last step is to apply the configuration change to the distribution.

```
[root@frontend nodes]# cd /home/install
[root@frontend install]# rocks-dist dist
...
```

You can now install the system using the new partitioning scheme.

7.1.6.3 Other changes

By default, a compute node will attempt to reinstall itself whenever it does a hard restart, e.g., after a power failure. You can disable this behavior by executing the next two commands.

```
[root@frontend root]# cluster-fork '/etc/rc.d/init.d/rocks-grub stop'
```

```
compute-0-0:
```

```
Rocks GRUB: Setting boot action to 'boot current kernel': [ OK ]
```

```
...
```

```
[root@frontend root]# cluster-fork '/sbin/chkconfig --del rocks-grub'
```

```
compute-0-0:
```

```
...
```

The command *cluster-fork* is used to execute a command on every machine in the cluster. In this example, the two commands enclosed in quotes will be executed on each compute node. Of course, if you really wanted to, you could log onto each, one at a time, and execute those commands. *cluster-fork* is a convenient tool to have around. Additional information can be found in the Rocks user's guide. There is no manpage at this time.

Creating and installing custom kernels on the compute nodes, although more involved, is nonetheless straightforward under Rocks. You'll first need to create a compute node, build a new kernel on the compute node, package it using *rpm*, copy it to the frontend, rebuild the Rocks distribution with *rocks-dist*, and reinstall the compute nodes. The details are provided in the Rocks user's guide along with descriptions of other customizations you might want to consider.

7.2 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
```

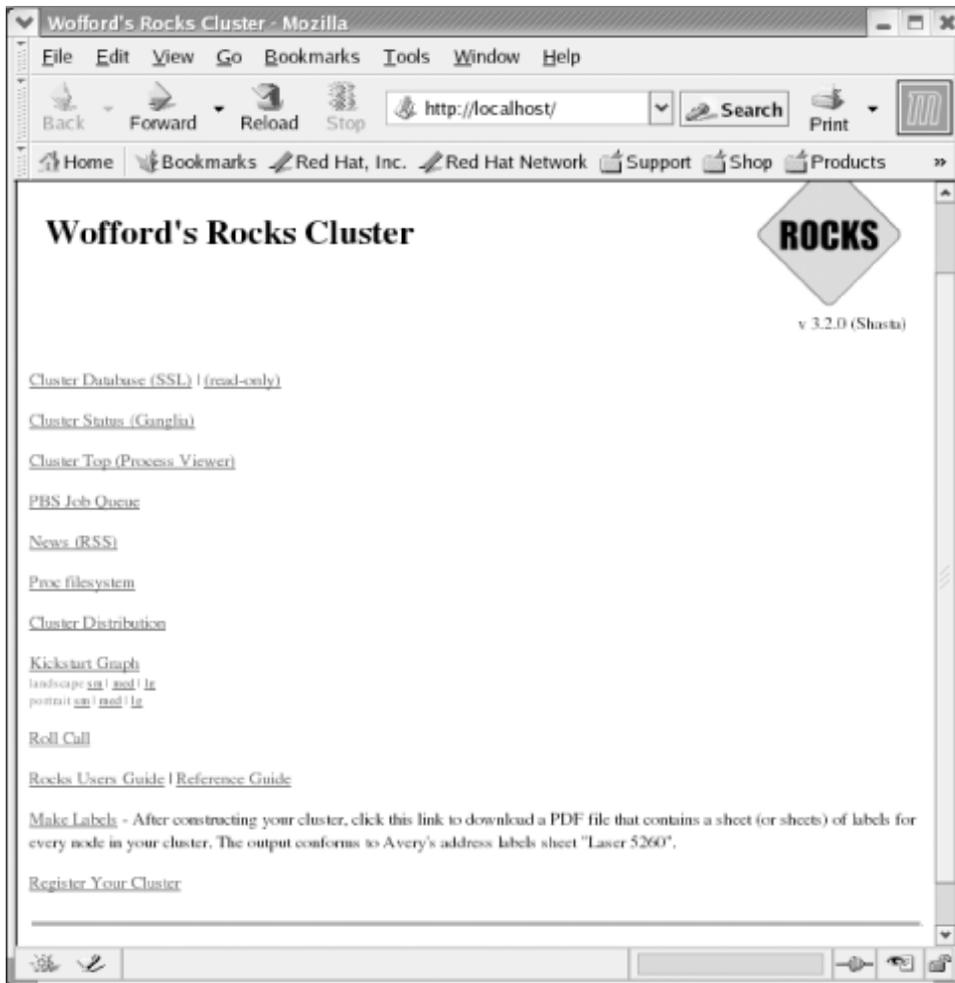
Then restart the *iptables* service.

```
[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.

Figure 7-2. Rocks database page



Cluster Status (Ganglia)

This link provides a way into Ganglia's home page. Ganglia, a cluster monitoring package, is described in [Chapter 10](#).

Cluster Top (Process Viewer)

This link takes you to a page that displays the top processes running on the cluster. This is basically the Unix *top* command, but provides cluster-wide information. The columns are similar to those provided by *top* except for the first two. The first, TN, gives the age of the information in seconds, and the second, HOST, is the host name for the cluster node that the process is running on. You can look at the *top(1)* manpage for information on how to interpret this page. [Figure 7-3](#) shows the Cluster Top screen for an idle cluster.

PBS Job Queue

PBS is described in [Chapter 11](#). You should see the PBS link only if you've installed the PBS roll.

News (RSS)

This is an alert system that sends RSS-style news items for events within the cluster. It is documented in the Rocks Reference Guide.

Figure 7-3. Cluster Top

Wofford's Rocks Cluster Cluster Top

Fri, 30 Jul 2004 16:40:14 +0000 Physical Job Assignments

Show only processes by user: Go

TN	HOST	PID	USER	CMD	%CPU	%MEM	SIZE	DATA	SHARED	VM
112	frontend.local	2316	root	Xvnc	1.96	5.08	1104	6572	3052	9668
55	compute-0-1.local	3727	root	greceptor	1.31	2.59	516	1632	1628	4328
14	compute-0-0.local	3730	root	greceptor	1.31	2.44	440	1600	1464	4164
118	frontend.local	1572	root	greceptor	0.66	1.14	388	1032	1124	3656

Proc filesystem

This link takes you into the `/proc` subdirectory. The files in this subdirectory contain dynamic information about the state of the operating system. You can examine files to see the current configuration, and, in some cases, change the file to alter the configuration. This page is accessible only on a local system.

Cluster Distribution

The Cluster Distribution link is a link into the `/home/install` directory on the frontend. This directory holds the RPM packages used to construct the cluster. This page is accessible only on a local system.

Kickstart Graph

This link provides a graphical representation of the information used to create the Kickstart file. This is generated on the fly. Different display sizes are available.

Roll Call

This link returns a page that lists the various rolls that have been installed on your cluster.

Rocks User's Guide/Reference Guide

These are online versions of the Rocks documentation that have been alluded to so often in this chapter.

Make Labels

This link generates a PDF document containing labels for each node in the cluster. The labels contain the cluster name, node name, MAC address, and the Rocks logo. If your cluster name is too long, the logo will obscure it. You should be able to print the document on a standard sheet of labels such as Avery 5260 stock.

Register Your Cluster

This will take you to the Rocks registration site, so you can add your cluster to the list of other Rocks clusters.

Finally, there is a link to the Rocks home page.



7.3 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 `-machinefilefilename` 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]$ /opt/mpich/gnu/bin/mpirun -nolocal \  
>-np 2 /opt/hpl/gnu/bin/xhpl  
...
```

(Add a machine file if you like.) You can find more details in the Rocks user manual.

Part III: Building Custom Clusters

This section describes individual components and software that you should consider when building your cluster. Most of these components are part of the OSCAR and Rocks distribution but can be installed independently. Thus, by using the material in this section, you could bypass OSCAR or Rocks and build a custom cluster. Or you could use these chapters to learn more about the software that is part of OSCAR and Rocks.

These chapters are largely independent and can be read in any order with one minor exception. [Chapter 12](#) uses the C3 tools introduced in [Chapter 10](#). However, you should have little trouble understanding [Chapter 12](#) even if you haven't read [Chapter 10](#). In practice, you will probably want to install the software described in Chapters 9 through 12 and then return to [Chapter 8](#) to clone your systems. Since the other chapters describe manually installing software, you might want to glance over [Chapter 8](#) before you begin those chapters so you'll know what can be automated.

Chapter 8. Cloning Systems

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.

8.1 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.

8.1.1 Distributing Files

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.

Table 8-1.

Types of Files
Accounting files, e.g., <i>/etc/passwd, /etc/shadow, /etc/group, /etc/gshadow</i>
Configuration files, e.g., <i>/etc/motd, /etc/fstab, /etc/hosts, /etc/printcap.local</i>
Security configuration files such as firewall rulesets or public keys
Packages for software you wish to install
Configuration files for installed software
User scripts
Kernal images and kernal source files

Many of these are one-time copies, but others, like the accounting files, will need to be updated frequently.

You have a lot of options. Some approaches work best when moving sets of files but can be tedious when dealing with just one or two files. If you are dealing with a number of files, you'll

need some form of repository. (While you could pack a collection of files into a single file using tar, this approach works well only if the files aren't changing.) You could easily set up your own HTTP or FTP server for both packages and customized configuration files, or you could put them on a floppy or CD and carry the disk to each machine. If you are putting together a repository of files, perhaps the best approach is to use NFS.

With NFS, you won't need to copy anything. But while this works nicely with user files, it can create problems with system files. For example, you may not want to mount a single copy of */etc* using NFS since, depending on your flavor of Linux, there may be files in the */etc* that are unique to each machine, e.g., */etc/HOSTNAME*. The basic problem with NFS is that the granularity (a directory) is too coarse. Nonetheless, NFS can be used as a first step in distributing files. For example, you might set up a shared directory with all the distribution RPMs along with any other software you want to add. You can then mount this directory on the individual machines. Once mounted, you can easily copy files where you need them or install them from that directory. For packages, this can easily be done with a shell script.

While any of these approaches will work and are viable approaches on an occasional basis, they are a little clunky, particularly if you need to move only a file or two. Fortunately, there are also a number of commands designed specifically to move individual files between machines. If you have enabled the *r*-service commands, you could use *rcp*. A much better choice is *scp*, the SSH equivalent. You could also consider *rdist*. Debian users should consider *apt-get.cpush*, one of the tools supplied in C3 and described in [Chapter 10](#), is another choice. One particularly useful command is *rsync*, which will be described next.

8.1.1.1 Pushing files with rsync

rsync is GNU software written by Andrew Tridgell and Paul Mackerras. *rsync* is sometimes described as a faster, more flexible replacement for *rcp*, but it is really much more. *rsync* has several advantages. It can synchronize a set of files very quickly because it sends only the difference in the files over the link. It can also preserve file settings. Finally, since other tools described later in this book such as SystemImager and C3 use it, a quick review is worthwhile.

rsync is included in most Linux distributions. It is run as a client on the local machine and as a server on the remote machine. With most systems, before you can start the *rsync* daemon on the machine that will act as the server, you'll need to create both a configuration file and a password file.^[1]

[1] Strictly speaking, the daemon is unnecessary if you have SSH or RSH.

A configuration file is composed of optional global commands followed by one or more module sections. Each module or section begins with a module name and continues until the next module is defined. A module name associates a symbolic name to a directory. Modules are composed of parameter assignments in the form *option = value*. An example should help clarify this.

```
# a sample rsync configuration file -- /etc/rsyncd.conf
#
[systemfiles]
# source/destination directory for files
path = /etc
# authentication -- users, hosts, and password file
auth users = root, sloanjd
```

```

hosts allow = amy basil clara desmond ernest fanny george hector james

secrets file = /etc/rsyncd.secrets

# allow read/write

read only = false

# UID and GID for transfer

uid = root

gid = root

```

There are no global commands in this example, only the single module *[systemfiles]*. The name is an arbitrary string (hopefully not too arbitrary) enclosed in square brackets. For each module, you must specify a `path` option, which identifies the target directory on the server accessed through the module.

The default is for files to be accessible to all users without a password, i.e., *anonymous rsync*. This is not what we want, so we use the next three commands to limit access. The `auth user` option specifies a list of users that can access a module, effectively denying access to all other users. The `hosts allow` option limits the machines that can use this module. If omitted, then all machines will have access. In place of a list of machines, an address/mask pattern can be used. The `secrets file` specifies the name of a password file used for authentication. The file is used only if the `auth user` option is also used. The format of the secrets file is *user:password*, one entry per line. Here is an example:

```
root:RSpw012...
```

The secrets file should be readable only by root, and should not be writable or executable. *rsync* will balk otherwise.

By default, files are read only; i.e., files can be downloaded from the server but not uploaded to the server. Set the `read only` option to `false` if you want to allow writing, i.e., uploading files from clients to the server. Finally, the `uid` and `gid` options set the user and group identities for the transfer. The configuration file is described in detail in the manpage *rsyncd.conf(5)*. As you might imagine, there are a number of other options not described here.

rsync usually uses *rsh* or *ssh* for communications (although it is technically possible to bypass these). Consequently, you'll need to have a working version of *rsh* or *ssh* on your system before using *rsync*.

To move files between machines, you will issue an *rsync* command on a local machine, which will contact an *rsync* daemon on a remote machine. Thus, to move files *rsync* must be installed on each client and the remote server must be running the *rsync* daemon. The *rsync* daemon is typically run by *xinetd* but can be run as a separate process if it is started using the `--daemon` option. To start *rsync* from *xinetd*, you need to edit the file */etc/xinetd.d/rsync*, change the line `disable = yes` to `disable = no`, and reinitialize or restart *xinetd*. You can confirm it is listening by using *netstat*.

```

[root@fanny xinetd.d]# netstat -a | grep rsync

tcp        0      0  *:rsync          ::*              LISTEN

```

rsync uses TCP port 873 by default.

rsync can be used in a number of different ways. Here are a couple of examples to get you started. In this example, the file *passwd* is copied from *fanny* to *george* while preserving the group, owner, permissions, and time settings for the file.

```
[root@fanny etc]# rsync -gopt passwd george::systemfiles
```

Password:

Recalls `systemfiles` is the module name in the configuration file. Note that the system prompts for the password that is stored in the `/etc/rsyncd.secrets` file on *george*. You can avoid this step (useful in scripts) with the `--password-file` option. This is shown in the next example when copying the file *shadow*.

```
[root@fanny etc]# rsync -gopt --password-file=rsyncd.secrets shadow /
```

george::systemfiles

If you have the *rsync* daemon running on each node in your cluster, you could easily write a script that would push the current accounting files to each node. Just be sure you get the security right.

In the preceding examples, *rsync* was used to push files. It can also be used to pull files. (*fanny* has the same configuration files as *george*.)

```
[root@george etc]# rsync -gopt fanny::systemfiles/shadow /etc/shadow
```

Notice that the source file is actually `/etc/shadow` but the `/etc` is implicit because it is specified in the configuration file.

rsync is a versatile tool. It is even possible to clone running systems with *rsync*. Other command forms are described in the manpage *rsync(1)*.

8.2 Automating Installations

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.

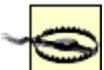
If you are only doing a few machines, you can do the installations manually from the checklist if you are very careful. But this can be an error-prone activity, so even small clusters can benefit from automated installs. If you are building a large cluster, you'll definitely need some tools. There are 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 its 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.

8.2.1 Kickstart

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 *DrakX* for Mandrake Linux and *Fully Automatic Installation (FAI)* for Debian. A Kickstart installation can be done using a local CD-ROM or hard drive, or over a network using FTP, NFS, or HTTP. We'll look at using a local CD-ROM and using NFS over a network. NFS is preferable when working with a large number of machines.



Warning! With any network-based approach, if you have problems, the first thing to check is your firewall setting for your servers!

Anaconda is the Red Hat installation program. It is written in Python with some custom modules in C. Anaconda is organized in stages. The first stage is an installer which loads kernel modules needed later. It is this loader that goes to the appropriate installation source. Finally, Anaconda has an auto-install mechanism, Kickstart, that allows installs to be scripted via the Kickstart configuration file.

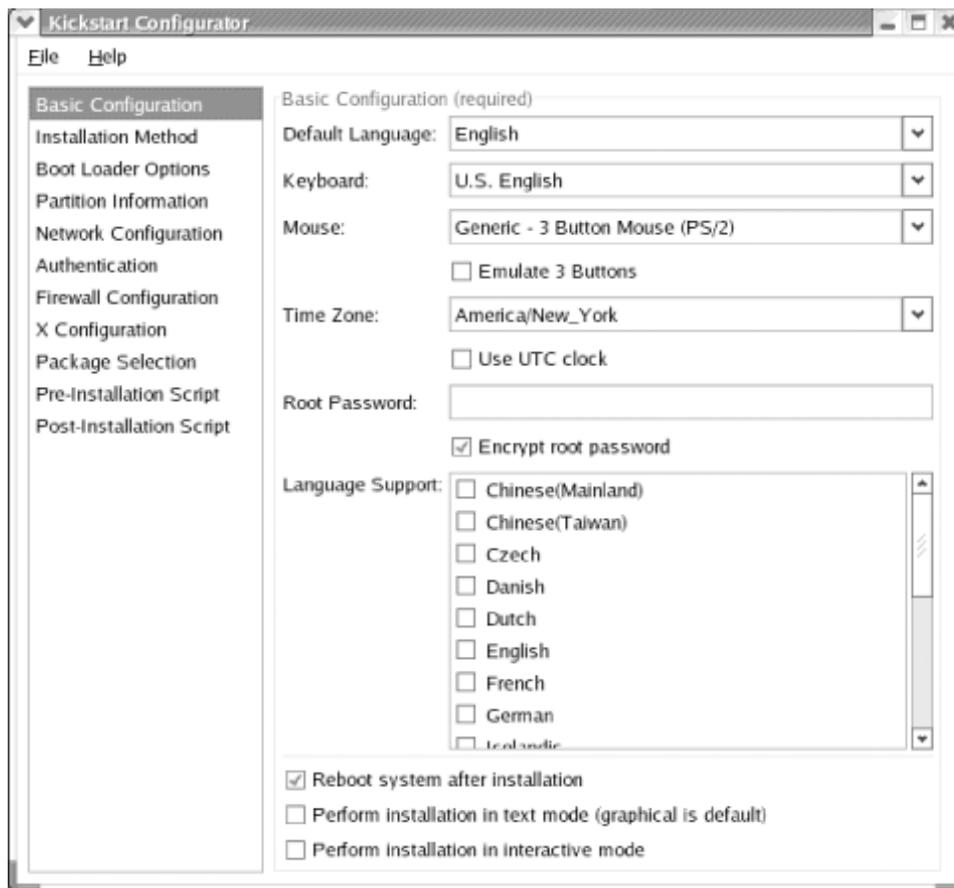
8.2.1.1 Configuration file

The first step in using Kickstart is to create a Kickstart configuration file. Once you have the

configuration file, you'll create a boot disk and start the installation. You have two options in creating a configuration file—you can edit an existing configuration file or you can use Red Hat's *Kickstart Configurator* program to create a new file. While the configuration program has a nice GUI and is easy to use, older versions don't give you the option of reopening an existing configuration file. So with the older version, you'll need to get everything right the first time, start over from scratch, or manually edit the file that it creates after the fact.

Using *KickstartConfigurator* is straightforward. Since it provides a GUI, you'll need to be running the X Window System. You can start it from a console window with the command `/usr/sbin/ksconfig` or, if you are using *gnome*, from *Main Menu Button Programs System Kickstart Configurator*. [Figure 8-1](#) shows the initial window.

Figure 8-1. Kickstart Configurator



Simply work your way down the lists on the left setting the fields on the right as needed. Most of what you'll see will be familiar questions from a normal installation, although perhaps in slightly more detail. On the second screen, *Installation Method*, you'll be asked for the installation method—CD-ROM, FTP, etc. The last two screens ask for pre-installation and post-installation scripts, allowing you to add additional tasks to the install. When you are done, save the file.

Alternatively, you could use an existing configuration file. The Red Hat installation program creates a Kickstart file for the options you select when you do an installation. This is saved as `/root/anaconda-ks.cfg`. (There is also a template for a configuration file on the Red Hat documentation disk called *sample.ks*, but it is a bit sparse.) If you have already done a test installation, you may have something very close to what you need, although you may want to tweak it a bit.

Once you have a configuration file, you may need to make a few changes. Often, manually editing an existing configuration file is the easiest way to get exactly what you want. Since the configuration is a simple text file, this is a very straightforward process. The configuration file is divided into four sections that must be in the order they are described here.

The command section comes first and contains basic system information such as keyboard and mouse information, the disk partition, etc. Here is part of a command section with comments explaining each command:

```
# Kickstart file

# Do a clean install rather than an upgrade (optional).

install

# Install from a CD-ROM; could also be nfs, hard drive, or
# a URL for FTP or HTTP (required).

cdrom

# language used during installation (required)

lang en_US

# languages to install on system (required)

langsupport --default en_US.iso885915 en_US.iso885915

# type of keyboard (required)

keyboard us

# type of mouse (required)

mouse genericps/2 --device psaux -emulthree

# X configuration (optional)

xconfig --card "Matrox Millennium G200" --videoram 8192 --hsync 30.0-60.0
--vsync 47.5-125.0 --resolution 1024x768 --depth 16 --startxonboot

# network setup (optional)

network --device eth0 --bootproto dhcp

# root password (required)

rootpw --iscrypted $1$İZ5ÜİİÜÑ$Ulh7W6TkpQ303eTHtk4wG1

# firewall setup (optional)

firewall --medium --dhcp --port ssh:tcp

# system authentication (required)

authconfig --enablesshadow --enablemd5
```

```

# timezone (required)

timezone --utc America/New_York

# bootloader (required)

bootloader --md5pass=$1$Åq9erÖβE$HoYKj.adlPZyv4mGtc62W.

# remove old partitions from disk (optional)

clearpart --all --drives=hda

#partition information (required)

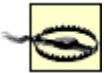
part /boot --fstype ext3 --size=50 --ondisk=hda

part / --fstype ext3 --size=1100 --grow --ondisk=hda

part swap --size=256 --ondisk=hda

```

Other options and details can be found in the first chapter of *The Official Red Hat Linux Customization Guide* on the Red Hat documentation disk.



If you omit any of the required commands, the install will pause and prompt you for that information, which is not what you want for an automatic installation.

The second part of the configuration file lists the packages that will be installed. This section begins with the line `%packages`. Here is a part of a sample listing for this section:

```

%packages

@ Printing Support

@ Classic X Window System

@ X Window System

@ GNOME

@ Sound and Multimedia Support

@ Network Support

@ Software Development

@ Workstation Common

...

balsa

gnumeric-devel

esound-devel

ImageMagick-c++-devel

```

```
mozilla-chat
```

```
...
```

Often you need to list only a component, not the individual packages. In this example, the lines starting with @ are all components. The remaining lines are all individual packages.

The last two sections, the pre-install and post-install configuration sections, are optional. These are commands that are run immediately before and immediately after installation. Here is an example that adds a user:

```
%post
```

```
/usr/sbin/useradd sloanjd
```

```
chfn -f 'Joe Sloan' sloanjd
```

```
/usr/sbin/usermod -p '$1$IgùYUDi$oyWJSirX8I0XElXVGXesG2.' Sloanjd
```

Note that a pre-install section is not run in a *chroot* environment, while a post-install section is.^[2] Basically, these sections provide a primitive way of doing custom configurations. This can be useful for small changes but is awkward for complex tasks. For more details about the configuration file, see the Red Hat documentation.

^[2] A *chroot* environment restricts access to the part of the filesystem you are working in, denying access to the remainder of the filesystem.

8.2.1.2 Using Kickstart

Once you have the Kickstart file, you need to place the file where it will be available to the system you are configuring. This can be done in several ways depending on how you will boot the system. For a CD-ROM installation, you could simply copy the file over to a floppy.

```
[root@amy root]# mount /mnt/floppy
```

```
[root@amy root]# cp ks.cfg /mnt/floppy/ks.cfg
```

```
[root@amy root]# umount /mnt/floppy
```

Reboot your system from an installation CD-ROM. (If your system won't boot from a CD-ROM, you could create a floppy boot disk and copy the configuration file onto it.) With this approach, you'll need to tell the system where to find the configuration file. At the boot prompt, enter the command

```
boot:linux ks=floppy
```

While you will be able to complete the installation without typing anything else, you will still need to swap CD-ROMs. This probably isn't what you had in mind, but it is a good, quick way to test your Kickstart file.

If you want to do a network installation, you can provide the installation files via FTP, NFS, or HTTP. You will need to set up the corresponding server, make the appropriate changes to the Kickstart configuration file and copy it to the server, and create a network boot disk. (A network or PXE boot is also an option.) If you want to do an unattended installation, you will also need a DHCP server to provide both the IP address and the location of the Kickstart

configuration file. Using a boot disk with an NFS server is probably the most common approach.

To set up a NFS server, you'll need to identify a machine with enough free space to hold all the installation CD-ROMs, copy over the contents of the CD-ROMs, and configure the NFS server software. For example, to install Red Hat 9, you might begin by creating the directory `/export/9.0` and copying over the distribution files.

```
[root@fanny root]# mkdir -p /export/9.0

[root@fanny root]# mount /mnt/cdrom

[root@fanny root]# cp -arv /mnt/cdrom/RedHat /export/9.0

...

[root@fanny root]# eject cdrom
```

You'll repeat the last three steps for each CD-ROM.

To configure NFS, you'll need to install the NFS package if it is not already installed, edit `/etc/exports` so that the target can mount the directory with the files, e.g., `/export/9.0`, and start or restart NFS. For example, you might add something like the following lines to `/etc/exports`.

```
/export/9.0      george hector ida james
/kickstart      george hector ida james
```

This allows the four listed machines access to the installation directory and the directory holding the Kickstart configuration file. You'll start or restart NFS with either `/sbin/service nfs start` or `/sbin/service nfs restart`.

Since you are doing a network install, you'll need to replace the entry `CDROM` in `ks.cfg` with information about the NFS server such as

```
nfs --server 10.0.32.144 --dir /export/9.0

network --device eth0 --bootproto dhcp
```

The second line says to use DHCP, which is the default if this information isn't provided. While not always necessary, it may be safer in some circumstances to use IP addresses rather than host names.

If you aren't using PXE, you'll need a network boot disk. It is tempting to think that, since we have specified an NFS install in the Kickstart file, any boot disk should work. Not so! Put a blank floppy in your floppy drive, mount the first distribution CD-ROM, change to the `images` subdirectory, and then use the following command:

```
[root@amy images]# dd if=bootnet.img of=/dev/fd0 bs=1440k

...
```

If you don't need to do an unattended installation, the simplest approach is to copy the configuration file to the boot floppy and tell the boot loader where to find the file, just as you did with the CD-ROM installation. If you want to do an unattended installation, things are a little more complicated.

For an unattended installation, you will need to copy the Kickstart configuration file onto your NFS server and edit the boot disk configuration file. While you can place the file in the installation directory of your NFS server, a more general approach is to create a separate directory for Kickstart configuration files such as */kickstart*. You'll need to export this directory via NFS as shown earlier. If you only need one configuration file, *ks.cfg* is the usual choice. However, if you create multiple Kickstart configuration files, you can use a convention supported by Kickstart. Name each machine using the format *IP-number-kickstart* where *IP-number* is replaced by the IP address of the target node such as *10.0.32.146-kickstart*. This allows you to maintain a different configuration file for each machine in your cluster.

To access the file, you need to tell the client where to find the configuration file. For testing, you can do this manually at the boot loader. For example, you might enter something like

```
boot:linux ks=nfs:10.0.32.144:/kickstart/
```

This tells the loader to use the NFS server 10.0.32.144 and look in the */kickstart* directory. It will look for a file using the name format *IP-number-kickstart*. Alternatively, you could give a complete file name.

For an unattended installation, you will need to edit *syslinux.cfg* on the boot disk, changing the line

```
default
```

to something like

```
default linux ks=nfs:10.0.32.144:/kickstart/
```

You might also shorten the timeout. Once done, you just insert the floppy and power up the node. The remainder of the installation will take place over your network.

While *Kickstart* does what it was designed to do quite well, there are some severe limitations to what it can do. As a package-based installation, there is no easy way to deal with needs that aren't packaged-based. For example, if you recompile your kernel, modify configuration files, or install non-package software, you'll need to do some kind of scripting to deal with these special cases. That may be OK for one or two changes, but it can become tedious very quickly. If you need to make a number of customizations, you may be better served with an image-based tool like *g4u* or *SystemImager*.

8.2.2 g4u

Image copying is useful in any context where you have a large number of identical machines. While we will be using it to clone machines in a high-performance cluster, it could also be used in setting up a web server farm, a corporate desktop environment, or a computer laboratory. With image copying, you begin by building a sample machine, installing all the software needed, and doing any desired customizations. Then you copy over an image of the disk to other machines, causing all the added software and customizations to get copied as well.

g4u is a simple disk image installer. It allows you to copy the image of a computer's disk to a server and then install that image on other machines in your cluster. The design philosophy for *g4u* is very simple. *g4u* is indifferent to what is on the disk—it just copies bits. It doesn't matter what version of Unix or what file system you use. It doesn't care if a disk sector is unused—it still gets copied. The image is compressed while on the server, but otherwise is an exact copy of the disk. If the image includes configuration files that are specific to the original machine, e.g., a static IP address or a host-name file, you will have to correct these after

installing the image. (You can avoid most problems of this sort if you use DHCP to configure your systems.) g4u works best when used with disks with the same size and geometry but, under limited circumstances, it may be finessed to work with other disks. Image copying is the simplest approach to learn and to use and is usable with almost any operating system.

There are three things you will need to do before using g4u. If you don't already have an FTP server, you will need to create one to store the images. You will need to download the g4u software. And, while not strictly required, you should prepare your source system for cloning. All of these are very straightforward.

To set up an FTP server, you'll need to install the software, edit the configuration files, and start the daemon. Several FTP server implementations are available. Select and install your favorite. The *vsftpd* (*Very Secure FTP*) package is a good choice for this purpose. You'll need to edit the appropriate configuration files, */etc/vsftpd/vsftpd.conf*, */etc/vsftpd.ftpusers*, and */etc/vsftpd.user_list*. Then start the service.

```
[root@fanny etc]# /etc/init.d/vsftpd start
```

```
Starting vsftpd for vsftpd: [ OK ]
```

(When you are through cloning systems, you may want to disable FTP until you need it again because it poses a security risk. Just replace **start** with **stop** in the above.) Consult the documentation with your distribution or the appropriate manpages.

The g4u software consists of a NetBSD boot disk with the image-copying software. While it is possible to download the sources, it is much simpler if you just download a disk image with the software. You can download either a floppy image or a CD-ROM ISO image in either zipped or uncompressed format from <http://www.feyrer.de/g4u/>. (The uncompressed ISO image is smaller than 1.5 MB so downloads go quickly.) Once you have downloaded the image, unzip it if it is compressed and create your disk. With a floppy, you can use a command similar to the following, adjusting the version number as needed:

```
[root@fanny root]# cat g4u-1.16.fs > /dev/fd0
```

(With Windows, you can use *rawrite.exe*, which can also be downloaded from the web site.) For a CD-ROM, use your favorite software.

Since g4u creates a disk image, it copies not only files but unused sectors as well. If there is a lot of garbage in the unused sectors on the disk, they will take up space in the compressed image, and creating that image will take longer. You can minimize this problem by writing zeros out to the unused sectors before you capture the image. (Long strings of zeros compress quickly and use very little space.) The g4u documentation recommends creating a file of zeros that grows until it fills all the free space on the system, and then deleting that file.

```
[root@ida root]# dd if=/dev/zero of=/0bits bs=20971520
```

```
dd: writing `'/0bits': No space left on device
```

```
113+0 records in
```

```
112+0 records out
```

```
[root@ida root]# rm /0bits
```

```
rm: remove `'/0bits'? y
```

Once the file is deleted, the unused sectors will still contain mostly zeros and should compress nicely. While you don't have to do this, it will significantly reduce storage needs and transfer

time.

To use g4u, you will need to capture the original disk and then copy it to the new machines. Begin by shutting down the source machine and then booting it with the g4u disk. As the system boots, you'll see some messages, including a list of commands, and then a command-line prompt. To capture and upload the disk, use the *uploaddisk* command. For example,

```
#uploaddisk sloanjd@fanny.wofford.int ida.g4u
```

The arguments to *uploaddisk* are the user's FTP server and the saved images. You'll see a few more messages and then the system will prompt you for the user's FTP password. As the disk image is captured and uploaded to the FTP server, the software will display dots on the screen. When the upload is complete, the software will display some statistics about the transfer.

To create new systems from the image, the process is almost the same. Boot the new system from the g4u disk and use the *slurpdisk* command, like so:

```
#slurpdisk sloanjd@fanny.wofford.int ida.g4u
```

You'll be prompted for a password again and see similar messages. However, the download tends to go much faster than the upload. When the user prompt returns, remove the g4u disk and reboot the system. Log in and make any needed configuration changes. That's really all there is to it!

8.2.3 SystemImager

SystemImager is a part of the *Systems Installation Suite (SIS)*, a set of tools for building an image for a cluster node and then copying it to other nodes. In many ways it is quite similar to g4u. However, there are several major differences in both the way it works and in the added functionality it provides. It is also a much more complicated tool how to learn to use. These differences will be apparent as you read through this section.

As with g4u, with SIS you will set up a single node as a model, install the operating system and any additional software you want, and configure the machine exactly the way you want it. Next, copy the image of this machine to a server, and then from the server to the remaining machines in the cluster.

SystemImager is also useful in maintaining clusters since it provides an easy way to synchronize files among machines. For example, if you have a security patch to install on all the machines in the cluster, you could install it on your model computer and then update the cluster. Since SIS uses *rsync*, this is very efficient. Only the files changed by the patch will be copied.

The Systems Installation Suite is made up of three tools, *SystemConfigurator*, *SystemImager*, and *SystemInstaller*. From a pragmatic perspective, SystemImager is the place to begin and, depending upon your needs, may be the only part of the suite you will need to master.

SystemInstaller is generally used to build a pre-installation image on the image server without having to first create a model system. For example, OSCAR uses SystemInstaller to do just this. But if you are happy building the model system, which is strongly recommended since it gives you an opportunity to test your configuration before you copy it, there is no reason to be in a hurry to learn the details of SystemInstaller.

SystemConfigurator allows you to do a post-installation configuration of your system. While it is a useful standalone tool, it is integrated into SystemImager so that its use is transparent to the user. So while you will need to install SystemConfigurator, you don't need to learn the details of SystemConfigurator to get started using SIS. Consequently, this section focuses on

SystemImager.

Since SystemImager uses client-server architecture, you will need to set up two machines initially before you can begin cloning systems. The *image server* manages the installation, holds the clone image, and usually provides other needed services such as DHCP. You will also need to set up the model node or *golden client*. Once you have created the golden client, its image is copied to the server and can then be installed on the remaining machines within the cluster.

The installation of SystemImager can be divided into four multistep phases—setting up the image server, setting up the golden client, transferring the image to the image server, and copying the image to the remaining nodes in the cluster. Each of these phases is described in turn. If you installed OSCAR, this setup has already been done for you. However, OSCAR users may want to skim this material to get a better idea of how OSCAR works and can be used.

8.2.3.1 Image server setup

In setting up the image server, you will need to select a server, install Linux and other system software as needed, install the SystemImager software on the server, and determine both how you will assign IP addresses to clients and how you will start the download.

You'll want to take care in selecting your server. Typically, the SystemImager server will also act as the head node for your cluster and will provide additional network services such as DHCP. While it is possible to distribute some of this functionality among several machines, this isn't usually done and won't be discussed here. If you already have a server, this is a likely choice provided it has enough space.

Unlike g4u, the images SystemImager creates are stored as uncompressed directory trees on the server. This has a number of advantages. First, it works nicely with *rsync*. And as a live filesystem, you can *chroot* to it and make changes or even install packages (if you are a brave soul). You'll only copy useful files, not unused sectors. While this approach has a number of advantages, even a single image can take up a lot of space. Heterogeneous clusters will require multiple images. Taken together, this implies you'll want a speedy machine with lots of disk space for your server.

Because of dependencies, you should install all of SIS even if you plan to use only SystemImager. You have a couple of choices as to how you do this. There is a Perl installation script that can be downloaded and run. It will take care of downloading and installing everything else you need. Of course, you'll need Internet access from your cluster for this to work. Alternatively, you can download DEB or RPM packages and install. Downloading these packages and burning them onto a CD-ROM is one approach to setting up an isolated cluster. This chapter describes the installation process using RPM packages.

Since SIS supports a wide variety of different Linux releases, you'll need to select the correct packages for your distribution, and you'll need a number of packages to install SystemImager. These can be downloaded from SourceForge. Go to <http://sisuite.sourceforge.net> and follow the links to SystemConfigurator, SystemImager, and SystemInstaller, as needed, to download the individual packages. If in doubt, you can read the release notes for details on many of the packages.

There may be additional dependencies that you'll also need to address. For a simple Red Hat install, you'll need to install the following packages, if they are not already on your system, in this order: *rsync*, *perl-AppConfig*, *perl-XML-Simple*, *systemconfigurator*, *systemimager-common*, *systemimager-server*, *perl-MLDBM*, and *systeminstaller*. You'll also need a boot package specific to your architecture. For example, you would use *systemimager-boot-i386-standard* for the Intel 386 family. *rsync* is usually already installed. Install these as you would install any RPM.

