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 the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Gove, Darryl.
Multicore application programming : for Windows, Linux, and Oracle Solaris / Darryl Gove.
p. cm.
Includes bibliographical references and index.
1. Parallel programming (Computer science) I. Title.
QA76.642.G68 2011
005.2’75–dc22
2010033284

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-10: 0-321-71137-8

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, IN.
First printing, October 2010
Contents at a Glance

Preface xv
Acknowledgments xix
About the Author xxi

1 Hardware, Processes, and Threads 1
2 Coding for Performance 31
3 Identifying Opportunities for Parallelism 85
4 Synchronization and Data Sharing 121
5 Using POSIX Threads 143
6 Windows Threading 199
7 Using Automatic Parallelization and OpenMP 245
8 Hand-Coded Synchronization and Sharing 295
9 Scaling with Multicore Processors 333
10 Other Parallelization Technologies 383
11 Concluding Remarks 411

Bibliography 417
Index 419
This page intentionally left blank
# Contents

Preface xv

Acknowledgments xix

About the Author xxi

1 Hardware, Processes, and Threads 1
   Examining the Insides of a Computer 1
   The Motivation for Multicore Processors 3
      Supporting Multiple Threads on a Single Chip 4
      Increasing Instruction Issue Rate with Pipelined Processor Cores 9
   Using Caches to Hold Recently Used Data 12
   Using Virtual Memory to Store Data 15
   Translating from Virtual Addresses to Physical Addresses 16
   The Characteristics of Multiprocessor Systems 18
      How Latency and Bandwidth Impact Performance 20
   The Translation of Source Code to Assembly Language 21
      The Performance of 32-Bit versus 64-Bit Code 23
      Ensuring the Correct Order of Memory Operations 24
      The Differences Between Processes and Threads 26
   Summary 29

2 Coding for Performance 31
   Defining Performance 31
   Understanding Algorithmic Complexity 33
      Examples of Algorithmic Complexity 33
      Why Algorithmic Complexity Is Important 37
      Using Algorithmic Complexity with Care 38
   How Structure Impacts Performance 39
      Performance and Convenience Trade-Offs in Source Code and Build Structures 39
      Using Libraries to Structure Applications 42
      The Impact of Data Structures on Performance 53
3 Identifying Opportunities for Parallelism 85
   Using Multiple Processes to Improve System Productivity 85
   Multiple Users Utilizing a Single System 87
   Improving Machine Efficiency Through Consolidation 88
      Using Containers to Isolate Applications Sharing a Single System 89
      Hosting Multiple Operating Systems Using Hypervisors 89
   Using Parallelism to Improve the Performance of a Single Task 92
      One Approach to Visualizing Parallel Applications 92
      How Parallelism Can Change the Choice of Algorithms 93
      Amdahl’s Law 94
      Determining the Maximum Practical Threads 97
      How Synchronization Costs Reduce Scaling 98
   Parallelization Patterns 100
      Data Parallelism Using SIMD Instructions 101
      Parallelization Using Processes or Threads 102
      Multiple Independent Tasks 102
      Multiple Loosely Coupled Tasks 103
      Multiple Copies of the Same Task 105
      Single Task Split Over Multiple Threads 106
5 Using POSIX Threads 143
   Creating Threads 143
      Thread Termination 144
      Passing Data to and from Child Threads 145
      Detached Threads 147
      Setting the Attributes for Pthreads 148
   Compiling Multithreaded Code 151
   Process Termination 153
   Sharing Data Between Threads 154
      Protecting Access Using Mutex Locks 154
      Mutex Attributes 156
      Using Spin Locks 157
      Read-Write Locks 159
      Barriers 162
      Semaphores 163
      Condition Variables 170
   Variables and Memory 175
   Multiprocess Programming 179
      Sharing Memory Between Processes 180
      Sharing Semaphores Between Processes 183
      Message Queues 184
      Pipes and Named Pipes 186
      Using Signals to Communicate with a Process 188
   Sockets 193
   Reentrant Code and Compiler Flags 197
   Summary 198

6 Windows Threading 199
   Creating Native Windows Threads 199
      Terminating Threads 204
      Creating and Resuming Suspended Threads 207
      Using Handles to Kernel Resources 207
   Methods of Synchronization and Resource Sharing 208
      An Example of Requiring Synchronization Between
      Threads 209
      Protecting Access to Code with Critical Sections 210
      Protecting Regions of Code with Mutexes 213
Contents

Slim Reader/Writer Locks 214
Semaphores 216
Condition Variables 218
Signaling Event Completion to Other Threads or Processes 219
Wide String Handling in Windows 221
Creating Processes 222
  Sharing Memory Between Processes 225
  Inheriting Handles in Child Processes 228
  Naming Mutexes and Sharing Them Between Processes 229
  Communicating with Pipes 231
  Communicating Using Sockets 234
Atomic Updates of Variables 238
Allocating Thread-Local Storage 240
Setting Thread Priority 242
Summary 244

7 Using Automatic Parallelization and OpenMP 245
  Using Automatic Parallelization to Produce a Parallel Application 245
    Identifying and Parallelizing Reductions 250
    Automatic Parallelization of Codes Containing Calls 251
    Assisting Compiler in Automatically Parallelizing Code 254
  Using OpenMP to Produce a Parallel Application 256
    Using OpenMP to Parallelize Loops 258
    Runtime Behavior of an OpenMP Application 258
    Variable Scoping Inside OpenMP Parallel Regions 259
    Parallelizing Reductions Using OpenMP 260
    Accessing Private Data Outside the Parallel Region 261
    Improving Work Distribution Using Scheduling 263
    Using Parallel Sections to Perform Independent Work 267
    Nested Parallelism 268
Using OpenMP for Dynamically Defined Parallel Tasks 269
Keeping Data Private to Threads 274
Controlling the OpenMP Runtime Environment 276
Waiting for Work to Complete 278
Restricting the Threads That Execute a Region of Code 281
Ensuring That Code in a Parallel Region Is Executed in Order 285
Collapsing Loops to Improve Workload Balance 286
Enforcing Memory Consistency 287
An Example of Parallelization 288
Summary 293

8 Hand-Coded Synchronization and Sharing 295
Atomic Operations 295
Using Compare and Swap Instructions to Form More Complex Atomic Operations 297
Enforcing Memory Ordering to Ensure Correct Operation 301
Compiler Support of Memory-Ordering Directives 303
Reordering of Operations by the Compiler 304
Volatile Variables 308
Operating System–Provided Atomics 309
Lockless Algorithms 312
Dekker’s Algorithm 312
Producer-Consumer with a Circular Buffer 315
Scaling to Multiple Consumers or Producers 318
Scaling the Producer-Consumer to Multiple Threads 319
Modifying the Producer-Consumer Code to Use Atomics 326
The ABA Problem 329
Summary 332

9 Scaling with Multicore Processors 333
Constraints to Application Scaling 333
Performance Limited by Serial Code 334
For a number of years, home computers have given the illusion of doing multiple tasks simultaneously. This has been achieved by switching between the running tasks many times per second. This gives the appearance of simultaneous activity, but it is only an appearance. While the computer has been working on one task, the others have made no progress. An old computer that can execute only a single task at a time might be referred to as having a single processor, a single CPU, or a single “core.” The core is the part of the processor that actually does the work.

Recently, even home PCs have had multicore processors. It is now hard, if not impossible, to buy a machine that is not a multicore machine. On a multicore machine, each core can make progress on a task, so multiple tasks really do make progress at the same time.

The best way of illustrating what this means is to consider a computer that is used for converting film from a camcorder to the appropriate format for burning onto a DVD. This is a compute-intensive operation—a lot of data is fetched from disk, a lot of data is written to disk—but most of the time is spent by the processor decompressing the input video and converting that into compressed output video to be burned to disk.

On a single-core system, it might be possible to have two movies being converted at the same time while ignoring any issues that there might be with disk or memory requirements. The two tasks could be set off at the same time, and the processor in the computer would spend some time converting one video and then some time converting the other. Because the processor can execute only a single task at a time, only one video is actually being compressed at any one time. If the two videos show progress meters, the two meters will both head toward 100% completed, but it will take (roughly) twice as long to convert two videos as it would to convert a single video.

On a multicore system, there are two or more available cores that can perform the video conversion. Each core can work on one task. So, having the system work on two films at the same time will utilize two cores, and the conversion will take the same time as converting a single film. Twice as much work will have been achieved in the same time.