```
[root@fanny sysimager]# rpm -vih perl-AppConfig-1.52-4.noarch.rpm
```

```
Preparing... ##### [100%]
 1:perl-AppConfig ##### [100%]
```

Repeat the process with each package. There is also an X interface to SystemInstaller called *tk/sis*. If you want to install this, you will need to install *perl-DBI*, *perl-TK*, and *systeminstall-x77*. (If you have problems with circular dependencies, you might put the package names all on the same line and use `rpm -Uvh` to install them.)

The SIS installation will create a directory */etc/systemimager* containing the configuration files used by SystemImager. By default, SystemImager is not started. You can use the command `service systemimager start` to manually start it. SystemImager starts the *rsync* daemon using the configuration file in */etc/systemimager*, so if *rsync* is already running on your system, you'll need to turn it off first. As with any manual start, if you restart the system, you'll need to restart SystemImager. (With a recent release, the names of several services have changed. To ensure you are using the appropriate names, look in */etc/init.d* to see what is installed.)

There are a couple of other things you might want to set up on your server if you don't already have them. With SIS, there are four installation methods. You can boot the machine you are installing the image on from a floppy, from CD-ROM, from its hard drive, or over the network using a PXE-based network adapter. (The hard drive option is used for upgrading systems rather than for new installs.)

If you are going to do a network boot, you will need a TFTP server. SIS includes a command, *mkbootserver*, which will handle the configuration for you, but you must first install some packages—*tftp-server*, *tftp*, and *pxe*. Once these packages are installed, the script *mkbootserver* will take care of everything else. As needed, it will create the */tftpboot* directory, modify */etc/services*, modify */etc/inetd.conf* or */etc/xinetd.d/tftp*, verify that the TFTP server works, configure PXE creating */etc/pxe.conf*, verify the *pxe* daemon is running, verify the network interface is up, and pass control to the *mkdhcpserver* command to configure a DHCP server. Once *mkbootserver* has been run, your server should be appropriately configured for booting clients and installing images via PXE. Of course, you'll need a PXE-enabled network adapter in your client.

Even if you aren't booting via PXE, you will probably still want to use DHCP to assign IP addresses. This isn't absolutely necessary since you can create a configuration diskette for each machine with the appropriate information, but it is probably the easiest way to go. Using DHCP implies you'll need a DHCP server, i.e., both server software and a configuration file. Setting up the software is usually just a matter of installing the *dhcp* package.

```
[root@fanny root]# rpm -vih dhcp-3.0p11-23.i386.rpm
warning: dhcp-3.0p11-23.i386.rpm: V3 DSA signature: NOKEY, key ID db42a60e

Preparing... ##### [100%]
 1:dhcp ##### [100%]
```

To create a configuration file, typically */etc/dhcpd.conf*, use the *mkdhcpserver* script. You'll need to collect information about your network such as the IP address range, broadcast address, network mask, DNS servers, and the network gateway before you run this script. Here is an example of using *mkdhcpserver* for a simple network.

```
[root@fanny root]# mkdhcpserver
```

Welcome to the SystemImager "mkdhcpserver" command. This command will

prepare this computer to be a DHCP server by creating a dhcpd.conf file for use with your ISC DHCP server (v2 or v3).

If there is an existing file, it will be backed up with the .beforesystemimager extension.

Continue? (y/[n]): **y**

Type your response or hit <Enter> to accept [defaults]. If you don't have a response, such as no first or second DNS server, just hit <Enter> and none will be used.

What is your DHCP daemon major version number (2 or 3)? [2]: **2**

Use of uninitialized value in concatenation (.) or string at /usr/sbin/mkdhcpserver line 202, <STDIN> line 2.

What is the name of your DHCP daemon config file? []: **/etc/dhcpd.conf**

What is your domain name? [localdomain.domain]: **wofford.int**

What is your network number? [192.168.1.0]: **10.0.32.0**

What is your netmask? [255.255.255.0]: **255.255.248.0**

What is the starting IP address for your dhcp range? [192.168.1.1]: **10.0.32.145**

What is the ending IP address for your dhcp range? [192.168.1.100]: **10.0.32.146**

What is the IP address of your first DNS server? []: **10.0.80.3**

What is the IP address of your second DNS server? []: **10.0.80.2**

What is the IP address of your third DNS server? []:

What is the IP address of your default gateway? [192.168.1.254]: **10.0.32.2**

What is the IP address of your image server? [192.168.1.254]: **10.0.32.144**

What is the IP address of your boot server? []: **10.0.32.144**

What is the IP address of your log server? []:

Will your clients be installed over SSH? (y/[n]): **y**

What is the base URL to use for ssh installs? [http://10.0.32.144/
systemimager/boot/]:

What... is the air-speed velocity of an unladen swallow? []:

Wrong!!! (with a Monty Python(TM) accent...)

Press <Enter> to continue...

Ahh, but seriously folks...

Here are the values you have chosen:

#####

ISC DHCP daemon version:	2
DHCP daemon using fixed-address patch:	n
ISC DHCP daemon config file:	/etc/dhcpd.conf
DNS domain name:	wofford.int
Network number:	10.0.32.0
Netmask:	255.255.248.0
Starting IP address for your DHCP range:	10.0.32.145
Ending IP address for your DHCP range:	10.0.32.146
First DNS server:	10.0.80.3
Second DNS server:	10.0.80.2
Third DNS server:	
Default gateway:	10.0.32.2
Image server:	10.0.32.144
Boot server:	10.0.32.144
Log server:	
Log server port:	

SSH files download URL: <http://10.0.32.144/systemimager/boot/>

#####

Are you satisfied? (y/[n]): **y**

The dhcp server configuration file (/etc/dhcpd.conf) file has been created for you. Please verify it for accuracy.

If this file does not look satisfactory, you can run this command again to re-create it: "mkdhcpserver"

WARNING!: If you have multiple physical network interfaces, be sure to edit the init script that starts dhcpd to specify the interface that is connected to your DHCP clients. Here's an example:

Change "/usr/sbin/dhcpd" to "/usr/sbin/dhcpd eth1".

Depending on your distribution, you may be able to set this with the "INTERFACES" variable in either "/etc/default/dhcp" or in your dhcpd initialization script (usually "/etc/init.d/dhcpd").

Also, be sure to start or restart your dhcpd daemon. This can usually be done with a command like "/etc/init.d/dhcpd restart" or similar.

Would you like me to restart your DHCP server software now? (y/[n]): **y**

Shutting down dhcpd: [FAILED]

Starting dhcpd: [OK]

As you can see, the script is very friendly. There is also important information buried in the output, such as the warning about restarting the DHCP daemon. Be sure you read it carefully. If you already have a DHCP configuration file, it is backed up, usually as */etc/dhcpd.conf.beforessystemimager*. You may need to merge information from your old file into the newly created file.

As previously noted, you don't have to use DHCP. You can create a configuration disk with a file *local.cfg* for each machine with the information provided by DHCP. Here is an example.

```
HOSTNAME=hector  
  
DOMAINNAME=wofford.int  
  
DEVICE=eth0  
  
IPADDR=10.0.32.146  
  
NETMASK=255.255.248.0  
  
NETWORK=10.0.32.0  
  
BROADCAST=10.0.39.255  
  
GATEWAY=10.0.32.2  
  
IMAGESERVER=10.0.32.144  
  
IMAGENAME=ida.image
```

Regardless of how you are booting for your install, the software will look for a floppy with this file and use the information if provided. In this example, the client names that have been automatically generated are not being used, so it is necessary to rename the installation scripts on the image server. We'll come back to this.

8.2.3.2 Golden client setup

The golden client is a model for the other machines in your cluster. Setting up the golden client requires installing and configuring Linux, the SystemImager software, and any other software you want on each client. You will also need to run the *prepareclient* script to collect image information and start the *rsync* daemon for the image transfer.

Because you are using an image install, your image should contain everything you want on the cluster nodes, and should be compatible with the node's hardware. In setting up the client, think about how it will be used and what you will need. Doing as much of this as possible will save you work in the long run. For example, if you generate SSH keys prior to cloning systems, you won't have to worry about key distribution. However, getting the software right from the start isn't crucial. SystemImager includes a script to update clients, and since it uses *rsync*, updates go fairly quickly. Nonetheless, this is something of a nuisance, so you'll want to minimize updates as much as possible. If possible, set up your client and test it in the environment in which it will be used.

Getting the hardware right is more important. The hardware doesn't have to be identical on every node, but it needs to be close. For network and video adapters, you'll want the same chipset. Although disk sizes don't have to be identical, it is better to select for your golden client a machine with the smallest disk size in your cluster. And you can't mix IDE and SCSI systems. Having said all this, remember that you can have multiple images. So if you have a cluster with three different sets of hardware, you can create three images and do three sets of installs. [\[3\]](#)

^[3] To some extent, you can install an image configured for different hardware and use kudzu to make corrections once the system reboots. For example, I've done this with network adapters. When the system boots for the first time, I delete the image's adapter and configure the actual adapter in the machine. (Actually, SystemConfigurator should be able to manage NIC detection and setup.)

Once you have built your client, you'll need to install the SystemImager client software. This

is done in much the same manner as with the server but there is less to install. For a typical Red Hat install, you'll need *perl-AppConfig*, *systemconfigurator*, *systemimager-common*, and *systemimager-client* packages at a minimum.

Once all the software has been installed and configured, there is one final step in preparing the client. This involves collecting information about the client needed to build the image by running the *prepareclient* script. The script is very friendly and describes in some detail what it is doing.

```
[root@ida sis]# prepareclient
```

```
Welcome to the SystemImager prepareclient command. This command may modify the following files to prepare your golden client for having its image retrieved by the imageserver. It will also create the /etc/systemimager directory and fill it with information about your golden client. All modified files will be backed up with the .before_systemimager-3.0.1 extension.
```

```
/etc/services:
```

```
This file defines the port numbers used by certain software on your system. I will add appropriate entries for rsync if necessary.
```

```
/etc/inetd.conf:
```

```
This is the configuration file for the inet daemon, which starts up certain server software when the associated client software connects to your machine. SystemImager needs to run rsync as a standalone daemon on your golden client until it's image is retrieved by your image server. I will comment out the rsync entry in this file if it exists. The rsync daemon will not be restarted when this machine is rebooted.
```

```
/tmp/rsyncd.conf.13129:
```

```
This is a temporary configuration file that rsync needs on your golden client in order to make your filesystem available to your image server.
```

```
See "prepareclient -help" for command line options.
```

```
Continue? (y/[n]): y
```

```
***** WARNING *****
```

This utility starts an rsync daemon that makes all of your files accessible by anyone who can connect to the rsync port of this machine. This is the case until you reboot, or kill the 'rsync --daemon' process by hand. By default, once you use `getimage` to retrieve this image on your image server, these contents will become accessible to anyone who can connect to the rsync port on your imageserver. See `rsyncd.conf(5)` for details on restricting access to these files on the imageserver. See the `systemimager-ssh` package for a more secure method of making images available to clients.

```
***** WARNING *****
```

```
Continue? (y/[n]): y
```

```
Signaling xinetd to restart...
```

```
Using "sfdisk" to gather information about /dev/hda... done!
```

```
Starting or re-starting rsync as a daemon....done!
```

This client is ready to have its image retrieved. You must now run the "`getimage`" command on your imageserver.

As you can see from the output, the script runs the `rsync` server daemon on the client. For this reason, you should wait to run this script until just before you are ready to transfer the image to the image server. Also, be sure to disable this `rsync` server after copying the client image to the image server.

8.2.3.3 Retrieving the image

This is perhaps the simplest phase of the process. To get started, run the `getimage` script. You'll need to specify the name or address of the client and a name for the image. It should look something like this:

```
[root@fanny scripts]# getimage -golden-client ida -image ida.image
```

```
This program will get the "ida.image" system image from "ida"
```

```
making the assumption that all filesystems considered part
```

```
of the system image are using ext2, ext3, jfs, FAT, reiserfs, or xfs.
```

```
This program will not get /proc, NFS, or other filesystems
```

not mentioned above.

***** WARNING *****

All files retrieved from a golden client are, by default, made accessible to anyone who can connect to the rsync port of this machine. See rsyncd.conf(5) for details on restricting access to these files on the imageserver. See the systemimager-ssh package for a more secure (but less effecient) method of making images available to clients.

***** WARNING *****

See "getimage -help" for command line options.

Continue? ([y]/n): **y**

Retrieving /etc/systemimager/mounted_filesystems from ida to check for mounted filesystems...

----- ida mounted_filesystems RETRIEVAL PROGRESS -----

receiving file list ... done

/var/lib/systemimager/images/ida.image/etc/systemimager/mounted_filesystems

wrote 138 bytes read 114 bytes 504.00 bytes/sec

total size is 332 speedup is 1.32

----- ida mounted_filesystems RETRIEVAL FINISHED -----

Retrieving image ida.image from ida

----- ida.image IMAGE RETRIEVAL PROGRESS -----

...

At this point you'll see the names of each of the files whiz by. After the last file has been transferred, the script will print a summary.

...

wrote 92685 bytes read 2230781 bytes 10489.69 bytes/sec

total size is 1382212004 speedup is 594.89

----- ida.image IMAGE RETRIEVAL FINISHED -----

Press <Enter> to continue...

IP Address Assignment

There are four ways to assign IP addresses to the client systems on an ongoing basis:

1) DHCP

A DHCP server will assign IP addresses to clients installed with this image. They may be assigned a different address each time. If you want to use DHCP, but must ensure that your clients receive the same IP address each time, see "man mkdhcpstatic".

2) STATIC

The IP address the client uses during autoinstall will be permanently assigned to that client.

3) REPLICANT

Don't mess with the network settings in this image. I'm using it as a backup and quick restore mechanism for a single machine.

Which method do you prefer? [1]:

You have chosen method 1 for assigning IP addresses.

```
Are you satisfied? ([y]/n): y
```

```
Would you like to run the "addclients" utility now? (y/[n]): n
```

Unless you have edited `/etc/systemimager/systemimager.conf`, the image will be stored in the directory `/var/lib/systemimager/images` as the subdirectory `ida.image`.

The `getimage` command runs `mkautoinstallscript`, which creates the auto-install script `/var/lib/systemimager/scripts/ida.image.master` in this case, and gives you the option to move onto the next step. But before you do, you may want to kill the `rsync` daemon on the golden client.

```
[root@ida sysconfig]# ps -aux | grep rsync | grep -v grep
```

```
root      13142  0.0  0.4 1664  576 ?        S    15:46   0:00 rsync
```

```
--daemon --
```

```
[root@ida sysconfig]# kill 13142
```

8.2.3.4 Cloning the systems

The final steps of distributing the image to the clients require creating the installation scripts for the clients, preparing any needed boot media, and then booting the clients to initiate the process.^[4]

^[4] The latest release of SIS includes a program `flamethrower`. This is used to multicast images speeding the file distribution process on multicast enabled networks. `flamethrower` is not discussed in this chapter.

As noted above, you should now have an initial auto-install script. The next script you'll run is `addclients`, which does three things—it automatically generates host names for each node, it creates symbolic links to the auto-install script, one for each client, and it populates the `/etc/hosts` table.

```
[root@fanny root]# addclients
```

```
Welcome to the SystemImager "addclients" utility
```

```
...
```

A copy of the host table and the install scripts for the individual machines are located in the directory `/var/lib/systemimager/scripts`. If you don't want to use the automatically generated names, you'll need to edit `/etc/hosts` and `/var/lib/systemimager/scripts/hosts`, replacing the automatically generated names with the names you want. You'll also need to rename the individual install scripts in `/var/lib/systemimager/scripts` to match your naming scheme. Of course, if you are happy with the generated names, you can skip all this.

If you are using a network or PXE boot, you can restart the clients now. If you are booting from a floppy or CD-ROM, you'll first need to make a boot disk. You can use the scripts `mkautoinstalldiskette` or `mkautoinstallcd` to make, respectively, a boot diskette or boot CD-ROM. Here is an example of making a CD-ROM.

```
[root@fanny root]# mkautoinstallcd -out autoinstall.iso
```

Here is a list of available flavors:

standard

Which flavor would you like to use? [standard]:

...

Note that the default or `standard` flavor was used. This was created when the package `systemimager-boot-i386-standard` was installed. With the CD-ROM script, an ISO image is generated that can be used to burn a CD-ROM. Fortunately, this is a relatively small file, so it can easily be moved to another system with a CD-ROM burner. If you elect to use the diskette script instead, it will mount, format, and record the diskette for you. If you don't want to use DHCP, put the file `local.cfg` on a separate diskette even if you are using a CD-ROM to boot. When booting from a diskette, you'll need to put `local.cfg` on that diskette. Be warned, you may run out of space if you use a diskette. If you aren't using a local configuration file, you need only one boot disk. You need a diskette for each machine, however, if you are using the local configuration file. If you upgrade SystemImager, remember to regenerate your boot disks as they are release dependent.

Now that you have the boot disk, all you need to do is reboot the client from it. The client will locate the image server and then download and run the installation script. You can sit back and watch the magic for a while. After a short time, your systems should begin to beep at you. At this point, you can remove any diskettes or CD-ROMs and reboot the systems. Your node is installed.

There is one last script you may want to run if you are using DHCP. The script `mkdhcpstatic` can update your DHCP configuration file, associating IP addresses with MAC addresses. That is, if you run this script, each IP address will be tied to a specific machine based on the MAC address of the machine to which it was first assigned. Since IP addresses are handed out in numerical order, by booting the individual machines in a specific order and then running `mkdhcpstatic`, you can control IP assignments.

8.2.3.5 Other tasks

As if building your network isn't enough, SystemImager can also be used to maintain and update your clients. The script `updateclient` is used to resynchronize a client with an image. Its calling syntax is similar to `getimage`.

```
[root@hector root]# updateclient -server fanny -image ida.image
```

```
Updating image from module ida.image...
```

```
receiving file list ... done
```

```
...
```

You'll see a lot of file names whiz by at this point.

```
...
```

```
wrote 271952 bytes  read 72860453 bytes  190201.31 bytes/sec
```

```
total size is 1362174476  speedup is 18.63
```

Running bootloader...

Probing devices to guess BIOS drives. This may take a long time.

Installation finished. No error reported.

This is the contents of the device map `/boot/grub/device.map`.

Check if this is correct or not. If any of the lines is incorrect, fix it and re-run the script ``grub-install``.

```
(fd0)    /dev/fd0
```

```
(hd0)    /dev/hda
```

Probing devices to guess BIOS drives. This may take a long time.

Probing devices to guess BIOS drives. This may take a long time.

It should be noted that the script is fairly intelligent. It will not attempt to update some classes of files, such as log files, etc.

SystemInstaller also provides several commands for manipulating images. The commands *cpimage*, *mvimage*, *lsimage*, and *rmimage* are, as you might guess, analogous to *cp*, *mv*, *ls*, and *rm*.

8.3 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.^[1] 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.

[1] Steven Baum's site, <http://stommel.tamu.edu/~baum/npaci.html>, while ostensibly about Rocks, contains a very long list of cluster applications for those who want to write their own applications.

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.

9.1 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.

9.2 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.^[2]

^[2] In fairness, there may be some very rare occasions where efficiency concerns might dictate this approach.

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.

The MPI specification was developed by the MPI Forum, a collaborative effort with support from both academia and industry. It is suitable for both small clusters and "big-iron" implementations. It was designed with functionality, portability, and efficiency in mind. By providing a well-designed set of function calls, the library provides a wide range of functionality that can be implemented in an efficient manner. As a clearly defined standard, the library can be implemented on a variety of architectures, allowing code to move easily among machines.

MPI has gone through a couple of revisions since it was introduced in the early '90s. Currently, people talk of MPI-1 (typically meaning Version 1.2) and MPI-2. MPI-1 should provide for most of your basic needs, while MPI-2 provides enhancements.

While there are several different implementations of MPI, there are two that are widely used—LAM/MPI and MPICH. Both LAM/MPI and MPICH go beyond simply providing a library. Both include programming and runtime environments providing mechanisms to run programs across the cluster. Both are widely used, robust, well supported, and freely available. Excellent documentation is provided with both. Both provide all of MPI-1 and considerable portions of MPI-2, including ROMIO, Argonne National Laboratory's freely available high-performance IO system. (For more information on ROMIO, visit <http://www.mcs.anl.gov/romio>.) At this time, neither is totally thread-safe. While there are differences, if you are just getting started, you should do well with either product. And since

both are easy to install, with very little extra work you can install both.



< Day Day Up >



9.3 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*.

9.3.1 Installing LAM/MPI

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.

```
[root@fanny root]# rpm -vih lam-7.0.6-1.i586.rpm
```

```
Preparing...                               ##### [100%]
      1:lam                                  ##### [100%]
```

The files will be installed under the `/usr` directory. The space used is minimal. You can use the `laminfo` command to see the details of the installation, including compiler bindings and which modules are installed, etc.

If you need more control over the installation, you'll want to do a manual install: fetch the source, compile, install, and configure. The manual installation is only slightly more involved. However, it does take considerably longer, something to keep in mind if you'll be repeating the installation on each machine in your cluster. But if you are building an image, this is a one-time task. The installation requires a POSIX-compliant operating system, an appropriate compiler (e.g., GNU 2.95 compiler suite) and utilities such as `sed`, `grep`, and `awk`, and a modern `make`. You should have no problem with most versions of Linux.

First, you'll need to decide where to put everything, a crucial step if you are installing more than one version of MPI. If care isn't taken, you may find that part of an installation has been overwritten. In this example, the source files are saved in `/usr/local/src/lam-7.0.6` and the installed code in `/usr/local/lam-7.0.6`. First, download the appropriate file from

<http://www.lam-mpi.org/> to `/usr/local/src`. Next, uncompress and unpack the file.

```
[root@fanny src]# bunzip2 lam-7.0.6.tar.bz2
```

```
[root@fanny src]# tar -xvf lam-7.0.6.tar
```

...

```
[root@fanny src]# cd lam-7.0.6
```

You'll see a lot of files stream by as the source is unpacked. If you want to capture this output, you can *tee* it to a log file. Just append `| tee tar.log` to the end of the line and the output will be copied to the file `tar.log`. You can do something similar with subsequent commands.

Next, create the directory where the executables will be installed and configure the code specifying that directory with the `--prefix` option. You may also include any other options you desire. The example uses a configuration option to specify SSH as well. (You could also set this through an environmental variable `LAMRSH`, rather than compiling it into the code—something you must do if you use a package installation.)

```
[root@fanny lam-7.0.6]# mkdir /usr/local/lam-7.0.6
```

```
[root@fanny lam-7.0.6]# ./configure --prefix=/usr/local/lam-7.0.6 \
```

```
>--with-rsh="ssh -x"
```

If you don't have a FORTRAN compiler, you'll need to add `--without-fc` to the `configure` command. A description of other configuration options can be found in the documentation. However, the defaults are quite reasonable and will be adequate for most users. Also, if you aren't using the GNU compilers, you need to set and export compiler variables. The documentation advises that you use the same compiler to build LAM/MPI that you'll use when using LAM/MPI.

Next, you'll need to make and install the code.

```
[root@fanny lam-7.0.6]# make
```

...

```
[root@fanny lam-7.0.6]# make install
```

...

You'll see a lot of output with these commands, but all should go well. You may also want to make the examples and clean up afterwards.

```
[root@fanny lam-7.0.6]# make examples
```

...

```
[root@fanny lam-7.0.6]# make clean
```

...

Again, expect a lot of output. You only need to make the examples on the cluster head. Congratulations, you've just installed LAM/MPI. You can verify the settings and options with

the *laminfo* command.

9.3.2 User Configuration

Before you can use LAM, you'll need to do a few more things. First, you'll need to create a host file or *schema*, which is basically a file that contains a list of the machines in your cluster that will participate in the computation. In its simplest form, it is just a text file with one machine name per line. If you have multiple CPUs on a host, you can repeat the host name or you can append a CPU count to a line in the form `cpu=n`, where *n* is the number of CPUs. However, you should realize that the actual process scheduling on the node is left to the operating system. If you need to change identities when logging into a machine, it is possible to specify that username for a machine in the schema file, e.g., `user=smith`. You can create as many different schemas as you want and can put them anywhere on the system. If you have multiple users, you'll probably want to put the schema in a public directory, for example, */etc/lamhosts*.

You'll also want to set your `$PATH` variable to include the LAM executables, which can be trickier than it might seem. If you are installing both LAM/MPI and MPICH, there are several programs (e.g., *mpirun*, *mpicc*, etc.) that have the same name with both systems, and you need to be able to distinguish between them. While you could rename these programs for one of the packages, that is not a good idea. It will confuse your users and be a nuisance when you upgrade software. Since it is unlikely that an individual user will want to use both packages, the typical approach is to set the path to include one but not the other. Of course, as the system administrator, you'll want to test both, so you'll need to be able to switch back and forth. OSCAR's solution to this problem is a package called *switcher* that allows a user to easily change between two configurations. *switcher* is described in [Chapter 6](#).

A second issue is making sure the path is set properly for both interactive and noninteractive or non-login shells. (The path you want to add is */usr/local/lam-7.0.6/bin* if you are using the same directory layout used here.) The processes that run on the compute nodes are run in noninteractive shells. This can be particularly confusing for *bash* users. With *bash*, if the path is set in *.bash_profile* and not in *.bashrc*, you'll be able to log onto each individual system and run the appropriate programs, but you won't be able to run the programs remotely. Until you realize what is going on, this can be a frustrating problem to debug. So, if you use *bash*, don't forget to set your path in *.bashrc*. (And while you are setting paths, don't forget to add the manpages when setting up your paths, e.g., */usr/local/lam-7.0.6/man*.)

It should be downhill from here. Make sure you have *ssh-agent* tuning and that you can log onto other machines without a password. Setting up and using SSH is described in [Chapter 4](#). You'll also need to ensure that there is no output to *stderr* whenever you log in using SSH. (When LAM sees output to *stderr*, it thinks something bad is happening and aborts.) Since you'll get a warning message the first time you log into a system with SSH as it adds the remote machine to the known hosts, often the easiest thing to do (provided you don't have too many machines in the cluster) is to manually log into each machine once to get past this problem. You'll only need to do this once. *recon*, described in the subsection on testing, can alert you to some of these problems.

Also, the directory */tmp* must be writable. Don't forget to turn off or reconfigure your firewall as needed.

9.3.3 Using LAM/MPI

The basic steps in creating and executing a program with LAM are as follows:

1. Booting the runtime system with *lamboot*.

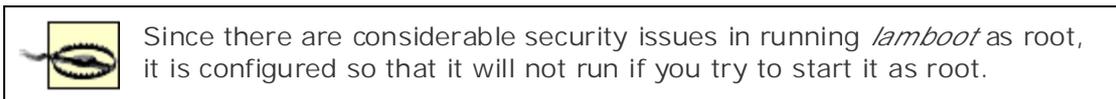
2. Writing and compiling a program with the appropriate compiler, e.g., *mpicc*.^[3]

^[3] Actually, you don't need to boot the system to compile code.

3. Execute the code with the *mpirun* command.
4. Clean up any crashed processes with *lamclean* if things didn't go well.
5. Shut down the runtime system with the command *lamhalt*.

Each of these steps will now be described.

In order to use LAM, you will need to launch the runtime environment. This is referred to as booting LAM and is done with the *lamboot* command. Basically, *lamboot* starts the *lamd* daemon, the message server, on each machine.



You specify the schema you want to use as an argument.

```
[sloanjd@fanny sloanjd]$ lamboot -v /etc/lamhosts
```

```
LAM 7.0.6/MPI 2 C++/ROMIO - Indiana University
```

```
n-1<9677> ssi:boot:base:linear: booting n0 (fanny.wofford.int)
```

```
n-1<9677> ssi:boot:base:linear: booting n1 (george.wofford.int)
```

```
...
```

```
n0<15402> ssi:boot:base:linear: finished
```

As noted above, you must be able to log onto the remote systems without a password and without any error messages. (If this command doesn't work the first time, you might give this a couple of tries to clear out any one time error messages.) If you don't want to see the list of nodes, leave out the *-v*. You can always use the *lamnodes* command to list the nodes later if you wish.

```
[sloanjd@fanny sloanjd]$ lamnodes
```

```
n0      10.0.32.144:1:origin,this_node
```

```
n1      10.0.32.145:1:
```

```
...
```

You'll only need to boot the system once at the beginning of the session. It will remain loaded until you halt it or log out. (Also, you can omit the schema and just use the local machine. Your code will run only on the local node, but this can be useful for initial testing.)

Once you have entered your program using your favorite editor, the next step is to compile and link the program. You could do this directly by typing in all the compile options you'll

need. But it is much simpler to use one of the wrapper programs supplied with LAM. The programs *mpicc*, *mpiCC*, and *mpif77* will respectively invoke the C, C++, and FORTRAN 77 compilers on your system, supplying the appropriate command-line arguments for LAM. For example, you might enter something like the following:

```
[sloanjd@fanny sloanjd]$ mpicc -o hello hello.c
```

(*hello.c* is one of the examples that comes with LAM and can be found in */usr/local/src/lam-7.0.6/examples/hello* if you use the same directory structure used here to set up LAM.) If you want to see which arguments are being passed to the compiler, you can use the `-showme` argument. For example,

```
[sloanjd@fanny sloanjd]$ mpicc -showme -o hello hello.c
```

```
gcc -I/usr/local/lam-7.0.6/include -pthread -o hello hello.c -L/usr/local/  
lam-7.0.6/lib -llammpio -llamf77mpi -lmpi -llam -lutil
```

With `-showme`, the program isn't compiled; you just see the arguments that would have been used had it been compiled. Any other arguments that you include in the call to *mpicc* are passed on to the underlying compiler unchanged. In general, you should avoid using the `-g` (debug) option when it isn't needed because of the overhead it adds.

To compile the program, rerun the last command without `-showme` if you haven't done so. You now have an executable program. Run the program with the *mpirun* command. Basically, *mpirun* communicates with the remote LAM daemon to fork a new process, set environment variables, redirect I/O, and execute the user's command. Here is an example:

```
[sloanjd@fanny sloanjd]$ mpirun -np 4 hello
```

```
Hello, world! I am 0 of 4
```

```
Hello, world! I am 1 of 4
```

```
Hello, world! I am 2 of 4
```

```
Hello, world! I am 3 of 4
```

As shown in this example, the argument `-np 4` specified that four processes be used when running the program. If more machines are available, only four will be used. If fewer machines are available, some machines will be used more than once.

Of course, you'll need the executable on each machine. If you're using NFS to mount your home directories, this has already been taken care of if you are working in that directory. You should also remember that *mpirun* can be run on a single machine, which can be helpful when you want to test code away from a cluster.

If a program crashes, there may be extraneous processes running on remote machines. You can clean these up with the *lamclean* command. This is a command you'll use only when you are having problems. Try *lamclean* first and if it hangs, you can escalate to *wipe*. Rerun *lamboot* after using *wipe*. This isn't necessary with *lamclean*. Both *lamclean* and *wipe* take a `-v` for verbose output.

Once you are done, you can shut down LAM with the *lamhalt* command, which kills the *lamd* daemon on each machine. If you wish, you can use `-v` for verbose output. Two other useful LAM commands are *mpitask* and *mpimsg*, which are used to monitor processes across the cluster and to monitor the message buffer, respectively.

9.3.4 Testing the Installation

LAM comes with a set of examples, tests, and tools that you can use to verify that it is properly installed and runs correctly. We'll start with the simplest tests first.

The *recon* tool verifies that LAM will boot properly. *recon* is not a complete test, but it confirms that the user can execute commands on the remote machine, and that the LAM executables can be found and executed.

```
[sloanjd@fanny bin]$ recon
```

```
-----  
Woo hoo!
```

```
recon has completed successfully. This means that you will most likely  
be able to boot LAM successfully with the "lamboot" command (but this  
is not a guarantee). See the lamboot(1) manual page for more  
information on the lamboot command.
```

```
If you have problems booting LAM (with lamboot) even though recon  
worked successfully, enable the "-d" option to lamboot to examine each  
step of lamboot and see what fails. Most situations where recon  
succeeds and lamboot fails have to do with the hboot(1) command (that  
lamboot invokes on each host in the hostfile).
```

```
-----  
Since lamboot is required to run the next tests, you'll need to run these tests as a non-  
privileged user. Once you have booted LAM, you can use the tping command to check basic  
connectivity. tping is similar to ping but uses the LAM echo server. This confirms that both  
network connectivity and that the LAM daemon is listening. For example, the following  
command sends two one-byte packets to the first three machines in your cluster.
```

```
[sloanjd@fanny sloanjd]$ tping n1-3 -c2
```

```
1 byte from 3 remote nodes: 0.003 secs
```

```
1 byte from 3 remote nodes: 0.002 secs
```

```
2 messages, 2 bytes (0.002K), 0.006 secs (0.710K/sec)
```

```
roundtrip min/avg/max: 0.002/0.003/0.003
```

If you want to probe every machine, use `n` without a count.

The LAM test suite is the most comprehensive way to test your system. It can be used to confirm that you have a complete and correct installation. Download the test suite that corresponds to your installation and then uncompress and unpack it.

```
[sloanjd@fanny sloanjd]$ bunzip2 lamtests-7.0.6.tar.bz2
```

```
[sloanjd@fanny sloanjd]$ tar -xvf lamtests-7.0.6.tar
```

...

This creates the directory *lamtests-7.0.6* with the tests and a set of directions in the file *README*. Next, you should start LAM with *lamboot* if you haven't already done so. Then change to the *test* directory and run *configure*.

```
[sloanjd@fanny sloanjd]$ cd lamtests-7.0.6
```

```
[sloanjd@fanny lamtests-7.0.6]$ ./configure
```

...

Finally, run *make*.

```
[sloanjd@fanny lamtests-7.0.6]$ make -k check
```

...

You'll see lots of output scroll past. Don't be concerned about an occasional error message while it is running. What you want is a clean bill of health when it is finally done. You can run specific tests in the test suite by changing into the appropriate subdirectory and running *make*.

9.4 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.

9.4.1 Installing

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.)

ch_nt

This is a version of MPI for Windows NT machines.

The best choice for most clusters is either the *ch_p4* model or *ch_p4mpd* model. The *ch_p4mpd* model assumes a homogenous architecture while *ch_p4* works with mixed architectures. If you have a homogenous architecture, *ch_p4mpd* should provide somewhat

better performance. This section will describe the `ch_p4` since it is more versatile.

The first step in installing MPICH is to download the source code for your system. MPICH is not available in binary (except for Windows NT). Although the available code is usually updated with the latest patches, new patches are occasionally made available, so you'll probably want to check the patch list at the site. If necessary, apply the patches to your download file following the directions supplied with the patch file.

Decide where you want to install the software. This example uses `/usr/local/src/mpich`. Then download the source to the appropriate directory, uncompress it, and unpack it.

```
[root@fanny src]# gunzip mpich.tar.gz
```

```
[root@fanny src]# tar -xvf mpich.tar
```

...

Expect lots of output! Change to the directory where the code was unpacked, make a directory for the installation, and run `configure`.

```
[root@fanny src]# cd mpich-1.2.5.2
```

```
[root@fanny mpich-1.2.5.2]# mkdir /usr/local/mpich-1.2.5.2
```

```
[root@fanny mpich-1.2.5.2]# ./configure --prefix=/usr/local/mpich-1.2.5.2 \
```

```
> -rsh=ssh
```

...

As with LAM, this installation configures MPICH to use SSH.^[4] Other configuration options are described in the installation and user's guides.

^[4] Alternatively, you could use the environmental variable `$RSHCOMMAND` to specify SSH.

Next, you'll make, install, and clean up.

```
[root@fanny mpich-1.2.5.2]# make
```

...

```
[root@fanny mpich-1.2.5.2]# make install
```

...

```
[root@fanny mpich-1.2.5.2]# make clean
```

...

Again, you'll see lots of output after each of these steps. The first `make` builds the software while the `make install`, which is optional, puts it in a public directory. It is also a good idea to make the tests on the head node.

MPICH on Windows Systems

For those who need to work in different environments, it is worth noting that MPICH will run under Windows NT and 2000. (While I've never tested it in a cluster setting, I have used MPICH on XP to compile and run programs.)

To install, download the self-extracting archive. By default, this will install the runtime DLLs, the development libraries, *jumpshot*, and a PDF of the user's manual. I've used this combination without problems with Visual Studio.NET and CodeWarrior. It is said to work with GCC but I haven't tested it.

Installing MPICH on a laptop can be very helpful at times, even if you aren't attaching the laptop to a cluster. You can use it to initially develop and test code. In this mode, you would run code on a single machine as though it were a cluster. This is not the same as running the software on a cluster, and you definitely won't see any performance gains, but it will allow you to program when you are away from your cluster. Of course, you can also include Windows machines in your cluster as compute nodes. For more information, see the MPICH `ch_nt` manual.

Before you can use MPICH, you'll need to tell it which machines to use by editing the file *machine.architecture*. For Linux clusters, this is the file *machine.LINUX* and is located in the directory `./share` under installation directory. If you use the same file layout used here, the file is `/usr/local/mpich-1.2.5.2/share/machines.LINUX`. This file is just a simple list of machines with one hostname per line. For SMP systems, you can append a `:n` where *n* is the number of processors in the host. This file plays the same role as the schema with LAM. (You can specify a file with a different set of machines as a command-line argument when you run a program if desired.)

9.4.2 User Configuration

Since individual users don't set up schemas for MPICH, there is slightly less you need to do compared to LAM. Besides this difference, the user setup is basically the same. You'll need to set the `$PATH` variable appropriately (and `$MANPATH`, if you wish). The same concerns apply with MPICH as with LAM—you need to distinguish between LAM and MPICH executables if you install both, and you need to ensure the path is set for both interactive and noninteractive logins. You'll also need to ensure that you can log onto each machine in the cluster using SSH without a password. (For more information on these issues, see the subsection on user configuration under LAM/MPI.)

9.4.3 Using MPICH

Unlike LAM, you don't need to boot or shut down the runtime environment when running an MPICH program. With MPICH you'll just need to write, compile, and run your code. The downside is, if your program crashes, you may need to manually kill errant processes on compute nodes. But this shouldn't be a common problem. Also, you'll be able to run programs as root provided you distribute the binaries to all the nodes. (File access can be an issue if you don't export root's home directory via NFS.)

The first step is to write and enter your program using your favorite text editor. Like LAM, MPICH supplies a set of wrapper programs to simplify compilation—*mpicc*, *mpiCC*, and *mpif77*, and *mpif90* for C, C++, FORTRAN 77, and FORTRAN 90, respectively. Here is an example of compiling a C program:

```
[sloanjd@fanny sloanjd]$ mpicc -o cpi cpi.c
```

cpi.c is one of the sample programs included with MPICH. It can be found in the directory *../examples/basic* under the source directory.

You can see the options supplied by the wrapper program without executing the code by using the `-show` option. For example,

```
[sloanjd@fanny sloanjd]$ mpicc -show -o cpi cpi.c
```

```
gcc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1 -DHAVE_
STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1 -L/opt/mpich-1.2.5.10-ch_p4-gcc/
lib -o cpi cpi.c -lmpich
```

Obviously, you'll want to use the wrapper programs rather than type in arguments manually.

To run a program, you use the *mpirun* command. Again, before the code will run, you must have copies of the binaries on each machine and you must be able to log into each machine with SSH without a password. Here is an example of running the code we just compiled.