Multicore systems have the capability to do more work per unit time than single-core systems—two films can be converted in the same time that one can be converted on a single-core system. However, it’s possible to split the work in a different way. Perhaps the multiple cores can work together to convert the same film. In this way, a system with two cores could convert a single film twice as fast as a system with only one core.
This book is about using and developing for multicore systems. This is a topic that is often described as complex or hard to understand. In some way, this reputation is justified. Like any programming technique, multicore programming can be hard to do both correctly and with high performance. On the other hand, there are many ways that multicore systems can be used to significantly improve the performance of an application or the amount of work performed per unit time; some of these approaches will be more difficult than others.

Perhaps saying “multicore programming is easy” is too optimistic, but a realistic way of thinking about it is that multicore programming is perhaps no more complex or no more difficult than the step from procedural to object-oriented programming. This book will help you understand the challenges involved in writing applications that fully utilize multicore systems, and it will enable you to produce applications that are functionally correct, that are high performance, and that scale well to many cores.

Who Is This Book For?
If you have read this far, then this book is likely to be for you. The book is a practical guide to writing applications that are able to exploit multicore systems to their full advantage. It is not a book about a particular approach to parallelization. Instead, it covers various approaches. It is also not a book wedded to a particular platform. Instead, it pulls examples from various operating systems and various processor types. Although the book does cover advanced topics, these are covered in a context that will enable all readers to become familiar with them.

The book has been written for a reader who is familiar with the C programming language and has a fair ability at programming. The objective of the book is not to teach programming languages, but it deals with the higher-level considerations of writing code that is correct, has good performance, and scales to many cores.

The book includes a few examples that use SPARC or x86 assembly language. Readers are not expected to be familiar with assembly language, and the examples are straightforward, are clearly commented, and illustrate particular points.

Objectives of the Book
By the end of the book, the reader will understand the options available for writing programs that use multiple cores on UNIX-like operating systems (Linux, Oracle Solaris, OS X) and Windows. They will have an understanding of how the hardware implementation of multiple cores will affect the performance of the application running on the system (both in good and bad ways). The reader will also know the potential problems to avoid when writing parallel applications. Finally, they will understand how to write applications that scale up to large numbers of parallel threads.
Structure of This Book

This book is divided into the following chapters.

Chapter 1 introduces the hardware and software concepts that will be encountered in the rest of the book. The chapter gives an overview of the internals of processors. It is not necessarily critical for the reader to understand how hardware works before they can write programs that utilize multicore systems. However, an understanding of the basics of processor architecture will enable the reader to better understand some of the concepts relating to application correctness, performance, and scaling that are presented later in the book. The chapter also discusses the concepts of threads and processes.

Chapter 2 discusses profiling and optimizing applications. One of the book’s premises is that it is vital to understand where the application currently spends its time before work is spent on modifying the application to use multiple cores. The chapter covers all the leading contributors to performance over the application development cycle and discusses how performance can be improved.

Chapter 3 describes ways that multicore systems can be used to perform more work per unit time or reduce the amount of time it takes to complete a single unit of work. It starts with a discussion of virtualization where one new system can be used to replace multiple older systems. This consolidation can be achieved with no change in the software. It is important to realize that multicore systems represent an opportunity to change the way an application works; they do not require that the application be changed. The chapter continues with describing various patterns that can be used to write parallel applications and discusses the situations when these patterns might be useful.

Chapter 4 describes sharing data safely between multiple threads. The chapter leads with a discussion of data races, the most common type of correctness problem encountered in multithreaded codes. This chapter covers how to safely share data and synchronize threads at an abstract level of detail. The subsequent chapters describe the operating system-specific details.

Chapter 5 describes writing parallel applications using POSIX threads. This is the standard implemented by UNIX-like operating systems, such as Linux, Apple's OS X, and Oracle's Solaris. The POSIX threading library provides a number of useful building blocks for writing parallel applications. It offers great flexibility and ease of development.

Chapter 6 describes writing parallel applications for Microsoft Windows using Windows native threading. Windows provides similar synchronization and data sharing primitives to those provided by POSIX. The differences are in the interfaces and requirements of these functions.

Chapter 7 describes opportunities and limitations of automatic parallelization provided by compilers. The chapter also covers the OpenMP specification, which makes it relatively straightforward to write applications that take advantage of multicore processors.

Chapter 8 discusses how to write parallel applications without using the functionality in libraries provided by the operating system or compiler. There are some good reasons for writing custom code for synchronization or sharing of data. These might be for
finer control or potentially better performance. However, there are a number of pitfalls that need to be avoided in producing code that functions correctly.

Chapter 9 discusses how applications can be improved to scale in such a way as to maximize the work performed by a multicore system. The chapter describes the common areas where scaling might be limited and also describes ways that these scaling limitations can be identified. It is in the scaling that developing for a multicore system is differentiated from developing for a multiprocessor system; this chapter discusses the areas where the implementation of the hardware will make a difference.

Chapter 10 covers a number of alternative approaches to writing parallel applications. As multicore processors become mainstream, other approaches are being tried to overcome some of the hurdles of writing correct, fast, and scalable parallel code.

Chapter 11 concludes the book.
Identifying Opportunities for Parallelism

This chapter discusses parallelism, from the use of virtualization to support multiple operating systems to the use of multiple threads within a single application. It also covers the concepts involved in writing parallel programs, some ways of visualizing parallel tasks, and ways of architecting parallel applications. The chapter continues with a discussion of various parallelization strategies, or patterns. It concludes by examining some of the limitations to parallelization. By the end of the chapter, you should be in a position to understand some of the ways that a system can support multiple applications and that an existing application might be modified to utilize multiple threads. You will also be able to identify places in the code where parallelization might be applicable.

Using Multiple Processes to Improve System Productivity

Consider a home computer system. This will probably have only one active user at a time, but that user might be running a number of applications simultaneously. A system where there is a single core produces the illusion of simultaneous execution of multiple applications by switching between the active applications many times every second. A multicore system has the advantage of being able to truly run multiple applications at the same time.

A typical example of this happens when surfing the Web and checking e-mail. You may have an e-mail client downloading your e-mail while at the same time your browser is rendering a web page in the background. Although these applications will utilize multiple threads, they do not tend to require much processor time; their performance is typically dominated by the time it takes to download mail or web pages from remote machines. For these applications, even a single-core processor often provides sufficient processing power to produce a good user experience. However, a single-core processor can get saturated if the e-mail client is indexing mail while an animation-heavy web page is being displayed.
In fact, these applications will probably already take advantage of multiple threads. Figure 3.1 shows a newly opened instance of Mozilla Firefox launching 20 threads. A consequence of this is that just by having a multicore processor, the performance of the system will improve because multiples of those threads can be executed simultaneously—and this requires no change to the existing applications.

Alternatively, there are a number of tasks we perform on our personal computer systems that are inherently compute intensive, such as playing computer games, encoding audio for an MP3 player, transforming one video format into another suitable for burning to DVD, and so on. In these instances, having multiple cores can enable the work to take less time by utilizing additional cores or can keep the system responsive while the task is completed in the background.

Figure 3.2 shows the system stack when a single user runs multiple applications on a system.

![Windows Process Explorer showing thread activity in Mozilla Firefox](image)

Figure 3.1 Windows Process Explorer showing thread activity in Mozilla Firefox.
It is also possible to have multiple users in a home environment. For example, on Windows, it is quite possible for one user to be logged in and using the computer while another user, having logged in earlier, has set some other applications running. For example, you may have left some DVD-authoring package running in the background while another user logs into their account to check their e-mail.

**Multiple Users Utilizing a Single System**

In business and enterprise computing, it is much more common to encounter systems with multiple simultaneous users. This is often because the computer and software being shared are more powerful and more costly than the typical consumer system. To maximize efficiency, a business might maintain a database on a single shared system. Multiple users can simultaneously access this system to add or retrieve data. These users might just as easily be other applications as well as humans.

For many years, multiuser operating systems like UNIX and Linux have enabled sharing of compute resources between multiple users. Each user gets a “slice” of the available compute resources. In this way, multicore systems provide more compute resources for the users to share.

Figure 3.3 illustrates the situation with multiple users of the same system.

Multicore systems can be very well utilized running multiple applications, running multiple copies of the same application, and supporting multiple simultaneous users. To the OS, these are all just multiple processes, and they will all benefit from the capabilities of a multicore system.