```
[sloanjd@fanny sloanjd]$ mpirun -np 4 cpi
```

```
Process 0 of 4 on fanny.wofford.int
```

```
pi is approximately 3.1415926544231239, Error is 0.000000008333307
```

```
wall clock time = 0.008783
```

```
Process 2 of 4 on hector.wofford.int
```

```
Process 1 of 4 on george.wofford.int
```

```
Process 3 of 4 on ida.wofford.int
```

The argument `-np 4` specified running the program with four processes. If you want to specify a particular set of machines, use the `-machinefile` argument.

```
[sloanjd@fanny sloanjd]$ mpirun -np 4-machinefile machines cpi
```

```
Process 0 of 4 on fanny.wofford.int
```

```
pi is approximately 3.1415926544231239, Error is 0.000000008333307
```

```
wall clock time = 0.007159
```

```
Process 1 of 4 on george.wofford.int
```

```
Process 2 of 4 on fanny.wofford.int
```

```
Process 3 of 4 on george.wofford.int
```

In this example, four processes were run on the two machines listed in the file *machines*. Notice that each machine was used twice. You can view the *mpicc(1)* and *mpirun(1)* manpage for more details.

9.4.4 Testing the Installation

You can test connectivity issues and the like with the MPICH-supplied script *tstmachines*, which is located in the `../sbin` directory under the MPICH installation. This script takes the architecture as an argument. For example,

```
[sloanjd@fanny sloanjd]$ /usr/local/mpich-1.2.5.2/sbin/tstmachines LINUX
```

If all is well, the script runs and terminates silently. If there is a problem, it makes suggestions on how to fix the problem. If you want more reassurance that it is actually doing something, you can run it with the `-v` argument.

For more thorough testing, MPICH provides a set of tests with the distribution. You'll find a thorough collection of tests supplied with the source files. These are in the directory `../examples/test`. You run these tests by executing the command:

```
[sloanjd@fanny test]$ make testing | tee make.log
```

...

You'll need to do this in the *test* directory. This directory must be shared among all the nodes on the cluster, so you will have to either mount this directory on all the machines or copy its contents over to a mounted directory. When this runs, you'll see a lot of output as your cluster is put through its paces. The output will be copied to the file *make.log*, so you'll be able to peruse it at your leisure.

9.4.5 MPE

The *Multi-Processing Environment (MPE)* library extends MPI. MPE provides such additional facilities as libraries for creating log files, an X graphics library, graphical visualization tools, routines for serializing sections of parallel code, and debugger setup routines. While developed for use with MPICH, MPE can be used with any MPI implementation. MPE is included with MPICH and will be built and installed. MPE includes both a library for collecting information and a viewer for displaying the collected information. A user's guide is available that provides greater detail. Use of MPE is described in greater detail in [Chapter 17](#).

MPE includes four viewers—*upshot*, *nupshot*, *jumpshot-2*, and *jumpshot-3*. These are not built automatically since the software required for the build may not be present on every machine. Both *upshot* and *nupshot* require Tcl/Tk and Wish. *jumpshot-2* and *jumpshot-3* require Java.

There are three different output formats for MPE log files—*alog*, an ASCII format provided for backwards compatibility; *clog*, *alog*'s binary equivalent; and *slog*, a scalable format capable of handling very large files. *upshot* reads *alog* files, *nupshot* and *jumpshot-2* read *clog* files, and *jumpshot-3* reads *slog* files. MPE includes two utilities, *clog2slog* and *clog2alog*, to convert between formats. The basic functionality of the viewers is similar, so installing any one of them will probably meet your basic needs.

Although the requirements are different, the compilation process is similar for each tool. You can build the viewers collectively or individually. For example, to compile *jumpshot-3*, you'll need to install Java if you don't already have it. JDK-1.1, JDK-1.2, or JDK-1.3 can be used. (*jumpshot-2* compiles only with JDK-1.1.) If you don't have the appropriate Java, you can download it from <http://www.blackdown.org> or <http://java.sun.com> and follow the installation directions given at the respective site. Once Java has been installed, make sure that you add its directory to your path. Next, change to the `../mpe/viewer/jumpshot-3` subdirectory under the MPICH directory, for example, `/usr/local/src/mpich-1.2.5.2/mpe/viewers/jumpshot-3`. Now you can configure and build *jumpshot-3*.

```
[root@fanny jumpshot-3]# ./configure
```

```
...
```

```
[root@fanny jumpshot-3]# make
```

```
...
```

```
[root@fanny jumpshot-3]# make install
```

```
...
```

jumpshot-3 will be installed in the `/usr/local/bin` directory as *jumpshot*. (You will only need to install it on the head node.) For details on the installation of the other viewer, see the MPE installation and user's guide.

To test your installation, you'll need to compile a program using the `-mpilog` option and run the code to create a log file.

```
[sloanjd@fanny sloanjd]$ mpicc -mpilog -o cpi cpi.c
```

```
[sloanjd@fanny sloanjd]$ mpirun cpi
```

```
...
```

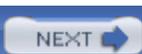
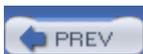
When you run the code, the log file *cpi.clog* will be created. You'll need to convert this to a format that *jumpshot-3* can read.

```
[sloanjd@fanny sloanjd]$ clog2slog cpi.clog
```

The conversion routines are in the directory `../mpich-1.2.5.2/bin`. Now you can view the output. Of course, you must have a graphical login for this to work. With this command, several windows should open on your display.

```
[sloanjd@fanny sloanjd]$ jumpshot cpi.slog
```

As noted, the use of MPE will be described in greater detail in [Chapter 17](#).



9.5 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.

9.5.1 Debuggers

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* 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%]
```

Depending on what is installed on your system, you may run into a few dependencies. For example, *ddd* requires *openmotif*.

9.5.2 HDF5

Depending on the nature of the programming you do, there may be other useful libraries that you'll want to install. One such package that OSCAR includes is *Hierarchical Data Format (Version 5)* or *HDF5*. HDF5 is a freely available software package developed by the HDF5 group at the National Center for Supercomputing Applications (NCSA). The official web site is <http://hdf.ncsa.uiuc.edu/HDF5/>.

HDF5 is both a file format standard and a library with utilities specifically designed for storing scientific data. It supports very large files and is designed and tuned for efficient storage on parallel computing systems. Data is stored in two parts, a header and a data array. The header contains the information needed to interpret the data array. That is, it describes and annotates the data set. The data sets are essentially multidimensional arrays of items. The

API is available only in C. While HDF5 is beyond the scope of this book, you should be aware it exists should you need it. An extensive tutorial, as well as other documentation, is available at the software's web site.

9.5.3 SPRNG

Scalable Parallel Random Number Generators (SPRNG) is a library that provides six different state-of-the-art random number generators for use with parallel programs. *SPRNG* integrates nicely with MPI. Its use is described in [Chapter 15](#).

SPRNG is freely available from <http://sprng.cs.fsu.edu/>. At the time this was written, the latest version *sprng2.0a.tgz*. First, download the package and move it to an appropriate directory, e.g., */usr/local/src*. The next step is to unpack it.

```
[root@amy src]# gunzip sprng2.0a.tgz
[root@amy src]# tar -xvf sprng2.0a.tar
```

...

Then change to the directory to where you just unpacked the source. Before you can build it, you need to edit a couple of files. In the first section of the file *make.CHOICES*, select the appropriate platform. Typically, this will be INTEL for Linux clusters. Make sure the line

```
PLAT = INTEL
```

is uncommented and the lines for other platforms are commented out. Because you want to use it with MPI, in the second section, uncomment the line

```
MPIDEF = -DSPRNG_MPI
```

You should also comment out the two lines in the third section if *libgmp.a* is not available on your system.

You should also edit the appropriate architecture file in the *SRC* subdirectory, typically *make.INTEL*. You'll need to make two sets of changes for a Linux cluster. First, change all the *gcc* optimization flags from *-O3* to *-O1*. Next, change all the paths to MPI to match your machine. For the setup shown in this chapter, the following lines were changed:

```
MPIDIR = -L/usr/local/mpich-1.2.5.2/lib
```

and

```
CFLAGS = -O1 -DLittleEndian $(PMLCGDEF) $(MPIDEF) -D$(PLAT) \
```

```
-I/usr/local/mpich-1.2.5.2/include -I/usr/local/mpich-1.2.5.2/include
```

```
CLDFLAGS = -O1
```

```
FFLAGS = -O1 $(PMLCGDEF) $(MPIDEF) -D$(PLAT) \
```

```
-I/usr/local/mpich-1.2.5.2/include -I/usr/local/mpich-1.2.5.2/include -I.
```

```
F77LDFLAGS = -O1
```

Once you've done this, run *make* from the root of the source tree. If you want to play with the MPI examples, run *make mpi* in the *EXAMPLES* subdirectory.

To use the library, you must adjust your compile paths to include the appropriate directories. For example, to use SPRNG with OSCAR and MPICH, the following changes should work.

```
MPIDIR = -L/opt/mpich-1.2.5.10-ch_p4-gcc/lib
```

```
MPILIB = -lmpich
```

```
# Please include mpi header file path, if needed
```

```
CFLAGS = -O1 -DLittleEndian $(PMLCGDEF) $(MPIDEF) -D$(PLAT) -I/opt/mpich-  
1.2.5.10-ch_p4-gcc/include -I/opt/mpich-1.2.5.10-ch_p4-gcc/include
```

```
CLDFLAGS = -O1
```

```
F77LDFLAGS = -O1 $(PMLCGDEF) $(MPIDEF) -D$(PLAT) -I/opt/mpich-1.2.5.10-ch_p4-  
gcc/include -I/opt/mpich-1.2.5.10-ch_p4-gcc/include -I.
```

```
F77LDFLAGS = -O1
```

Note this installation is specific to one version of MPI. See [Chapter 15](#) for the details of using SPRNG.

9.6 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.

9.6.1 Adding MPE

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
```

(Be sure these paths match your system.)

Now you can change to the MPE directory and run `configure`, `make`, and `make install`.

```
[root@amy src]# cd mpich-1.2.5.2/mpe
[root@amy mpe]# ./configure
...
[root@amy mpe]# make
...
[root@amy mpe]# make install
...
```

You should now have MPE on your system. If you used the same directories as used here, it will be in `/usr/local/src/mpich-1.2.5.2/mpe`.

9.7 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.

10.1 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.^[1] 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.

[1] A Python/Tk GUI known as *C2G* has also been developed.

10.1.1 Installing C3

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](#).

Once you have met the prerequisites, you can download, unpack, and install C3. To download it, go to <http://www.csm.ornl.gov/torc/C3/> and follow the link to the download page. You can download sources or an RPM package. In this example, sources are used. If you install from RPMs, install the full install RPM and profile RPM on servers and the client RPM on clients. Note that with the simple installation you only need to install C3 on the head node of your cluster. However, you will need *SSH* and the like on every node.

Once you have unpacked the software and read the *README* files, you can run the install script *Install-c3*.

```
[root@fanny src]# gunzip c3-4.0.1.tar.gz
```

```
[root@fanny src]# tar -xvf c3-4.0.1.tar
```

```
[root@fanny src]# cd c3-4.0.1
```

```
[root@fanny c3-4.0.1]# ./Install-c3
```

The install script will copy the scripts to */opt/c3-4* (for Version 4 at least), set paths, and install man pages. There is nothing to compile.

The next step is creating a configuration file. The default file is `/etc/c3.conf`. However, you can use other configuration files if you wish by explicitly referencing them in C3 commands using the `-f` option with the file name.

Here is a very simple configuration file:

```
cluster local {  
  
    fanny.wofford.int  
  
    george.wofford.int  
  
    hector.wofford.int  
  
    ida.wofford.int  
  
    james.wofford.int  
  
}
```

This example shows a configuration for a single cluster. In fact, the configuration file can contain information on multiple clusters. Each cluster will have its own cluster description block, which begins with the identifier `cluster` followed by a name for a cluster. The name can be used in C3 commands to identify the specific cluster if you have multiple cluster description blocks. Next, the machines within the cluster are listed within curly braces. The first machine listed is the head node. To remove ambiguity, the head node entry can consist of two parts separated by a colon—the head node's external interface to the left of the colon and the head node's internal interface to the right of the colon. (Since *fanny* has a single interface, that format was not appropriate for this example.) The head node is followed by the compute nodes. In this example, the compute nodes are listed one per line. It is possible to specify a range. For example, `node[01-64]` would specify 64 machines with the names *node1*, *node2*, etc. The cluster definition block is closed with another curly brace. Of course, all machine names must resolve to IP addresses, typically via the `/etc/hosts` file. (The commands *cname* and *cnum*, described later in this section, can be discerning the details surrounding node indices.)

Within the compute node list, you can also use the qualifiers `exclude` and `dead`. `exclude` is applied to range qualifiers and immediately follow a range specification. `dead` applies to individual machines and precedes the machine name. For example,

```
node[1-64]  
  
exclude 60  
  
alice  
  
dead bob  
  
carol
```

In this list *node60* and *bob* are designated as being unavailable. Starting with Version 3 of C3, it is possible to use ranges in C3 commands to restrict actions to just those machines within the range. The order of the machines in the configuration file determines their numerical position within the range. In the example, the 67 machines defined have list positions 0 through 66. If you deleted *bob* from the file instead of marking it as dead, *carol's* position would change from 66 to 65, which could cause confusion. By using `exclude` and `dead`, you effectively remove a machine from a cluster without renumbering the remaining machines. `dead` can also be used with a dummy machine to switch from 0-indexing to 1-indexing. For

example, just add the following line to the beginning of the machine list:

```
dead place_holder
```

Once done, all the machines in the list move up one position. For more details on the configuration file, see the *c3.conf(5)* and *c3-scale(5)* manpages.

Once you have created your configuration file, there is one last thing you need to do before C3 is ready to go. For the command *ckill* to work properly, the Perl script *ckillnode* must be installed on each individual machine. Fortunately, the rest of C3 is installed and functional, so you can use it to complete the installation. Just issue these commands:

```
[root@fanny root]# cexec mkdir /opt/c3-4

***** local *****

----- george.wofford.int-----

...

[root@fanny root]# cpush /opt/c3-4/ckillnode

building file list ... building file list ... building file list ... building
file list ... done

...
```

The first command makes the directory */opt/c3-4* on each machine in your cluster and the second copies the file *ckillnode* to each machine. You should see a fair amount of output with each command. If you are starting SSH manually, you'll need to start it before you try this.

10.1.2 Using C3 Commands

Here is a brief description of C3's more useful utilities.

10.1.2.1 cexec

This command executes a command string on each node in a cluster. For example,

```
[root@fanny root]# cexec mkdir tmp

***** local *****

----- george.wofford.int-----

----- hector.wofford.int-----

----- ida.wofford.int-----

----- james.wofford.int-----
```

The directory *tmp* has been created on each machine in the local cluster. *cexec* has a serial version *cexecs* that can be used for testing. With the serial version, the command is executed to completion on each machine before it is executed on the next machine. If there is any

ambiguity about the order of execution for the parts of a command, you should use double quotes within the command. Consider:

```
[root@fanny root]# cexec "ps | grep a.out"
```

...

The quotes are needed here so *grep* will be run on each individual machine rather than have the full output from *ps* shipped to the head node.

10.1.2.2 cget

This command is used to retrieve a file from each machine in the cluster. Since each file will initially have the same name, when the file is copied over, the cluster and host names are appended. Here is an example.

```
[root@fanny root]# cget /etc/motd
```

```
[root@fanny root]# ls
```

```
motd_local_george.wofford.int
```

```
motd_local_hector.wofford.int
```

```
motd_local_ida.wofford.int
```

```
motd_local_james.wofford.int
```

cget ignores links and subdirectories.

10.1.2.3 ckill

This script allows you to kill a process running on each node in your cluster. To use it, specify the process by name, not by number, because it is unlikely that the processes will have the same process ID on each node.

```
[root@fanny root]# ckill -u sloanjd a.out
```

```
uid selected is 500
```

```
uid selected is 500
```

```
uid selected is 500
```

```
uid selected is 500
```

You may also specify an owner as shown in the example. By default, the local user name will be used.

10.1.2.4 cpush

This command is used to move a file to each node on the cluster.

```
[root@fanny root]# cpush /etc/motd /root/motd.bak

building file list ... done

building file list ... done

motd

motd

building file list ... done

motd

wrote 119 bytes  read 36 bytes  62.00 bytes/sec
total size is 39  speedup is 0.25

wrote 119 bytes  read 36 bytes  62.00 bytes/sec
total size is 39  speedup is 0.25

wrote 119 bytes  read 36 bytes  62.00 bytes/sec
total size is 39  speedup is 0.25

building file list ... done

motd

wrote 119 bytes  read 36 bytes  62.00 bytes/sec

total size is 39  speedup is 0.25
```

As you can see, statistics for each move are printed. If you only specify one file, it will use the same name and directory for the source and the destination.

10.1.2.5 **crm**

This routine deletes or removes files across the cluster.

```
[root@fanny root]# crm /root/motd.bak
```

Like its serial counterpart, you can use the `-i`, `-r` and `-v` options for interactive, recursive, and verbose deletes, respectively. Please note, the `-i` option only prompts once, not for each node. Without options, `crm` silently deletes files.

10.1.2.6 **cshutdown**

This utility allows you to shut down the nodes in your cluster.

```
[root@fanny root]# cshutdown -r t 0
```

In this example, the time specified was 0 for an immediate reboot. (Note the absence of the

hyphen for the `t` option.) Additional options are supported, e.g., to include a shutdown message.

10.1.2.7 `clist`, `cname`, and `cnum`

These three commands are used to query the configuration file to assist in determining the appropriate numerical ranges to use with C3 commands. `clist` lists the different clusters in the configuration file.

```
[root@amy root]# clist

cluster  oscar_cluster  is a direct local cluster
cluster  pvfs_clients   is a direct local cluster
cluster  pvfs_iod       is a direct local cluster
```

`cname` lists the names of machines for a specified range.

```
[root@fanny root]# cname local:0-1

nodes from cluster:  local

cluster:  local ; node name:  george.wofford.int
cluster:  local ; node name:  hector.wofford.int
```

Note the use of 0 indexing.

`cnum` determines the index of a machine given its name.

```
[root@fanny root]# cnum ida.wofford.int

nodes from cluster:  local

ida.wofford.int is at index 2 in cluster local
```

These can be very helpful because it is easy to lose track of which machine has which index.

10.1.2.8 Further examples and comments

Here is an example using a range:

```
[root@fanny root]# cpush local:2-3 data

...
```

`local` designates which cluster is within your configuration file. Because compute nodes are numbered from 0, this will push the file `data` to the third and fourth nodes in the cluster. (That is, it will send the file from *fanny* to *ida* and *james*, skipping over *george* and *hector*.) Is that what you expected? For more information on ranges, see the manpage `c3-range(5)`.

Note that the name used in C3 commands must match the name used in the configuration file. For C3, *ida* and *ida.wofford.int* are not equal even if there is an alias *ida* that resolves to

ida.wofford.int. For example,

```
[root@fanny root]# cnum ida.wofford.int

nodes from cluster: local

ida.wofford.int is at index 2 in cluster local

[root@fanny root]# cnum ida

nodes from cluster: local
```

When in doubt about what form to use, just refer back to */etc/c3.conf*.

In addition to the commands just described, the C3 command *cpushimage* can be used with SystemImager to push an image from server to nodes. There are also several user-contributed utilities. While not installed, these can be found in the C3 source tree in the subdirectory *contrib*. User-contributed scripts can be used as examples for writing other scripts using C3 commands.

C3 commands take a number of different options not discussed here. For a brief description of other options, use the `--help` option with individual commands. For greater detail, consult the manpage for the individual command.

10.2 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 *gmetric* command under *cron*, you could track computer temperature over time.

Finally, Ganglia also provides an execution environment. *gexec* allows you to run commands across the cluster transparently and forward *stdin*, *stdout*, and *stderr*. This discussion will focus on the three core elements of Ganglia—*gmond*, *gmetad*, and the web frontend.

10.2.1 Installing and Using Ganglia

Ganglia can be installed by compiling the sources or using RPM packages. The installation of the software for the management station, i.e., the node that collects information from the other nodes and maintains the database, is somewhat more involved. With large clusters, you may want to use a machine as a dedicated monitor. For smaller clusters, you may be able to get by with your head node if it is reasonably equipped. We'll look at the installation of the management node first since it is more involved.

10.2.1.1 RRDTool

Before you begin, there are several prerequisites for installing Ganglia. First, your network and hosts must be multicast enabled. This typically isn't a problem with most Linux installations. Next, the management station or stations, i.e., the machine on which you'll install *gmetad* and the web frontend, will also need *RRDtool* and Perl and a PHP-enabled web server.^[2] (Since you will install only *gmond* on your compute nodes, these do not require Apache or *RRDtool*.)

[2] It appears that only the include file and library from *RRDtool* is needed, but I have not verified this. Perl is required for *RRDtool*, not Ganglia.

RRDtool is a round-robin database. As you add information to the database, the oldest data is dropped from the database. This allows you to store data in a compact manner that will not expand endlessly over time. Sources can be downloaded from <http://www.rrdtool.org/>. To install it, you'll need to unpack it and run *configure*, *make*, and *make install*.

```
[root@fanny src]# gunzip rrdtool-1.0.48.tar.gz
```

```
[root@fanny src]# tar -vxf rrdtool-1.0.48.tar
```

```
...
```

```
[root@fanny src]# cd rrdtool-1.0.48
```

```
[root@fanny rrdtool-1.0.48]# ./configure
```

```
...
```

```
[root@fanny rrdtool-1.0.48]# make
```

```
[root@fanny rrdtool-1.0.48]# make install
```

```
...
```

You'll see a lot of output along the way. In this example, I've installed it under */usr/local/src*. If you want to install it in a different directory, you can use the `--prefix` option to specify the directory when you run *configure*. It doesn't really matter where you put it, but when you build Ganglia you'll need to tell Ganglia where to find the *RRDtool* library and include files.

10.2.1.2 Apache and PHP

Next, check the configuration files for Apache to ensure the PHP module is loaded. For Red Hat 9.0, the primary configuration file is *httpd.conf* and is located in */etc/httpd/conf/*. It, in turn, includes the configuration files in */etc/httpd/conf.d/*, in particular *php.conf*. What you are looking for is a configuration command that loads the PHP module somewhere in one of the Apache configuration files. That is, one of the configuration files should have some lines like the following:

```
LoadModule php4_module modules/libphp4.so
```

```
...
```

```
<Files *.php>
```

```
    SetOutputFilter PHP
```

```
    SetInputFilter PHP
```

```
    LimitRequestBody 524288
```

```
</Files>
```

If you used the package system to set up Apache and PHP, this should have been done for you. Finally, make sure Apache is running.

10.2.1.3 Ganglia monitor core

Next, you'll need to download the appropriate software. Go to <http://ganglia.sourceforge.net/>. You'll have a number of choices, including both source files and RPM files, for both Ganglia and related software. The Ganglia monitor core contains both *gmond* and *gmetad* (although by default it doesn't install *gmetad*). Here is an example of using the monitor core download to install from source files. First, unpack the software.

```
[root@fanny src]# gunzip ganglia-monitor-core-2.5.6.tar.gz
[root@fanny src]# tar -xvf ganglia-monitor-core-2.5.6.tar
...
```

As always, once you have unpacked the software, be sure to read the *README* file.

Next, change to the installation directory and build the software.

```
[root@fanny src]# cd ganglia-monitor-core-2.5.6
[root@fannyganglia-monitor-core-2.5.6]# ./configure \
>CFLAGS="-I/usr/local/rrdtool-1.0.48/include" \
>CPPFLAGS="-I/usr/local/rrdtool-1.0.48/include" \
>LDFLAGS="-L/usr/local/rrdtool-1.0.48/lib" --with-gmetad
...
[root@fanny ganglia-monitor-core-2.5.6]# make
...
[root@fanny ganglia-monitor-core-2.5.6]# make install
...
```

As you can see, this is a pretty standard install with a couple of small exceptions. First, you'll need to tell configure where to find the *RRDtool* to include file and library by setting the various flags as shown above. Second, you'll need to explicitly tell configure to build *gmetad*. This is done with the `--with-gmetad` option.

Once you've built the software, you'll need to install and configure it. Both *gmond* and *gmetad* have very simple configuration files. The sample files *gmond/gmond.conf* and *gmetad/gmetad.conf* are included as part of the source tree. You should copy these to */etc* and edit them before you start either program. The sample files are well documented and straightforward to edit. Most defaults are reasonable. Strictly speaking, the *gmond.conf* file is not necessary if you are happy with the defaults. However, you will probably want to update the cluster information at a minimum. The *gmetad.conf* file must be present and you'll need to identify at least one data source. You may also want to change the identity information in it.

For *gmetad.conf*, the data source entry is a list of the machines that will be monitored. The format is the identifier `data_source` followed by a unique string identifying the cluster. Next is an optional polling interval. Finally, there is a list of machines and optional port numbers. Here is a simple example:

```
data_source "my cluster" 10.0.32.144 10.0.32.145 10.0.32.146 10.0.32.147
```


If you see output such as this, everything is up and running. (Since you are going to the *localhost*, this should work even if your firewall is blocking TELNET.)

10.2.1.4 Web frontend

The final step in setting up the monitoring station is to install the frontend software. This is just a matter of downloading the appropriate file and unpacking it. Keep in mind that you must install this so that it is reachable as part of your website. Examine the `DocumentRoot` in your Apache configuration file and install the package under this directory. For example,

```
[root@fanny root]# grep DocumentRoot /etc/httpd/conf/httpd.conf
```

```
...
```

```
DocumentRoot "/var/www/html"
```

```
...
```

Now that you know where the document root is, copy the web frontend to this directory and unpack it.

```
[root@fanny root]# cp ganglia-webfrontend-2.5.5.tar.gz /var/www/html/
```

```
[root@fanny root]# cd /var/www/html
```

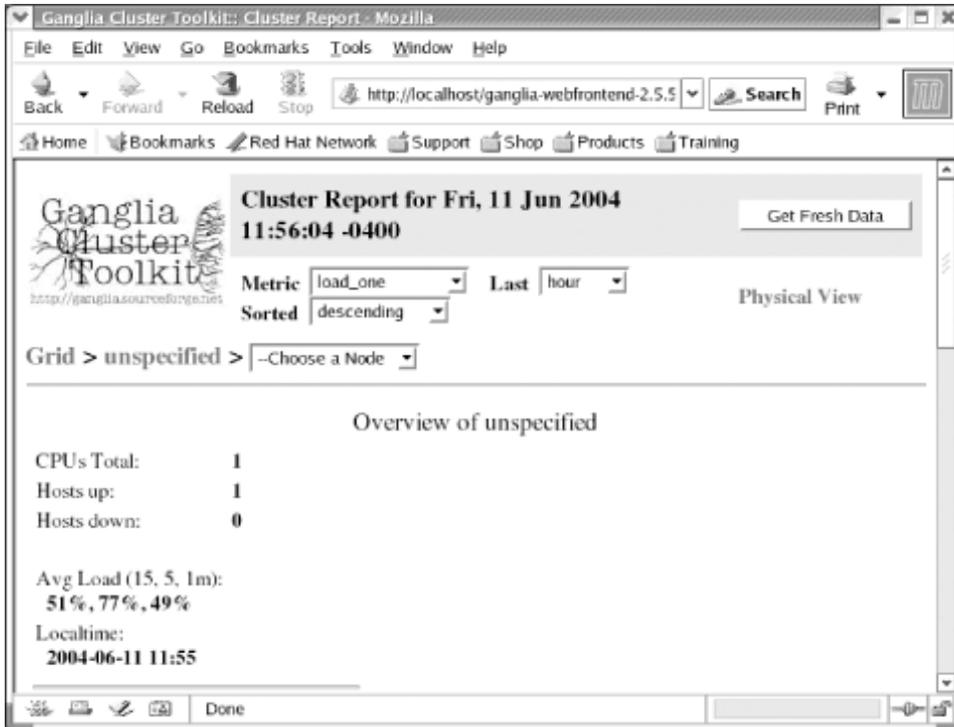
```
[root@fanny html]# gunzip ganglia-webfrontend-2.5.5.tar.gz
```

```
[root@fanny html]# tar -xvf ganglia-webfrontend-2.5.5.tar
```

There is nothing to build in this case. The configuration file is *conf.php*. Among other things, you can use this to change the appearance of your web site by changing the display themes.

At this point, you should be able to examine the state of this machine. (You'll still need to install *gmond* on the individual nodes before you can look at the rest of the cluster.) Start your web browser and visit your site, e.g., <http://localhost/ganglia-webfrontend-2.5.5/>. You should see something like [Figure 10-1](#).

Figure 10-1. Ganglia on a single node



This shows the host is up. Next, we need to install *gmond* on the individual nodes so we can see the rest of the cluster. You could use the same technique used above—just skip over the prerequisites and the *gmetad* steps. But it is much easier to use RPM. Just download the package to an appropriate location and install it. For example,

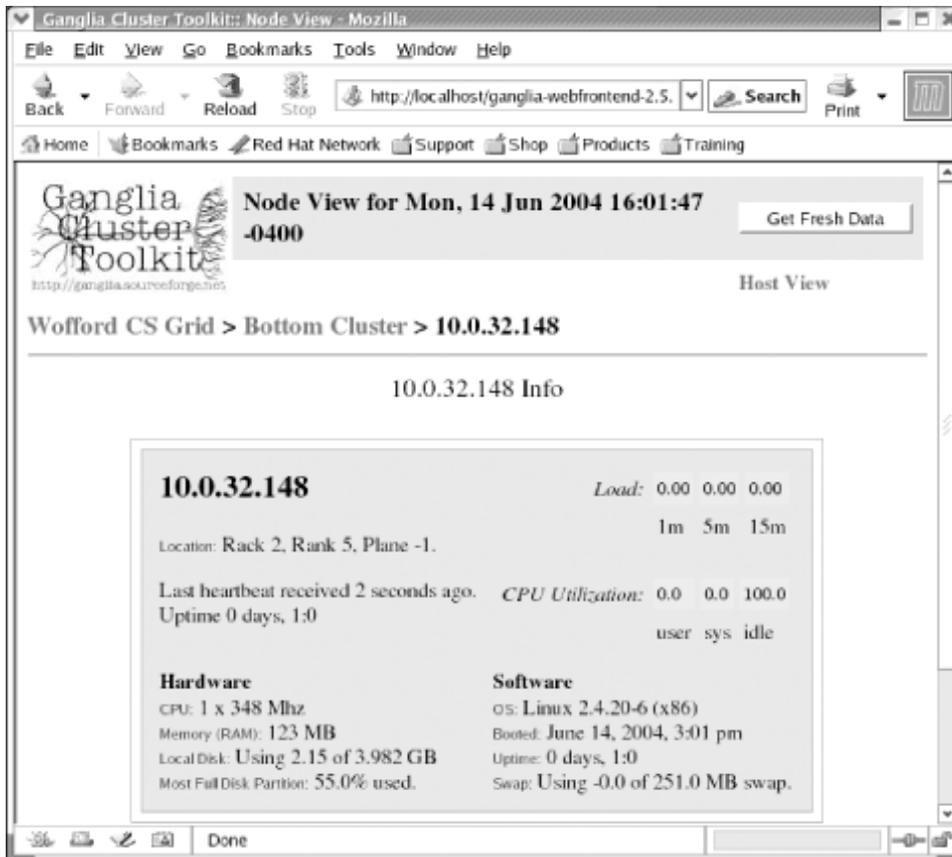
```
[root@george root]# rpm -vih ganglia-monitor-core-gmond-2.5.6-1.i386.rpm
Preparing...                               ##### [100%]
 1:ganglia-monitor-core-gm##### [100%]
Starting GANGLIA gmond: [ OK ]
```

gmond is installed in */usr/sbin* and its configuration file in */etc*. Once you've installed *gmond* on a machine, it should appear on your web page when you click on refresh. Repeat the installation for your remaining nodes.

Once you have Ganglia running, you may want to revisit the configuration files. With Ganglia running, it will be easier to see exactly what effect a change to a configuration file has. Of course, if you change a configuration file, you'll need to restart the appropriate services before you will see anything different.

You should have no difficulty figuring out how to use Ganglia. There are lots of "hot spots" on the pages, so just click and see what you get. The first page will tell you how many machines are up and down and their loads. You can select a physical view or collect information on individual machines. [Figure 10-2](#) shows information for an individual machine. You can also change the metric displayed. However, not all metrics are supported. The Ganglia documentation supplies a list of supported metrics by architecture.

Figure 10-2. Ganglia Node View



As you can see, these screen captures were made when the cluster was not otherwise in use. Otherwise the highlighted load figures would reflect that activity.

10.3 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](#).)

PBS is a powerful and versatile system. While this chapter sticks to the basics, you should keep in mind that there is a lot more to PBS than described here. Look at the *Administrator Guide* to learn more, particularly if you need help with more advanced features.

11.1 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.

11.1.1 Architecture

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.

The *pbs_server* daemon, the job server, runs on the server system and is the heart of the PBS system. It provides basic batch services such as receiving and creating batch jobs, modifying the jobs, protecting jobs against crashes, and running the batch jobs. User commands and the other daemons communicate with the *pbs_server* over the network using TCP. The user commands need not be installed on the server.

The job server manages one or more queues. (Despite the name, queues are not restricted to first-in, first-out scheduling.) A scheduled job waiting to be run or a job that is actually running is said to be a member of its queue. The job server supports two types of queues, execution and routing. A job in an execution queue is waiting to execute while a job in a routing queue is waiting to be routed to a new destination for execution.

The *pbs_mom* daemon executes the individual batch jobs. This job executor daemon is often called the *MOM* because it is the "mother" of all executing jobs and must run on every system within the cluster. It creates an execution environment that is as nearly identical to the user's session as possible. MOM is also responsible for returning the job's output to the user.

The final daemon, *pbs_sched*, implements the cluster's job-scheduling policy. As such, it communicates with the *pbs_server* and *pbs_mom* daemons to match available jobs with available resources. By default, a first-in, first-out scheduling policy is used, but you are free to set your own policies. The scheduler is highly extensible.

PBS provides both a GUI interface as well as 1003.2d-compliant command-line utilities. These commands fall into three categories: management, operator, and user commands. Management and operator commands are usually restricted commands. The commands are used to submit, modify, delete, and monitor batch jobs.

11.1.2 Installing OpenPBS

While detailed installation directions can be found in the *PBS Administrator Guide*, there are enough "gotchas" that it is worth going over the process in some detail. Before you begin, be sure you look over the *Administrator Guide* as well. Between the guide and this chapter, you should be able to overcome most obstacles.

Before starting with the installation proper, there are a couple of things you need to check. As noted, PBS provides both command-line utilities and a graphical interface. The graphical interface requires Tcl/Tk 8.0 or later, so if you want to use it, make sure Tcl/Tk is installed. You'll want to install Tcl/Tk before you install PBS. For a Red Hat installation, you can install Tcl/Tk from the packages supplied with the operating system. For more information on Tcl/Tk, visit the web site <http://www.scriptics.com/>. In order to build the GUI, you'll also need the X11 development packages, which Red Hat users can install from the supplied RPMs.

The first step in the installation proper is to download the software. Go to the OpenPBS web site (<http://www-unix.mcs.anl.gov/openpbs/>) and follow the links to the download page. The first time through, you will be redirected to a registration page. With registration, you will receive by email an account name and password that you can use to access the actual download page. Since you have to wait for approval before you receive the account information, you'll want to plan ahead and register a couple of days before you plan to download and install the software. Making your way through the registration process is a little annoying because it keeps pushing the commercial product, but it is straightforward and won't take more than a few minutes.

Once you reach the download page, you'll have the choice of downloading a pair of RPMs or the patched source code. The first RPM contains the full PBS distribution and is used to set up the server, and the second contains just the software needed by the client and is used to set up compute nodes within a cluster. While RPMs might seem the easiest way to go, the available RPMs are based on an older version of Tcl/Tk (Version 8.0). So unless you want to backpedal—i.e., track down and install these older packages, a nontrivial task—installing the source is preferable. That's what's described here.

Download the source and move it to your directory of choice. With a typical installation, you'll end up with three directory trees—the source tree, the installation tree, and the working directory tree. In this example, I'm setting up the source tree in the directory `/usr/local/src`. Once you have the source package where you want it, unpack the code.

```
[root@fanny src]# gunzip OpenPBS_2_3_16.tar.gz
```

```
[root@fanny src]# tar -vxpzf OpenPBS_2_3_16.tar
```

When untarring the package, use the `-p` option to preserve permissions bits.

Since the OpenPBS code is no longer supported, it is somewhat brittle. Before you can compile the code, you will need to apply some patches. What you install will depend on your configuration, so plan to spend some time on the Internet: the OpenPBS URL given above is a good place to start. For Red Hat Linux 9.0, start by downloading the scaling patch from <http://www-unix.mcs.anl.gov/openpbs/> and the *errno* and *gcc* patches from <http://bellatrix.pcl.ox.ac.uk/~ben/pbs/>. (Working out the details of what you need is the annoying side of installing OpenPBS.) Once you have the patches you want, install them.