Multiuser operating systems enforce separation between the applications run by different users. If a program one user was running were to cause other applications to crash or to write randomly to disk, the damage is limited to only those applications owned by that user or the disk space they have permission to change.
Chapter 3 Identifying Opportunities for Parallelism

Such containment and security is critical for supporting multiple simultaneous users. As the number of users increases, so does the chance that one of them will do something that could “damage” the rest of the system. This could be something as simple as deleting critical files or enabling someone to get unauthorized access to the system.

**Improving Machine Efficiency Through Consolidation**

Multicore computing is really just the continuing development of more powerful system architectures. Tasks that used to require a dedicated machine can now be performed using a single core of a multicore machine. This is a new opportunity to consolidate multiple tasks from multiple separate machines down to a single multicore machine. An example might be using a single machine for both a web server and e-mail where previously these functions would be running on their own dedicated machines.

There are many ways to achieve this. The simplest would be to log into the machine and start both the e-mail and web server. However, for security reasons, it is often necessary to keep these functions separated. It would be unfortunate if it were possible to send a suitably formed request to the web server allowing it to retrieve someone’s e-mail archive.

The obvious solution would be to run both servers as different users. This could use the default access control system to stop the web server from getting access to the e-mail server’s file. This would work, but it does not guard against user error. For example, someone might accidentally put one of the mail server’s files under the wrong permissions, leaving the mail open to reading or perhaps leaving it possible to install a back door into the system. For this reason, smarter technologies have evolved to provide better separation between processes running on the same machine.
Using Containers to Isolate Applications Sharing a Single System

One such technology is containerization. The implementations depend on the particular operating system, for example, Solaris has Zones, whereas FreeBSD has Jails, but the concept is the same. A control container manages the host operating system, along with a multitude of guest containers. Each guest container appears to be a complete operating system instance in its own right, and an application running in a guest container cannot see other applications on the system either in other guest containers or in the control container. The guests do not even share disk space; each guest container can appear to have its own root directory system.

The implementation of the technology is really a single instance of the operating system, and the illusion of containers is maintained by hiding applications or resources that are outside of the guest container. The advantage of this implementation is very low overhead, so performance comes very close to that of the full system. The disadvantage is that the single operating system image represents a single point of failure. If the operating system crashes, then all the guests also crash, since they also share the same image. Figure 3.4 illustrates containerization.

Hosting Multiple Operating Systems Using Hypervisors

Two other approaches that enforce better isolation between guests’ operating systems also remove the restriction that the guests run the same operating system as the host. These approaches are known as type 1 and type 2 hypervisors.

Type 1 hypervisors replace the host operating system with a very lightweight but high-level system supervisor system, or hypervisor, that can load and initiate multiple operating system instances on its own. Each operating system instance is entirely isolated from the others while sharing the same hardware.
Each operating system appears to have access to its own machine. It is not apparent, from within the operating system, that the hardware is being shared. The hardware has effectively been *virtualized*, in that the guest operating system will believe it is running on whatever type of hardware the hypervisor indicates.

This provides the isolation that is needed for ensuring both security and robustness, while at the same time making it possible to run multiple copies of different operating systems as guests on the same host. Each guest believes that the entire hardware resources of the machine are available. Examples of this kind of hypervisor are the Logical Domains provided on the Sun UltraSPARC T1 and T2 product lines or the Xen hypervisor software on x86. Figure 3.5 illustrates a type 1 hypervisor.

![Figure 3.5 Type 1 hypervisor](image)

A type 2 hypervisor is actually a normal user application running on top of a host operating system. The hypervisor software is architected to host other operating systems. Good examples of type 2 hypervisors are the open source VirtualBox software, VMware, or the Parallels software for the Apple Macintosh. Figure 3.6 illustrates a type 2 hypervisor.

Clearly, it is also possible to combine these strategies and have a system that supports multiple levels of virtualization, although this might not be good for overall performance.

Even though these strategies are complex, it is worth exploring why virtualization is an appealing technology.

- **Security.** In a virtualized or containerized environment, it is very hard for an application in one virtualized operating system to obtain access to data held in a different one. This also applies to operating systems being hacked; the damage that a hacker can do is constrained by what is visible to them from the operating system that they hacked into.

- **Robustness.** With virtualization, a fault in a guest operating system can affect only those applications running on that operating system, not other applications running in other guest operating systems.
- **Configuration isolation.** Some applications expect to be configured in particular ways: They might always expect to be installed in the same place or find their configuration parameters in the same place. With virtualization, each instance believes it has the entire system to itself, so it can be installed in one place and not interfere with another instance running on the same host system in a different virtualized container.

- **Restricted control.** A user or application can be given root access to an instance of a virtualized operating system, but this does not give them absolute control over the entire system.

- **Replication.** There are situations, such as running a computer lab, where it is necessary to be able to quickly reproduce multiple instances of an identical configuration. Virtualization can save the effort of performing clean reinstalls of an operating system. A new guest operating system can be started, providing a new instance of the operating system. This new instance can even use a preconfigured image, so it can be up and running easily.

- **Experimentation.** It is very easy to distribute a virtualized image of an operating system. This means a user can try a new operating system without doing any damage to their existing configuration.

- **Hardware isolation.** In some cases, it is possible to take the running image of a virtualized operating system and move that to a new machine. This means that old or broken hardware can be switched out without having to make changes to the software running on it.

- **Scaling.** It is possible to dynamically respond to increased requests for work by starting up more virtual images. For example, a company might provide a web-hosted computation on-demand service. Demand for the service might peak on weekday evenings but be very low the rest of the time. Using virtualization, it
would be possible to start up new virtual machines to handle the load at the times when the demand increases.

- **Consolidation.** One of the biggest plays for virtualization is that of consolidating multiple old machines down to fewer new machines. Virtualization can take the existing applications, and their host operating systems can move them to a new host. Since the application is moved with its host operating system, the transition is more likely to be smooth than if the application had to be reconfigured for a new environment.

All these characteristics of virtualization make it a good fit for *cloud computing*. Cloud computing is a service provided by a remote farm of machines. Using virtualization, each user can be presented with root access to an unshared virtual machine. The number of machines can be scaled to match the demand for their service, and new machines can quickly be brought into service by replicating an existing setup. Finally, the software is isolated from the physical hardware that it is running on, so it can easily be moved to new hardware as the farm evolves.

**Using Parallelism to Improve the Performance of a Single Task**

Virtualization provides one way of utilizing a multicore or multiprocessor system by extracting parallelism at the highest level: running multiple tasks or applications simultaneously. For a user, a compelling feature of virtualization is that utilizing this level of parallelism becomes largely an administrative task.

But the deeper question for software developers is how multiple cores can be employed to improve the throughput or computational speed of a single application. The next section discusses a more tightly integrated parallelism for enabling such performance gains.

**One Approach to Visualizing Parallel Applications**

One way to visualize parallelization conceptually is to imagine that there are two of you; each thinks the same thoughts and behaves in the same way. Potentially, you could achieve twice as much as one of you currently does, but there are definitely some issues that the two of you will have to face.

You might imagine that your double could go out to work while you stay at home and read books. In this situation, you are implicitly controlling your double: You tell them what to do.

However, if you’re both identical, then your double would also prefer to stay home and read while you go out to work. So, perhaps you would have to devise a way to determine which of you goes to work today—maybe splitting the work so that one would go one week, and the other the next week.
Of course, there would also be problems on the weekend, when you both would want to read the same newspaper at the same time. So, perhaps you would need two copies of the paper or work out some way of sharing it so only one of you had the paper at a time.

On the other hand, there would be plenty of benefits. You could be painting one wall, while your double is painting another. One of you could mow the lawn while the other washes the dishes. You could even work together cooking the dinner; one of you could be chopping vegetables while the other is frying them.

Although the idea of this kind of double person is fanciful, these examples represent very real issues that arise when writing parallel applications. As a thought experiment, imagining two people collaborating on a particular task should help you identify ways to divide the task and should also indicate some of the issues that result.

The rest of the chapter will explore some of these opportunities and issues in more detail. However, it will help in visualizing the later parts of the chapter if you can take some of these more “human” examples and draw the parallels to the computational problems.

Parallelism provides an opportunity to get more work done. This work might be independent tasks, such as mowing the lawn and washing the dishes. These could correspond to different processes or perhaps even different users utilizing the same system. Painting the walls of a house requires a little more communication—you might need to identify which wall to paint next—but generally the two tasks can proceed independently. However, when it comes to cooking a meal, the tasks are much more tightly coupled. The order in which the vegetables are chopped should correspond to the order in which they are needed. You might even need messages like “Stop what you’re doing and get me more olive oil, now!” Preparing a meal requires a high amount of communication between the two workers.