```
[root@fanny src]# cp openpbs-gcc32.patch /usr/local/src/OpenPBS_2_3_16/
```

```

[root@fanny src]# cp openpbs-errno.patch /usr/local/src/OpenPBS_2_3_16/
[root@fanny src]# cp ncsa_scaling.patch /usr/local/src/OpenPBS_2_3_16/
[root@fanny src]# cd /usr/local/src/OpenPBS_2_3_16/
[root@fanny OpenPBS_2_3_16]# patch -p1 -b < openpbs-gcc32.patch
patching file buildutils/exclude_script
[root@fanny OpenPBS_2_3_16]# patch -p1 -b < openpbs-errno.patch
patching file src/lib/Liblog/pbs_log.c
patching file src/scheduler.basl/af_resmom.c
[root@fanny OpenPBS_2_3_16]# patch -p1 -b < ncsa_scaling.patch
patching file src/include/acct.h
patching file src/include/cmds.h
patching file src/include/pbs_ifl.h
patching file src/include/qmgr.h
patching file src/include/server_limits.h

```

The scaling patch changes built-in limits that prevent OpenPBS from working with larger clusters. The other patches correct problems resulting from recent changes to the *gcc* compiler.^[1]

[1] Even with the patches, I found it necessary to manually edit the file *srv_connect.c*, adding the line `#include<error.h>` with the other `#include` lines in the file. If you have this problem, you'll know because *make* will fail when referencing this file. Just add the line and remake the file.

As noted, you'll want to keep the installation directory separate from the source tree, so create a new directory for PBS. */usr/local/OpenPBS* is a likely choice. Change to this directory and run *configure*, *make*, *make install*, and *make clean* from it.

```

[root@fanny src]# mkdir /usr/local/OpenPBS
[root@fanny src]# cd /usr/local/OpenPBS
[root@fanny OpenPBS]# /usr/local/src/OpenPBS_2_3_16/configure \
>--set-default-server=fanny --enable-docs --with-scp
...
[root@fanny OpenPBS]# cd /usr/local/src/OpenPBS_2_3_16/
[root@fanny OpenPBS-2.3.16]# make
...
[root@fanny OpenPBS-2.3.16]# /usr/local/src/OpenPBS
[root@fanny OpenPBS]# make install

```

...

```
[root@fanny OpenPBS]# make clean
```

...

In this example, the configuration options set *fanny* as the server, create the documentation, and use *scp* (SSH secure copy program) when moving files between remote hosts. Normally, you'll create the documentation only on the server. The *Administrator Guide* contains several pages of additional options.

By default, the procedure builds all the software. For the compute nodes, this really isn't necessary since all you need is *pbs_mom* on these machines. Thus, there are several alternatives that you might want to consider when setting up the clients. You could just go ahead and build everything like you did for the server, or you could use different build options to restrict what is built. For example, the option `--disable-server` prevents the *pbs_server* daemon from being built. Or you could build and then install just *pbs_mom* and the files it needs. To do this, change to the MOM subdirectory, in this example */usr/local/OpenPBS/src/resmom*, and run *make install* to install just MOM.

```
[root@ida OpenPBS]# cd /usr/local/OpenPBS/src/resmom
```

```
[root@ida resmom]# make install
```

...

Yet another possibility is to use NFS to mount the appropriate directories on the client machines. The *Administrator Guide* outlines these alternatives but doesn't provide many details. Whatever your approach, you'll need *pbs_mom* on every compute node.

The *make install* step will create the */usr/spool/PBS* working directory, and will install the user commands in */usr/local/bin* and the daemons and administrative commands in */usr/local/sbin*. *make clean* removes unneeded files.

11.1.3 Configuring PBS

Before you can use PBS, you'll need to create or edit the appropriate configuration files, located in the working directory, e.g., */usr/spool/PBS*, or its subdirectories. First, the server needs the node file, a file listing the machines it will communicate with. This file provides the list of nodes used at startup. (This list can be altered dynamically with the *qmgr* command.) In the subdirectory *server_priv*, create the file *nodes* with the editor of your choice. The nodes file should have one entry per line with the names of the machines in your cluster. (This file can contain additional information, but this is enough to get you started.) If this file does not exist, the server will know only about itself.

MOM will need the configuration file *config*, located in the subdirectory *mom_priv*. At a minimum, you need an entry to start logging and an entry to identify the server to MOM. For example, your file might look something like this:

```
$logevent 0x1ff
```

```
$clienthost fanny
```

The argument to *\$logevent* is a mask that determines what is logged. A value of `0x0ff` will log all events excluding debug messages, while a value of `0x1ff` will log all events including debug messages. You'll need this file on every machine. There are a number of other options,

such as creating an access list.

Finally, you'll want to create a *default_server* file in the working directory with the fully qualified domain name of the machine running the server daemon.

PBS uses ports 15001-15004 by default, so it is essential that your firewall doesn't block these ports. These can be changed by editing the */etc/services* file. A full list of services and ports can be found in the *Administrator Guide* (along with other configuration options). If you decide to change ports, it is essential that you do this consistently across your cluster!

Once you have the configuration files in place, the next step is to start the appropriate daemons, which must be started as root. The first time through, you'll want to start these manually. Once you are convinced that everything is working the way you want, configure the daemons to start automatically when the systems boot by adding them to the appropriate startup file, such as */etc/rc.d/rc.local*. All three daemons must be started on the server, but the *pbs_mom* is the only daemon needed on the compute nodes. It is best to start *pbs_mom* before you start the *pbs_server* so that it can respond to the server's polling.

Typically, no options are needed for *pbs_mom*. The first time (and only the first time) you run *pbs_server*, start it with the option `-t create`.

```
[root@fanny OpenPBS]# pbs_server -t create
```

This option is used to create a new server database. Unlike *pbs_mom* and *pbs_sched*, *pbs_server* can be configured dynamically after it has been started.

The options to *pbs_sched* will depend on your site's scheduling policies. For the default FIFO scheduler, no options are required. For a more detailed discussion of command-line options, see the manpages for each daemon.

11.1.4 Managing PBS

We'll begin by looking at the command-line utilities first since the GUI may not always be available. Once you have mastered these commands, using the GUI should be straightforward. From a manager's perspective, the first command you'll want to become familiar with is *qmgr*, the queue management command. *qmgr* is used to create job queues and manage their properties. It is also used to manage nodes and servers providing an interface to the batch system. In this section we'll look at a few basic examples rather than try to be exhaustive.

First, identify the *pbs_server* managers, i.e., the users who are allowed to reconfigure the batch system. This is generally a one-time task. (Keep in mind that not all commands require administrative privileges. Subcommands such as the *list* and *print* can be executed by all users.) Run the *qmgr* command as follows, substituting your username:

```
[root@fanny OpenPBS]# qmgr
```

```
Max open servers: 4
```

```
Qmgr:set server managers=sloanjd@fanny.wofford.int
```

```
Qmgr:quit
```

You can specify multiple managers by adding their names to the end of the command, separated by commas. Once done, you'll no longer need root privileges to manage PBS.

Your next task will be to create a queue. Let's look at an example.

```
[sloanjd@fanny PBS]$ qmgr

Max open servers: 4

Qmgr:create queue workqueue

Qmgr:set queue workqueue queue_type = execution

Qmgr:set queue workqueue resources_max.cput = 24:00:00

Qmgr:set queue workqueue resources_min.cput = 00:00:01

Qmgr:set queue workqueue enabled = true

Qmgr:set queue workqueue started = true

Qmgr:set server scheduling = true

Qmgr:set server default_queue = workqueue

Qmgr:quit
```

In this example we have created a new queue named *workqueue*. We have limited CPU time to between 1 second and 24 hours. The queue has been enabled, started, and set as the default queue for the server, which must have at least one queue defined. All queues must have a type, be enabled, and be started.

As you can see from the example, the general form of a *qmgr* command line is a command (*active, create, delete, set, unset, list, or print*) followed by a target (*server, queue, or node*) followed by an attribute assignment. These keywords can be abbreviated as long as there is no ambiguity. In the first example in this section, we set a server attribute. In the second example, the target was the queue that we were creating for most of the commands.

To examine the configuration of the server, use the command

```
Qmgr:print server
```

This can be used to save the configuration you are using. Use the command

```
[root@fanny PBS]# qmgr -c "print server" > server.config
```

Note, that with the *-c* flag, *qmgr* commands can be entered on a single line. To re-create the queue at a later time, use the command

```
[root@fanny PBS]# qmgr < server.config
```

This can save a lot of typing or can be automated if needed. Other actions are described in the documentation.

Another useful command is *pbsnodes*, which lists the status of the nodes on your cluster.

```
[sloanjd@amy sloanjd]$ pbsnodes -a
```

```
oscarnode1.oscardomain
```

```
state = free
```

```
np = 1

properties = all

ntype = cluster

oscarnode2.oscardomain

state = free

np = 1

properties = all

ntype = cluster

...
```

On a large cluster, that can create a lot of output.

11.1.5 Using PBS

From the user's perspective, the place to start is the *qsub* command, which submits jobs. The only jobs that the *qsub* accepts are scripts, so you'll need to package your tasks appropriately. Here is a simple example script:

```
#!/bin/sh

#PBS -N demo

#PBS -o demo.txt

#PBS -e demo.txt

#PBS -q workq

#PBS -l mem=100mb

mpiexec -machinefile /etc/myhosts -np 4 /home/sloanjd/area/area
```

The first line specified the shell to use in interpreting the script, while the next few lines starting with **#PBS** are directives that are passed to PBS. The first names the job, the next two specify where output and error output go, the next to last identifies the queue that is used, and the last lists a resource that will be needed, in this case 100 MB of memory. The blank line signals the end of PBS directives. Lines that follow the blank line indicate the actual job.

Once you have created the batch script for your job, the *qsub* command is used to submit the job.

```
[sloanjd@amy area]$ qsub pbsdemo.sh
```

11.amy

When run, *qsub* returns the job identifier as shown. A number of different options are available, both as command-line arguments to *qsub* or as directives that can be included in the script. See the *qsub (1B)* manpage for more details.

There are several things you should be aware of when using *qsub*. First, as noted, it expects a script. Next, the target script cannot take any command-line arguments. Finally, the job is launched on one node. The script must ensure that any parallel processes are then launched on other nodes as needed.

In addition to *qsub*, there are a number of other useful commands available to the general user. The commands *qstat* and *qdel* can be used to manage jobs. In this example, *qstat* is used to determine what is on the queue:

```
[sloanjd@amy area]$ qstat
```

Job id	Name	User	Time Use	S	Queue
11.amy	pbsdemo	sloanjd	0	Q	workq
12.amy	pbsdemo	sloanjd	0	Q	workq

qdel is used to delete jobs as shown.

```
[sloanjd@amy area]$ qdel 11.amy
```

```
[sloanjd@amy area]$ qstat
```

Job id	Name	User	Time Use	S	Queue
12.amy	pbsdemo	sloanjd	0	Q	workq

qstat can be called with the job identifier to get more information about a particular job or with the *-s* option to get more details.

A few of the more useful ones include the following:

qalter

This is used to modify the attributes of an existing job.

qhold

This is used to place a hold on a job.

qmove

This is used to move a job from one queue to another.

qorder

This is used to change the order of two jobs.

qrun

This is used to force a server to start a job.

If you start with the *qsub (1B)* manpage, other available commands are listed in the "See Also" section.

Figure 11-1. xpbs -admin

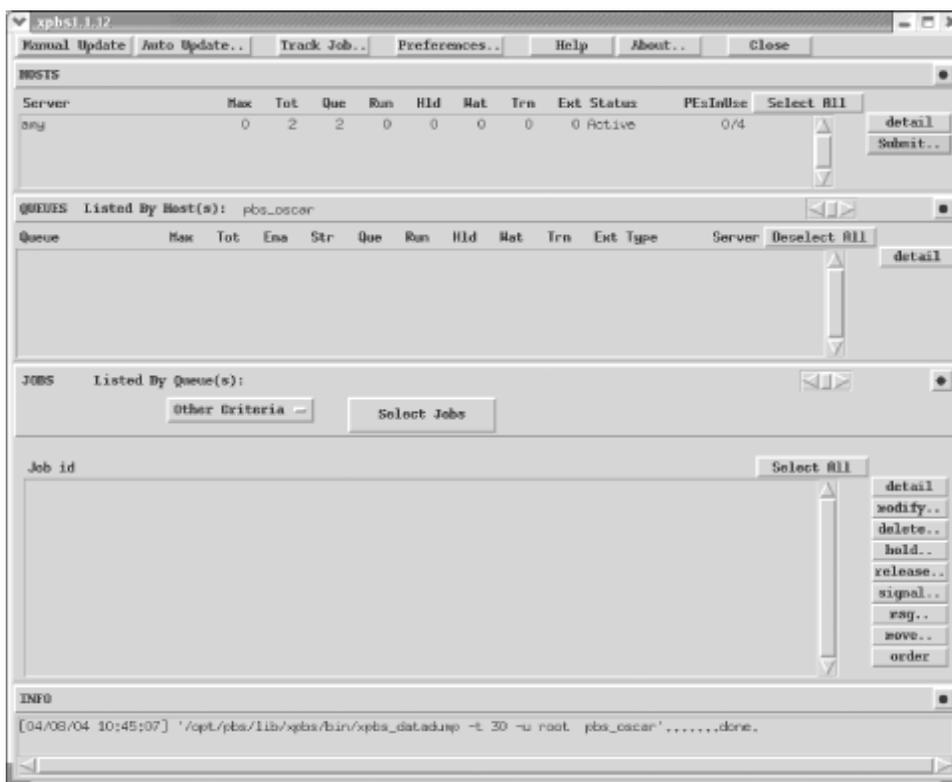
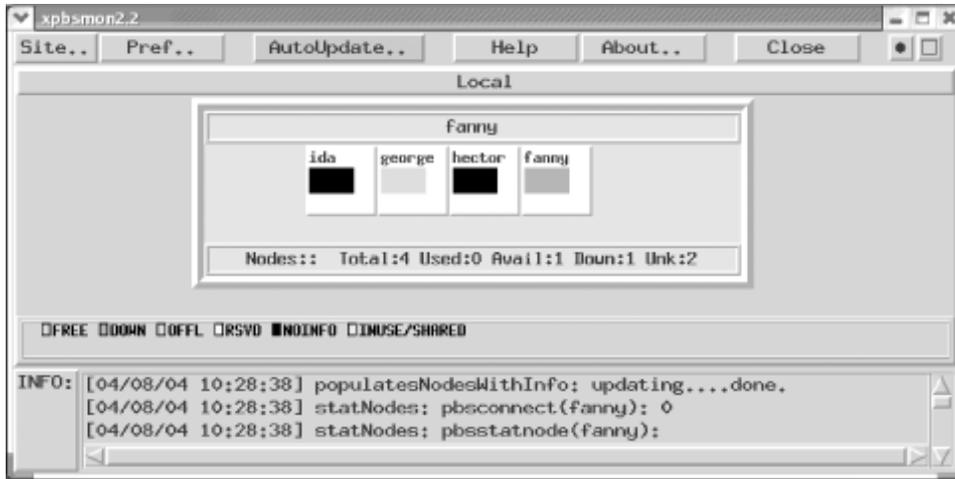


Figure 11-2. xpbsmon



11.1.6 PBS's GUI

PBS provides two GUIs for queue management. The command *xpbs* will start a general interface. If you need to do administrative tasks, you should include the argument *-admin*. [Figure 11-1](#) shows the *xpbs* GUI with the *-admin* option. Without this option, the general appearance is the same, but a number of buttons are missing. You can terminate a server; start, stop, enable, or disable a queue; or run or rerun a job. To monitor nodes in your cluster, you can use the *xpbsmon* command, shown for a few machines in [Figure 11-2](#).

11.1.7 Maui Scheduler

If you need to go beyond the schedulers supplied with PBS, you should consider installing Maui. In a sense, Maui picks up where PBS leaves off. It is an external scheduler—that is, it does not include a resource manager. Rather, it can be used in conjunction with a resource manager such as PBS to extend the resource manager's capabilities. In addition to PBS, Maui works with a number of other resource managers.

Maui controls how, when, and where jobs will be run and can be described as a policy engine. When used correctly, it can provide extremely high system utilization and should be considered for any large or heavily utilized cluster that needs to optimize throughput. Maui provides a number of very advanced scheduling options. Administration is through the master configuration file *maui.cfg* and through either a text-based or a web-based interface.

Maui is installed by default as part of OSCAR and Rocks. For the most recent version of Maui or for further documentation, you should visit the Maui web site, <http://www.supercluster.org>.

11.2 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.

A parallel filesystem might not provide optimal performance for serial programs or single tasks. Because high-performance filesystems are designed for a different purpose, they should not be thought of as replacements for NFS. Rather, they complement the functionality provided by NFS. Many clusters benefit from both NFS and a high-performance filesystem.

There's more good news. If you need a high-performance filesystem, there are a number of alternatives. If you have very deep pockets, you can go for hardware-based solutions. With *network attached storage (NAS)*, a dedicated server is set up to service file requests for the network. In a sense, NAS owns the filesystem. Since serving files is NAS's only role, NAS servers tend to be highly optimized file servers. But because these are still traditional servers, latency can still be a problem.

The next step up is a *storage area network (SAN)*. Typically, a SAN provides direct block-level access to the physical hardware. A SAN typically includes high-performance networking as well. Traditionally, SANs use fibre channel (FC) technology. More recently, IP-based storage technologies that operate at the block level have begun to emerge. This allows the creation of a SAN using more familiar IP-based technologies.

Because of the high cost of hardware-based solutions, they are outside the scope of this book. Fortunately, there are also a number of software-based filesystems for clusters, each with its own set of features and limitations. While many of the following might not be considered a high-performance filesystem, you might consider one of the following, depending upon your needs. However, you should be very careful before adopting any of these. Like most software,

these should be regarded as works in progress. While they may be ideal for some uses, they may be problematic for others. Caveat emptor! These packages are generally available as both source tar balls and as RPMs.

ClusterNFS

This is a set of patches for the NFS server daemon. The clients run standard NFS software. The patches allow multiple diskless clients to mount the same root filesystem by "reinterpreting" file names. ClusterNFS is often used with Mosix. If you are building a diskless cluster, this is a package you might want to consider (<http://clusternfs.sourceforge.net/>).

Coda

Coda is a distributed filesystem developed at Carnegie Mellon University. It is derived from the Andrew File System. Coda has many interesting features such as performance enhancement through client side persistent caching, bandwidth adaptation, and robust behavior with partial network failures. It is a well documented, ongoing project. While it may be too early to use Coda with large, critical systems, this is definitely a distributed filesystem worth watching (<http://www.coda.cs.cmu.edu/index.html>).

InterMezzo

This distributed filesystem from CMU was inspired by Coda. InterMezzo is designed for use with high-availability clusters. Among other features, it offers automatic recovery from network outages (<http://www.inter-mezzo.org/>).

Lustre

Lustre is a cluster filesystem designed to work with very large clusters—up to 10,000 nodes. It was developed and is maintained by Cluster File Systems, Inc. and is available under a GPL. Since Lustre patches the kernel, you'll need to be running a 2.4.X kernel (<http://www.lustre.org/>).

OpenAFS

The Andrew File System was originally created at CMU and now developed and supported by IBM. OpenAFS is source fork released by IBM. It provides scalable client-server-based architecture with transparent data migration. Consider OpenAFS a potential replacement for NFS (<http://www.openafs.org/>).

Parallel Virtual File System (PVFS)

PVFS provides high-performance, parallel filesystem. The remainder of this chapter describes PVFS in detail (<http://www.parl.clemson.edu/pvfs/>).

This is only a partial listing of what is available. If you are looking to implement a SAN, you might consider *Open Global File System (OpenGFS)* (<http://opengfs.sourceforge.net/>). Red Hat markets a commercial, enterprise version of OpenGFS. If you are using IBM hardware, you might want to look into *General Parallel File System (GPFS)* (

1.ibm.com/servers/eserver/clusters/software/gpfs.html). In this chapter we will look more closely at PVFS, an open source, high-performance filesystem available for both Rocks and OSCAR.

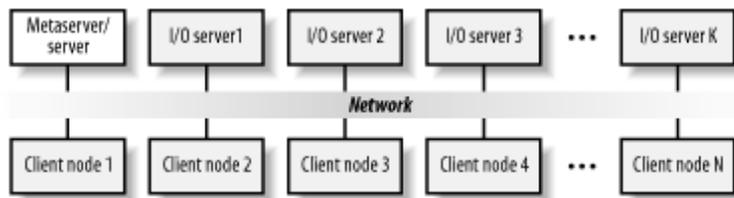
12.1 PVFS

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.

The remaining nodes are the client nodes. These are the actual compute nodes within the clusters, i.e., where the parallel jobs execute. With PVFS, client nodes and I/O servers can overlap. For a small cluster, it may make sense for all nodes to be both client and I/O nodes. Similarly, the metadata server can also be an I/O server or client node, or both. Once you start writing data to these machines, it is difficult to change the configuration of your system. So give some thought to what you need.

12.1.1 Installing PVFS on the Head Node

Installing and configuring PVFS is more complicated than most of the other software described in this book for a couple of reasons. First, you will need to decide how to partition your cluster. That is, you must decide which machine will be the metadata server, which machines will be clients, and which machines will be I/O servers. For each type of machine, there is

different software to install and a different configuration. If a machine is going to be both a client and an I/O server, it must be configured for each role. Second, in order to limit the overhead of accessing the filesystem through the kernel, a kernel module is used. This may entail further tasks such as making sure the appropriate kernel header files are available or patching the code to account for differences among Linux kernels.

This chapter describes a simple configuration where *fanny* is the metadata server, a client, and an I/O server, and all the remaining nodes are both clients and I/O servers. As such, it should provide a fairly complete idea about how PVFS is set up. If you are configuring your cluster differently, you won't need to do as much. For example, if some of your nodes are only I/O nodes, you can skip the client configuration steps on those machines.

In this example, the files are downloaded, compiled, and installed on *fanny* since *fanny* plays all three roles. Once the software is installed on *fanny*, the appropriate pieces are pushed to the remaining machines in the cluster.

The first step, then, is to download the appropriate software. To download PVFS, first go to the PVFS home page (<http://www.parl.clemson.edu/pvfs/>) and follow the link to files. This site has links to several download sites. (You'll want to download the documentation from this site before moving on to the software download sites.) There are two tar archives to download: the sources for PVFS and for the kernel module.

You should also look around for any patches you might need. For example, at the time this was written, because of customizations to the kernel, the current version of PVFS would not compile correctly under Red Hat 9.0. Fortunately, a patch from <http://www.mcs.anl.gov/~robl/pvfs/redhat-ntpl-fix.patch.gz> was available.^[1] Other patches may also be available.

^[1] Despite the URL, this was an uncompressed text file at the time this was written.

Once you have the files, copy the files to an appropriate directory and unpack them.

```
[root@fanny src]# gunzip pvfs-1.6.2.tgz
[root@fanny src]# gunzip pvfs-kernel-1.6.2-linux-2.4.tgz
[root@fanny src]# tar -xvf pvfs-1.6.2.tar
...
[root@fanny src]# tar -xvf pvfs-kernel-1.6.2-linux-2.4.tar
...
```

It is simpler if you install these under the same directory. In this example, the directory `/usr/local/src` is used. In the documentation that comes with PVFS, a link was created to the first directory.

```
[root@fanny src]# ln -s pvfs-1.6.0 pvfs
```

This will save a little typing but isn't essential.



Be sure to look at the README and INSTALL files that come with the sources.

Next, apply any patches you may need. As noted, with this version the kernel module sources need to be patched.

```
[root@fanny src]# mv redhat-ntpl-fix.patch pvfs-kernel-1.6.2-linux-2.4/
[root@fanny src]# cd pvfs-kernel-1.6.2-linux-2.4
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# patch -p1 -b <
\>redhat-ntpl-fix.patch
patching file config.h.in
patching file configure
patching file configure.in
patching file kpvfsc.c
patching file kpvfscdev.c
patching file pvfscdev.c
patching file pvfscdev.c
```

Apply any other patches that might be needed.

The next steps are compiling PVFS and the PVFS kernel module. Here are the steps for compiling PVFS:

```
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# cd /usr/local/src/pvfs
[root@fanny pvfs]# ./configure
...
[root@fanny pvfs]# make
...
[root@fanny pvfs]# make install
...
```

There is nothing new here.

Next, repeat the process with the kernel module.

```
[root@fanny src]# cd /usr/local/src/pvfs-kernel-1.6.2-linux-2.4
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# ./configure
...
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# make
...
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# make install
install -c -d /usr/local/sbin
```

```
install -c mount.pvfs /usr/local/sbin
```

```
install -c pvfsd /usr/local/sbin
```

NOTE: pvfs.o must be installed by hand!

NOTE: install mount.pvfs by hand to /sbin if you want 'mount -t pvfs' to work

This should go very quickly.

As you see from the output, the installation for the kernel requires some additional manual steps. Specifically, you need to decide where you want to put the kernel module. The following works for Red Hat 9.0.

```
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# mkdir \  
>/lib/modules/2.4.20-6/kernel/fs/pvfs  
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# cp pvfs.o \  
>/lib/modules/2.4.20-6/kernel/fs/pvfs/pvfs.o
```

If you are doing something different, you may need to poke around a bit to find the right location.

12.1.2 Configuring the Metadata Server

If you have been following along, at this point you should have all the software installed on the head node, i.e., the node that will function as the metadata server for the filesystem. The next step is to finish configuring the metadata server. Once this is done, the I/O server and client software can be installed and configured.

Configuring the meta-server is straightforward. First, create a directory to store filesystem data.

```
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# mkdir /pvfs-meta
```

Keep in mind, this directory is used to store information about the PVFS filesystem. The actual data is not stored in this directory. Once PVFS is running, you can ignore this directory.

Next, create the two metadata configuration files and place them in this directory. Fortunately, PVFS provides a script to simplify the process.

```
[root@fanny pvfs-kernel-1.6.2-linux-2.4]# cd /pvfs-meta  
[root@fanny pvfs-meta]# /usr/local/bin/mkmgconf
```

This script will make the .iodtab and .pvfsdir files
in the metadata directory of a PVFS file system.

Enter the root directory (metadata directory):

```
/pvfs-meta/
```

Enter the user id of directory:

root

Enter the group id of directory:

root

Enter the mode of the root directory:

777

Enter the hostname that will run the manager:

fanny

Searching for host...success

Enter the port number on the host for manager:

(Port number 3000 is the default)

3000

Enter the I/O nodes: (can use form node1, node2, ... or

nodename{#-#, #, #})

fannygeorgehectoridajames

Searching for hosts...success

I/O nodes: fanny george hector ida james

Enter the port number for the iods:

(Port number 7000 is the default)

7000

Done!

Running this script creates the two configuration files *.pvfsdir* and *.iodtab*. The file *.pvfsdir* contains permission information for the metadata directory. Here is the file the *mkmgrconf* script creates when run as shown.

84230

0

0

0040777

3000

fanny

```
/pvfs-meta/
```

```
/
```

The first entry is the inode number of the configuration file. The remaining entries correspond to the questions answered earlier.

The file `.iodtab` is a list of the I/O servers and their port numbers. For this example, it should look like this:

```
fanny:7000
```

```
george:7000
```

```
hector:7000
```

```
ida:7000
```

```
james:7000
```

Systems can be listed by name or by IP number. If the default port (7000) is used, it can be omitted from the file.



The `.iodtab` file is an ordered list of I/O servers. Once PVFS is running, you should not change the `.iodtab` file. Otherwise, you will almost certainly render existing PVFS files inaccessible.

12.1.3 I/O Server Setup

To set up the I/O servers, you need to create a data directory on the appropriate machines, create a configuration file, and then push the configuration file, along with the other I/O server software, to the appropriate machines. In this example, all the nodes in the cluster including the head node are I/O servers.

The first step is to create a directory with the appropriate ownership and permissions on all the I/O servers. We start with the head node.

```
[root@fanny /]# mkdir /pvfs-data
```

```
[root@fanny /]# chmod 700 /pvfs-data
```

```
[root@fanny /]# chown nobody.nobody /pvfs-data
```

Keep in mind that these directories are where the actual pieces of a data file will be stored. However, you will not access this data in these directories directly. That is done through the filesystem at the appropriate mount point. These PVFS data directories, like the meta-server's `metadata` directory, can be ignored once PVFS is running.

Next, create the configuration file `/etc/iod.conf` using your favorite text editor. (This is optional, but recommended.) `iod.conf` describes the `iod` environment. Every line, apart from comments, consists of a key and a corresponding value. Here is a simple example:

```
# iod.conf-iod configuration file
```

```
datadir /pvfs-data

user nobody

group nobody

logdir /tmp

rootdir /

debug 0
```

As you can see, this specifies a directory for the data, the user and group under which the I/O daemon *iod* will run, the log and root directories, and a debug level. You can also specify other parameters such as the port and buffer information. In general, the defaults are reasonable, but you may want to revisit this file when fine-tuning your system.

While this takes care of the head node, the process must be repeated for each of the remaining I/O servers. First, create the directory and configuration file for each of the remaining I/O servers. Here is an example using the C3 utilities. (C3 is described in [Chapter 10](#).)

```
[root@fanny /]# cexec mkdir /pvfs-data

...

[root@fanny /]# cexec chmod 700 /pvfs-data

...

[root@fanny /]# cexec chown nobody.nobody /pvfs-data

...

[root@fanny /]# cpush /etc/iod.conf

...
```

Since the configuration file is the same, it's probably quicker to copy it to each machine, as shown here, rather than re-create it.

Finally, since the *iod* daemon was created only on the head node, you'll need to copy it to each of the remaining I/O servers.

```
[root@fanny root]# cpush /usr/local/sbin/iod

...
```

While this example uses C3's *cpush*, you can use whatever you are comfortable with.

If you aren't configuring every machine in your cluster to be an I/O server, you'll need to adapt these steps as appropriate for your cluster. This is easy to do with C3's range feature.

12.1.4 Client Setup

Client setup is a little more involved. For each client, you'll need to create a PVFS device file,

copy over the kernel module, create a mount point and a PVFS mount table, and copy over the appropriate executable along with any other utilities you might need on the client machine. In this example, all nodes including the head are configured as clients. But because we have already installed software on the head node, some of the steps aren't necessary for that particular machine.

First, a special character file needs to be created on each of the clients using the *mknod* command.

```
[root@fanny /]# cexec mknod /dev/pvfsd c 60 0
```

...

/dev/pvfsd is used to communicate between the *pvfsd* daemon and the kernel module *pvfs.o*. It allows programs to access PVFS files, once mounted, using traditional Unix filesystem semantics.

We will need to distribute both the kernel module and the daemon to each node.

```
[root@fanny /]# cpush /usr/local/sbin/pvfsd
```

...

```
[root@fanny /]# cexec mkdir /lib/modules/2.4.20-6/kernel/fs/pvfs/
```

...

```
[root@fanny /]# cpush /lib/modules/2.4.20-6/kernel/fs/pvfs/pvfs.o
```

...

The kernel module registers the filesystem with the kernel while the daemon performs network transfers.

Next, we need to create a mount point.

```
[root@fanny root]# mkdir /mnt/pvfs
```

```
[root@fanny /]# cexec mkdir /mnt/pvfs
```

...

This example uses */mnt/pvfs*, but */pvfs* is another frequently used alternative. The mount directory is where the files appear to be located. This is the directory you'll use to access or reference files.

The *mount.pvfs* executable is used to mount a filesystem using PVFS and should be copied to each client node.

```
[root@fanny /]# cpush /usr/local/sbin/mount.pvfs /sbin/
```

...

mount.pvfs can be invoked by the *mount* command on some systems, or it can be called directly.

Finally, create */etc/pvfstab*, a mount table for the PVFS system. This needs to contain only a

single line of information as shown here:

```
fanny:/pvfs-meta /mnt/pvfs pvfs port=3000 0 0
```

If you are familiar with */etc/fstab*, this should look very familiar. The first field is the path to the metadata information. The next field is the mount point. The third field is the filesystem type, which is followed by the port number. The last two fields, traditionally used to determine when a filesystem is dumped or checked, aren't currently used by PVFS. These fields should be zeros. You'll probably need to change the first two fields to match your cluster, but everything else should work as shown here.

Once you have created the mount table, push it to the remaining nodes.

```
[root@fanny /]# cpush /etc/pvfstab
...
[root@fanny /]# cexec chmod 644 /etc/pvfstab
...
```

Make sure the file is readable as shown.

While it isn't strictly necessary, there are some other files that you may want to push to your client nodes. The installation of PVFS puts a number of utilities in */usr/local/bin*. You'll need to push these to the clients before you'll be able to use them effectively. The most useful include *mgr-ping*, *iod-ping*, *pvstat*, and *u2p*.

```
[root@fanny root]# cpush /usr/local/bin/mgr-ping
...
[root@fanny root]# cpush /usr/local/bin/iod-ping
...
[root@fanny root]# cpush /usr/local/bin/pvstat
...
[root@fanny pvfs]# cpush /usr/local/bin/u2p
...
```

As you gain experience with PVFS, you may want to push other utilities across the cluster.

If you want to do program development using PVFS, you will need access to the PVFS header files and libraries and the *pvfstab* file. By default, header and library files are installed in */usr/local/include* and */usr/local/lib*, respectively. If you do program development only on your head node, you are in good shape. But if you do program development on any of your cluster nodes, you'll need to push these files to those nodes. (You might also want to push the manpages as well, which are installed in */usr/local/man*.)

12.1.5 Running PVFS

Finally, now that you have everything installed, you can start PVFS. You need to start the

appropriate daemons on the appropriate machines and load the kernel module. To load the kernel module, use the *insmod* command.

```
[root@fanny root]# insmod /lib/modules/2.4.20-6/kernel/fs/pvfs/pvfs.o
[root@fanny root]# cexec insmod /lib/modules/2.4.20-6/kernel/fs/pvfs/pvfs.o
...
```

Next, run the *mgr* daemon on the metadata server. This is the management daemon.

```
[root@fanny root]# /usr/local/sbin/mgr
```

On each I/O server, start the *iod* daemon.

```
[root@fanny root]# /usr/local/sbin/iod
[root@fanny root]# cexec /usr/local/sbin/iod
...
```

Next, start the *pvfsd* daemon on each client node.

```
[root@fanny root]# /usr/local/sbin/pvfsd
[root@fanny root]# cexec /usr/local/sbin/pvfsd
...
```

Finally, mount the filesystem on each client.

```
[root@fanny root]# /usr/local/sbin/mount.pvfs fanny:/pvfs-meta /mnt/pvfs
[root@fanny /]# cexec /sbin/mount.pvfs fanny:/pvfs-meta /mnt/pvfs
...
```

PVFS should be up and running.^[2]

^[2] Although not described here, you'll probably want to make the necessary changes to your startup file so that this is all done automatically. PVFS provides scripts *enablemgr* and *enableiod* for use with Red Hat machines.

To shut PVFS down, use the *umount* command to unmount the filesystem, e.g., *umount /mnt/pvfs*, stop the PVFS processes with *kill* or *killall*, and unload the *pvfs.o* module with the *rmmmod* command.

12.1.5.1 Troubleshooting

There are several things you can do to quickly check whether everything is running. Perhaps the simplest is to copy a file to the mounted directory and verify that it is accessible on other nodes. If you have problems, there are a couple of other things you might want to try to narrow things down.

First, use *ps* to ensure the daemons are running on the appropriate machines. For example,

```
[root@fanny root]# ps -aux | grep pvfsd
root      15679  0.0  0.1  1700  184 ?        S
Jun21    0:00 /usr/local/sbin/pvfsd
```

Of course, *mgr* should be running only on the metadata server and *iod* should be running on all the I/O servers (but nowhere else).

Each process will create a log file, by default in the `/tmp` directory. Look to see if these are present.

```
[root@fanny root]# ls -l /tmp
total 48
-rwxr-xr-x  1 root    root          354 Jun 21 11:13 iolog.OxLkSR
-rwxr-xr-x  1 root    root           0 Jun 21 11:12 mgrlog.z3tg11
-rwxr-xr-x  1 root    root          119 Jun 21 11:21 pvfsdlog.msBrCV
...
```

The garbage at the end of the filenames is generated to produce a unique filename.

The mounted PVFS will be included in the listing given with the `mount` command.

```
[root@fanny root]# mount
...
fanny:/pvfs-meta on /mnt/pvfs type pvfs (rw)
...
```

This should work on each node.

In addition to the fairly obvious tests just listed, PVFS provides a couple of utilities you can turn to. The utilities *iod-ping* and *mgr-ping* can be used to check whether the I/O and metadata servers are running and responding on a particular machine.

Here is an example of using *iod-ping*:

```
[root@fanny root]# /usr/local/bin/iod-ping
localhost:7000 is responding.

[root@fanny root]# cexec /usr/local/bin/iod-ping
***** local *****
----- george.wofford.int-----
localhost:7000 is responding.
----- hector.wofford.int-----
```

```
localhost:7000 is responding.
```

```
----- ida.wofford.int-----
```

```
localhost:7000 is responding.
```

```
----- james.wofford.int-----
```

```
localhost:7000 is responding.
```

The *iod* daemon seems to be OK on all the clients. If you run *mgr-ping*, only the metadata server should respond.

12.2 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

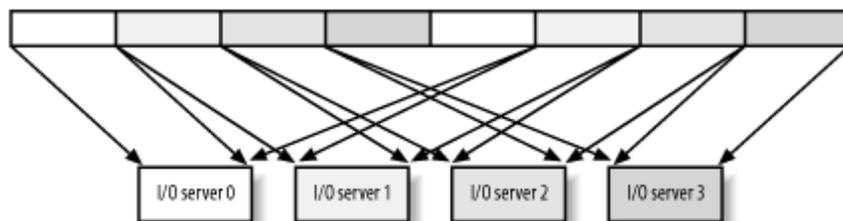
The number of I/O servers among which the file is partitioned. Typically, this defaults to the total number of I/O servers.

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 *u2p* utility. With *u2p*, the command-line option *-s* sets the stripe size; *-b* specifies the base; and *-n* specifies the number of nodes. Here is an example:

```
[root@fanny /]# u2p -s16384 data /mnt/data

1 node(s); ssize = 8192; buffer = 0; nanMBps (0 bytes total)

[root@fanny /]# pvstat /mnt/data

/mnt/data: base = 0, pcount = 1, ssize = 8192
```

Typically, *u2p* is used to convert an existing file for use with a parallel program.

While Unix system call *read* and *write* will work with the PVFS without any changes, large numbers of small accesses will not perform well. The buffered routines from the standard I/O library (e.g., *fread* and *fwrite*) should work better provided an adequate buffer is used.

To make optimal use of PVFS, you will need to write your programs to use PVFS explicitly. This can be done using the native PVFS access provided through the *libpvfs.a* library. Details can be found in *Using the Parallel Virtual File System*, part of the documentation available at the PVFS web site. Programming examples are included with the source in the *examples* subdirectory. Clearly, you should understand your application's data requirements before you begin programming.

Alternatively, PVFS can be used with the ROMIO interface from <http://www.mcs.anl.gov>. The ROMIO is included with both MPICH and LAM/MPI. (If you compile ROMIO, you need to specify PVFS support. Typically, you use the compile flags *-lib=/usr/local/lib/libpvfs.a* and *-file_system=pvfs+nfs+ufs*.) ROMIO provides two important optimizations, data sieving and two-phase I/O. Additional information is available at the ROMIO web site.

12.3 Notes for OSCAR and Rocks Users

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.

Part IV: Cluster Programming

The final section of this book describes programming tools. If you will be writing your own applications for your cluster, these chapters should get you started.

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.

13.1 MPI

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.

13.1.1 Core MPI

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.

```
#include "mpi.h"

#include <stdio.h>

int main( int argc, char * argv[ ] )
{
    int processId;        /* rank of process */
    int noProcesses;     /* number of processes */
    int nameSize;        /* length of name */
    char computerName[MPI_MAX_PROCESSOR_NAME];
```

```

MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);

MPI_Comm_rank(MPI_COMM_WORLD, &processId);

MPI_Get_processor_name(computerName, &nameSize);

fprintf(stderr, "Hello from process %d on %s\n", processId, computerName);

MPI_Finalize( );

return 0;
}

```

This example introduces five MPI functions, defined through the inclusion of the header file for the MPI library, *mpi.h*, and included when the MPI library is linked to the program. While this example uses C, similar libraries are available for C++ and FORTRAN.

Four of these functions, *MPI_Init*, *MPI_Comm_size*, *MPI_Comm_rank*, and *MPI_Finalize*, are seen in virtually every MPI program. We will look at each in turn. (Notice that all MPI identifiers begin with *MPI_*.)

13.1.1.1 MPI_Init

MPI_Init is used to initialize an MPI session. All MPI programs must have a call to *MPI_Init*. *MPI_Init* is called once, typically at the start of a program. You can have lots of other code before this call, or you can even call *MPI_Init* from a subroutine, but you should call it before any other MPI functions are called. (There is an exception: the function *MPI_Initialized* can be called before *MPI_Init*. *MPI_Initialized* is used to see if *MPI_Init* has been previously called.)

In C, *MPI_Init* can be called with the *addresses* for *argc* and *argv* as shown in the example. This allows the program to take advantage of command-line arguments. Alternatively, these addresses can be replaced with a *NULL*.

13.1.1.2 MPI_Finalize

MPI_Finalize is called to shut down MPI. *MPI_Finalize* should be the last MPI call made in a program. It is used to free memory, etc. It is the user's responsibility to ensure that all pending communications are complete before a process calls *MPI_Finalize*. You must write your code so that every process calls *MPI_Finalize*. Notice that there are no arguments.

13.1.1.3 MPI_Comm_size

This routine is used to determine the total number of processes running in a *communicator*

(the communications group for the processes being used). It takes the communicator as the first argument and the address of an integer variable used to return the number of processes. For example, if you are executing a program using five processes and the default communicator, the value returned by `MPI_Comm_size` will be five, the total number of processes being used. This is number of processes, but not necessarily the number of machines being used.

In the example, both `MPI_Comm_size` and `MPI_Comm_rank` used the default communicator, `MPI_COMM_WORLD`. This communicator includes all the processes available at initialization and is created automatically for you. Communicators are used to distinguish and group messages. As such, communicators provide a powerful encapsulation mechanism. While it is possible to create and manipulate your own communicators, the default communicator will probably satisfy most of your initial needs.

13.1.1.4 MPI_Comm_rank

`MPI_Comm_rank` is used to determine the rank of the current process within the communicator. `MPI_Comm_rank` takes a communicator as its first argument and the address of an integer variable is used to return the value of the rank.

Basically, each process is assigned a different process number or rank within a communicator. Ranks range from 0 to one less than the size returned by `MPI_Comm_size`. For example, if you are running a set of five processes, the individual processes will be numbered 0, 1, 2, 3, and 4. By examining its rank, a process can distinguish itself from other processes.

The values returned by `MPI_Comm_size` and `MPI_Comm_rank` are often used to divvy up a problem among processes. For example, suppose that for some problem you want to divide the work among five processors. This is a decision you make when you run your program; your choice is not coded into the program since it may not be known when the program is written. Once the program is running, it can call `MPI_Comm_size` to determine the number of processes attacking the problem. In this example, it would return five. Each of the five processes now knows that it needs to solve one fifth of the original problem (assuming you've written the code this way).

Next, each individual process can examine its rank to determine its role in the calculation. Continuing with the current example, each process needs to decide which fifth of the original problem to work on. This is where `MPI_Comm_rank` comes in. Since each process has a different rank, it can use its rank to select its role. For example, the process with rank 0 might work on the first part of the problem; the process with rank 1 will work on the second part of the problem, etc.

Of course, you can divide up the problem differently if you like. For example, the process with rank 0 might collect all the results from the other processes for the final report rather than participate in the actual calculation. Or each process could use its rank as an index to an array to discover what parameters to use in a calculation. It is really up to you as a programmer to determine how you want to use this information.

13.1.1.5 MPI_Get_processor_name

`MPI_Get_processor_name` is used to retrieve the host name of the node on which the individual process is running. In the sample program, we used it to display host names. The first argument is an array to store the name and the second is used to return the actual length of the name.

`MPI_Get_processor_name` is a nice function to have around, particularly when you want to debug code, but otherwise it isn't used all that much. The first four MPI functions, however, are core functions and will be used in virtually every MPI program you'll write. If you drop the relevant declarations, the call to `MPI_Get_processor_name`, and the `fprintf`, you'll have a

template that you can use when writing MPI programs.

Although we haven't used it, each of the C versions of these five functions returns an integer error code. With a few exceptions, the actual code is left up to the implementers. Error codes can be translated into meaningful messages using the `MPI_Error_string` function. In order to keep the code as simple as possible, this book has adopted the (questionable) convention of ignoring the returned error codes.

Here is an example of compiling and running the code:

```
[sloanjd@amy sloanjd]$ mpicc hello.c -o hello
```

```
[sloanjd@amy sloanjd]$ mpirun -np 5 hello
```

```
Hello from process 0 on amy
```

```
Hello from process 2 on oscarnode2.oscardomain
```

```
Hello from process 1 on oscarnode1.oscardomain
```

```
Hello from process 4 on oscarnode4.oscardomain
```

```
Hello from process 3 on oscarnode3.oscardomain
```

There are a couple of things to observe with this example. First, notice that there is no apparent order in the output. This will depend on the speed of the individual machines, the loads on the machines, and the speeds of the communications links. Unless you take explicit measures to control the order of execution among processors, you should make no assumptions about the order of execution.

Second, the role of `MPI_Comm_size` should now be clearer. When running the program, the user specifies the number of processes on the command line. `MPI_Comm_size` provided a way to get that information back into the program. Next time, if you want to use a different number of processes, just change the command line and your code will take care of the rest.

13.2 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.

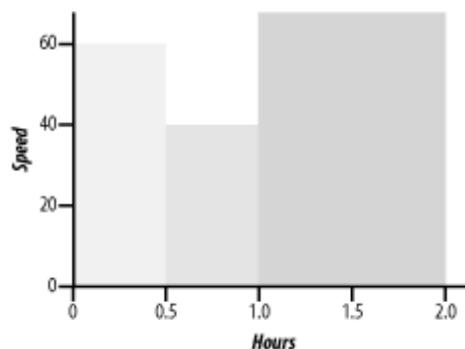
If you are familiar with numerical integration, you can skim this section quickly and move on to the next. Although this problem is a bit mathematical, it is straightforward and the mathematics shouldn't create much of a problem. Each step in the problem in this section is carefully explained, and you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

13.2.1 Background

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.

Figure 13-1. Area is distance traveled



Of course, in practice, your speed will change smoothly rather than in steps so that you won't be able to fit rectangles perfectly into the area. But the area under the curve does give the exact answer to the problem, and you can approximate the area by adding up rectangles. Generally, the more rectangles you use, the better your approximation.

In [Figure 13-2](#), three rectangles are used to estimate the area under a curve for a similar problem. In [Figure 13-3](#), six rectangles are used. The shaded areas in each determine the error in the approximation of the total area. However, since some of these areas are above the curve and some below, they tend to cancel each other out, at least in part. Unfortunately, this is not always the case.^[1]

[1] Those of you who remember your calculus recognize that we are calculating definite integrals. But as a numerical technique, this approximation will work even if you can't do the integration. (Ever run across an integral you couldn't evaluate?) You may also be asking why we aren't using the trapezoid rule. We are trying to keep things simple. The trapezoid rule is left as an exercise for those of you who remember it.

Figure 13-2. Approximating with three rectangles

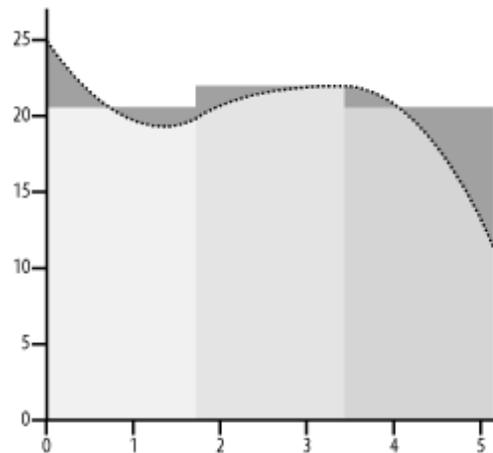
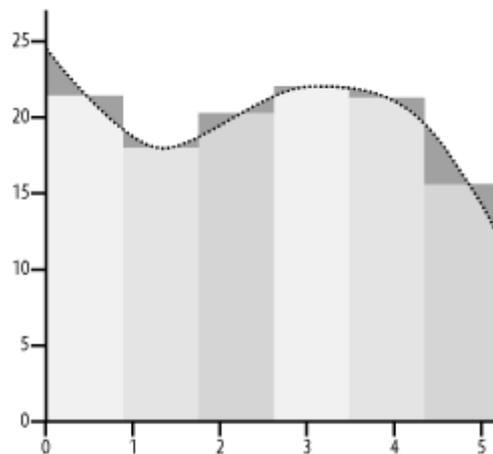


Figure 13-3. Approximating with six rectangles

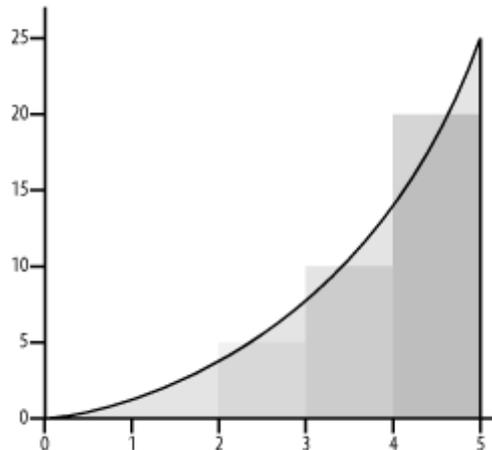


We can do this calculation without bothering to do the graph. All we need are the heights and widths of the rectangles. Widths are easy—we just divide the trip duration by however many rectangles we want. For the heights, we will need a way to calculate the speed during the rectangle. What we really want is the average speed. As an approximation, the simplest

approach is to use the speed at the middle of a rectangle. For most problems of this general type, some function or rule is used to calculate this value.

Let's turn this into to a more generic problem. Suppose you want to know the area under the curve $f(x) = x^2$ between 2 and 5. This is the shaded region in [Figure 13-4](#), which shows what the graph of this problem would look like if we use three rectangles.

Figure 13-4. Area under x^2 from 2 to 5 with three rectangles



With three rectangles, the width of each will be 1. To find the height, we take the center of each rectangle (2.5, 3.5, and 4.5) and evaluate the function ($2.5^2 = 6.25$, $3.5^2 = 12.25$, and $4.5^2 = 20.25$). Multiplying height by width for each rectangle and adding the results gives an area of 38.75. (Using calculus, we know the exact answer is 39.0, so we aren't far off.)

13.2.2 Single-Processor Program

Being computer types, we'll want to write a program to do the calculation. This will allow us to easily use many more rectangles to get better results and will allow us to easily change the endpoints and functions we use. Here is the code in C:

```
#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[ ] )

{
```

```

int          i;

double       area, at, height, width;

area = 0.0;

width = (upperLimit - lowerLimit) / numberRects;

for (i = 0; i < numberRects; i++)
{
    at = lowerLimit + i * width + width / 2.0;

    height = f(at);

    area = area + width * height;
}

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

return 0;
}

```

After entering the code with our favorite text editor, we can compile and run it.

```
[sloanjd@cs sloanjd]$ gcc rect.c -o rect
```

```
[sloanjd@cs sloanjd]$ ./rect
```

```
The area from 2.000000 to 5.000000 is: 38.999100
```

This is a much better answer.

This code should be self-explanatory, but a few comments can't hurt. First, macros are used to define the problem parameters, including the function that we are looking at. For the parameters, this lets us avoid the I/O issue when we code the MPI solution. The macro for the function is used to gain the greater efficiency of inline code while maintaining the clarity of a separate function. While this isn't much of an issue here, it is good to get in the habit of using macros. The heart of the code is a loop that, for each rectangle, first calculates the height of the rectangle and then calculates the area of the rectangle, adding it to a running total. Since we want to calculate the height of the rectangle at the middle of the interval, we add `width/2.0` when calculating `at`, the location we feed into the function. Obviously, there are a few things we can do to tighten up this code, but let's not worry about that right now

13.3 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++.

13.3.1 A C Solution

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;

}

```

This code is fairly straightforward, and you've already seen most of it. As before, we begin with the definition of the problem function and parameters. This is followed by the declaration of the variable that we'll need, first for the MPI calls and then for the problem. There are a few MPI variables whose use will be described shortly. Next comes the section that sets up MPI, but there is nothing new here.

The first real change comes in the next section where we adjust the problem size. In this example, we are calculating the area between 2 and 5. Since each process only needs to do part of this calculation, we need to divide the problem among the processes so that each process gets a different part (and all the parts are accounted for.) `MPI_Comm_size` is used to determine the number of parts the problem will be broken into, `noProcesses`. That is, we divide the total range (2 to 5) equally among the processes and adjust the start of the range for an individual process based on its rank. For example, with four processes, one process could calculate from 2 to 2.75, one from 2.75 to 3.5, one from 3.5 to 4.25, and one from 4.25 to 5.

In the next section of code, each process calculates the area for its part of the problem. This code keeps the number of rectangles fixed in this example rather than adjust it to the number of processes. That is, regardless of the number of processes used, each will use the same number of rectangles to solve its portion of the problem. Thus, if the number of processes increases from one run to the next, the answer won't come back any quicker but, up to a point, the answer should be more accurate. If your goal is speed, you could easily set the total number of rectangles for the problem, and then, in each process, divide that number by the number of processes or use some similar strategy.

Once we have completed this section, we need to collect and combine all our individual results. This is the new stuff. One process will act as a collector to which the remaining processes will send their results. Using the process with rank 0 as the receiver is the logical choice. The remaining processes act as senders. There is nothing magical about this choice apart from the fact that there will always be a process of rank 0. If a different process is selected, you'll need to ensure that a process with that rank exists. A fair amount of MPI code development can be done on a single processor system and then moved to a multiprocessor environment, so this isn't, as it might seem, a moot point.

The test (`ProcessId == 0`) determines what will be done by the collector process and what will be done by all the remaining processes. The first branch following this test will be executed by the single process with a rank of 0. The second branch will be executed by each of the remaining processes. It is just this sort of test that allows us to write a single program that will execute correctly on each machine in the cluster with different processes doing different things.

13.3.2 Transferring Data

The defining characteristic of message passing is that the transfer of data from one process to another requires operations to be performed by both processes. This is handled by `MPI_Send` and `MPI_Recv`. The first branch after this test contains a loop that will execute once for each of the remaining nodes in the cluster. At each execution of the body of the loop, the rank 0 process collects information from one of the other processes. This is done with the call to `MPI_Recv`. Each of the other processes executes the second branch after the test once. Each process uses the call to `MPI_Send` to pass its results back to process 0. For example, for 100 processes, there are 99 calls to `MPI_Send` and 99 calls to `MPI_Recv`. (Process 0 already knows what it calculated.) Let's look at these two functions more closely.

13.3.2.1 MPI_Send

`MPI_Send` is used to send information from one process to another process.^[2] A call to `MPI_Send` must be matched with a corresponding call to `MPI_Recv` in the receiving process. Information is both typed and tagged. Typing is needed to support communications in a heterogeneous environment. The type information is used to insure that the necessary conversions to data representation are applied as data moves among machines in a transparent manner.

^[2] Actually, a process can send a message to itself, but this possibility can get tricky so we'll ignore it.

The first three arguments to `MPI_Send`, collectively, are used to specify the transmitted data. The first argument gives the address of the data, the second gives the number of items to be sent, and the third gives the data type. In this sample code, we are sending the area, a single `double`, so we specify `MPI_DOUBLE` as the type. In addition to `MPI_DOUBLE`, the other possible types are `MPI_BYTE`, `MPI_CHAR`, `MPI_UNSIGNED_CHAR`, `MPI_SHORT`, `MPI_UNSIGNED_SHORT`, `MPI_INT`, `MPI_UNSIGNED_INT`, `MPI_LONG`, `MPI_UNSIGNED_LONG`, `MPI_LONG_DOUBLE`, `MPI_FLOAT`, and `MPI_PACKED`.

The next argument is the destination. This is just the rank of the receiver. The destination is followed by a tag. Since MPI provides buffering, several messages can be outstanding. The tag is used to distinguish among multiple messages. This is a moot point in this example. `MPI_COMM_WORLD` is the default communicator, which has already been described.

13.3.2.2 MPI_Recv

The arguments to `MPI_Recv` are similar but include one addition, a status field. `MPI_STATUS` is a type definition for a structure that holds information about the actual message size, its source, and its tag. In C, the status variable is a structure composed of three fields—`MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`—that contain the source, tag, and error code, respectively. With `MPI_Recv`, you can use a wildcard for either or both the source and the tag—`MPI_ANY_SOURCE` and `MPI_ANY_TAG`. The status field allows you to determine the actual source and tag in this situation.

You should be aware that `MPI_Send` and `MPI_Recv` are both blocking calls. For example, if you try to receive information that hasn't been sent, your process will be blocked or wait until it is sent before it can continue executing. While this is what you might expect, it can lead to nasty surprises if your code isn't properly written since you may have two processes waiting for each other.

Here is the output:

```
[sloanjd@amy sloanjd]$ mpicc mpi-rect.c -o mpi-rect
```

```
[sloanjd@amy sloanjd]$ mpirun -np 5 mpi-rect
```

```
The area from 2.000000 to 5.000000 is: 38.999964
```

Of course, all of this assumed you wanted to program in C. The next two sections provide alternatives to C.

13.3.3 MPI Using FORTRAN

Let's take a look at the same program written in FORTRAN.

```
program main

include "mpif.h"

parameter (NORECS = 50, DLIMIT = 2.00, ULIMIT = 5.00)

integer dst, err, i, noprocs, procid, src, tag

integer status(MPI_STATUS_SIZE)

double precision area, at, height, lower, width, total, range

f(x) = x * x

***** MPI setup *****

call MPI_INIT(err)

call MPI_COMM_SIZE(MPI_COMM_WORLD, noprocs, err)

call MPI_COMM_RANK(MPI_COMM_WORLD, procid, err)

***** adjust problem size for subproblem *****

range = (ULIMIT - DLIMIT) / noprocs

width = range / NORECS

lower = DLIMIT + range * procid

***** calculate area for subproblem *****

area = 0.0;

do 10 i = 0, NORECS - 1

at = lower + i * width + width / 2.0

height = f(at)

area = area + width * height
```


FORTRAN 77 programs begin with `include "mpi.f"`. FORTRAN 90 may substitute `use mpi` if the MPI implementation supports modules.

In creating the MPI specification, a great deal of effort went into having similar binding in C and FORTRAN. The biggest difference is the way error codes are handled. In FORTRAN there are explicit parameters included as the last argument to each function call. This will return either `MPI_SUCCESS` or an implementation-defined error code.

In C, function arguments tend to be more strongly typed than in FORTRAN, and you will notice that C tends to use addresses when the function is returning a value. As you might expect, the parameters to `MPI_Init` have changed. Finally, `MPI_STATUS` is an array rather than a structure in FORTRAN.

Overall, the differences between C and FORTRAN aren't that great. You should have little difficulty translating code from one language to another.

13.3.4 MPI Using C++

Here is the same code in C++:

```
#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);

noProcesses = MPI::COMM_WORLD.Get_size( );

processId = MPI::COMM_WORLD.Get_rank( );

/* 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::COMM_WORLD.Recv(&area, 1, MPI::DOUBLE, src, tag);

        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::COMM_WORLD.Send(&area, 1, MPI::DOUBLE, dest, tag);

};

/* finish */

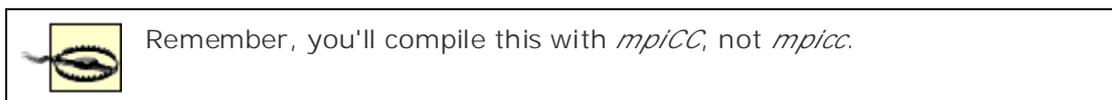
MPI::Finalize( );

return 0;

}

```

If you didn't skip the section on FORTRAN, you'll notice that there are more differences in going from C to C++ than in going from C to FORTRAN.



The C++ bindings were added to MPI as part of the MPI-2 effort. Rather than try to follow the binding structure used with C and FORTRAN, the C++ bindings were designed to exploit MPI's object-oriented structure. Consequently, most functions become members of C++ classes. For example, `MPI::COMM_WORLD` is an instance of the communicator class. `Get_rank` and `Get_size` are methods in the class. All classes are part of the MPI namespace.

Another difference you'll notice is that `Get_size` and `Get_rank` return values. Since the usual style of error handling in C++ is throwing exceptions, which MPI follows, there is no need to return error codes.

Finally, you notice that the type specifications have changed. In this example, we see `MPI::DOUBLE` rather than `MPI_DOUBLE` which is consistent with the naming conventions being adopted here. We won't belabor this example. By looking at the code, you should have a pretty clear idea of how the bindings have changed with C++.

Now that we have a working solution, let's look at some ways it can be improved. Along the way we'll see two new MPI functions that can make life simpler.

13.4 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, numberReacts;

    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);

tag = 0;

if (processId == 0)          /* if rank is 0, collect parameters */
{
    fprintf(stderr, "Enter number of steps:\n");

    scanf("%d", &numberRects);

    fprintf(stderr, "Enter low end of interval:\n");

    scanf("%lf", &lowerLimit);

    fprintf(stderr, "Enter high end of interval:\n");

    scanf("%lf", &upperLimit);

    for (dest=1; dest < noProcesses; dest++) /* distribute parameters */
    {
        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);
    }
}
else                          /* all other processes receive */
{ src = 0;

    MPI_Recv(&numberRects, 1, MPI_INT, src, 0, MPI_COMM_WORLD, &status);

    MPI_Recv(&lowerLimit, 1, MPI_DOUBLE, src, 1, MPI_COMM_WORLD, &status);

    MPI_Recv(&upperLimit, 1, MPI_DOUBLE, src, 2, MPI_COMM_WORLD, &status);
}

```

```

/* 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 = 3;

if (processId == 0)          /* if rank is 0, collect results */

{ total = area;

  fprintf(stderr, "Area for process 0 is: %f\n", area);

  for (src=1; src < noProcesses; src++)

  {

    MPI_Recv(&area, 1, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, &status);

    fprintf(stderr, "Area for process %d is: %f\n", src, area);

    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;

}
```

The solution is straightforward. We need to partition the problem so that the input is only attempted by the rank 0 process. It then enters a loop to send the parameters to the remaining processes.

While this approach certainly works, it introduces a lot of overhead. While it might be tempting to calculate a few of the derived parameters (e.g. `range` or `width`) and distribute them as well, this is a false economy. Communication is always costly, so we'll let each process calculate these values for themselves. Anyway, they would have been idle while the rank 0 process did the calculations.

13.5 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.

13.5.1 Broadcast Functions

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 still 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`.

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 call, 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.

Here is our numerical integration code using `MPI_Bcast` (and `MPI_Reduce`, a function we will discuss next). 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 noProcesses, processId;

/* 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);

if (processId == 0)      /* if rank is 0, collect parameters */
{
    fprintf(stderr, "Enter number of steps:\n");
    scanf("%d", &numberRects);
    fprintf(stderr, "Enter low end of interval:\n");
    scanf("%lf", &lowerLimit);
    fprintf(stderr, "Enter high end of interval:\n");
    scanf("%lf", &upperLimit);
}

MPI_Bcast(&numberRects, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&lowerLimit, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&upperLimit, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* 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;

}

MPI_Reduce(&area, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

/* collect information and print results */

if (processId == 0)          /* if rank is 0, print results */

{ fprintf (stderr, "The area from %f to %f is: %f\n",

          lowerLimit, upperLimit, total );

}

/* finish */

MPI_Finalize( );

return 0;

}

```

Notice that we have eliminated the control structures as well as the need for separate `MPI_Send` and `MPI_Recv` calls.

13.5.1.2 MPI_Reduce

You'll also notice that we have used a new function, `MPI_Reduce`. The process of collecting data is so common that MPI includes functions that automate this process. The idea behind `MPI_Reduce` is to specify a data item to be accumulated, a storage location or variable to accumulate in, and an operator to use when accumulating. In this example, we want to add up all the individual areas, so `area` is the data to accumulate, `total` is the location where we accumulate the data, and the operation is adding or `MPI_SUM`.

More specifically, `MPI_Reduce` has seven arguments. The first two are the addresses of the send and receive buffers. The third is the number of elements in the send buffer, while the fourth gives the type of the data. Both send and receive buffers will manipulate the same number of elements which will be of the same type. The next operation identifies the function

used to combine elements. `MPI_SUM` is used to add elements. MPI defines a dozen different operators. These include operators to find the sum of the data values (`MPI_SUM`), their product (`MPI_PROD`), the largest and smallest values (`MPI_MAX` and `MPI_MIN`), and numerous logical operations for both logical and bitwise comparisons using AND, OR, and XOR (`MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI BOR`, `MPI_LXOR`, and `MPI_BXOR`). The data type must be compatible with the selected operation.

The next to the last argument identifies the root of the communications, i.e., the rank of the process that will accumulate the final answer, and the last argument is the communicator. These must have identical values in every process. Notice that only the root process will have the accumulated result. If all of the processes need the result, there is an analogous function `MPI_Allreduce` that is used in the same way.

Notice how the use of `MPI_Reduce` has simplified our code. We have eliminated a control structure, and, apart from the single parameter in our recall to `MPI_Reduce`, we no longer need to distinguish among processes. Keep in mind that it is up to the implementer to determine the best way to implement these functions. Details will vary. For example, the "broadcast" in `MPI_Bcast` simply means that the data is sent to all the processes. It does not necessarily imply that an Ethernet-style broadcast will be used, although that is one obvious implementation strategy. When implementing for other networks, other strategies may be necessary.

In this chapter we have introduced the six core MPI functions—`MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Recv`, and `MPI_Finalize`—as well as several others that simplify MPI coding. These six core functions have been described as the six indispensable MPI functions, the functions that you really can't do without. On the other hand, most MPI programs, with a little extra work, could be rewritten with just these six functions. Congratulations! You are now an MPI programmer.

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.

14.1 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.

14.1.1 Non-Blocking Communication

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.

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`.

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 `MPI_Recv`. `MPI_Wait` blocks until the operation identified by the request handle completes. When it returns, the request handle is set to a special constant, `MPI_REQUEST_NULL`, indicating that there is no longer a pending operation associated with the request handle.

Code for `MPI_Irecv` and `MPI_Wait` might look something like this fragment:

```
...  
  
int datum1, datum2;  
  
MPI_Status status;  
  
MPI_Request handle;
```

```

if (processId == 0)
{
    MPI_Send(&datum1, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &handle);
    ...
}
else
{
    MPI_Irecv(&datum2, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &handle);
    ...
    MPI_Wait(&handle, &status);
}
...

```

In this example, the contents of `datum1` are received in `datum2`. As shown here, it is OK to mix blocking and non-blocking commands. For example, you can use `MPI_Send` to send a message that will be received by `MPI_Irecv` as shown in the example.

14.1.1.3 MPI_Test

`MPI_Test` is a non-blocking alternative to `MPI_Wait`. It takes three arguments: the request handle, a flag, and a status variable. If the exchange is complete, the value returned in the flag variable is `true`, the request handle is set to `MPI_REQUEST_NULL`, and the status variable will contain information about the exchange. If the flag is still set to `false`, then the exchange hasn't completed, the request variable is unchanged, and the status variable is undefined.

14.1.1.4 MPI_Iprobe

If you want to check up on messages without actually receiving them, use `MPI_Iprobe`. (There is also a blocking variant called `MPI_Probe`.) `MPI_Iprobe` can be called multiple times without actually receiving the message. Once you know the exchange has finished, you can use `MPI_Test` or `MPI_Wait` to actually receive the message. `MPI_Iprobe` takes five arguments: the rank of the source, the message tag, the communicator, a flag, and a status object. If the flag is `true`, the message has been received and the status object can be examined. If `false`, the status is undefined.

14.1.1.5 MPI_Cancel

If you have a pending, non-blocking communication operation, it can be aborted with the `MPI_Cancel` command. `MPI_Cancel` takes a request handle as its only argument. You might use `MPI_Cancel` in conjunction with `MPI_Iprobe`. If you don't like the status information returned by `MPI_Iprobe`, you can use `MPI_Cancel` to abort the exchange.

14.1.1.6 MPI_Sendrecv and MPI_Sendrecv_replace

If you need to exchange information between a pair of processes, you can use `MPI_Sendrecv` or `MPI_Sendrecv_replace`. With the former, both the send and receive buffers must be distinct. With the latter, the received message overwrites the sent message. These are both blocking commands.

While these examples should give you an idea of some of the functions available, there are other point-to-point functions not described here. For example, there is a set of commands to create and manipulate persistent connections similar to communication ports (`MPI_Send_init`, `MPI_Start`, etc.). You can specify dummy sources and destinations for messages (`MPI_PROC_NULL`). There are variants on `MPI_Wait` and `MPI_Test` for processing lists of pending communication operations (`MPI_Testany`, `MPI_Testall`, `MPI_Testsome`, `MPI_Waitany`, etc.) Additional communication modes are also supported: synchronous-mode communication and ready-mode communication.



14.2 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.

14.2.1 Gather and Scatter

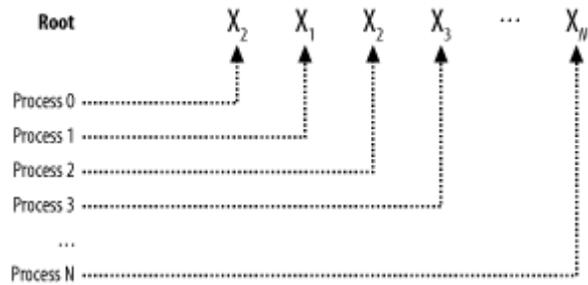
After `MPI_Bcast` and `MPI_Reduce`, the two most useful collective operations are `MPI_Gather` and `MPI_Scatter`.

14.2.1.1 MPI_Gather

`MPI_Gather`, as the name implies, gathers information from all the processes in a communication group. It takes eight arguments. The first three arguments define the information that is sent. These are the starting address of the send buffer, the number of items in the buffer, and the data type for the items. The next three arguments, which define the received information, are address of the receive buffer, the number of items for a single receive, and the data type. The seventh argument is the rank of the receiving or root process. And the last argument is the communicator. Keep in mind that each process, including the root process, sends the contents of its send buffer to the receiver. The root process receives the messages, which are stored in rank order in the receive buffer. While this may seem similar to `MPI_Reduce`, notice that the data is simply received. It is not combined or reduced in any way.

[Figure 14-1](#) shows the flow of data with a gather operation. The root or receiver can be any of the processes. Each process sends a different piece of the data, shown in the figure as the different x 's.

Figure 14-1. Gathering data



Here is an example using `MPI_Gather`.

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

int main( int argc, char * argv[ ] )
{
    int processId;

    int a[4] = {0, 0, 0, 0};
    int b[4] = {0, 0, 0, 0};

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &processId);

    if (processId == 0) a[0] = 1;
    if (processId == 1) a[1] = 2;
    if (processId == 2) a[2] = 3;
    if (processId == 3) a[3] = 5;

    if (processId == 0)
        fprintf(stderr, "Before: b[ ] = [%d, %d, %d, %d]\n", b[0], b[1], b[2],
b[3]);

    MPI_Gather(&a[processId], 1, MPI_INT, b, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```

    if (processId == 0)
        fprintf(stderr, "After:  b[ ] = [%d, %d, %d, %d]\n", b[0], b[1], b[2],
b[3]);

    MPI_Finalize( );

    return 0;
}

```

While this is a somewhat contrived example, you can clearly see how `MPI_Gather` works. Pay particular attention to the arguments in this example. Note that both the address of the item sent and the address of the receive buffer (in this case just an array name) are used. Here is the output:

```

[sloanjd@amy C12]$ mpirun -np 4 gath

Before: b[ ] = [0, 0, 0, 0]

After:  b[ ] = [1, 2, 3, 5]

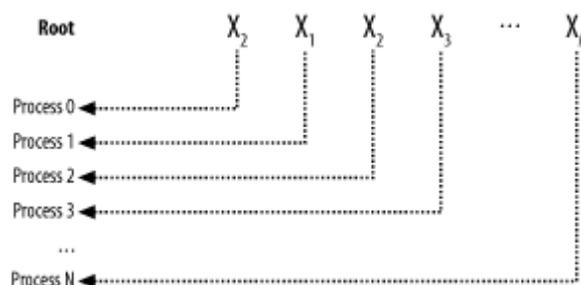
```

`MPI_Gather` has a couple of useful variants. `MPI_Gatherv` has an additional argument, an integer array giving displacements. `MPI_Gatherv` is used when the amount of data varies from process to process. `MPI_Allgather` functions just like `MPI_Gather` except that all processes receive copies of the data. There is also an `MPI_Allgatherv`.

14.2.1.2 MPI_Scatter

`MPI_Scatter` is the dual or inverse of `MPI_Gather`. If you compare [Figure 14-2](#) to [Figure 14-1](#), the only difference is the direction of the arrows. The arguments to `MPI_Scatter` are the same as `MPI_Gather`. You can think of `MPI_Scatter` as splitting the send buffer and sending a piece to each receiver, i.e., each receiver receives a unique piece of data. `MPI_Scatter` also has a vector variant `MPI_Scatterv`.

Figure 14-2. Scattering data



Here is another contrived example, this time with `MPI_Scatter`:

```

#include "mpi.h"

```

```

#include <stdio.h>

int main( int argc, char * argv[ ] )
{
    int processId, b;

    int a[4] = {0, 0, 0, 0};

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &processId);

    if (processId == 0) { a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 5; }

    MPI_Scatter(a, 1, MPI_INT, &b, 1, MPI_INT, 0, MPI_COMM_WORLD);

    fprintf(stderr, "Process %d: b = %d \n", processId, b);

    MPI_Finalize( );

    return 0;
}

```

Notice that we are sending an array but receiving its individual elements in this example. In summary, `MPI_Gather` sends an element and receives an array while `MPI_Scatter` sends an array and receives an element.

As with point-to-point communication, there are additional collective operations (for example, `MPI_Alltoall`, `MPI_Reduce_scatter`, and `MPI_Scan`). It is even possible to define your own reduction operator (using `MPI_Op_create`) for use with `MPI_Reduce`, etc. Because the amount and nature of the data that you need to share varies with the nature of the problem, it is worth becoming familiar with MPI's collective functions. You may need them sooner than you think.

14.3 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.^[1]

^[1] Although it sounds like there is only one default communicator, there are actually two. The other default communicator is `MPI_COMM_SELF`. Since this is defined for each process and contains only that process, it isn't all that useful when defining new communicators.

14.3.1 Communicator Commands

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.