The more communication is required, the more likely it is that the effect of the two workers will not be a doubling of performance. An example of communication might be to indicate which order the vegetables should be prepared in. Inefficiencies might arise when the person cooking is waiting for the other person to complete chopping the next needed vegetable.

The issue of accessing resources, for example, both wanting to read the same newspaper, is another important concern. It can sometimes be avoided by duplicating resources—both of you having your own copies—but sometimes if there is only a single resource, we will need to establish a way to share that resource.

In the next section, we will explore this thought experiment further and observe how the algorithm we use to solve a problem determines how efficiently the problem can be solved.

**How Parallelism Can Change the Choice of Algorithms**

Algorithms have characteristics that make them more or less appropriate for a multi-threaded implementation. For example, suppose you have a deck of playing cards that are in a random order but you would like to sort them in order. One way to do this would
be to hold the unsorted cards in one hand and place each card into its appropriate place in the other hand. There are \( N \) cards, and a binary search is needed to locate each card into its proper place. So, going back to the earlier discussion on algorithmic complexity, this is an \( O(n \times \log(n)) \) algorithm.

However, suppose you have someone to help, and you each decide to sort half the pack. If you did that, you would end up with two piles of sorted cards, which you would then have to combine. To combine them, you could each start with a pile of cards, and then whoever had the next card could place it onto the single sorted stack. The complexity of the sort part of this algorithm would be \( O(n \times \log(n)) \) (for a value of \( n \) that was half the original), and the combination would be \( O(n) \). So although we have increased the number of “threads,” we do not guarantee a doubling of performance.

An alternative way of doing this would be to take advantage of the fact that playing cards have an existing and easily discernible order. If instead of sorting the cards, you just place them at the correct place on a grid. The grid could have the “value” of the card as the \( x \)-axis and the “suit” of the card as the \( y \)-axis. This would be an \( O(n) \) operation since the time it takes to place a single card does not depend on the number of cards that are present in the deck. This method is likely to be slightly slower than keeping the cards in your hands because you will have to physically reach to place the cards into the appropriate places in the grid. However, if you have the benefit of another person helping, then the deck can again be split into two, and each person would have to sort only half the cards. Assuming you don’t obstruct each other, you should be able to attain a near doubling of performance. So, comparing the two algorithms, using the grid method might be slower for a single person but would scale better with multiple people.

The point here is to demonstrate that the best algorithm for a single thread may not necessarily correspond to the best parallel algorithm. Further, the best parallel algorithm may be slower in the serial case than the best serial algorithm.

Proving the complexity of a parallel algorithm is hard in the general case and is typically handled using approximations. The most common approximation to parallel performance is Amdahl’s law.

**Amdahl’s Law**

Amdahl’s law is the simplest form of a scaling law. The underlying assumption is that the performance of the parallel code scales with the number of threads. This is unrealistic, as we will discuss later, but does provide a basic starting point. If we assume that \( S \) represents the time spent in serial code that cannot be parallelized and \( P \) represents the time spent in code that can be parallelized, then the runtime of the serial application is as follows:

\[
\text{Runtime} = S + P
\]

The runtime of a parallel version of the application that used \( N \) processors would take the following:
Runtime = \( S + \frac{P}{N} \)

It is probably easiest to see the scaling diagrammatically. In Figure 3.7, we represent the runtime of the serial portion of the code and the portion of the code that can be made to run in parallel as rectangles.

![Figure 3.7 Single-threaded runtime](image)

If we use two threads for the parallel portion of the code, then the runtime of that part of the code will halve, and Figure 3.8 represents the resulting processor activity.

![Figure 3.8 Runtime with two threads](image)

If we were to use four threads to run this code, then the resulting processor activity would resemble Figure 3.9.

![Figure 3.9 Runtime with four threads](image)

There are a couple of things that follow from Amdahl’s law. As the processor count increases, performance becomes dominated by the serial portion of the application. In
the limit, the program can run no faster than the duration of the serial part, S. Another observation is that there are diminishing returns as the number of threads increases: At some point adding more threads does not make a discernible difference to the total runtime.

These two observations are probably best illustrated using the chart in Figure 3.10, which shows the parallel speedup over the serial case for applications that have various amounts of code that can be parallelized.

![Figure 3.10 Scaling with diminishing parallel regions](image)

If all the code can be made to run in parallel, the scaling is perfect; a code run with 18 threads will be 18x faster than the serial version of the code. However, it is surprising to see how fast scaling declines as the proportion of code that can be made to run in parallel drops. If 99% of the application can be converted to parallel code, the application would scale to about 15x the serial performance with 18 threads. At 95% serial, this would drop to about 10x the serial performance. If only half the application can be run in parallel, then the best that can be expected is for performance to double, and the code would pretty much attain that at a thread count of about 8.

There is another way of using Amdahl’s law, and that is to look at how many threads an application can scale to given the amount of time it spends in code that can be parallelized.
Determining the Maximum Practical Threads

If we take Amdahl’s law as a reasonable approximation to application scaling, it becomes an interesting question to ask how many threads we should expect an application to scale to.

If we have an application that spends only 10% of its time in code that can be parallelized, it is unlikely that we’ll see much noticeable gain when using eight threads over using four threads. If we assume it took 100 seconds to start with, then four threads would complete the task in 92.5 seconds, whereas eight threads would take 91.25 seconds. This is just over a second out of a total duration of a minute and a half. In case the use of seconds might be seen as a way of trivializing the difference, imagine that the original code took 100 days; then the difference is equivalent to a single day out of a total duration of three months.

There will be some applications where every last second is critical and it makes sense to use as many resources as possible to increase the performance to as high as possible. However, there are probably a large number of applications where a small gain in performance is not worth the effort.

We can analyze this issue assuming that a person has a tolerance, T, within which they cease to care about a difference in performance. For many people this is probably 10%; if the performance that they get is within 10% of the best possible, then it is acceptable. Other groups might have stronger or weaker constraints.

Returning to Amdahl’s law, recall that the runtime of an application that has a proportion P of parallelizable code and S of serial code and that is run with N threads is as follows:

\[
\text{Runtime}_N = S + \frac{P}{N}
\]

The optimal runtime, when there are an infinite number of threads, is S. So, a runtime within T percent of the optimal would be as follows:

\[
\text{Acceptable runtime} = S \times (1 + T)
\]

We can compare the acceptable runtime with the runtime with N threads:

\[
S \times (1 + T) = \left( S + \frac{P}{N} \right)
\]

We can then rearrange and solve for N to get the following relationship for N:

\[
N = \frac{P}{ST} = \frac{P}{(1 - P)T}
\]
Using this equation, Figure 3.11 shows the number of threads necessary to get a runtime that is within 10% of the best possible.

Reading this chart, it is clear that an application will have only limited scalability until it spends at least half of its runtime in code that can be parallelized. For an application to scale to large numbers of cores, it requires that 80%+ of the serial runtime is spent in parallelizable code.

If Amdahl’s law were the only constraint to scaling, then it is apparent that there is little benefit to using huge thread counts on any but the most embarrassingly parallel applications. If performance is measured as throughput (or the amount of work done), it is probable that for a system capable of running many threads, those threads may be better allocated to a number of processes rather than all being utilized by a single process.

However, Amdahl’s law is a simplification of the scaling situation. The next section will discuss a more realistic model.

**How Synchronization Costs Reduce Scaling**

Unfortunately, there are overhead costs associated with parallelizing applications. These are associated with making the code run in parallel, with managing all the threads, and with the communication between threads. You can find a more detailed discussion in Chapter 9, “Scaling on Multicore Systems.”

In the model discussed here, as with Amdahl’s law, we will ignore any costs introduced by the implementation of parallelization in the application and focus entirely on the costs of synchronization between the multiple threads. When there are multiple threads cooperating to solve a problem, there is a communication cost between all the
threads. The communication might be the command for all the threads to start, or it might represent each thread notifying the main thread that it has completed its work.

We can denote this synchronization cost as some function $F(N)$, since it will increase as the number of threads increases. In the best case, $F(N)$ would be a constant, indicating that the cost of synchronization does not change as the number of threads increases. In the worst case, it could be linear or even exponential with the number threads. A fair estimate for the cost might be that it is proportional to the logarithm of the number of threads ($F(N) = K \times \ln(N)$); this is relatively easy to argue for since the logarithm represents the cost of communication if those threads communicated using a balanced tree. Taking this approximation, then the cost of scaling to $N$ threads would be as follows:

$$\text{Runtime} = S + \frac{P}{N} + K \ln(N)$$

The value of $K$ would be some constant that represents the communication latency between two threads together with the number of times a synchronization point is encountered (assuming that the number of synchronization points for a particular application and workload is a constant). $K$ will be proportional to memory latency for those systems that communicate through memory, or perhaps cache latency if all the communicating threads share a common level of cache. Figure 3.12 shows the curves resulting from an unrealistically large value for the constant $K$, demonstrating that at some thread count the performance gain over the serial case will start decreasing because of the synchronization costs.