```
#include "mpi.h"

#include <stdio.h>

int main( int argc, char * argv[ ] )
{
    int processId, i, flag = 0;

    int processes[1] = {0};

    MPI_Group  worldGroup, newGroup;

    MPI_Comm  newComm;
```

```

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &processId);

MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);

MPI_Group_excl(worldGroup, 1, processes, &newGroup);

MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);

fprintf(stderr, "Before: process: %d  Flag: %d\n", processId, flag);

if (processId == 1) flag = 1;

if (processId != 0)

    MPI_Bcast(&flag, 1, MPI_INT, 0, newComm);

fprintf(stderr, "After  process: %d  Flag: %d\n", processId, flag);

if (processId !=0)

{ MPI_Comm_free(&newComm);

MPI_Group_free(&newGroup);

}

MPI_Finalize( );

return 0;

}

```

Relevant portions of this program appear in boldface.

The first things to notice about the program are the new type declarations using `MPI_Group` and `MPI_Comm`. `MPI_Group` allows us to define handles for manipulating process groups. In this example, we need two group handles, one for the existing group from `MPI_COMM_WORLD` and one for the new group, we are defining. The `MPI_Comm` type is used to define a variable for the new communicator being created.

14.3.1.1 MPI_Comm_group

Next, we need to extract the group from `MPI_COMM_WORLD`, our starting point for the new group. We use the function `MPI_Comm_group` to do this. It takes two arguments: the first is the communicator; the second argument is a handle used to return the group for the specified communicator, in this case, `MPI_COMM_WORLD`'s group.

14.3.1.2 MPI_Group_incl and MPI_Group_excl

Once we have an existing group, we can use it to create a new group. In this example, we exclude process from the original group using the `MPI_Group_excl` command. Exclusion is the easiest way to handle this particular case since only one process needs to be specified. `MPI_Group_incl` should be used when it is simpler to list processes to include rather than exclude. The four arguments to `MPI_Group_incl` and `MPI_Group_excl` are the same: the first argument is the original group you are using as a starting point; the second argument is the number of processes that will be included or excluded; the third argument is an integer array giving the ranks of the processes to be included or excluded; and the last parameter is the address of the group's handle.

In this example, since process 0 is excluded, we have used the array `process` to list the single process rank that we want excluded. We could have accomplished the same thing with the array

```
int processes[3] = {1, 2, 3};
```

and the call

```
MPI_Group_incl(worldGroup, 3, processes, &newGroup);
```

Either way works fine.

14.3.1.3 MPI_Comm_create

Finally, we need to turn the new group into a communicator. This is done with the `MPI_Comm_create` command, which takes three arguments: the original communicator, the new group, and the address for the new communicator's handle. Once this call is made, we have our communicator.

In the code sample given above, the next block of code shows how the new communicator could be used. In the example, there is a variable `flag` initially set to 0. It is changed in process 1 to 1 and then broadcast to the remaining processes within the new communicator. Here is what the output for four processes looks like.

```
[sloanjd@amy COMM]$ mpirun -np 4 comm
```

```
Process: 0   Flag: 0
```

```
Process: 0   Flag: 0
```

```
Process: 1   Flag: 0
```

```
Process: 2   Flag: 0
```

```
Process: 3   Flag: 0
```

```
Process: 1   Flag: 1
```

```
Process: 2   Flag: 1
```

```
Process: 3   Flag: 1
```

Note that the value changes for every process except process 0.

There are a couple of things worth noting about how the new communicator is used. First, notice that only the relevant processes are calling `MPI_Bcast`. Process 0 has been excluded.

Had this not been done, the call in process 0 would have returned a null communicator error since it is not part of the communicator. The other thing to note is that the process with rank 1 in `MPI_COMM_WORLD` has a rank of 0 in the new communicator. Thus, the fourth argument to `MPI_Bcast` is 0, not 1.

14.3.1.4 `MPI_Comm_free` and `MPI_Group_free`

It is good housekeeping to release any communicators or groups you are no longer using. For these two functions, the handles will be set to `MPI_COMM_NULL` and `MPI_GROUP_NULL`, respectively. While releasing these isn't absolutely necessary, it can be helpful at times. For example, doing so may alert you to the inadvertent use of what should be defunct groups or communicators. Each of these two functions takes the address of the communicator or of the group as an argument, respectively. It doesn't matter which function you call first.

Since process 0 is not part of the new communicator in the last example, we need to guard against using the new communicator within process 0. This isn't too difficult when a single process is involved but can be a bit of a problem when more processes are involved. So in some instances, splitting communicators is a better approach. Here is a simple example.

```
#include "mpi.h"

#include <stdio.h>

int main( int argc, char * argv[ ] )
{
    int processId, i, flag = 0, color = 0;

    MPI_Comm    newComm;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &processId);

    if (processId == 0 || processId == 1) color = 1;

    MPI_Comm_split(MPI_COMM_WORLD, color, processId, &newComm);

    fprintf(stderr, "Process: %d    Flag: %d\n", processId, flag);

    if (processId == 0) flag = 1;

    MPI_Bcast(&flag, 1, MPI_INT, 0, newComm);

    fprintf(stderr, "Process: %d    Flag: %d\n", processId, flag);

    MPI_Comm_free(&newComm);
```

```

MPI_Finalize( );

return 0;

}

```

Notice that, in this example, the communicator is manipulated directly without resorting to dealing with groups.

14.3.1.5 MPI_Comm_split

The function `MPI_Comm_split` is at the heart of this example. It is used to break a communicator into any number of pieces. The first argument is the original communicator. The second argument, often referred to as the *color*, is used to determine which communicator a process will belong to. All processes that make the call to `MPI_Comm_split` with the same color will be in the same communicator. Processes with different values (or colors) will be in different communicators. In this example, processes 0 and 1 have a color of 1 so they are in one communicator while processes 2 and above have a color of 0 and are in a separate communicator. (If the color is `MPI_UNDEFINED`, the process is excluded from any of the new communicators.) The third argument, often called the *key*, is used to determine the rank ordering for processes within a communicator. When keys are the same, the original rank is used to break the tie. The last argument is the address of the new communicator.

[Table 14-1](#) gives a slightly more complicated example of how this might work. Using the data in this table, three new communicators are created. The first communicator consists of processes A, C, and D with ranks in the new communicator of 1, 0, and 2, respectively. The second communicator consists of processes B and E with ranks 0 and 1, respectively. The last communicator consists of the single process F with a rank of 0. Process G is not included in any of the new communicators.

Table 14-1. Communicator assignments

Process	A	B	C	D	E	F	G
Original rank	0	1	2	3	4	5	6
Color	1	2	1	1	2	3	MPI_UNDEFINED
Key	3	3	2	3	3	a	0

Returning to the code given above, with four processes, two communicators will be created. Both will be called `newComm`. The first will have the original processes 0 and 1 with the same ranks in the new communicator. The second will have the original processes 2 and 3 with new ranks 0 and 1, respectively. Notice that a communicator is defined for every process, all with the same name.

These two examples should give you an idea of why communicators are useful and how they are used. Group management functions include functions to access groups (e.g., `MPI_Group_size`, `MPI_Group_rank`, and `MPI_Group_compare`) and functions to construct groups (e.g., `MPI_Group_difference`, `MPI_Group_union`, `MPI_Group_incl`, and `MPI_Group_range_incl`). There are also a number of different communicator management functions (e.g., `MPI_Comm_size`, `MPI_Comm_dup`, `MPI_Comm_compare`, and `MPI_Comm_create`).

14.4 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.

14.4.1 User-Defined Types

A user-defined data type can be used in place of the predefined data types included with MPI. Such a type can be used as the data type in any MPI communication function. MPI type-constructor functions are used to describe the memory layout for these new types in terms of primitive types. User-defined or -derived data types are opaque objects that specify the sequence of the primitive data types used and a sequence of displacements or offsets.

Here is the numerical integration problem adapted to use a user-defined data type.

```
#include "mpi.h"

#include <stdio.h>

/* problem parameters */

#define f(x)          ((x) * (x))

int main( int argc, char * argv[ ] )
{
    /* MPI variables */

    int          noProcesses, processId;

    int          blocklengths[3] = {1, 1, 1};

    MPI_Aint     displacements[3] = {0, sizeof(double), 2*sizeof(double)};

    MPI_Datatype rectStruct; /* the new type */

    MPI_Datatype types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};

    /* problem variables */

    int          i;

    double       area, at, height, lower, width, total, range;

    struct ParamStruct
    {
        double lowerLimit;

        double upperLimit;

        int    numberRects;
    } params;

    /* MPI setup */

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);

    MPI_Comm_rank(MPI_COMM_WORLD, &processId);
```

```

/* define type */

MPI_Type_struct(3, blocklengths, displacements, types, &rectStruct);

MPI_Type_commit(&rectStruct);

if (processId == 0)          /* if rank is 0, collect parameters */
{
    fprintf(stderr, "Enter number of steps:\n");

    scanf("%d", &params.numberRects);

    fprintf(stderr, "Enter low end of interval:\n");

    scanf("%lf", &params.lowerLimit);

    fprintf(stderr, "Enter high end of interval:\n");

    scanf("%lf", &params.upperLimit);
}

MPI_Bcast(&params, 1, rectStruct, 0, MPI_COMM_WORLD);

/* adjust problem size for subproblem*/

range = (params.upperLimit - params.lowerLimit) / noProcesses;

width = range / params.numberRects;

lower = params.lowerLimit + range * processId;

/* calculate area for subproblem */

area = 0.0;

for (i = 0; i < params.numberRects; i++)
{
    at = lower + i * width + width / 2.0;

    height = f(at);

    area = area + width * height;
}

```

```

MPI_Reduce(&area, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

/* collect information and print results */

if (processId == 0)          /* if rank is 0, collect results */

{ fprintf(stderr, "The area from %f to %f is: %f\n",

           params.lowerLimit, params.upperLimit, total );

}

/* finish */

MPI_Finalize( );

return 0;

}

```

To simplify the new MPI type definition, a structure type `ParamStruct` was defined. `params` is an instance of that structure. To access the individual elements of the structure, constructs such as `params.numberRects` must be used. These constructs have not been highlighted. All other changes related to user-defined types appear in boldface in the code.

14.4.1.1 MPI_Type_struct

`MPI_Type_struct` was used to define the new type. This function takes five arguments. The first four are input parameters while the last is the output parameter. The first is an integer that gives the number of blocks of elements in the type. In our example, we have three blocks of data. Our blocks are two doubles and an integer, but a block could be an aggregate data type such as an array. The next argument is an array giving the lengths of each block. In this example, because we've used scalars instead of arrays for our three blocks, the argument is just an array of 1's, one for each block. The third argument is an array of displacements. A displacement is determined by the size of the previous blocks, and the first displacement is always zero. Note the use of the type `MPI_Aint`. This bit of MPI magic is an integer type defined to be large enough to hold any address on the target architecture.^[2] The fourth argument is an array of primitive data types. Basically, you can think of an MPI data type as a set of pairs, each pair defining the basic MPI type and its displacement in bytes.

^[2] MPI also supplies a function `MPI_Address` that can be used to calculate an offset. It takes a variable and returns its byte address in memory.

14.4.1.2 MPI_Type_commit

Before a derived type can be used, it must be committed. You can think of this as "compiling" the new data type. The only argument to `MPI_Type_commit` is the type being defined.

As you can see from the example, the new type is used just like any existing type once defined. Keep in mind that this is a very simplistic example. Much more complicated structures can be built. MPI provides a rich feature set for user-defined data types.

14.4.2 Packing Data

Another alternative to packaging data is to use the MPI functions `MPI_Pack` and `MPI_Unpack`. `MPI_Pack` allows you to store noncontiguous data in contiguous memory while `MPI_Unpack` is used to retrieve that data.

14.4.2.1 MPI_Pack

`MPI_Pack` takes seven arguments. The first three define the message to be packed: the input buffer, the number of input components, and the data type of each component. The next two parameters define the buffer where the information is packed: the output buffer and the buffer size. The next to the last argument gives the current position in the buffer in bytes, while the last parameter is the communicator for the message.

Here is an example of how data is packed.

```
position = 0;

MPI_Pack(&numberRects, 1, MPI_INT, buffer, 50, &position, MPI_COMM_WORLD);

MPI_Pack(&lowerLimit, 1, MPI_DOUBLE, buffer, 50, &position, MPI_COMM_WORLD);

MPI_Pack(&upperLimit, 1, MPI_DOUBLE, buffer, 50, &position, MPI_COMM_WORLD);
```

In this instance, `buffer` has been defined as an array of 50 `chars` and `position` is an `int`. Notice that the value of `position` is automatically incremented as it is used.

14.4.2.2 MPI_Unpack

The first argument is the input buffer, a contiguous storage area containing the number of bytes specified in the second argument. The third argument is the position where unpacking should begin. The fourth and fifth arguments give the output buffer and the number of components to unpack. The next to last argument is the output data type while the last argument is the communicator for the message. You need not unpack the entire message.

Here is an example of unpacking the data just packed.

```
if (processId !=0)

    { position = 0;

      MPI_Unpack(buffer, 50, &position, &numberRects, 1, MPI_INT,

        MPI_COMM_WORLD);

      MPI_Unpack(buffer, 50, &position, &lowerLimit, 1, MPI_DOUBLE,

        MPI_COMM_WORLD);

      MPI_Unpack(buffer, 50, &position, &upperLimit, 1, MPI_DOUBLE,

        MPI_COMM_WORLD);

    }
```

This is the call to `MPI_Bcast` used to send the data.

```
MPI_Bcast(buffer, 50, MPI_PACKED, 0, MPI_COMM_WORLD);
```

As you can see, it's pretty straightforward. The most likely mistake is getting parameters in the wrong order or using the wrong type.

In general, if you have an array of data, the first approach (using a count) is the easiest. If you have lots of different data scattered around your program, packing and unpacking is likely to be the best choice. If the data are stored at regular intervals and of the same type, e.g., the column of a matrix, a derived type is usually a good choice.

This chapter has only scratched the surface. There is a lot more to know about MPI. For more information, consult any of the books described in the [Appendix A](#).

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.

15.1 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.

15.2 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.

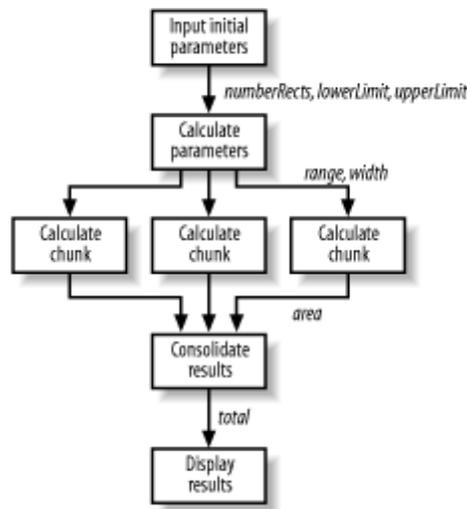
A *data* (or *task*) *dependency graph* (or *diagram*) is one way of visually representing a program. This can be helpful when investigating and describing potential concurrency. The idea is to break the algorithm into pieces of code or tasks based on the data required by that task. A graph is then drawn for the algorithm that shows the set of tasks as nodes connected by arrows indicating the flow of data between connected pairs of tasks.

[Figure 15-1](#) is a data dependency graph for the numerical integration program developed in [Chapter 13](#). The amount of detail will vary in these graphs depending on your purpose. In this case, I've kept things very simple. If you desire, you can increase the detail to the point of having a single node for each instruction and arrows for each variable. [\[1\]](#)

[1] Some authors distinguish between data and task dependency graphs and between dependencies and interactions. Feel free to adjust your graphs as you see fit.

The idea is that graphs such as these help you think about and locate potential concurrencies in your code. If you have two blocks of code or tasks that don't depend on each other and have everything they need to execute, these are potentially parallelizable tasks. Data flow graphs can be used for both data and task partitioning. Data flow graphs should also provide you with some idea of the critical paths through code, i.e., those paths that will likely take the longest amount of time to complete. You won't be able to shorten the runtime of a program to less time than it takes to complete the critical path. In other words, if you want to shorten the runtime of a program, you must shorten the critical path.

Figure 15-1. Data flow for numerical integration



There are some limitations to this approach. You'll need to give loops some thought when drawing these graphs, since the body of a loop is a potentially parallelizable piece of code. The essential step in parallelizing the numerical integration problem in [Chapter 13](#) was packaging as individual tasks, with pieces of the loop used to calculate the area using the individual rectangles. You should also realize that the graph provides no information about the relative execution time for each task. Finally, and perhaps most important, the graph doesn't clearly indicate how idle time might show up. Depending on how we code the task *Consolidate Results* in [Figure 15-1](#), most of the *Calculate Chunk* blocks may be idle waiting for an opportunity to report their results. (Moreover, depending on how they are coded, the individual *Calculate Chunk* tasks may not all be of the same length.)

15.2.1 Decomposition Strategies

There are several different decomposition strategies worth considering. Roughly speaking, decomposition strategies fall into two different categories—*data decomposition*, sometimes called *data partitioning*, and *control decomposition* or *task partitioning*. With data decomposition, the data is broken into individual chunks and distributed among processes that are essentially similar. With control decomposition, the problem is divided in such a way that each process is doing a different calculation. In practice, many algorithms show characteristics of both strategies.

15.2.1.1 Data decomposition

Data decomposition is generally much easier to program than control decomposition and is usually the best approach when trying to adapt serial algorithms for parallel use. Data decomposition also tends to scale very well, a crucial consideration when dealing with problems that may grow.

The numerical integration program from the last chapter used data decomposition. Each process had a different set of bounds, so the area that each calculated was different, but the procedure was the same.

One of the most common approaches to data decomposition is a *divide-and-conquer* strategy. This works particularly well with recursive algorithms. If a problem can be treated as a set of independent subproblems, it is an ideal candidate for data decomposition. Consider the problem of finding the largest value in a large collection of data. The data could be divided into different sets, the largest in each set could be found, and finally, this collection of largest values could be examined. Finding the largest value in each of the smaller sets could be

handled by a different processor. Finding the final answer is an ideal use of `MPI_Reduce`. This is a pretty trivial example of how divide and conquer works.

For a more involved example, consider the merge sort algorithm.^[2] The serial algorithm takes a set of data, divides it into smaller sets of data, sorts these smaller individual data sets, and then merges the sorted sets back together. To sort a smaller set of data, merge sort uses the same strategy recursively. Eventually, the smaller sets of data are reduced to sets of single items that are obviously sorted. Merging sorted data is straightforward since you only have to compare the first item in each group and select accordingly until you've worked your way through the smaller sets.

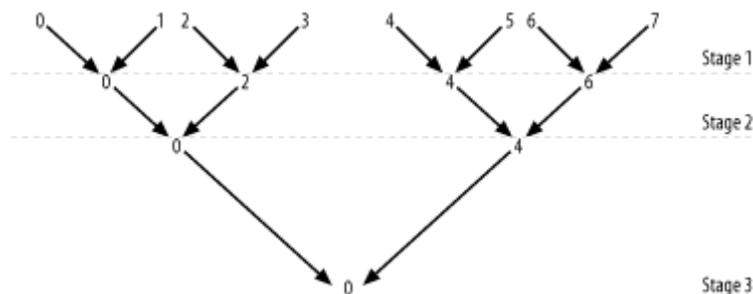
[2] The sorting algorithm described here is just one possible approach, not necessarily the best. Sorting in a parallel environment is particularly difficult and is an area of active, ongoing research.

In a parallel environment, you'll want to divide the data equally among the available processors, but you probably won't want to continue dividing up the data beyond that point because of the communications overhead. Once you have the data distributed, you'll need to sort it locally on each individual processor. You could use the serial version of merge sort or some other serial sorting algorithm.

Merging the data back together will be more problematic. Not only will you need code to merge two data sets, but you'll need to develop a communications strategy to do this efficiently. If you use a single process to collect and merge data, you will have a large amount of idle time. A more appropriate strategy is to have pairs of processes merge their data, i.e., one sends its data and dies while the other receives the sent data and merges that data with its own. Repeat this strategy with the remaining processes until only a single process remains. It will have all the data sorted.

For example, if you have eight processes, processes 0 and 1, processes 2 and 3, processes 4 and 5, and processes 6 and 7 could all merge their data at the same time. Next, processes 0 and 2 and processes 4 and 6 could merge their data simultaneously. Finally, processes 0 and 4 could merge their data. This strategy, shown in [Figure 15-2](#), has three sets of parallel merges or stages. This is much more efficient than having process 0 merge its data repeatedly with each of the other seven processes sequentially, a seven-stage procedure.

Figure 15-2. Merging data



With this strategy, for instance, 1,024 processes could merge their data in 10 stages. It would take 1,023 stages with a single receiving process, roughly 100 times as long.

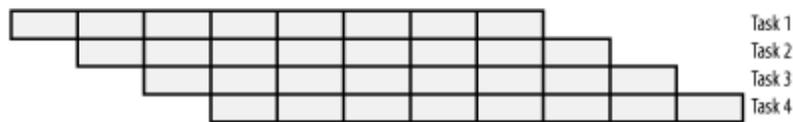
15.2.1.2 Control decomposition

With control decomposition, each processor has different tasks. One common model for control decomposition is pipelining or stream parallelism. With pipelining, each task, except the first and last, plays the role of both producer and consumer. A task receives or consumes data, processes that data, and then sends the results on to the next consumer. For example,

consider a set of processes designed to manipulate a video stream. The first process might crop a frame, the second might adjust brightness within the frame, the third might adjust color levels, etc. Each process does something different and will require radically different code.

Note that the second process must wait for the first process to finish before it can begin since the second process consumes and processes the data produced by the first. Similarly, the third process can't begin until the second sends its data, and so on. Getting enough data into the system so that all processes are active is referred to as priming the pipeline. [Figure 15-3](#) shows how processes overlap.

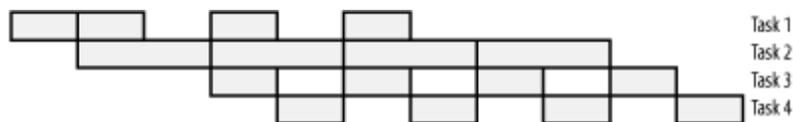
Figure 15-3. Ideal process overlap



You must have a lot more data than processes for this approach to be efficient. Otherwise, the idle time at both the beginning and at the end will render this approach pointless. Granularity is a key consideration here. If the granularity is coarse, priming the pipeline is particularly costly.

A second issue with pipelining is balance. Each of the processes must run for about the same amount of time. If one process takes much longer than the other processes, they will be idle much of the time and overall efficiency will be lost. (This is a likely problem with video processing, for example, as described.) [Figure 15-4](#) shows the effect of having one process take longer. Note the idle time.

Figure 15-4. Process overlap with idle time



However, even though task 2 takes twice as long as the other tasks in this four-task example, there is still a speedup using the pipeline.

A number of algorithms fall between these two extremes. That is, they appear to have elements of both strategies. For example, a common approach in artificial intelligence is to describe algorithms in terms of a search space. A fundamental part of a chess-playing program is to examine a number of different moves to see which appears to be the best. Since it evaluates each move in the same manner, it is reasonable to approach this as a data decomposition problem. Each process will be given a different board configuration, i.e., a different set of data. But once the data has been distributed, the different processes go their different ways. One process may terminate quickly having determined the board position is abysmal (a process known as pruning), while another may be following a hot move recursively through several levels.

15.3 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.

15.3.1 Communication Overhead

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.

In practice, you'll need to take an iterative approach to find the right balance between these two extremes. It may take several tries to work out the details. There are three useful factors. The most important is task size. Keep in mind that tasks may be uniform, i.e., all the same size, or nonuniform. Decomposing into uniform pieces will usually minimize idle time, but this isn't always true. First, you will need to be able to distribute data efficiently so that some processes aren't waiting. Second, if some of the compute nodes are faster than others or if some are more heavily loaded, the benefit of uniformity can be lost and may even be a disadvantage.

Some tasks are inherently nonuniform. Consider searching through an array of data for an item. In one instance, you may be able to find the item very quickly. In another instance, it may take much longer. If two processes are sorting data, depending on the algorithm, the one that receives a nearly sorted set of data may have a tremendous advantage over similar processes sorting a highly random set of data.

In addition to task size, there is the issue of task generation. For some problems, task generation is clearly defined. Task generation is said to be static for these problems. For example, if we want to sort a million numbers, we can clearly determine in advance how we want to generate the tasks. But not all problems are static. Consider the problem of playing chess. The boards you will want to consider will depend on a number of factors that vary from game to game, so they aren't known in advance. Both the number and size of the task will depend on how the pieces are positioned on the board. For such problems, task generation is said to be dynamic.

A third consideration is the communication pattern that the problem will generate. Like tasks, communications may be static (the pattern is known in advance) or dynamic. In general, static communication is easier to program since dynamic communication tends to be unpredictable and error prone.

When programming, there are several very straightforward ways to minimize the impact of communications. First, try to reduce the volume of the data you send. Avoid sending unnecessary data. Can one process duplicate a calculation more efficiently than a pair of processes can exchange a value? Next, try to minimize the number of messages sent. If possible, package data so that it can be sent in a single message rather than as a series of messages. Look for hotspots in your communication pattern. When possible, overlap communications with computation to minimize network congestion. Finally, when feasible, use the collective operations in your message-passing library to optimize communication.

There are a number of other important questions that need to be answered to fully characterize communication patterns. Do all the processes need to communicate with each other or can communication be managed through a single process? Then there is the issue of communication timing, i.e., is communication synchronized? Can all the data be distributed at once, or will it be necessary to update the data as the program runs? Is communication unidirectional or bidirectional? What is the source and destination for data, i.e., does it come from another process, is it sent to another process, or is the filesystem used? There are no right or wrong answers to these questions, but you do need to know the answers to understand what's going on.

15.3.2 Load Balancing

As previously noted, idle time is a major source of overhead. The best way to minimize idle time is to balance the computing requirements among the available processors. There are several sources of idle time in parallel programs. One source is a mismatch between tasks and processors. If you try to run five processes on four processors, two of the processes will be competing for the same processor and will take twice as long as the other processes. Another source of idle time is nonuniform tasks as shown in [Figure 15-4](#). Differences in processor speeds, memory, or workload on cluster nodes can also result in some processes taking longer than expected to complete, leaving other processes idle as they wait to send data to or receive data from those processes.

One way to minimize the overhead resulting from idle time is load balancing. Depending on the context, load balancing can mean different things. In the larger context of operating systems, load balancing may mean running different programs or processes on different machines. In the current context of parallel programming, it refers to a technique of breaking a program into tasks and distributing those tasks based on processor availability.

An example should help. Suppose you have 100 nodes in your cluster, some fast and some slow. If you divide your problem into 100 tasks and send one task to each node, then you won't finish until the slowest, most heavily loaded node finishes. If, however, you divide your problem into 1,000 tasks and write your code so that when a processor finishes one task it receives another, the faster and less loaded processors can take on a larger share of the work while the slower processors will do less. If all goes well, you will finish quicker.

This is the basic idea behind a *work pool*. The work is distributed by maintaining a pool of tasks that are sent to processors whenever a processor becomes idle. Typically, a *master-slave* arrangement is used—one (sometimes more) processor acts as a master distributing work and collecting results, while the remaining processes act as slaves that process a single task, return the results to the master, and wait for their next task. Typically, slaves are idle only toward the end of the program's execution when there are fewer uncompleted tasks than slaves.

In order to use a work pool effectively, you need to reduce the granularity of your tasks so

that you have more tasks than slaves. The key issue, when reducing the granularity, is at what point communication overhead begins to outweigh the benefits of reduced idle time. In general, a work pool works best when the communication overhead is small compared to the amount of computing needed. You should also be aware that the master process can become a bottleneck if it must deal with too many tasks. This may happen if the task size is too small.

Here is the numerical integration problem rewritten using a master-slave, work pool approach.

```
#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;
    MPI_Status status;

    /* problem variables */
    int          i, chunk, numberChunks, 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);

    if (processId == 0)          /* if rank is 0, collect parameters */
    {
        fprintf(stderr, "Enter number of chunk to divide problem into:\n");
        scanf("%d", &numberChunks);
    }
}
```

```

    fprintf(stderr, "Enter number of steps per chunk:\n");

    scanf("%d", &numberRects);

    fprintf(stderr, "Enter low end of interval:\n");

    scanf("%lf", &lowerLimit);

    fprintf(stderr, "Enter high end of interval:\n");

    scanf("%lf", &upperLimit);

}

MPI_Bcast(&numberChunks, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&numberRects, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&lowerLimit, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&upperLimit, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* collect information and print results */

/* if rank is 0, assign chunk, collect results, print results */
if (processId == 0)
{
    total = 0.0;

    if (noProcesses - 1 < numberChunks) chunk = noProcesses - 1;

    else chunk = 0;

    for (i = 1; i <= numberChunks; i++)
    {
        MPI_Recv(&area, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);

        fprintf(stderr, "Area for process %d, is: %f\n", status.MPI_TAG,
                area);

        total = total + area;

        if (chunk != 0 && chunk < numberChunks) chunk++;

        else chunk = 0;

        MPI_Send(&chunk, 1, MPI_INT, status.MPI_TAG, chunk, MPI_COMM_WORLD);
    }

    fprintf(stderr, "The area from %f to %f is: %f\n",

```

```

        lowerLimit, upperLimit, total );
    }
else
/* all other processes, calculate area for chunk and send results */
{
    if (processId > numberChunks) chunk = 0; /* too many processes */
    else chunk = processId;

    while (chunk != 0)
    { /* adjust problem size for subproblem */
        range = (upperLimit - lowerLimit) / numberChunks;

        width = range / numberRects;

        lower = lowerLimit + range * (chunk - 1);

        /* calculate area for this chunk */
        area = 0.0;

        for (i = 0; i < numberRects; i++)
        { at = lower + i * width + width / 2.0;

            height = f(at);

            area = area + width * height;
        }

        /* send results and get next chunk */
        dest = 0;

        MPI_Send(&area, 1, MPI_DOUBLE, dest, processId, MPI_COMM_WORLD);

        MPI_Recv(&chunk, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
            &status);
    }
}

/* finish */

```

```
MPI_Finalize( );  
  
return 0;  
  
}
```

There are two major sets of changes to this code. First, the number of regions (`numberChunks`) is now a parameter entered by the user. Previously, we divided the problem into the same number of regions as processors, i.e., each processor had its own well-defined region to evaluate. Now the total number of regions exceeds (or should exceed) the number of processes. The total number of regions is broadcast to each process so that the process can go ahead and begin calculating the area for its first region.

Process 0 is the master process and no longer calculates the area for a region. Rather, it keeps track of what needs to be done, assigns work, and collects results. All remaining processes are slaves and do the actual work. If one of these is heavily loaded, it may only calculate the area of one region while other, less-loaded nodes may calculate the area of several regions. Notice that a value of 0 for `chunk` signals a slave that no more regions need to be calculated.



15.4 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.

15.4.1 Parallel I/O

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.

ROMIO, from Argonne National Laboratory, is a freely available, portable, high-performance implementation of the MPI-IO standard that runs on a number of different architectures. It is included with MPICH and LAM/MPI and provides interfaces for both C and FORTRAN.

MPI-IO provides three types of data access mechanisms—using an explicit offset, using individual file pointers, or using shared file pointers. It also provides support for several different data representations.

15.4.2 MPI-IO Functions

MPI-IO optimizations include collective I/O, data sieving, and hints. With collective I/O, larger chunks of data are read with a single disk access. The data can then be distributed among the processes as needed. Data sieving is a technique that combines a number of smaller noncontiguous reads into one large read. The system selects and returns the sections

requested by the user and discards the rest. While this can improve disk performance, it can put a considerable strain on memory. Hints provide a mechanism to inform the filesystems about a program's data access patterns, e.g., desired caching policies or striping.

The following code fragment shows how MPI-IO functions might be used:

```
...

#define BUFFERSIZE 1000

...

int buffer[BUFFERSIZE];

MPI_File filehandle;

...

MPI_File_open(MPI_COMM_WORLD, "filename", MPI_MODE_RDONLY, MPI_INFO_NULL,
              &filehandle);

MPI_File_seek(filehandle, processId*BUFFERSIZE*sizeof(int), MPI_SEEK_SET);

MPI_File_read(filehandle, buffer, BUFFERSIZE, MPI_INT, &status);

MPI_File_close(&filehandle);

...
```

The last four function calls would be executed by each process and, like all MPI functions, would be sandwiched between calls to `MPI_Init` and `MPI_Finalize`. In this example, each process opens the file, moves to and reads a block of data from the file, and then closes it.

15.4.2.1 MPI_File_open

`MPI_File_open` is used to open a file. The first argument is the communication group. Every process in the group will open the file. The second argument is the file name. The third argument defines the type of access required to the file. `MPI_MODE_RDONLY` is read-only access. Nine different modes are supported including `MPI_MODE_RDWR` (reading and writing), `MPI_MODE_WRONLY` (write only), `MPI_MODE_CREATE` (create if it doesn't exist), `MPI_MODE_DELETE_ON_CLOSE` (delete file when done), and `MPI_MODE_APPEND` (set the file pointer at the end of the file). C users can use the bit-vector OR (`|`) to combine these constants. The next to last argument is used to pass hints to the filesystem. The constant `MPI_INFO_NULL` is used when no hint is available. Using hints does not otherwise change the semantics of the program. (See the MPI-2 documentation for the rather complex details of using hints.) The last argument is the file handle (on type `MPI_File`), an opaque object used to reference the file once opened.

15.4.2.2 MPI_File_seek

This function is used to position the file pointer. It takes three arguments: the file handle, an offset into the file, and an update mode. There are three update modes: `MPI_SEEK_SET` (set pointer to offset), `MPI_SEEK_CUR` (set pointer to current position plus offset), and `MPI_SEEK_END` (set pointer to end of file plus offset). In this example we have set the pointer to the offset. Notice that `processId` is used to calculate a different offset into the file for each

process.

15.4.2.3 MPI_File_read

`MPI_File_read` allows you to read data from the file specified by the first argument, the file handle. The second argument specifies the address of the buffer, while the third element gives the number of elements in the buffer. The fourth element specifies the type of the data read. The options are the same as with other MPI functions, such as `MPI_Send`. In this example, we are reading `BUFFERSIZE` integers into the array at `buffer`. The last argument is a structure describing the status of read operation. For example, the number of items actually read can be determined from `status` with the `MPI_Get_count` function.

15.4.2.4 MPI_File_close

`MPI_File_close` closes the file referenced by the file handle.

The four new functions in this sample example, along with `MPI_File_write`, are the core functions provided by MPI-IO. However, a large number of other MPI-IO functions are also available. These are described in detail in the MPI-2 documentation.

15.4.3 Random Numbers

Generating random (or pseudorandom) numbers presents a particular problem for parallel programming. Pseudorandom number generators typically produce a stream of "random" numbers where the next random number depends upon previously generated random numbers in some highly nonobvious way.^[3] While the numbers appear to be random, and are for most purposes, they are in fact calculated and reproducible provided you start with the same parameters, i.e., at the same point in the stream. By varying the starting parameters, it will appear that you are generating a different stream of random numbers. In fact, you are just starting at different points on the same stream. The period for a random number generator is the number of entries in the stream before the stream starts over again and begins repeating itself. For good random number generators, the periods are quite large and shouldn't create any problems for serial programs using random number generators.

^[3] As you can imagine, coming up with a good generator is very, very tricky.

For parallel programs, however, there are some potential risks. For example, if you are using a large number of random numbers on a number of different processors and using the same random number generator on each, then there is a chance that some of the streams will overlap. For some applications, such as parallel Monte Carlo simulations, this is extremely undesirable.

There are several ways around this. One approach is to have a single process serve as a random number generator and distribute its random numbers among the remaining processes. Since only a single generator is used, it is straightforward to ensure that no random number is used more than once. The disadvantage to this approach is the communication overhead required to distribute the random numbers. This can be minimized, somewhat, by distributing blocks of random numbers, but this complicates programming since each process must now manage a block of random numbers.

An alternative approach is to use the same random number generator in each process but to use different offsets into the stream. For example, if you are using 100 processes, process 0 would use the 1st, 101st, 201st, etc., random numbers in the stream. Process 1 would use the 2nd, 102nd, 202nd, etc., random numbers in the stream, etc. While this eliminates communication overhead, it adds to the complexity of the program.

Fortunately, there are libraries of random number generators designed specifically for use with parallel programs. One such library is *Scalable Parallel Random Number Generators* (*SPRNG*). This library actually provides six different state-of-the-art random number generators ([Table 15-1](#)). SPRNG works nicely with MPI. (You'll need to download and install SPRNG before you can use it. See [Chapter 9](#) for details.)

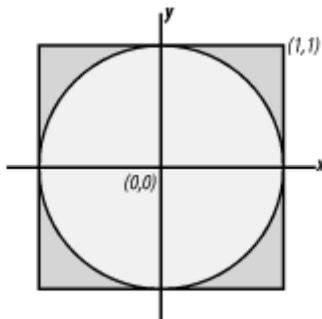
Table 15-1. SPRNG's random number generators

Code	Generator
0	Additive Lagged Fibonacci Generator
1	48-bit Linear Congruential Generator with Prime Addend
2	64-bit Linear Congruential Generator with Prime Addend
3	Combined Multiple Recursive Generator
4	Multiplicative Lagged Fibonacci Generator
5	Prime Modulus Linear Congruential Generator

To give you an idea of how to use SPRNG, we'll look at a simple Monte Carlo simulation that estimates the value of π (in case you've forgotten). The way the simulation works is a little like throwing darts.

Imagine throwing darts at a dart board with a circle in the center like the one in [Figure 15-5](#). Assuming that you are totally inept, the darts could land anywhere, but ignore those that miss the board completely. If you count the total number of darts thrown (`total`) and if you count those that land in the circle (`in`), then for random tosses, you'd expect the ratio `in/total` to be just the ratio of the area of the circle to the square. If the square is 1 foot on a side, the area of the circle is $\pi/4$ square feet. Using this information, you can estimate π as $4 * in / total$, i.e., four times the ratio of the area of the circle to the area of the square.

Figure 15-5. Monte Carlo dartboard



You'll need to throw a lot of darts to get a reasonable estimate. If you know a lot of inept dart enthusiasts, you can recruit them. Each one throws darts and keeps track of their total. If you add the results, you should get a better estimate.

The code that follows uses this technique to estimate the value of π . Multiple processes are used to provide a larger number of tosses and a better estimate. The code uses SPRNG to generate separate streams of random numbers, one for each process.