![Figure 3.12 Scaling with exaggerated synchronization overheads](image-url)
It is relatively straightforward to calculate the point at which this will happen:

\[
\frac{d \text{ runtime}}{dN} = -\frac{P}{N^2} + \frac{K}{N}
\]

Solving this for \( N \) indicates that the minimal value for the runtime occurs when

\[
N = \frac{P}{K}
\]

This tells us that the number of threads that a code can scale to is proportional to the ratio of the amount of work that can be parallelized and the cost of synchronization. So, the scaling of the application can be increased either by making more of the code run in parallel (increasing the value of \( P \)) or by reducing the synchronization costs (reducing the value of \( K \)). Alternatively, if the number of threads is held constant, then reducing the synchronization cost (making \( K \) smaller) will enable smaller sections of code to be made parallel (\( P \) can also be made smaller).

What makes this interesting is that a multicore processor will often have threads sharing data through a shared level of cache. The shared level of cache will have lower latency than if the two threads had to communicate through memory. Synchronization costs are usually proportional to the latency of the memory through which the threads communicate, so communication through a shared level of cache will result in much lower synchronization costs. This means that multicore processors have the opportunity to be used for either parallelizing regions of code where the synchronization costs were previously prohibitive or, alternatively, scaling the existing code to higher thread counts than were previously possible.

So far, this chapter has discussed the expectations that a developer should have when scaling their code to multiple threads. However, a bigger issue is how to identify work that can be completed in parallel, as well as the patterns to use to perform this work. The next section discusses common parallelization patterns and how to identify when to use them.

**Parallelization Patterns**

There are many ways that work can be divided among multiple threads. The objective of this section is to provide an overview of the most common approaches and to indicate when these might be appropriate.

Broadly speaking, there are two categories of parallelization, often referred to as *data parallel* and *task parallel*.

A data parallel application has multiple threads performing the same operation on separate items of data. For example, multiple threads could each take a chunk of itera-
tions from a single loop and perform those iterations on different elements in a single
array. All the threads would perform the same task but to different array indexes.

A task parallel application would have separate threads performing different operations
on different items of data. For example, an animated film could be produced having one
process render each frame and then a separate process take each rendered frame and
incorporate it into a compressed version of the entire film.

**Data Parallelism Using SIMD Instructions**

Although this book discusses data parallelism in the context of multiple threads cooper-
ating on processing the same item of data, the concept also extends into instruction sets.
There are instructions, called *single instruction multiple data* (SIMD) instructions, that load
a vector of data and perform an operation on all the items in the vector. Most processors
have these instructions: the SSE instruction set extensions for x86 processors, the VIS
instructions for SPARC processors, and the AltiVec instructions on Power/
PowerPC processors.

The loop shown in Listing 3.1 is ideal for conversion into SIMD instructions.

**Listing 3.1 Loop Adding Two Vectors**

```c
void vadd(double * restrict a, double * restrict b , int count)
{
    for (int i=0; i < count; i++)
    {
        a[i] += b[i];
    }
}
```

Compiling this on an x86 box without enabling SIMD instructions generates the
assembly language loop shown in Listing 3.2.

**Listing 3.2 Assembly Language Code to Add Two Vectors Using x87 Instructions**

```Assembly
loop:
    fldl (%edx)     // Load the value of a[i]
    faddl (%ecx)    // Add the value of b[i]
    fstpl (%edx)    // Store the result back to a[i]
    addl 8,%edx     // Increment the pointer to a
    addl 8,%ecx     // Increment the pointer to b
    addl 1,%esi     // Increment the loop counter
    cmp %eax,%esi    // Test for the end of the loop
    jle loop        // Branch back to start of loop if not complete
```

Compiling with SIMD instructions produces code similar to that shown in Listing 3.3.
Listing 3.3  Assembly Language Code to Add Two Vectors Using SSE Instructions

```
loop:
movupd (%edx),%xmm0 // Load a[i] and a[i+1] into vector register
movupd (%ecx),%xmm1 // Load b[i] and b[i+1] into vector register
addpd %xmm1,%xmm0  // Add vector registers
movpd %xmm0,(%edx) // Store a[i] and a[i+1] back to memory
addl 16,%edx      // Increment pointer to a
addl 16,%ecx      // Increment pointer to b
addl 2,%esi       // Increment loop counter
cmp %eax,%esi    // Test for the end of the loop
jle loop         // Branch back to start of loop if not complete
```

Since two double-precision values are computed at the same time, the trip count around the loop is halved, so the number of instructions is halved. The move to SIMD instructions also enables the compiler to avoid the inefficiencies of the stack-based x87 floating-point architecture.

SIMD and parallelization are very complementary technologies. SIMD is often useful in situations where loops perform operations over vectors of data. These same loops could also be parallelized. Simultaneously using both approaches enables a multicore chip to achieve high throughput. However, SIMD instructions have an additional advantage in that they can also be useful in situations where the amount of work is too small to be effectively parallelized.

**Parallelization Using Processes or Threads**

The rest of the discussion of parallelization strategies in this chapter will use the word *tasks* to describe the work being performed and the word *thread* to describe the instruction stream performing that work. The use of the word *thread* is purely a convenience. These strategies are applicable to a multithreaded application where there would be a single application with multiple cooperating threads and to a multiprocess application where there would be an application made up of multiple independent processes (with some of the processes potentially having multiple threads).

The trade-offs between the two approaches are discussed in Chapter 1, “Hardware, Processes, and Threads.” Similarly, these patterns do not need to be restricted to a single system. They are just as applicable to situations where the work is spread over multiple systems.

**Multiple Independent Tasks**

As discussed earlier in the chapter, the easiest way of utilizing a CMT system is to perform many independent tasks. In this case, the limit to the number of independent tasks is determined by resources that are external to those tasks. A web server might require a large memory footprint for caching recently used web pages in memory. A database server might require large amounts of disk I/O. These requirements would place load on
the system and on the operating system, but there would be no synchronization constraints between the applications running on the system.

A system running multiple tasks could be represented as a single system running three independent tasks, A, B, and C, as shown in Figure 3.13.

![Figure 3.13 Three independent tasks](image)

An example of this kind of usage would be consolidation of multiple machines down to a single machine. This consolidation might just be running the web server, e-mail server, and so on, on the same machine or might involve some form of virtualization where different tasks are isolated from each other.

This approach is very common but not terribly interesting from a parallelization strategy since there is no communication between the components. Such an approach would increase the utilization of the machine and could result in space or power savings but should not be expected to lead to a performance change (except that which is attained from the intrinsic differences in system performance).

One place where this strategy is common is in cluster, grid, or cloud computing. Each individual node (that is, system) in the cloud might be running a different task, and the tasks are independent. If a task fails (or a node fails while completing a task), the task can be retried on a different node. The performance of the cloud is the aggregate throughput of all the nodes.

What is interesting about this strategy is that because the tasks are independent, performance (measured as throughput) should increase nearly linearly with the number of available threads.

**Multiple Loosely Coupled Tasks**

A slight variation on the theme of multiple independent tasks would be where the tasks are different, but they work together to form a single application. Some applications do need to have multiple independent tasks running simultaneously, with each task generally independent and often different from the other running tasks. However, the reason this is an application rather than just a collection of tasks is that there is some element of communication within the system. The communication might be from the tasks to a central task controller, or the tasks might report some status back to a status monitor.
In this instance, the tasks themselves are largely independent. They may occasionally communicate, but that communication is likely to be asynchronous or perhaps limited to exceptional situations.

Figure 3.14 shows a single system running three tasks. Task A is a control or supervisor, and tasks B and C are reporting status to task A.

The performance of the application depends on the activity of these individual tasks. If the CPU-consuming part of the “application” has been split off into a separate task, then the rest of the components may become more responsive. For an example of this improved responsiveness, assume that a single-threaded application is responsible for receiving and forwarding packets across the network and for maintaining a log of packet activity on disk. This could be split into two loosely coupled tasks—one receives and forwards the packets while the other is responsible for maintaining the log. With the original code, there might be a delay in processing an incoming packet if the application is busy writing status to the log. If the application is split into separate tasks, the packet can be received and forwarded immediately, and the log writer will record this event at a convenient point in the future.

The performance gain arises in this case because we have shared the work between two threads. The packet-forwarding task only has to process packets and does not get delayed by disk activity. The disk-writing task does not get stalled reading or writing packets. If we assume that it takes 1ms to read and forward the packet and another 1ms to write status to disk, then with the original code, we can process a new packet every 2ms (this represents a rate of 5,000 packets per second). Figure 3.15 shows this situation.
If we split these into separate tasks, then we can handle a packet every 1ms, so throughput will have doubled. It will also improve the responsiveness because we will handle each packet within 1ms of arrival, rather than within 2ms. However, it still takes 2ms for the handling of each packet to complete, so the throughput of the system has doubled, but the response time has remained the same. Figure 3.16 shows this situation.

![Figure 3.16 Using two threads to perform packet forwarding and log writing](image)

**Multiple Copies of the Same Task**

An easy way to complete more work is to employ multiple copies of the same task. Each individual task will take the same time to complete, but because multiple tasks are completed in parallel, the throughput of the system will increase.

This is a very common strategy. For example, one system might be running multiple copies of a rendering application in order to render multiple animations. Each application is independent and requires no synchronization with any other.

Figure 3.17 shows this situation, with a single system running three copies of task A.

![Figure 3.17 Multiple copies of a single task](image)
Once again, the performance of the system is an increase in throughput, not an improvement in the rate at which work is completed.

**Single Task Split Over Multiple Threads**

Splitting a single task over multiple threads is often what people think of as parallelization. The typical scenario is distributing a loop’s iterations among multiple threads so that each thread gets to compute a discrete range of the iterations.

This scenario is represented in Figure 3.18 as a system running three threads and each of the threads handling a separate chunk of the work.

![Figure 3.18 Multiple threads working on a single task](image)

In this instance, a single unit of work is being divided between the threads, so the time taken for the unit of work to complete should diminish in proportion to the number of threads working on it. This is a reduction in completion time and would also represent an increase in throughput. In contrast, the previous examples in this section have represented increases in the amount of work completed (the throughput), but not a reduction in the completion time for each unit of work.

This pattern can also be considered a fork-join pattern, where the fork is the division of work between the threads, and the join is the point at which all the threads synchronize, having completed their individual assignments.

Another variation on this theme is the divide-and-conquer approach where a problem is recursively divided as it is divided among multiple threads.

**Using a Pipeline of Tasks to Work on a Single Item**

A pipeline of tasks is perhaps a less obvious strategy for parallelization. Here, a single unit of work is split into multiple stages and is passed from one stage to the next rather like an assembly line.

Figure 3.19 represents this situation. A system has three separate threads; when a unit of work comes in, the first thread completes task A and passes the work on to task B, which is performed by the second thread. The work is completed by the third thread performing task C. As each thread completes its task, it is ready to accept new work.
There are various motivations for using a pipeline approach. A pipeline has some amount of flexibility, in that the flow of work can be dynamically changed at runtime. It also has some implicit scalability because an implementation could use multiple copies of a particular time-consuming stage in the pipeline (combining the pipeline pattern with the multiple copies of a single task pattern), although the basic pipeline model would have a single copy of each stage.

This pattern is most critical in situations where it represents the most effective way the problem can be scaled to multiple threads. Consider a situation where packets come in for processing, are processed, and then are retransmitted. A single thread can cope only with a certain limit of packets per second. More threads are needed in order to improve performance. One way of doing this would be to increase the number of threads doing the receiving, processing, and forwarding. However, that might introduce additional complexity in keeping the packets in the same order and synchronizing the multiple processing threads.

In this situation, a pipeline looks attractive because each stage can be working on a separate packet, which means that the performance gain is proportional to the number of active threads. The way to view this is to assume that the original processing of a packet took three seconds. So, every three seconds a new packet could be dealt with. When the processing is split into three equal pipeline stages, each stage will take a second. More specifically, task A will take one second before it passes the packet of work on to task B, and this will leave the first thread able to take on a new packet of work. So, every second there will be a packet starting processing. A three-stage pipeline has improved performance by a factor of three. The issues of ordering and synchronization can be dealt with by placing the items in a queue between the stages so that order is maintained.

Notice that the pipeline does not reduce the time taken to process each unit of work. In fact, the queuing steps may slightly increase it. So, once again, it is a throughput improvement rather than a reduction in unit processing time.

One disadvantage to pipelines is that the rate that new work can go through the pipeline is limited by the time that it takes for the work of the slowest stage in the pipeline to complete. As an example, consider the case where task B takes two seconds. The second thread can accept work only every other second, so regardless of how much faster tasks A and C are to complete, task B limits the throughput of the pipeline to one task every two seconds. Of course, it might be possible to rectify this bottleneck by having
two threads performing task B. Here the combination would complete one task every second, which would match the throughput of tasks A and C. It is also worth considering that the best throughput occurs when all the stages in the pipeline take the same amount of time. Otherwise, some stages will be idle waiting for more work.

**Division of Work into a Client and a Server**

With a *client-server* configuration, one thread (the *client*) communicates requests to another thread (the *server*), and the other thread responds. The split into client and server might provide a performance improvement, because while the server is performing some calculation, the client can be responding to the user; the client might be the visible UI to the application, and the server might be the compute engine that is performing the task in the background. There are plenty of examples of this approach, such as having one thread to manage the redraw of the screen while other threads handle the activities of the application. Another example is when the client is a thread running on one system while the server is a thread running on a remote system; web browsers and web servers are obvious, everyday examples.

A big advantage of this approach is the sharing of resources between multiple clients. For example, a machine might have a single Ethernet port but have multiple applications that need to communicate through that port. The client threads would send requests to a server thread. The server thread would have exclusive access to the Ethernet device and would be responsible for sending out the packets from the clients and directing incoming packets to the appropriate client in an orderly fashion.

This client-server relationship can be represented as multiple clients: A, communicating with a server, B, as shown in Figure 3.20. Server B might also control access to a set of resources, which are not explicitly included in the diagram.

![Figure 3.20 Client-server division of work](image)

Implicit in the client-server pattern is the notion that there will be multiple clients seeking the attention of a single server. The single server could, of course, be implemented using multiple threads.

The client-server pattern does not improve responsiveness but represents a way of sharing the work between multiple threads, especially where the server thread actually does some work. Alternatively, it represents a way of sharing a common resource between
multiple clients (in which case any gains in throughput are a fortunate by-product rather than a design goal).

**Splitting Responsibility into a Producer and a Consumer**

A *producer-consumer model* is similar to both the pipeline model and the client-server. Here, the producer is generating units of work, and the consumer is taking those units of work and performing some kind of process on them.

For example, the movie-rendering problem described earlier might have a set of producers generating rendered frames of a movie. The consumer might be the task that has the work of ordering these frames correctly and then saving them to disk.

This can be represented as multiple copies of task A sending results to a single copy of task B, as shown in Figure 3.21. Alternatively, there could be multiple producers and a single consumer or multiple producers and consumers.

![Figure 3.21 Producer-consumer division of work](image)

Again, this approach does not necessarily reduce the latency of the tasks but provides an improvement in throughput by allowing multiple tasks to progress simultaneously. In common with the client-server task, it may also provide a way of reducing the complexity of combining the output from multiple producers of data.

**Combining Parallelization Strategies**

In many situations, a single parallelization strategy might be all that is required to produce a parallel solution for a problem. However, in other situations, there is no single strategy sufficient to solve the problem effectively, and it is necessary to select a combination of approaches.

The pipeline strategy represents a good starting point for a combination of approaches. The various stages in the pipeline can be further parallelized. For example, one stage might use multiple threads to perform a calculation on one item of data. A different stage might have multiple threads working on separate items of data.

When mapping a process to an implementation, it is important to consider all the ways that it is possible to exploit parallelism and to avoid limiting yourself to the first approach that comes to mind. Consider a situation where a task takes 100 seconds to
complete. Suppose that it's possible to take 80 of those seconds and use four threads to complete the work. Now the runtime for the task is 20 serial seconds, plus 20 seconds when four threads are active, for a total of 40 seconds. Suppose that it is possible to use a different strategy to spread the serial 20 seconds over two threads, leading to a performance gain of 10 seconds, so the total runtime is now 30 seconds: 10 seconds with two threads and 20 seconds with four threads. The first parallelization made the application two and a half times faster. The second parallelization made it 1.3x faster, which is not nearly as great but is still a significant gain. However, if the second optimization had been the only one performed, it would have resulted in only a 1.1x performance gain, not nearly as dramatic a pay-off as the 1.3x gain that it obtained when other parts of the code had already been made parallel.