```

#include <stdio.h>

#include <math.h>

#include "mpi.h"

#define SIMPLE_SPRNG /* simple interface */

#define USE_MPI /* MPI version of SPRNG */

#include "sprng.h"

main(int argc, char *argv[ ])
{
    int i, in, n, noProcesses, processId, seed, total;

    double pi;

    n = 1000;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);

    MPI_Comm_rank(MPI_COMM_WORLD, &processId);

    seed = make_sprng_seed( );

    init_sprng(3, seed, SPRNG_DEFAULT);

    print_sprng( );

    in = hits(n);

    MPI_Reduce(&in, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    /* estimate and print pi */

    if(processId == 0)

    { pi = (4.0 * total) / (n * noProcesses);

```

```

    printf("Pi estimate: %18.16f \n", pi);

    printf("Number of samples: %12d \n", n * noProcesses);
}

MPI_Finalize( );
}

/* count darts in target */

int hits(int n)
{
    int i, in = 0;

    double x, y;

    for (i = 0; i < n; i++)
    {
        x = sprng( );

        y = sprng( );

        if (x * x + y * y < 1.0) in++;
    }

    return in;
}

```

For simplicity, this code considers only the top right-hand corner (one-fourth) of the board. But since the board is symmetric, the ratio of the areas is the same. The code simulates randomly throwing darts by generating a pair of random numbers for the coordinates of where the dart might land, and then looks to see if they are within the circle by calculating the distance to the center of the circle.^[4] This is done in the function `hits`.

^[4] Strictly speaking, it is using the square of the distance to avoid evaluating a square root for each point, a costly operation.

Apart from the SPRNG code, shown in boldface, everything should look familiar. For MPI programming, before including the SPRNG header file, you need to define two macros. In this example, the macro `SIMPLE_SPRNG` is used to specify the simple interface, which should be adequate for most needs. The alternative or *default* interface provides for multiple streams per process. The macro `USE_MPI` is necessary to let the `init_sprng` routine make the necessary MPI calls to ensure separate streams for each process.

Before generating random numbers, a seed needs to be generated and an initialization routine called. The routine `make_sprng_seed` generates a seed using time and date information from

the system. When used with MPI, it broadcasts the seed to all the processes. `init_sprng` initializes the random number streams. (This call can be omitted if you want to use the defaults.) The first argument to `init_sprng` is an integer from 0 to 5 inclusive, specifying which of the random number generators to use. [Table 15-1](#) gives the possibilities. The second argument is the seed, an encoding of the start state for the random number generator, while the third argument is used to pass additional parameters required by some generators.

The call to `print_sprng`, also optional, will provide information about each of the streams as shown in the output below. Finally, to generate random numbers, a double between 0 and 1, the call `sprng` is used as seen in the `hits` routine.

Here is an example of compiling the code. On this system, SPRNG has been installed in the directory `/usr/local/src/sprng2.0`.

```
[sloanjd@fanny SPRNG]$ mpicc pi-mpi.c -I/usr/local/src/sprng2.0/include \  
>-L/usr/local/src/sprng2.0/lib -lsprng -lm -o pi-mpi
```

Note the inclusion of path and library information. (Look at the `Makefile` file in the `EXAMPLES` subdirectory in the installation tree for more hints on compiling.)

Here is part of the output for this program.

```
[sloanjd@fanny SPRNG]$ mpirun -np 4 pi-mpi
```

```
Combined multiple recursive generator
```

```
seed = 88724496, stream_number = 0      parameter = 0
```

```
Pi estimate: 3.0930000000000000
```

```
Number of samples:      4000
```

```
Combined multiple recursive generator
```

```
seed = 88724496, stream_number = 2      parameter = 0
```

```
...
```

The output for two of the streams is shown. It is similar for the other streams. If rounded, the answer is correct to two places. That's using 4,000 darts.

The documentation for SPRNG provides a number of the details glossed over here. And the installation also includes a large number of detailed examples.

These two examples, I/O and random numbers, should give you an idea of the types of problems you may encounter when writing parallel code. Dealing with problem areas like these may be critical when determining how your programs should be designed. At other times, performance may hinge on more general issues such as balancing parallelism with

overhead. It all depends on the individual problem you face. While good design is essential, often you will need to tweak your design based on empirical measurements. [Chapter 17](#) provides the tools you will need to do this.

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.

16.1 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.

This shouldn't intimidate you. While you may occasionally see explicitly parallel problems, most of the problems you are likely to see are not new. They are the same mistakes you'll have made with serial code. This is good news! It means you should already be familiar with most of the problems you'll see. It also suggests a strategy for developing and debugging code.

Start, whenever possible, with a serial version of the code. This will help you identify potential problems and work out the details of your code. Once you have the serial version fully debugged, you can move on to the parallel version. Depending on the complexity of the problem, the next step may be running the code with a small number of processes on the same machine. Only after this is working properly should you scale up the problem.

Since most problems are serial, we'll start with a quick review of debugging in general and then look at how we can expand traditional techniques to parallel programs.

16.2 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.

16.3 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 `lint` checks for are now caught by modern compilers (if you use the right flags). Fortunately, `lint` has been superceded with more extensive checkers. If you are running Linux, you probably already have `splint` installed.

Here is an example of using `splint`. The `-I` option is used to include the path to the file `mpi.h`.

```
[sloanjd@amy AREA]$ splint -I/opt/lam-7.0/include rect.c
```

```
Splint 3.0.1.7 --- 24 Jan 2003
```

```
...
```

```
rect.c:80:13: Return value (type int) ignored: MPI_Recv(&chunk,...
```

```
rect.c:85:5: Return value (type int) ignored: MPI_Finalize( )
```

```
Finished checking --- 29 code warnings
```

Most of the output has been deleted (out of embarrassment), but you should get the idea. Of course, this is a working program. It just could be better.

There are a number of other tools that you might want to investigate. For example, memory checkers will examine your code for potential memory leaks. Probably the best known is the

commercial product *purify*, but you might want to look at the GNU product *checkgcc*. Symbolic debuggers are described later in this chapter. And don't overlook profilers (described in [Chapter 17](#)). While not designed as debugging tools, they frequently reveal lots of problems.

16.4 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:

```
#include "mpi.h"

#include <stdio.h>

int main( int argc, char * argv[ ] )
{
    int datum1 = 19, datum2 = 23, datum3 = 27;

    int datum4, datum5, datum6;

    int noProcesses, processId;

    MPI_Status status;

    /* MPI setup */

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
```

```

MPI_Comm_rank(MPI_COMM_WORLD, &processId);

if (processId == 0)          /* for rank 0 */
{
    MPI_Recv(&datum4, 1, MPI_INT, 2, 3, MPI_COMM_WORLD, &status);

    MPI_Send(&datum1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    fprintf (stderr, "Received: %d\n", datum4);
}

else if (processId == 1)    /* for rank 1 */
{
    MPI_Recv(&datum5, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

    MPI_Send(&datum2, 1, MPI_INT, 2, 1, MPI_COMM_WORLD);

    fprintf (stderr, "Received: %d\n", datum5);
}

else                        /* for rank 2 */
{
    MPI_Recv(&datum6, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);

    MPI_Send(&datum3, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);

    fprintf (stderr, "Received: %d\n", datum6);
}

MPI_Finalize( );

return 0;
}

```

This code doesn't do anything worthwhile other than illustrate deadlock. It is designed to be run with three processes. You'll notice that each process waits for another process to send it information before it sends its own information. Thus process 0 is waiting for process 1 which is waiting for process 2 which is waiting for process 0. If you run this program, nothing happens—it hangs.

While this example is fairly straightforward and something that you probably could diagnose simply by reading the source, other examples of deadlock can be quite subtle and extraordinarily difficult to diagnose simply by looking at the source code.

Deadlock is one of the most common problems you'll face with parallel code. Another common problem is mismatching parameters in function calls, particularly MPI functions. This is something that you can check carefully while rereading your code.

16.5 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.

A technique that is often used with *printf*s is deleting extraneous code. The idea is, after making a copy of your program, to start deleting code and retesting to see whether the problem has disappeared. The goal is to produce the smallest piece of code that still exhibits the problem. This can be useful with some types of problems, particularly when you are trying to piece together how some feature of a language works. It can also be helpful when generating a bug report.

With parallel code, the `printf` approach can be problematic. Earlier in this book, you saw examples of how the output from different processes could be printed in a seemingly arbitrary order. Buffering further complicates matters. If your code is crashing, a process may die before its output is displayed. That output will be lost. Also, output can change timings which can limit its effectiveness if you are dealing with a race problem. Finally, print statements can seriously deteriorate performance.

If you are going to use the `printf` approach with parallel programs, there are two things you should do. First, if there is any possibility of the source of the output being confused, be sure to label the output with the process number or machine name. Second, follow your calls to `printf` with a call to `fflush` so that the output is actually printed at the moment the program generates it. For example,

```
...  
  
int  processId;  
  
char  processName[MPI_MAX_PROCESSOR_NAME];  
  
...  
  
MPI_Comm_rank(MPI_COMM_WORLD, &processId);  
  
MPI_Get_processor_name(processName, &nameSize);  
  
...  
  
fprintf(stdout, "Process %d on %s at checkpoint 1. \n", processId,  
processName);  
  
fflush(stdout);  
  
...
```

If you want to control the order of the output, you'll need to have the master process coordinate output.



16.6 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.^[1] You may need to install *ddd*, but the process is straightforward and is described in [Chapter 9](#).

[1] There are other friendly ways of running *gdb*. *xxgdb* is an X Windows System version. *gdb* is often run from within Emacs.

16.6.1 gdb

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;
}

```

If you try to run this, it doesn't print anything and doesn't return. With that kind of behavior, it is pretty easy to guess that there is something wrong with the loop. But let's play dumb for a moment and see how we could discover this using *gdb*.

Before you can run *gdb* with a program, you should compile that program with the `-g` option.

```
[sloanjd@amy DEBUG]$ gcc -g area.c -o area
```

The `-g` option generates code that produces debugging information. This is necessary to make the symbol table available so that you can refer to variables by name.

Unlike most compilers, *gcc* will allow you to use an optimization option (`-O`) with `-g`. Keep in mind that optimizing code may reorder instructions or may eliminate variables. This can be mind-boggling to a debugger so, in general, you should avoid optimizing your code when you plan to use a symbolic debugger. With *gcc*, you have some latitude, but beware!

Once the code is properly compiled, you can start *gdb*. There are several ways to do this, but the simplest is to pass the name of the program as a command-line argument. For example,

```
[sloanjd@amy DEBUG]$ gdb -q area
```

```
(gdb)
```

In this case, the `-q` option is used to suppress the general information that it prints by default.

When the program is loaded and ready to go, it returns the (gdb) prompt. You can get a list of command categories by typing `help` at the prompt. To see a listing of the commands within a category, type `help` followed by the category name, for example, `help data`. Commands may be abbreviated provided the shortened name is unambiguous. For example, `append` may be shortened to `app` but not to `ap` since it would be confused with `apropos`.

The most reasonable place to start is probably the `list` command (abbreviated `l`). `list` will begin listing your program, 10 lines at a time.

```
(gdb)l
7
8     double f(double x)
9     {
10        return x*x;
11    }
12
13    int main ( int argc, char * argv[ ] )
14    {
15        int i;
16        double        area = 0.0;
```

If you continue with `list`, it will display the next 10 lines of code. If you give it a single numeric value, it will list 10 lines starting at that line. If you give it two numeric values separated by a comma, it will treat those values as a range and print that code. For example,

```
(gdb)l 18,20
18        double        at, height;
19
20        for (i = 0; i <= numberSteps; i--)
```

If you enter `help list` at the prompt, you'll see a list of additional ways to use `list`.

Next, let's put a *breakpoint* on line 20. A breakpoint allows you to start a program and have it automatically stop when it reaches the target line. If the line is never reached, e.g., it is embedded in a conditional statement that fails, then the code won't stop. If the line is executed several times, such as a breakpoint within a loop, it will stop each time.

```
(gdb)b 20
```

```
Breakpoint 1 at 0x804836e: file area.c, line 20.
```

You can list breakpoints with the `info breakpoint` command. Type `help b` at the prompt to learn more breakpoints and the commands that can be used with them. (gdb also supports *watchpoints*, which stop when a watched variable changes, and *catchpoints*, which catch an exception).

Now let's run the program.

```
(gdb)run
```

```
Starting program: /home/sloanjd/DEBUG/area
```

```
Breakpoint 1, main (argc=1, argv=0xbfffe774) at area.c:20
```

```
20         for (i = 0; i <= numberSteps; i-)
```

You'll note that it stopped at our breakpoint as expected.

Let's look at a few variables to make sure everything has been initialized correctly.

```
(gdb)print area
```

```
$1 = 0
```

```
(gdb)print step
```

```
$2 = 0.059999999999999998
```

So far, everything looks good.

```
(gdb)print numberSteps
```

```
No symbol "numberSteps" in current context.
```

This may look like a problem, but it isn't. You'll recall that `numberSteps` isn't a program variable. It was defined with a `#define` statement. The preprocessor substitutes the value for the name throughout the program before compilation, so we won't be able to look at this with the debugger. That's not a big problem but something you should be aware of.

We can step through individual lines of code with the `next` command.

```
(gdb)n
```

```
21         {         at = lowerLimit + i * step + step / 2.0;
```

```
(gdb)n
```

```
22                 height = f(at);
```

```
(gdb)n
```

```
23                 area = area + step * height;
```

```
(gdb)n
```

```
24                 for (i = 0; i <= numberSteps; i++)
```

The `step` command is just like the `next` command except that `next` will treat a subroutine call as one instruction while `step` will enter into the subroutine.

We'll come back to `step` after we have looked at some of the variables.

```
(gdb)print area
$3 = 0.24725399999999992
(gdb)print height
$4 = 4.1208999999999989
(gdb)print step * height
$5 = 0.24725399999999992
```

Notice that `print` will handle expressions as well as simple variables, a real convenience. Everything still looks good.

Going back to `step`, here is the second iteration of the loop traced with `step`.

```
(gdb)s
21      {      at = lowerLimit + i * step + step / 2.0;
(gdb)s
22          height = f(at);
(gdb)s
f (x=2.0899999999999999) at area.c:10
10      return x*x;
(gdb)s
11      }
(gdb)s
main (argc=1, argv=0xbfffe774) at area.c:23
23          area = area + step * height;
(gdb)s
20      for (i = 0; i <= numberSteps; i++)
```

Notice that we are diving into the function `f`.

The body of the loop seems to be working correctly. Maybe there is something wrong with the print statement? To examine the values it is getting, we'll set a second breakpoint and resume execution.

```
(gdb)1 25,27
25
26      printf ("The area from %f to %f is: %f\n",
27          lowerLimit, upperLimit, area );
```

```
(gdb)b 26
```

```
Breakpoint 2 at 0x80483c4: file area.c, line 26.
```

```
(gdb)continue
```

```
Continuing.
```

At this point the program hangs. Since the body of the loop looks OK and we aren't getting to the `printf`, there must be something wrong with the loop control structure.

Let's interrupt the program (CTRL-C) and examine the counter `i`.

```
Program received signal SIGINT, Interrupt.
```

```
0x0804833d in f (x=-83775291.069999993) at area.c:10
```

```
10         return x*x;
```

```
(gdb)print i
```

```
No symbol "i" in current context.
```

```
(gdb)n
```

```
11     }
```

```
(gdb)n
```

```
main (argc=1, argv=0xbfffd774) at area.c:23
```

```
23         area = area + step * height;
```

```
(gdb)print i
```

```
$7 = -1396254885
```

When the program was interrupted, we were in the function `f` so `i` was out of scope. We needed to step through a couple of instructions to return to the main program to examine `i`. And when we did, we saw that something was obviously wrong.

We can change the value of `i` and continue.

```
(gdb)set var i=51
```

```
(gdb)continue
```

```
Continuing.
```

```
Breakpoint 2, main (argc=1, argv=0xbfffd774) at area.c:26
```

```
26     printf ("The area from %f to %f is: %f\n",
```

With an appropriate value of `i`, we exit the loop. Clearly, `i` isn't being updated appropriately.

We can continue until the end now, although our output won't make much sense, and then exit *gdb*.

```
(gdb)continue
```

Continuing.

```
The area from 2.000000 to 5.000000 is: 203220027199808325287936.000000
```

```
Program exited normally.
```

```
(gdb)q
```

Or we could have just quit where we were.

No doubt you noticed that the code had been changed from `i++` to `i--` long before the end of this section. This is definitely a problem that rereading the code should have found. Nevertheless, you should have an idea of how to use *gdb* at this point.

16.6.2 ddd

Data Display Debugger is a frontend for command-line debuggers (or inferior debugger, in *ddd* parlance). We'll use it with *gdb*, but it is not limited to *gdb*. You must have the X Window System running. Since *ddd* is a frontend to *gdb* and you already know how to use *gdb*, there isn't much new to learn. But *ddd* does have a few nice tricks. Although we won't go into it here, one of *ddd*'s real strengths is displaying complex data structures such as linked lists.

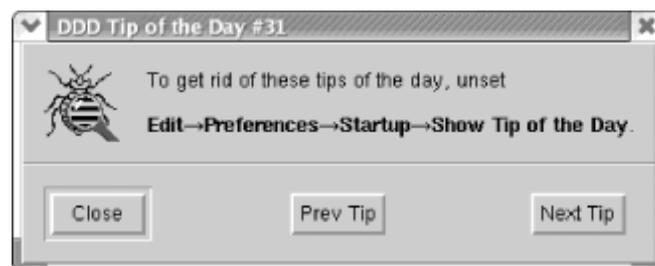
As with *gdb*, compile your program with the `-g` option. Next, open *ddd* with executable as an argument.

```
[sloanjd@amy DEBUG]$ ddd area
```

ddd's splash screen will appear briefly and then three windows will open. The top window is the *ddd* Tip of the Day window [\[2\]](#) as shown in [Figure 16-1](#).

^[2] Tip #31, the tip in this figure, tells you how to get rid of the *ddd* Tip of the Day.

Figure 16-1. ddd Tip of the Day #31

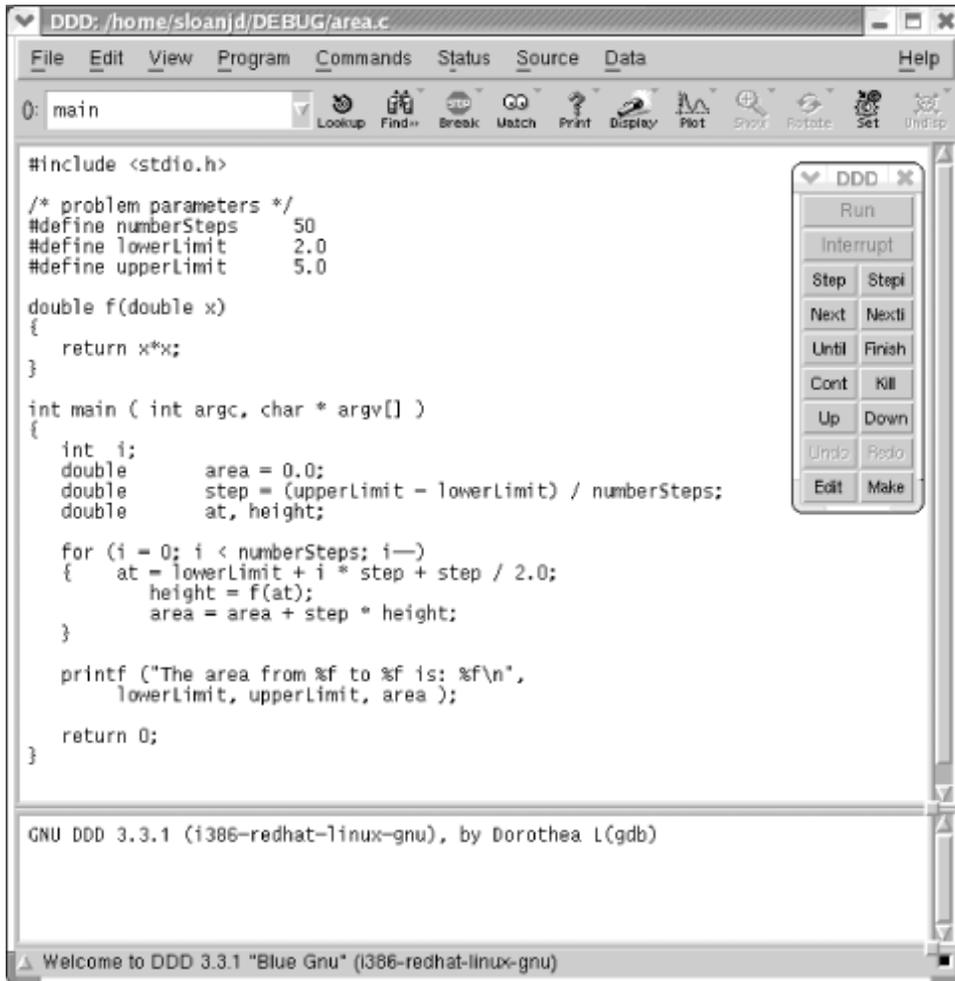


Read the tip, if you like, and then close the window.

The large window underneath the tip window is the main window you'll be working from. Off to the side you'll see a smaller window with a few *ddd* commands. The small command

window can be repositioned so that it doesn't overlap with the main window if you wish. [Figure 16-2](#) shows both of these windows.

Figure 16-2. ddd's main window



The window is pretty self-explanatory. The upper pane holds your source code, while the lower pane is a text interface to *gdb*. You can type *gdb* commands in the lower pane just as you did on the command line. *gdb* commands are also available via the menus at the top of the window, or you can use the command window to enter the most common commands. For example, if you want to edit the source, you can type `edit` in the command window (just as you would in *gdb*) or you can click on the edit button. Either way, you'll be thrown into an editor. (Sorry, you can't edit it directly in the upper pane.)

To add a breakpoint, you can select a line in the upper pane and then click on the break button (with the stop sign) on the tool bar. As you step through the code, a large green arrow at the edge of the top pane points to the current line. If you move the cursor over a variable, after a few seconds, a pop-up box will display the variable's current values.

The display can be reconfigured if you wish. For example, if you want to look at the machine code in addition to (or instead of) the source listings, you can open a machine language window (and close the source window). You can also resize windows and change fonts to your heart's content.

16.7 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.^[3] 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.

^[3] Actually, with some versions of *mpirun*, LAM/MPI, for instance, it is possible to start a debugger directly. Since this won't always work, a more general approach is described here.

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.

With this information, we can start a debugger.

```
[sloanjd@amy DEBUG]$ gdb -q area 19338
```

```
Attaching to program: /home/sloanjd/DEBUG/area, process 19338
```

```
Reading symbols from /lib/tls/libc.so.6...done.
```

```
Loaded symbols for /lib/tls/libc.so.6
```

```
Reading symbols from /lib/ld-linux.so.2...done.
```

```
Loaded symbols for /lib/ld-linux.so.2
```

```
0x080483a1 in main (argc=1, argv=0xbfffe4) at area.c:22
```

```
22             height = f(at);
```

```
(gdb)
```

When we attach to it, the program will stop running. It is now under our control. Of course, part of the program will have executed before we attached to it, but we can now proceed with our analysis using commands we have already seen.

Let's do the same thing with the deadlock program presented earlier in the chapter. First we'll compile and run it.

```
[sloanjd@amy DEADLOCK]$ mpicc -g dlock.c -o dlock
```

```
[sloanjd@amy DEADLOCK]$ mpirun -np 3 dlock
```

Notice that the `-g` option is passed transparently to the compiler. Don't forget to include it. (If you get an error message that the source is not available, you probably forgot.)

Then look for the process number and start *ddd*.

```
[sloanjd@amy DEADLOCK]$ ps -aux | grep dlock
```

```
sloanjd 19473  0.0  0.5 1600  676 pts/4    S   10:16   0:00 mpirun -np 3
```

```
dlock
```

```
sloanjd 19474  0.0  0.7 1904  904 ?        S   10:16   0:00 dlock
```

```
sloanjd 19475  0.0  0.5 3572  632 pts/3    S   10:17   0:00 grep dlock
```

```
[sloanjd@amy DEADLOCK]$ ddd dlock 19474
```

Notice that we see both the *mpirun* and the actual program. We are interested in the latter.

Once *ddd* is started, we can go to Status Backtrace to see where we are. A *backtrace* is a list of the functions that called the current one, extending back to the function with which the program began. As you can see in [Figure 16-3](#), we are at line 19, the call to `MPI_Recv`.

Figure 16-3. *ddd* with Backtrace



If you want to see what's happening on another processor, you can use `ssh` to connect to the machine and repeat the process. You will need to change to the appropriate directory so that the source will be found. Also, of course, the process number will be different so you must check for it again.

```
[sloanjd@amy DEADLOCK]$ ssh oscarnode1
```

```
[sloanjd@oscarnode1 sloanjd]$ cd DEADLOCK
```

```
[sloanjd@oscarnode1 DEADLOCK]$ ps -aux | grep dlock
```

```
sloanjd 23029 0.0 0.7 1908 896 ? S 10:16 0:00 dlock
```

```
sloanjd 23107 0.0 0.3 1492 444 pts/2 S 10:39 0:00 grep dlock
```

```
[sloanjd@oscarnode1 DEADLOCK]$ gdb -q dlock 23029
```

```
Attaching to program: /home/sloanjd/DEADLOCK/dlock, process 23029
```

```
Reading symbols from /usr/lib/libaio.so.1...done.
```

```
Loaded symbols for /usr/lib/libaio.so.1
```

```
Reading symbols from /lib/libutil.so.1...done.
```

```
Loaded symbols for /lib/libutil.so.1
```

```
Reading symbols from /lib/tls/libpthread.so.0...done.
```

```
[New Thread 1073927328 (LWP 23029)]
```

```
Loaded symbols for /lib/tls/libpthread.so.0
```

```
Reading symbols from /lib/tls/libc.so.6...done.
```

```
Loaded symbols for /lib/tls/libc.so.6
```

```
Reading symbols from /lib/ld-linux.so.2...done.
```

```
Loaded symbols for /lib/ld-linux.so.2
```

```
Reading symbols from /lib/libnss_files.so.2...done.
```

```
Loaded symbols for /lib/libnss_files.so.2
```

```
0xffffe002 in ?? ( )
```

```
(gdb)bt
```

```
#0 0xffffe002 in ?? ( )
```

```
#1 0x08066a23 in lam_ssi_rpi_tcp_low_fastrecv ( )
```

```
#2 0x08064dbb in lam_ssi_rpi_tcp_fastrecv ( )
```

```
#3 0x080575b4 in MPI_Recv ( )
```

```
#4 0x08049d4c in main (argc=1, argv=0xbfffdb44) at dlock.c:25
```

```
#5 0x42015504 in __libc_start_main ( ) from /lib/tls/libc.so.6
```

The back trace information is similar. The program is stalled at line 25, the `MPI_Recv` call for process with rank 1. `gdb` was used since this is a text-based window. If the node supports X Window System (by default, an OSCAR compute node won't), I could have used `ddd` by specifying the head node as the display.

16.8 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.

17.1 Why Profile?

You have probably heard it before—the typical program will spend over 90% of its execution time in less than 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.^[1]

^[1] In this chapter *optimization* means optimizing the time a program spends executing. Space optimizations will be ignored.

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 $O(n^2)$ 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 *asymptotic analysis*.

Asymptotic analysis can be problematic with parallel programs for several reasons. First, it may be difficult to estimate the cost of communications required by a parallel solution. This can be further complicated by the need for additional code to coordinate communications among the processors. Second, there is often a less than perfect overlap among the communicating processes. A processor may be idle while it waits for its next task (as with our numerical integration programs in earlier chapters). In particular, it may be difficult to predict when a processor will be idle and what effect this will have on overall performance. For these and other reasons, an empirical approach to estimating performance is often the preferred approach for parallel programs. That is, we directly measure performance of existing programs.

Thus, with parallel programs, the most appropriate strategy is to select the best algorithm you can and then empirically verify its actual performance.

17.2 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.

Larger problems that you may be able to identify and address include problems with memory access, I/O (I/O is always expensive), load balancing and task granularity, and communication patterns. Basically, anything that results in idle processors is worth examining.

Your extreme hotspots will be blocks of code that are executed repeatedly. These typically occur within loops or, especially, nested loops. For these, some hand optimization may be worthwhile. A number of techniques may be used, but they all boil down to eliminating unnecessary operations. Basically, you'll need to focus on and locate the instructions in question and look for ways to eliminate the number of instructions or replace them with less costly instructions. For example, moving instructions out of a loop will reduce the number of times the instructions are executed, while replacing an exponentiation with a multiplication can reduce the cost of an individual instruction.

A detailed description of the various techniques that can be used is outside the scope of this book. Several sources are listed in the [Appendix A](#). The remainder of this chapter describes tools that will help you locate inefficient code.

17.3 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.

If you don't want to worry about shell-specific commands, you can get pretty much the same information if you use Linux's external *time* command.

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

```
5.32user 0.00system 0:06.51elapsed 81%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (66major+12minor)pagefaults 0swaps
```

And if you want to be overwhelmed with information, use the `-v` option. (You might try `/bin/time` if you aren't running Linux.)

With MPICH or LAM/MPI you can run the `time` command by simply inserting it on the command line before the call to `mpirun`.

```
[sloanjd@amy PROFILE]$ time mpirun -np 4 rect
```

(This is not guaranteed to work with all versions of MPI.)

While you'll have already formed an opinion as to whether your code is taking too long well before you get around to timing it, `time` does let you to put some number on your impression so that you'll sound more professional when complaining about system performance. Also, the difference between the wall-clock time and the time your program takes will give you an idea of how much of the time is caused by your code and how much of the time depends on system load. (Of course, if you are timing an entire program, you needn't be concerned about any code reordering caused by optimizing compilers.) Finally, several timings with different input sizes can be used to get a very rough idea of how your program will scale.



17.4 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 `time` is that the granularity (1 second) is too large for many tasks.

You can get around the granularity problem by using a different function, `gettimeofday`, which provides microsecond granularity. `gettimeofday` is used with a structure composed of two long integers, one for seconds and one for microseconds. Its use is slightly more complicated. Here is an example:

```
...

#include <sys/time.h>

int main(void)
{
    struct timeval start, finish;

    struct timezone tz;

    ...

    gettimeofday(&start, &tz);

    printf("Time---seconds: %d  microseconds: %d \n",
           start.tv_sec, start.tv_usec);

    /* section to be timed */

    ...

    gettimeofday(&finish, &tz);

    printf("Time---seconds: %d  microseconds: %d \n",
           finish.tv_sec, finish.tv_usec);

    printf("\nElapsed time---seconds: %d  microseconds: %d \n",
           ((start.tv_usec > finish.tv_usec) ?
            finish.tv_sec - start.tv_sec - 1 :
            finish.tv_sec - start.tv_sec),
           (start.tv_usec > finish.tv_usec) ?
            1000000 + finish.tv_usec - start.tv_usec :
            finish.tv_usec - start.tv_usec);

    return 0;
}
```

```
}
```

The first argument to `gettimeofday` is the structure for the time. The second is used to adjust results for the appropriate time zone. Since we are interested in elapsed time, the time zone is treated as a dummy argument. The first two `printfs` in this example show how to display the individual counters. The last `printf` displays the elapsed time. Because two numbers are involved, calculating elapsed time is slightly more complicated than with `time`.

Keep in mind that both `time` and `gettimeofday` return wall-clock times. If the process is interrupted between calls, the elapsed time that you calculate will include the time spent during the interruption, even if it has absolutely nothing to do with your program. On the other hand, these functions should largely (but not completely) be immune to problems caused with code reordering for compiler optimizations, provided you stick to timing basic blocks. (A basic block is a block of contiguous code that has a single entry point at its beginning and a single exit point at its end).

Typically, timing commands are placed inside `#ifdef` statements so that they can be compiled only as needed. Other languages, such as FORTRAN, have similar timing commands. However, what's available varies from compiler to compiler, so be sure to check the appropriate documentation for your compiler.

17.4.1 Manual Timing with MPI

With the C library routines `time` and `gettimeofday`, you have to choose between poor granularity and the complications of dealing with a structure. With parallel programs, there is the additional problem of synchronizing processes across the cluster.

17.4.2 MPI Functions

MPI provides another alternative, three additional functions that can be used to time code.

17.4.2.1 MPI_Wtime

Like `time`, the function `MPI_Wtime` returns the number of seconds since some point in the past. Although this point in the past is not specified by the standard, it is guaranteed not to change during the life of a process. However, there are no guarantees of consistency among different processes across the cluster. The function call takes no arguments. Since the return value is a double, the function can provide a finer granularity than `time`. As with `time`, `MPI_Wtime` returns the wall-clock time. If the process is interrupted between calls to `MPI_Wtime`, the time the process is idle will be included in your calculated elapsed time. Since `MPI_Wtime` returns the time, unlike most MPI functions, it cannot return an error code.

17.4.2.2 MPI_Wtick

`MPI_Wtick` returns the actual resolution or granularity for the time returned by `MPI_Wtime` (rather than an error code). For example, if the system clock is a counter that is incremented every microsecond, `MPI_Wtick` will return a value of roughly 0.000001. It takes no argument and returns a double. In practice, `MPI_Wtick`'s primary use is to satisfy the user's curiosity.

17.4.2.3 MPI_Barrier

One problem with timing a parallel program is that one process may be idle while waiting for

another. If used naively, `MPI_Wtime`, `time`, or `gettimeofday` could return a value that includes both running code and idle time. While you'll certainly want to know about both of these, it is likely that the information will be useful only if you can separate them. `MPI_Barrier` can be used to synchronize processes within a communication group. When `MPI_Barrier` is called, individual processes in the group are blocked until all the processes have entered the call. Once all processes have entered the call, i.e., reached the same point in the code, the call returns for each process, and the processes are no longer blocked. `MPI_Barrier` takes a communicator as an argument and, like most MPI functions, returns an integer error code.

Here is a code fragment that demonstrates how these functions might be used:

```
...

#include "mpi.h"

...

int main( int argc, char * argv[ ] )
{
    double      start, finish;

    ...

    MPI_Barrier(MPI_COMM_WORLD);

    start = MPI_Wtime( );

    /* section to be timed */

    ...

    MPI_Barrier(MPI_COMM_WORLD);

    finish = MPI_Wtime( );

    if (processId == 0)

        fprintf(stderr, "Elapsed time: %f\n", finish-start);

    ...
}
```

Depending on the other code in the program, one or both of the calls to `MPI_Barrier` may not be essential. Also, when timing short code segments, you shouldn't overlook the cost of measurement. If needed, you can write a short code segment to estimate the cost of the calls to `MPI_Barrier` and `MPI_Wtime` by simply repeating the calls and calculating the difference.

17.4.3 PMPI

If you want to time MPI calls, MPI provides a wrapper mechanism that can be used to create profiling interface. Each MPI function has a dual function whose name begins with `PMPI` rather than `MPI`. For instance, you can use `PMPI_Send` just as you would `MPI_Send`, `PMPI_Recv` just as you would `MPI_Recv`, and so on. What this allows you to do is write your own version of any function and still have a way to call the original function. For example, if you want to write your own version of `MPI_Send`, you'll still be able to call the original version by simply calling

its dual, `PMPI_Send`. Of course, to get this to work, you'll need to link to your library of customized functions before you link to the standard MPI library.

Interesting, you say, but how is this useful? For profiling MPI commands, you can write a new version of any MPI function that calls a timing routine, then calls the original version, and, finally, calls the timing routine again when the original function returns. Here is an example for `MPI_Send`:

```
int MPI_Send(void * buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm)
{
    double start, finish;

    int err_code;

    start = MPI_Wtime( );

    err_code = PMPI_Send(buf, count, datatype, dest, tag, comm);

    finish = MPI_Wtime( );

    fprintf(stderr, "Elapsed time: %f\n", finish - start);

    return err_code;
}
```

For this function definition, the parameter list was copied from the MPI standard. For the embedded MPI function call, the return code from the call to `PMPI_Send` is saved and passed back to the calling program. This example just displays the elapsed time. An alternative would be to return it through a global variable or write it out to a file.

To use this code, you need to ensure that it is compiled and linked before the MPI library is linked into your program. One neat thing about this approach is that you'll be able to use it with precompiled modules even if you don't have their source. Of course it is a lot of work to create routines for every MPI routine, but we'll see an alternative when we look at profiling using MPE later in this chapter.

17.5 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.

Active profiles automatically add code to collect profile information. Code can be added either directly to the source code or to the object code generated by the compiler. While active compilers avoid some of the problems passive profilers face, they introduce their own set of problems. Perhaps the most significant of these is the confounding effect on timing measurement caused by the execution of the added instructions.

In this section, we'll look at two profilers, *gprof* and *gcov*. These profilers modify the application so that it will count how often functions are called and individual instructions are executed, respectively. (Additionally, *gprof* uses passive profiling to estimate execution times.) The two tools we'll examine use compiler options to add the profiling code to the object code generated by the compiler. While both of these tools were designed for serial programming and were not intended for use with parallel programs, they can, with a little added effort, be used with parallel programs. We'll examine their use with serial code first and then look at how they can be used with parallel code.

17.5.1 *gprof*

gprof generates profiles and call graphs (a report of which routines call which). There is both a

Berkeley Unix *gprof* and a GNU *gprof* that provide essentially the same information. *gprof* is available on most systems today. (If your system doesn't have *gprof*, you might look to see if *prof* is installed. While it's not as versatile as *gprof*, you'll still be able to generate the flat profiles described in this section.)

Using *gprof* is remarkably straightforward. You'll need to compile and link the program you want to profile using the `-pg` option. (Compiler optimizations shouldn't cause much trouble with *gprof*.) Next, run the program. This will create a file *gmon.out* with the profile information. If the file already exists from a previous run, it will be overwritten. The *gprof* program is then run to convert the binary output in *gmon.out* into a readable format.