**How Dependencies Influence the Ability Run Code in Parallel**

Dependencies within an application (or the calculation it performs) define whether the application can possibly run in parallel. There are two types of dependency: **loop-carried dependencies** and **memory-carried dependencies**.

With a loop-carried dependency, the next calculation in a loop cannot be performed until the results of the previous iteration are known. A good example of this is the loop to calculate whether a point is in the Mandelbrot set. Listing 3.4 shows this loop.

**Listing 3.4 Code to Determine Whether a Point Is in the Mandelbrot Set**

```c
int inSet(double ix, double iy)
{
    int iterations=0;
    double x = ix, y = iy, x2 = x*x, y2 = y*y;
    while ( (x2+y2 < 4) && (iterations < 1000) )
    {
        y  = 2 * x * y + iy;
        x  = x2 - y2 + ix;
        x2 = x * x;
        y2 = y * y;
        iterations++;
    }
    return iterations;
}
```

Each iteration of the loop depends on the results of the previous iteration. The loop terminates either when 1,000 iterations have been completed or when the point escapes a circle centered on the origin of radius two. It is not possible to predict how many iterations this loop will complete. There is also insufficient work for each iteration of the loop to be split over multiple threads. Hence, this loop must be performed serially.
Memory-carried dependencies are more subtle. These represent the situation where a memory access must be ordered with respect to another memory access to the same location. Consider the snippet of code shown in Listing 3.5.

**Listing 3.5**  **Code Demonstrating Ordering Constraints**

```c
int val=0;

void g()
{
    val = 1;
}

void h()
{
    val = val + 2;
}
```

If the routines `g()` and `h()` are executed by different threads, then the result depends on the order in which the two routines are executed. If `g()` is executed followed by `h()`, then the `val` will hold the result 3. If they are executed in the opposite order, then `val` will contain the result 1. This is an example of a memory-carried dependence because to produce the correct answer, the operations need to be performed in the correct order.

**Antidependencies and Output Dependencies**

Suppose one task, A, needs the data produced by another task, B; A depends on B and cannot start until B completes and releases the data needed by A. This is often referred to as *true dependency*. Typically, B writes some data, and A needs to read that data. There are other combinations of two threads reading and writing data. Table 3.1 illustrates the four ways that tasks might have a dependency.

**Table 3.1**  **Possible Ordering Constraints**

<table>
<thead>
<tr>
<th>First task</th>
<th>Second task Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read after read (RAR)</td>
<td>Write after read (WAR)</td>
</tr>
<tr>
<td>Write</td>
<td>Read after write (RAW)</td>
<td>Write after write (WAW)</td>
</tr>
<tr>
<td></td>
<td>No dependency</td>
<td>Antidependency</td>
</tr>
<tr>
<td></td>
<td>True dependency</td>
<td>Output dependency</td>
</tr>
</tbody>
</table>

When both threads perform read operations, there is no dependency between them, and the same result is produced regardless of the order the threads run in.
With an antidependency, or write after read, one task has to read the data before the second task can overwrite it. With an output dependency, or write after write, one of the two tasks has to provide the final result, and the order in which the two tasks write their results is critical. These two types of dependency can be most clearly illustrated using serial code.

In the code shown in Listing 3.6, there is an antidependency on the variable \texttt{data1}. The first statement needs to complete before the second statement because the second statement reuses the variable \texttt{data1}.

\begin{verbatim}
Listing 3.6  An Example of an Antidependency

void anti-dependency()
{
    result1 = calculation( data1 );  // Needs to complete first
    data1  = result2 + 1;           // Will overwrite data1
}
\end{verbatim}

If one of the statements was modified to use an alternative or temporary variable, for example, \texttt{data1_prime}, then both statements could proceed in any order. Listing 3.7 shows this modified code.

\begin{verbatim}
Listing 3.7  Fixing an Antidependency

void anti-dependency()
{
    data1_prime = data1;      // Local copy of data1
    result1 = calculation( data1_prime );
    data1  = result2 + 1;   // No longer has antidependence
}
\end{verbatim}

The code shown in Listing 3.8 demonstrates an output dependency on the variable \texttt{data1}. The second statement needs to complete after the first statement only because they both write to the same variable.

\begin{verbatim}
Listing 3.8  An Output Dependency

void output-dependency()
{
    data1  = result1 + 2;
    data1  = result2 + 2;  // Overwrites same variable
}
\end{verbatim}

If the first target variable was renamed \texttt{data1_prime}, then both statements could proceed in any order. Listing 3.9 shows this fix.
Listing 3.9  Fixing an Output Dependency

```c
void output-dependency()
{
    data1_prime = result1 + 2;
    data1     = result2 + 2; // No longer has output-dependence
}
```

What is important about these two situations is that both output and antidependencies can be avoided by *renaming* the data being written, so the final write operation goes to a different place. This might involve taking a copy of the object and having each task work on their own copy, or it might be a matter of duplicating a subset of the active variables. In the worst case, it could be resolved by both tasks working independently and then having a short bit of code that sets the variables to the correct state.

**Using Speculation to Break Dependencies**

In some instances, there is a clear potential dependency between different tasks. This dependency means it is impossible to use a traditional parallelization approach where the work is split between the two threads. Even in these situations, it can be possible to extract some parallelism at the expense of performing some unnecessary work. Consider the code shown in Listing 3.10.

Listing 3.10  Code with Potential for Speculative Execution

```c
void doWork( int x, int y )
{
    int value = longCalculation( x, y );
    if (value > threshold)
    {
        return value + secondLongCalculation( x, y );
    }
    else
    {
        return value;
    }
}
```

In this example, it is not known whether the second long calculation will be performed until the first one has completed. However, it would be possible to speculatively compute the value of the second long calculation at the same time as the first calculation is performed. Then depending on the return value, either discard the second value or use it. Listing 3.11 shows the resulting code parallelized using pseudoparallelization directives.
Listing 3.11 Speculatively Parallelized Code

```c
void doWork(int x, int y)
{
    int value1, value2;
    #pragma start parallel region
    {
        #pragma perform parallel task
        {
            value1 = longCalculation( x, y );
        }
        #pragma perform parallel task
        {
            value2 = secondLongCalculation( x, y );
        }
    }
    #pragma wait for parallel tasks to complete
    if (value1 > threshold)
    {
        return value1 + value2;
    }
    else
    {
        return value1;
    }
}
```

The #pragma directives in the previous code are very similar to those that are actually used in OpenMP, which we will discuss in Chapter 7, “OpenMP and Automatic Parallelization.” The first directive tells the compiler that the following block of code contains statements that will be executed in parallel. The two #pragma directives in the parallel region indicate the two tasks to be performed in parallel. A final directive indicates that the code cannot exit the parallel region until both tasks have completed.

Of course, it is important to consider whether the parallelization will slow performance down more than it will improve performance. There are two key reasons why the parallel implementation could be slower than the serial code.

- The overhead from performing the work and synchronizing after the work is close in magnitude to the time taken by the parallel code.
- The second long calculation takes longer than the first long calculation, and the results of it are rarely used.

It is possible to put together an approximate model of this situation. Suppose the first calculation takes $T_1$ seconds and the second calculation takes $T_2$ seconds; also suppose that the probability that the second calculation is actually needed is $P$. Then the total runtime for the serial code would be $T_1 + P \times T_2$. 
For the parallel code, assume that the calculations take the same time as they do in the serial case and the probability remains unchanged, but there is also an overhead from synchronization, S. Then the time taken by the parallel code is $S + \max(T_1, T_2)$.

Figure 3.22 shows the two situations.

We can further deconstruct this to identify the constraints on the two situations where the parallel version is faster than the serial version:

- If $T_1 > T_2$, then for the speculation to be profitable, $S + T_1 < T_1 + P \times T_2$, or $S < P \times T_2$. In other words, the synchronization cost needs to be less than the average amount of time contributed by the second calculation. This makes sense if the second calculation is rarely performed, because then the additional overhead of synchronization needed to speculatively calculate it must be very small.