Here is an example:

```
[sloanjd@amy PROFILE]$ gcc demo.c -pg -o demo
[sloanjd@amy PROFILE]$ ./demo
[sloanjd@amy PROFILE]$ gprof -b demo > demo.gprof
```

This example uses the `-b` option to suppress the annotations that *gprof* includes by default. You may want to leave this off the first time you run *gprof*, but if you run *gprof* often, you'll want to use this option. *gprof* has other options that you might want to investigate if you find you are using it a lot, such as excluding functions from the reports. In this example, I've also redirected the output to a file to make it easier to examine.

Here is the code that I profiled.

```
main( )
{
    int i;

    for (i=0; i<200; i++)
    {
        foo( );
        bar( );
        baz( ); }
}

foo( )
{
    int j;

    for (j=0; j<250; j++) bar( );
}

bar( )
{
    int k;

    for (k=0; k<50; k++);
}
```

```

baz( )

{ int m;

  for (m=0; m<100; m++) bang( );

}

```

```

bang( )

{ int n;

  for (n=0; n<200; n++) bar( );

}

```

As you can see, it doesn't do anything worthwhile, but it provides a reasonable demonstration of *gprof's* capabilities.

The output from *gprof* consists of three parts: a flat profile, a call graph, and a function index. Here is the first part of the output, the flat profile:

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
96.97	6.09	6.09	4050200	0.00	0.00	bar
1.27	6.17	0.08	20000	0.00	0.30	bang
1.27	6.25	0.08	200	0.40	30.87	baz
0.48	6.28	0.03	200	0.15	0.53	foo

This is not difficult to interpret. The routines are profiled, one per line, with their names given in the last column *name*. The column *% time* gives the amount of time spent in each routine. *self seconds* gives the time spent in an individual routine while *cumulative seconds* provides the total for the named routine along with the ones above it. *calls* reports the number of times the function is called while *self ms/call* gives the average time spent in the routine per call. *total ms/call* gives the average time spent in the function and its descendants. All times are given in milliseconds.

It should be clear that optimizing or eliminating calls to *bar* would provide the greatest improvement. What's not clear is the interrelationship among the calls. Since *bar* is called by several functions, improvement to these will help as well. To see this you need more information. This is provided by a call graph.

Here is the call graph for this code:

Call graph

granularity: each sample hit covers 4 byte(s) for 0.16% of 6.28 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	6.28		main [1]
		0.08	6.09	200/200	baz [2]
		0.03	0.08	200/200	foo [5]
		0.00	0.00	200/4050200	bar [4]

		0.08	6.09	200/200	main [1]
[2]	98.3	0.08	6.09	200	baz [2]
		0.08	6.01	20000/20000	bang [3]

		0.08	6.01	20000/20000	baz [2]
[3]	97.0	0.08	6.01	20000	bang [3]
		6.01	0.00	4000000/4050200	bar [4]

		0.00	0.00	200/4050200	main [1]
		0.08	0.00	50000/4050200	foo [5]
		6.01	0.00	4000000/4050200	bang [3]
[4]	97.0	6.09	0.00	4050200	bar [4]

		0.03	0.08	200/200	main [1]
[5]	1.7	0.03	0.08	200	foo [5]
		0.08	0.00	50000/4050200	bar [4]

The first thing you'll notice is that the numbers in the % time column don't add up to 100

percent. The reason is that each timing includes the total time spent in the function and its children. The next two columns give the actual time spent in the function (`self`) and in the functions it calls (`children`). The `called` column gives the number of times the function was called. When two numbers are given in this column, the top number is the number of calls by the parent to the child and the bottom number is the total number of calls to the child. For example, consider the next to last line in section [4].

```
6.01    0.00 4000000/4050200    bang [3]
```

Since section [4] describes calls to `bar`, this line tells us that `bang` makes 4,000,000 of the 4,050,200 total calls made to `bar`.

The table is sorted numerically with a unique number of each function. This number is printed next to each function to make it easier to look up individual functions. A function index is also included at the end of the report.

Index by function name

```
[3] bang           [2] baz
[4] bar           [5] foo
```

This is relatively useless for a small program but can be helpful with larger programs.

As noted above, the number of function calls is determined actively, so this number should be totally accurate. The percentage of time in each is determined passively with a sampling rate that may be too slow for short programs to capture enough data to be statistically reliable. You will have observed that the basic granularity of `gprof` is a routine. Depending on the code, this may not provide enough information. For example, it may not be immediately obvious whether the poor performance you are seeing is the result of the computational complexity of your code or of poor memory utilization if you are looking at a large routine. To improve resolution and gain more information, you could break a program into more routines. Or you could use `gcov`.

17.5.2 gcov

`gcov` is a test-coverage program that is often used as a profiler. When a test suite is designed for a program, one of the objects is to exercise all parts of the code. A test-coverage program records the number of times each line of code is executed. The idea is that you can use this coverage data to ensure that your test suite is adequate. Of course, this is also the sort of data you might want when profiling code.

`gcov` is part of the `gcc` development packages, so if you have `gcc` on your system, you should have `gcov`. It does not work with other compilers, but similar programs may be available for them (e.g., `tcov` with Solaris).

To use `gcov`, you need to compile code with two options, `-fprofile-arcs` and `-ftest-coverage`, which tell the compiler to add, respectively, the additional code needed to generate a flow graph and extra profiling information.

```
[sloanjd@amy PROFILE]$ gcc -fprofile-arcs -ftest-coverage demo.c -o demo
```

You should avoid optimizations when using `gcov`, since optimizations that rewrite code will make the results difficult to interpret. When the code is compiled, two new files will be created

with the same name as your program but with the extensions `.bb` and `.bbg`. The first contains a list of source files with line numbers corresponding to basic blocks. The second is used to reconstruct a flow graph for the program.

Once you have compiled the code, you'll need to run it.

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

This will create yet another file with the run data. It will have the same name as your program but with a `.da` extension. This is created in the directory where the original program was compiled.

Finally, you can run `gcov` to generate your report.

```
[sloanjd@amy PROFILE]$ gcov demo
```

```
100.00% of 13 source lines executed in file demo.c
```

```
Creating demo.c.gcov.
```

You'll notice that the command reports what percentage of the source is actually executed, something you would want to know if you are using it as a coverage tool rather than a profiler. The actual report is created in a file with the extension `.gcov`.

Here is an example for the demonstration program that we looked at earlier.

```
        main( )
    2   {   int i;
201       for (i=0; i<200; i++)
200       {   foo( );
200           bar( );
200           baz( ); }
        }

        foo( )
    400   {   int j;
200       for (j=0; j<250; j++) bar( );
        }

        bar( )
    8100400   {   int k;
    4050200       for (k=0; k<50; k++);
        }
```

```

        baz( )
400    { int m;
200        for (m=0; m<100; m++) bang( );
    }

        bang( )
40000  { int n;
20000  for (n=0; n<200; n++) bar( );
    }

```

As you can see, a count giving the number of times each line was executed is appended to each line. Take care; if you execute the program multiple times, these counts accumulate. If this isn't what you want, delete or rename the data file between runs.

17.5.3 Profiling Parallel Programs with *gprof* and *gcov*

Depending on the version of MPI you are using, you may be able to use *gprof* and *gcov* with your MPI programs. However, you'll need to make some adjustments. If we naively try to profile an MPI program with *gprof* or *gcov*, we run into problems. Here is an example using *rect* from [Chapter 14](#):

```
[sloanjd@amy PROFILE]$ mpicc -pg rect.c -o rect
```

```
[sloanjd@amy PROFILE]$ mpirun -np 4 rect
```

```
Enter number of chunk to divide problem into:
```

```
7
```

```
Enter number of steps per chunk:
```

```
20
```

```
Enter low end of interval:
```

```
2.0
```

```
Enter high end of interval:
```

```
5.0
```

```
Area for process 1, is: 2.107855
```

```
Area for process 2, is: 2.999984
```

```
Area for process 3, is: 4.049546
```

```

Area for process 1, is: 5.256543
Area for process 2, is: 6.620975
Area for process 3, is: 8.142841
Area for process 1, is: 9.822141

The area from 2.000000 to 5.000000 is: 38.999885

```

Everything appears to work. All the expected files are created. Here is the flat profile.

```
[sloanjd@amy PROFILE]$ gprof -bp rect
```

Flat profile:

Each sample counts as 0.01 seconds.

```
no time accumulated
```

%	cumulative	self		self	total	
time	seconds	seconds	calls	Ts/call	Ts/call	name
0.00	0.00	0.00	40	0.00	0.00	f
0.00	0.00	0.00	2	0.00	0.00	chunkArea
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__I_mainGCOV

We've obviously profiled something. The question is what. Because we are using NFS, all the processes are writing out their logfiles to the same directory. Since files are overwritten, what we have is the data from the last process to finish. For the data, we can see this is either process 2 or process 3.

If we are going to avoid this problem, we need to use the local filesystem on each node. For example, we could copy the compiled code over to */tmp* on each machine and run it from there.^[2]

^[2] This example uses *scp* to copy the file. If you have installed the C3 tools, discussed in [Chapter 9](#), then you can use *cpush* instead.

```
[sloanjd@amy PROFILE]$ cp rect /tmp/
```

```
[sloanjd@amy PROFILE]$ scp rect oscarnode1:/tmp/
```

```
rect          100% |*****| 372 KB  00:00
```

```
[sloanjd@amy PROFILE]$ scp rect oscarnode2:/tmp/
```

```
rect          100% |*****| 372 KB  00:00
```

```
[sloanjd@amy PROFILE]$ scp rect oscarnode3:/tmp/
```

```
rect          100% |*****| 372 KB  00:00
```

```
[sloanjd@amy PROFILE]$ cd /tmp
[sloanjd@amy tmp]$ mpirun -np 4 rect
Enter number of chunk to divide problem into:
7
Enter number of steps per chunk:
20
Enter low end of interval:
2.0
Enter high end of interval:
5.0
Area for process 1, is: 2.107855
Area for process 2, is: 2.999984
Area for process 3, is: 4.049546
Area for process 1, is: 5.256543
Area for process 2, is: 6.620975
Area for process 3, is: 8.142841
Area for process 1, is: 9.822141
The area from 2.000000 to 5.000000 is: 38.999885
```

Here is the flat profile for *oscardnode1*:

```
[sloanjd@oscardnode1 tmp]$ gprof -bp rect
```

Flat profile:

Each sample counts as 0.01 seconds.

no time accumulated

%	cumulative	self		self	total	
time	seconds	seconds	calls	Ts/call	Ts/call	name
0.00	0.00	0.00	60	0.00	0.00	f
0.00	0.00	0.00	3	0.00	0.00	chunkArea
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__I_mainGCOV

And here is the flat profile for *oscarnode2*:

```
[sloanjd@oscarnode2 tmp]$ gprof -bp rect
```

Flat profile:

Each sample counts as 0.01 seconds.

```
no time accumulated
```

%	cumulative	self		self	total	
time	seconds	seconds	calls	Ts/call	Ts/call	name
0.00	0.00	0.00	40	0.00	0.00	f
0.00	0.00	0.00	2	0.00	0.00	chunkArea
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__I_mainGCOV

If you compare these to the output, you'll see the calls to *chunkArea* now make sense—it is called three times by process 1 on *oscarnode1* and twice by process 2 on *oscarnode2*. (We could also look at the head node, *amy*, but because of the master/slave design of the program, there isn't much to see.)

Unfortunately, using *gcov* with a parallel program is even more complicated. You'll recall that the compile options used with *gcv* create two additional files and that a datafile is created when the program is run. To run *gcv*, you'll need all these files as well as the executable and the original source code file. To further complicate matters, the datafile that is created when the program is run is created in the directory where the source was compiled.

To use *gcv* on a cluster, first copy the source code file to */tmp* or a similar directory. Next, compile the program with the appropriate switches. Once you've done this, you'll need to copy the compiled code, the original source code, the *.bb* file, and the *.bbg* file to each node on the cluster. Now you can run the program. A datafile will be created on each cluster in */tmp*. Once you have done this, you can then log onto each node and run *gcv*. For example,

```
[sloanjd@oscarnode1 tmp]$ gcov rect
```

```
66.67% of 57 source lines executed in file rect.c
```

```
Creating rect.c.gcov.
```

Don't forget that the datafile accumulates information with *gcv*. If you want fresh data, you'll need to delete it from each node. This is a lot of copying, so you'll want to automate it as much as possible. You'll also need to clean up after you are done. In this example, I copied *rect*, *rect.c*, *rect.bb*, and *rect.bbg* to each node. Fortunately, for this demonstration I only needed to copy them to a few nodes.

A couple of warnings are in order. All of this is based on the assumption that each MPI process will be able to access and write to the local filesystem. With MPI, there is no guarantee this is the case. The approach outlined here seems to work with LAM/MPI and MPICH, but if you are using some other version of MPI, all bets are off.

17.6 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 that 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.

nupshot

This tool reads and displays *CLOG* files. Because it is implemented in an older version of Tcl/Tk, it is not automatically installed.

jumpshot-2

This tool reads and displays *CLOG* files and is implemented in Java 1.1. (Unlike *jumpshot-3*, *jumpshot-2* is not compatible with newer versions of Java.)

jumpshot-3

This tool reads and displays *SLOG* files and is implemented in Java.

To build each of these, you will need the appropriate version of TCL/TK or Java on your system.

Finally, MPE provides several utilities that simplify dealing with logfiles.

clog2slog

This utility that converts CLOG files into SLOG files.

clog2alog

This converts from CLOG to ALOG format.

slog_print and *clog_print*

These print programs for SLOG and CLOG files, respectively.

viewers

This utility invokes the appropriate visualization tool needed to display a logfile based on its format.

There are two basic approaches to generating logfiles with MPE. When you link to the appropriate MPE library, logfiles will be generated automatically using the PMPI profiling interface described earlier in this chapter. Alternatively, you can embed MPE commands in a program to manually collect information. It is also possible to combine these approaches in a single program.

17.6.1 Using MPE

In order to use MPE, you'll need to link your programs to the appropriate libraries. Since MPE has been integrated into the MPICH distribution, using MPICH is the easiest way to go because MPICH provides compiler flags that simplify compilation.

If you are using another version of MPI, instead of or in addition to MPICH, your first order of business will be locating the MPE libraries on your system and ensuring they are on your compile/link paths, typically */usr/local/lib*. If in doubt, use *whereis* to locate one of the libraries. They should all be in the same place.

```
[sloanjd@amy sloanjd]$ whereis libmpe.a
```

```
libmpe: /usr/local/lib/libmpe.a
```

Once you've got your path set correctly, using MPE shouldn't be difficult.

MPICH includes several demonstration programs, so you may find it easier if you test things out with these rather than with one of your own programs. In the next two examples, I'm using *cpi.c* and *cpiilog.c*, which are found in the *examples* directory under the MPICH source tree. *cpi.c* is an ordinary MPI program that estimates the value of π . It does not contain any MPE commands. We'll use it to see how the automatic profiling library works.

To compile *cpi.c* under MPICH, use the `-mpilog` compiler flag.

```
[sloanjd@amy MPEDEMO]$ mpicc cpi.c -mpilog -o cpi
```

It is only slightly more complicated with LAM/MPI. You'll need to be sure that the libraries can be found and you'll need to explicitly link both libraries, *liblmpe.a* and *libmpe.a* as shown:

```
[sloanjd@amy MPEDEMO]$ mpicc mpi.c -llmpe -lmpe -o mpi
```

(Be sure you link them in the order shown.)

When you run the program, you'll notice that a logfile is created.

```
[sloanjd@amy MPEDEMO]$ mpirun -np 4 mpi
```

```
Process 0 of 4 on amy
```

```
pi is approximately 3.1415926544231239, Error is 0.0000000008333307
```

```
wall clock time = 0.005883
```

```
Writing logfile.
```

```
Finished writing logfile.
```

```
Process 2 of 4 on oscarnde2.oscardomain
```

```
Process 1 of 4 on oscarnde1.oscardomain
```

```
Process 3 of 4 on oscarnde3.oscardomain
```

By default, a CLOG file will be created. You can change the default behavior by setting the environment variable `MPE_LOG_FORMAT`.^[3] For example,

^[3] While setting `MPE_LOG_FORMAT` works fine with MPICH, it doesn't seem to work with LAM/MPI.

```
[sloanjd@amy MPEDEMO]$ export MPE_LOG_FORMAT=SLOG
```

You can view the CLOG file directly with *jumpshot-2*, or you can convert it to a SLOG file with *clog2slog* utility and then view it with *jumpshot-3*. I'll use the latter approach since I haven't installed *jumpshot-2* on this system.

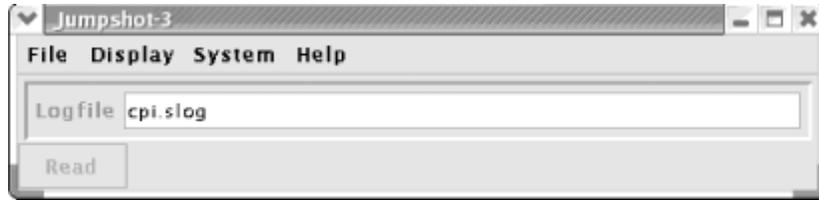
```
[sloanjd@amy MPEDEMO]$ clog2slog mpi.clog
```

```
[sloanjd@amy MPEDEMO]$ jumpshot mpi.slog
```

Remember that you'll need to execute that last command in an X Window System environment.

jumpshot-3 opens three windows. The first is the main window for *jumpshot-3*, which you can use to open other logfiles and change program defaults. If you close it, the other *jumpshot-3* windows will all close as well. See [Figure 17-1](#).

Figure 17-1. Main Jumpshot-3 window



The next window to open will be the legend. This gives the color code for the data display window, which opens last. See [Figure 17-2](#).

Figure 17-2. Legend



Since *cpi.c* only uses two MPI commands, only two are shown. If other MPI functions had been used in the program, they would have been added to the window. If the colored bullets are not visible when the window opens, which is often the case, just resize the window and they should appear.

The last window, the *View & Frame Selector*, displays the actual profile information. The graph is organized vertically by process and horizontally by time. Once you have this window open, you can use the options it provides to alter the way your data is displayed. See [Figure 17-3](#).

Figure 17-3. View & Frame Selector



You can find an introductory tutorial on *jumpshot-3* under the MPICH source tree in the directory `mpc/viewers/jumpshot-3/doc`. Both PDF and HTML versions are included.

As noted earlier, if you want more control over how your program is profiled, you can embed MPE profiling commands directly into the code. With MPICH, you'll compile it in exactly the same way, using the `-mpilog` flag. With LAM/MPI, you only need to link to the `libmpe.a` library.

```
[sloanjd@amy MPEDEMO]$ mpicc cpilog.c -lmpe -o cpilog
```

The file `cpilog.c`, compiled here, is an MPE demonstration program that includes embedded MPE commands. An explanation of these commands and an example are given in the next subsection of this chapter.

Before we leave compiling MPE programs, it is worth mentioning the other MPE libraries that are used in much the same way. With MPICH, the compiler flag `-mpianim` is used to link to the animation library, while the flag `-mpitrace` is used to link to the trace library. With LAM/MPI, you'll need to link these directly when you compile. For example, to use the trace library `libtmpe.a`, you might enter

```
[sloanjd@amy MPEDEMO]$ mpicc cpi.c -ltmpe -o cpi
```

With the trace library you'll get a trace printout for all MPI calls when you run the program. Here is a partial listing for `cpi.c`:

```
[sloanjd@amy MPEDEMO]$ mpirun -np 4 cpi
```

```
Starting MPI_Init...
```

```
Starting MPI_Init...
Starting MPI_Init...
Starting MPI_Init...
[0] Ending MPI_Init
[1] Ending MPI_Init
[2] Ending MPI_Init
[3] Ending MPI_Init
[1] Starting MPI_Comm_size...
[2] Starting MPI_Comm_size...
[3] Starting MPI_Comm_size...
[1] Ending MPI_Comm_size
[2] Ending MPI_Comm_size
[3] Ending MPI_Comm_size
[1] Starting MPI_Comm_rank...
[2] Starting MPI_Comm_rank...
[3] Starting MPI_Comm_rank...
[1] Ending MPI_Comm_rank
[2] Ending MPI_Comm_rank
[3] Ending MPI_Comm_rank
[2] Starting MPI_Get_processor_name...
[1] Starting MPI_Get_processor_name...
[3] Starting MPI_Get_processor_name...
[1] Ending MPI_Get_processor_name
[2] Ending MPI_Get_processor_name
[3] Ending MPI_Get_processor_name
Process 1 of 4 on oscarnode1.oscardomain
Process 2 of 4 on oscarnode2.oscardomain
Process 3 of 4 on oscarnode3.oscardomain
[2] Starting MPI_Bcast...
[1] Starting MPI_Bcast...
```

```
[3] Starting MPI_Bcast...
```

```
...
```

There's a lot more output that has been omitted. As you can see, the program output is interspersed with the trace. The number in the square bracket is the process.



17.7 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");
}

MPE_Start_log( );

MPI_Barrier(MPI_COMM_WORLD);

MPE_Log_event(evnt1a, 0, "start setup");

/* 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;
}

MPE_Log_event(evt1b, 0, "end setup");

MPI_Barrier(MPI_COMM_WORLD);

/* collect information and print results */

tag = 0;

if (processId == 0)          /* if rank is 0, collect results */
{
    MPE_Log_event(evt2a, 0, "start receive");

    total = area;

    for (src=1; src < noProcesses; src++)
    {
        MPI_Recv(&area, 1, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, &status);

        total = total + area;
    }

    MPE_Log_event(evt2b, 0, "end receive");

    MPE_Log_event(evt3a, 0, "start display");

    fprintf(stderr, "The area from %f to %f is: %f\n",

                lowerLimit, upperLimit, total );
}

else                          /* all other processes only send */
{
    MPE_Log_event(evt4a, 0, "start send");
}

```

```

    dest = 0;

    MPI_Send(&area, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);

    MPE_Log_event(evnt4b, 0, "end send");
}

if (processId == 0)
{
    finish = MPI_Wtime( );

    printf("Elapsed time = %f\n", finish-start);

    MPE_Log_event(evnt3b, 0, "end display");
}

/* finish */

MPE_Finish_log("rect2-log");

MPI_Finalize( );

return 0;
}

```

Let's examine the changes. First, you'll notice that the *mpe.h* header file has been included. Next, in this example, we want to look at four sections of code, so we've added variables to record event numbers for each, *evnt1a* through *evnt4b*. We'll need a pair of variables for each block of code. Event numbers are just distinct integers used to identify events. You could make up your own as long as you are consistent, but it is better to use the MPE function *MPE_Log_get_event_number*, which ensures that you have unique numbers. It is essential that you use it if you are putting these commands in functions stored in a library or functions that call libraries. With each pair of event numbers, we've associated a description and color using the *MPE_Describe_state* function. This is used to create the legend, color the graphs, etc. Notice that one event starts a block that you want measured and a second event ends it. Make sure your events are paired and are called exactly once.

You'll notice that all the other MPE function calls are bracketed between calls to *MPE_Init_log* and *MPE_Finish_log*. If you are combining your logging with MPE's default logging, i.e., linking your program to *lib/mpe.a*, these function calls should not be included. They will be called by *MPI_Init* and *MPI_Finish*, respectively. However, if you are using the MPE function call independently—that is, without using MPE's default logging—you'll need these two calls. Note that *MPE_Finish_log* allows you to specify a name for the logfile.

Once we've got everything set up, we are ready to call *MPI_Start_log* to begin recording events. Next, we simply put the sections of code we want to profile between pairs of calls to *MPE_Log_event*. For example, the initial *MPI_Bcast* call is profiled by surrounding it with the *MPE_Log_event* calls for *evnt1a* and *evnt1b*.

Once you've got the code instrumented, it is just a matter of compiling it with the appropriate MPE options, running the code, and then examining the logfiles. Here is the display for this program. This is shown in [Figure 17-4](#).

Figure 17-4. Timeline



In this example, I've opted to display the timeline for the program. For this particular display, I chose *MPI-Process* under *View Options* and clicked on the *Display* button under *Frame Operations*. (If you try this example, you may find it educational to run it with and without the calls to `MPI_Barrier`.)

17.8 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](#).

Part V: Appendix

This appendix offers useful sources of information for all the aspects of setting up and programming a cluster that were covered in this book.

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.

A.1 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.

There are a number of books available on parallel programming with MPI. For a general introduction, look to Gropp or Pacheco. Both will provide you with more examples and greater depth than I had space for in this book. Snir is an indispensable reference. If you are using PVM, Geist is the best place to start. For producing efficient code, Bentley is wonderful. Unfortunately, it is out of print, but you may be able to find it in a local library. Dowd is also useful.

Albitz, Paul and Cricket Liu. *DNS and BIND*. Fourth Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.

Barrett, Daniel and Richard Silverman. *SSH, the Secure Shell: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.

Bentley, Jon Louis. *Writing Efficient Programs*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1982.

Bookman, Charles. *Linux Clustering: Building and Maintaining Linux Clusters*. Indianapolis, IN: New Riders Publishing, 2002.

Bovet, Daniel and Marco Cesati. *Understanding the Linux Kernel*. Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2002.

Callaghan, Brent. *NFS Illustrated*. Reading, MA: Addison Wesley Professional, 1999.

- Cheswick, William, Steven Bellovin, and Aviel Rubin. *Firewalls and Internet Security: Repelling the Wiley Hacker*. Second Edition. Reading, MA: Addison Wesley Publishing Co., 2003.
- Comer, Douglas. *Internetworking with TCP/IP: Principles, Protocols, and Architectures*. Volume 1. Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2000.
- Culler, David, Jaswinder Pal Singh, with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1998.
- Dowd, Kevin and Charles Severance. *High Performance Computing*. Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 1998.
- Dongarra, Jack et al., eds. *Sourcebook of Parallel Computing*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2003.
- Frisch, Aileen. *Essential System Administration*. Sebastopol, CA: O'Reilly & Associates, Inc., 1991.
- Garfinkel, Simson, Gene Spafford, and Alan Schwartz. *Practical Unix & Internet Security*. Third Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 1993.
- Geist, Al et al. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press, 1994.
- Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Second Edition. Cambridge, MA: MIT Press, 1999.
- Hall, Eric A. *Internet Core Protocols: The Definitive Guide with CD-ROM*. Sebastopol, CA: O'Reilly & Associates, Inc., 2000.
- Hunt, Craig. *TCP/IP Network Administration*. Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 1998.
- Jain, Raj. *The Art of Computer Systems Performance Analysis*. New York, NY: John Wiley & Sons, 1991.
- Laurie, Ben and Peter Laurie. *Apache: The Definitive Guide*. Third Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2002.
- Limoncelli, Thomas and Christine Hogan. *The Practice of System and Network Administration*. Upper Saddle River, NJ: Addison Wesley, 2002.
- Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. Knoxville, TN: University of Tennessee, 1997.
- Musumeci, Gian-Palol and Mike Loukides. *System Performance Tuning*. Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2002.
- Nemeth, Evi et al. *Linux Administration Handbook*. Upper Saddle River, NJ: Prentice Hall, 2002.
- Oram, Andy, ed. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.
- Pacheco, Peter. *Parallel Programming with MPI*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997.
- Powers, Shelley et al.. *Unix Power Tools*. Third Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2003.
- Siever, Ellen, Aaron Weber, and Stephen Figgins. *Linux in a Nutshell*. Fourth Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2003.

Snir, Marc et al. *MPI: The Complete Reference*. 2 vols. Cambridge, MA: MIT Press, 1998.

Sonnenreich, Wes and Tom Yates. *Building Linux and OpenBSD Firewalls*. New York, NY: John Wiley & Sons, Inc., 2000.

Spurgeon, Charles. *Ethernet: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, Inc., 2000.

Stallman, Richard et. al. *Debugging with GDB: The GNU Source-Level Debugger*. Boston, MA: GNU Press, 2003.

Sterling, Thomas, ed. *Beowulf Cluster Computing with Linux*. Cambridge, MA: MIT Press, 2002.

Stern, Hal, Mike Eisler, and Ricardo Labiaga. *Managing NFS and NIS*. Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.

Stevens, W. Richard. *TCP/IP Illustrated*. Volume 1, *The Protocols*. Reading, MA: Addison Wesley Longman, 1994.

Tanenbaum, Andrew. *Computer Networks*. Fourth Edition. Saddle River, NJ: Pearson Education, 2002.

Thompson, Robert and Barbra Thompson. *Building the Perfect PC*. Sebastopol, CA: O'Reilly & Associates, Inc., 2004.

Thompson, Robert and Barbra Thompson. *PC Hardware in a Nutshell*. Third Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2003.

Wilkinson, Barry and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1999.

A.2 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.

A.2.1 General Cluster Information

<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>. If you want to know what computers are currently on the 500 top supercomputers list, visit this site.

<http://clusters.top500.org>. For the top 500 clusters, visit this site.

<http://www.redbooks.ibm.com>. Although IBM-Scentric, the Redbooks series contains a wealth of information. Look for the Redbooks on SANs or Globus.

A.2.2 Linux

<http://bccd.cs.uni.edu>. This is the home of the Bootable Cluster CD (BCCD).

<http://www.debian.org>. This is the home for Debian Linux.

<http://www.gentoo.org>. This is the home for Gentoo Linux.

<http://www.knoppix.org>. This is the home for Knoppix. Click on a flag for the language of your choice. For a cluster version of Knoppix, visit <http://bofh.be/clusterknoppix/>.

<http://www.kernel.org>. The Linux Kernels Archive provides kernel sources.

<http://www.mandrakesoft.com>. This is the home for Mandrake Linux

<http://plumpos.sourceforge.net>. This is the home for PlumpOS.

<http://www.redhat.com>. This is the home for Red Hat Linux.

<http://www.suse.com>. This is the home for SUSE Linux.

A.2.3 Cluster Software

<http://bioinformatics.org/biobrew/>. Visit the Biobrew site for information on a Rocks-based bioinformatics cluster.

<http://www.mosix.org>. This is site for the Mosix project.

<http://openmosix.sourceforge.net>. This is the site for the openMosix project.

<http://www.openmosixview.com>. If you are running openMosix, visit this site for the openMosixView tools.

<http://howto.ipng.be/Mosix-HOWTO/>. Visit this site for the most recent version of Kris Buytaert's openMosix HOWTO.

<http://mcaserta.com/maask/>. For more information on the MigSHM openMosix patch, visit this site.

<http://oscar.openclustergroup.org>. This is the home of the Open Cluster Group and the site to visit for OSCAR.

<http://www.openclustergroup.org/HA-OSCAR/>. This is the home for the high availability OSCAR branch.

<http://rocks.npaci.edu>. This is the home for Rocks.

<http://stommel.tamu.edu/~baum/npaci.html>. This is Steven Baum's Rocks site, another good source of information on Rocks. This has very nice sections on grids and on applications available for clusters.

<http://www.scyld.com>. This is the home for Scyld Beowulf. An earlier, nonsupported version of Scyld Beowulf can be purchased at <http://www.linuxcentral.com>.

A.2.4 Grid Computing and Tools

<http://www.globus.org>. If you are interested in grid computing, you should start by visiting the Globus Alliance site.

<http://gridengine.sunsource.net>. This is the home for the Sun Grid Engine.

<http://www.nsf-middleware.org>. This is the home for the NSF middleware grid software.

<http://www.opensce.org>. This is the home for the OpenSCE (Scalable Cluster Environment) project.

<http://nws.cs.ucsb.edu>. This is the home for the Network Weather Service.

<http://www.ncsa.uiuc.edu/Divisions/ACES/GPT/>. This is the home for the Grid Packaging Tools.

http://www.citi.umich.edu/projects/kerb_pki/. This is the home for KX.509 and KCA.

<http://grid.ncsa.uiuc.edu/ssh/>. This is the home for GSI OpenSSH.

<http://grid.ncsa.uiuc.edu/myproxy/>. This is the home for MyProxy.

<http://rocks.npaci.edu/gridconfig/>. A user's manual for the Rocks Gridconfig Tools can be found at this site.

A.2.5 Cloning and Management Software

<http://www.csm.ornl.gov/torc/C3/>. This is the home for Cluster Command and Control (C3).

<http://sourceforge.net/projects/clumon>. If you need Clumon, it can be downloaded from SourceForge.

<http://www.feyrer.de/g4u/>. This is the home for *g4u*.

<http://ganglia.sourceforge.net>. Ganglia can be downloaded from this site.

<http://oss.sgi.com/projects/pcp/>. This is the home for SGI's Performance Co-Pilot.

<http://sisuite.sourceforge.net>. This is the home for System Installation Suite (SIS). It provides links to System Configurator, SystemImager, and System Installer.

A.2.6 Filesystems

<http://clusternfs.sourceforge.net>. This is the home for ClusterNFS.

<http://www.coda.cs.cmu.edu/index.html>. This is the home for the Coda file system

<http://www-1.ibm.com/servers/eserver/clusters/software/gpfs.html>. This site provides information for GPFS, the General Parallel File System.

<http://www.inter-mezzo.org>. This is the home for the InterMezzo filesystem.

<http://www.lustre.org>. This is the home for the Luster filesystem.

<http://www.openafs.org>. This is the home for the Open Andrew filesystem.

<http://opengfs.sourceforge.net>. This is the home for OpenGFS Project.

<http://www.parl.clemson.edu/pvfs/>. This is the home to the PVFS. For PVFS2, go to <http://www.pvfs.org/pvfs2/>.

A.2.7 Parallel Benchmarks

<http://www.netlib.org>. The Netlib Repository is a good place to start if you need benchmarks.

<http://hint.byu.edu>. This is the home for the Hierarchical Integration (HINT) benchmark.

<http://www.netlib.org/benchmark/hpl/>. This is the home for High Performance Linpack benchmark.

<http://www.iozone.org>. This is the home for Iozone, an I/O and file system benchmark tool.

<http://dast.nlanr.net/Projects/Iperf/>. This is the home for Iperf, a network performance measurement tool.

<http://science.nas.nasa.gov/Software/NPB/>. This is the home for the NAS Parallel Benchmarks.

A.2.8 Programming Software

<http://www.gnu.org/software/ddd/>. This is the Data Display Debugger's home page.

<http://gcc.gnu.org>. This is the home page for the *gcc* compiler project. This project includes *gprof* and *gcov*.

<http://hdf.ncsa.uiuc.edu/HDF5/>. This is the home page for HDF5.

<http://java.sun.com>. This is Sun's Java page. Java can also be downloaded from <http://www.blackdown.org>.

<http://www.lam-mpi.org>. This is the home page for the LAM/MPI project.

<http://www.mpi-forum.org>. This is the home page for the MPI Forum. Visit this site for standards documents and other information on MPI.

<http://www-unix.mcs.anl.gov/mpi/mpich/>. This is the home page for MPICH.

<http://www.netlib.org/pvm3/>. This is the home page for PVM.

<http://www.mcs.anl.gov/romio/>. This is the home page for ROMIO.

<http://www.splint.org>. This is the home page for SPLINT.

<http://sprng.cs.fsu.edu/>. This is the home page for SPRNG, the Scalable Parallel Random Number Generator.

<http://www.scriptics.com>. This is the home page for Tcl/Tk.

<http://vmi.ncsa.uiuc.edu>. This is NCSA's page for the Virtual Machine Interface or VMI.

A.2.9 Scheduling Software

<http://www.cs.wisc.edu/condor>. This is the home for the Condor Project.

<http://www.supercluster.org>. This is the Center for HPC Cluster Resource Management and Scheduling. Visit this site for Maui and Torque.

<http://umbc7.umbc.edu/ngs/ngsmain.html>. This is the home for the Network Queuing Systems software.

<http://www.openpbs.org>. This is the home for the open software branch for PBS. For OpenPBS patches, you might visit <http://www-unix.mcs.anl.gov/openpbs/> and <http://bellatrix.pcl.ox.ac.uk/~ben/pbs/>.

<http://www.pbspro.com> This is the home for the commercial branch for PBS.

A.2.10 System Software and Utilities

<http://www.chkrootkit.org>. This is the home for the *chkrootkit* security program.

<http://modules.sourceforge.net>. The modules package, on which *switcher* is based, can be downloaded from SourceForge.

<http://www.myri.com>. This commercial site is the home for Myricom, the creators of Myrinet.

<http://sourceforge.net/projects/pfilter>. If you want the *pfilter* software, you can download it from SourceForge.

<http://www.rrdtool.org>. This is the home for Tobi Oetiker's RRDtool.

<http://samba.anu.edu.au/rsync>. Visit this site for more information or the latest version of *rsync*.

<http://www.tripwire.org>. This is the home for the *tripwire* security auditing tool.

<http://vsftpd.beasts.org>. This is the home for Very Secure FTP.



Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The cover image of cowboys herding cattle is a 19th-century engraving from the Dover Pictorial Archive. Using their horsemanship and lariat skills, cowboys in the American West managed herds of several thousand cattle. These abilities would become especially valuable after the Civil War, when an increased demand for beef in the northern and eastern parts of the country left Texas ranchers needing a way to transport their product. Cowboys would drive Texas Longhorn cattle over 1,000 miles north to railroad cow towns in Kansas and Nebraska. These grueling journeys would take several months to complete, with those in charge of the herd working, eating, and sleeping on the open plain.

Adam Witwer was the production editor and copyeditor for High Performance Linux Clusters with OSCAR, Rocks, openMosix, and MPI. Leanne Soylemez was the proofreader. Claire Cloutier and Sanders Kleinfeld provided quality control. John Bickelhaupt wrote the index.

Emma Colby designed the cover of this book, based on a series design by Hanna Dyer and Edie Freedman. Clay Fernald produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. The chapter opening images are from *Marvels of the New West: A Vivid Portrayal of the Stupendous Marvels in the Vast Wonderland West of the Missouri River*, by William Thayer (The Henry Bill Publishing Co., 1888). This book was converted to FrameMaker 5.5.6 by Joe Wizda with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand MX and Adobe Photoshop CS. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Adam Witwer.

The online edition of this book was created by the Safari production group (John Chodacki, Ken Douglass, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.