- If $T_2 > T_1$ (as shown in Figure 3.21), then for speculation to be profitable, $S + T_2 < T_1 + P \times T_2$ or $P > (T_2 + S - T_1)/T_2$. This is a more complex result because the second task takes longer than the first task, so the speculation starts off with a longer runtime than the original serial code. Because $T_2 > T_1$, $T_2 + S - T_1$ is always $> 0$. $T_2 + S - T_1$ represents the overhead introduced by parallelization. For the parallel code to be profitable, this has to be lower than the cost contributed by executing $T_2$. Hence, the probability of executing $T_2$ has to be greater than the ratio of the additional cost to the original cost. As the additional cost introduced by the parallel code gets closer to the cost of executing $T_2$, then $T_2$ needs to be executed increasingly frequently in order to make the parallelization profitable.

The previous approach is *speculative execution*, and the results are thrown away if they are not needed. There is also *value speculation* where execution is performed, speculating on the value of the input. Consider the code shown in Listing 3.12.
Listing 3.12  **Code with Opportunity for Value Speculation**

```c
void doWork(int x, int y)
{
    int value = longCalculation( x, y );
    return secondLongCalculation( value );
}
```

In this instance, the second calculation depends on the value of the first calculation. If the value of the first calculation was predictable, then it might be profitable to speculate on the value of the first calculation and perform the two calculations in parallel. Listing 3.13 shows the code parallelized using value speculation and pseudoparallelization directives.

Listing 3.13  **Parallelization Using Value Speculations**

```c
void doWork(int x, int y)
{
    int value1, value2;
    static int last_value;
    #pragma start parallel region
    {
        #pragma perform parallel task
        {
            value1 = longCalculation( x, y );
        }
        #pragma perform parallel task
        {
            value2 = secondLongCalculation( lastValue );
        }
    }
    #pragma wait for parallel tasks to complete
    if (value1 == lastvalue)
    {
        return value2;
    }
    else
    {
        lastValue = value1;
        return secondLongCalculation( value1 );
    }
}
```

The value calculation for this speculation is very similar to the calculation performed for the speculative execution example. Once again, assume that T1 and T2 represent the
costs of the two routines. In this instance, P represents the probability that the speculation is incorrect. S represents the synchronization overheads. Figure 3.23 shows the costs of value speculation.

The original code takes $T_1 + T_2$ seconds to complete. The parallel code takes $\max(T_1, T_2) + S + P \times T_2$. For the parallelization to be profitable, one of the following conditions needs to be true:

- If $T_1 > T_2$, then for the speculation to be profitable, $T_1 + S + P \times T_2 < T_1 + T_2$. So, $S < (1-P) \times T_2$. If the speculation is mostly correct, the synchronization costs just need to be less than the costs of performing $T_2$. If the synchronization is often wrong, then the synchronization costs need to be much smaller than $T_2$ since $T_2$ will be frequently executed to correct the misspeculation.

- If $T_2 > T_1$, then for the speculation to be profitable, $T_2 + S + P \times T_2 < T_1 + T_2$. So, $S < T_1 - P \times T_2$. The synchronization costs need to be less than the cost of $T_1$ after the overhead of recomputing $T_2$ is included.

As can be seen from the preceding discussion, speculative computation can lead to a performance gain but can also lead to a slowdown; hence, care needs to be taken in using it only where it is appropriate and likely to provide a performance gain.

**Critical Paths**

One way of looking at parallelization is by examining the *critical paths* in the application. A critical path is the set of steps that determine the minimum time that the task can
complete in. A serial program might complete tasks A, B, C, and D. Not all of the tasks need to have dependencies. B might depend on the results of A, and D might depend on the results of B and C, but C might not depend on any previous results. This kind of data can be displayed on a graph such as the one in Figure 3.24.

It is relatively straightforward to identify the critical path in a process once the dependencies and durations have been identified. From this graph, it is apparent that task C could be performed in parallel with tasks A and B. Given timing data, it would be possible to estimate the expected performance of this parallelization strategy.

**Identifying Parallelization Opportunities**

The steps necessary to identify parallelization opportunities in codes are as follows:

1. Gather a representative runtime profile of the application, and identify the regions of code where the most time is currently being spent.

2. For these regions, examine the code for dependencies, and determine whether the dependencies can be broken so that the code can be performed either as multiple parallel tasks or as a loop over multiple parallel iterations. At this point, it may also be worth investigating whether a different algorithm or approach would give code that could be more easily made parallel.

3. Estimate the overheads and likely performance gains from this parallelization strategy. If the approach promises close to linear scaling with the number of threads, then it is probably a good approach; if the scaling does not look very efficient, it may be worth broadening the scope of the analysis.

4. Broaden the scope of the analysis by considering the routine that calls the region of interest. Is it possible to make this routine parallel?

The important point to remember is that parallelization incurs synchronization costs, so the more work that each thread performs before it needs synchronization, the better the code will scale. Consequently, it is always worth looking further up the call stack of a region of code to determine whether there is a more effective parallelization point. For example, consider the pseudocode shown in Listing 3.14.
In this example, there are two long-running tasks; each performs some manipulation of a packet of data. It is quite possible that the two tasks, `doOneTask()` and `doSecondTask()`, could be performed in parallel. However, that would introduce one synchronization point after every packet that is processed. So, the synchronization cost would be $O(N)$ where $N$ is the number of packets.

Looking further up the stack, the calling routine, `handleStream()`, iterates over a stream of packets. So, it would probably be more appropriate to explore whether this loop could be made to run in parallel. If this was successful, then there would be a synchronization point only after an entire stream of packets had been handled, which could represent a significant reduction in the total synchronization costs.

**Summary**

This chapter has discussed the various strategies that can be used to utilize systems more efficiently. These range from virtualization, which increases the productivity of the system through increasing the number of active applications, to the use of parallelization techniques that enable developers to improve the throughput or speed of applications.

It is important to be aware of how the amount of code that is made to run in parallel impacts the scaling of the application as the number of threads increases. Consideration of this will enable you to estimate the possible performance gains that might be attained from parallelization and determine what constraints need to be met for the parallelization to be profitable.

The chapter introduces various parallelization strategies, and these should provide you with insights into the appropriate strategy for the situations you encounter. Successful parallelization of applications requires identification of the dependencies present in code. This chapter demonstrates ways that the codes can be made parallel even in the presence of dependencies.
This chapter has focused on the strategies that might be employed in producing parallel applications. There is another aspect to this, and that is the handling of data in parallel applications. The individual threads need to coordinate work and share information. The appropriate method of sharing information or synchronizing will depend on the implementation of the parallelization strategy. The next chapter will discuss the various mechanisms that are available to support sharing data between threads and the ways that threads can be synchronized.
ABA problem, 329–332
ABI (application binary interface), 22
accept socket routine, 194
Access patterns for arrays, 58–59
Accessor functions, 40
Accessor patterns for cross-file optimization, 67
Acquire barriers, 302
acquire method, 129
AcquireSRWLockExclusive routine, 214
AcquireSRWLockShared routine, 214
Adaptive mutex locks, 157
Addition
atomic operations, 239
with mutex locks, 301
with reductions, 250, 261
vectors, 101–102
Addresses
sockets, 194
virtual, 16–18
Affinity
purpose, 8
setting, 376
Algorithmic complexity, 33
considerations, 38–39
examples, 33–37
importance, 37–38
Algorithms
limits, 350–352
lockless. See Lockless algorithms
Aliasing pointers, 61, 70–74
Alignment
caches, 12, 56, 359
loads, 316
memory segments, 183
Alternatives languages, 399–401
AMD64 instruction set, 23
Amdahl’s law, 94–98, 333
AND operations
atomic operations, 239
reductions, 250, 261
Anonymous pipes
POSIX threads, 186–187
Windows threads, 231, 233
Antidependencies, 111–113
Application binary interface (ABI), 22
Application scaling constraints, 333
algorithmic limit, 350–352
hot locks, 340–345
insufficient work, 347–350
library code, 345–347
serial code, 334–336
superlinear scaling, 336–337
workload imbalance, 338–339
Applications

build structure, 39–42
data structure, 53–60
execution paths, 61
isolating, 89
library structure, 42–53
memory maps, 49–50
processes, 26

Archive libraries, 41

Arguments for processes, 224–225

Arrays

access patterns, 58–59
incrementing values in, 254
searches in, 59–60
sums of numbers in, 250
thread-private data, 141–142

asm keyword, 296, 307–308

Associativity of caches, 13–14, 359–360

atomic_add routine, 309
atomic_add_float routine, 299, 301
atomic_add_int routine, 296–297
atomic directive, 283
atomic_inc_int routine, 296–297

Atomic operations

hand-coded synchronization. See
Hand-coded synchronization
increments in C and C++ proposals, 395–396
OpenMP, 283
producer-consumer systems, 326–328
synchronization, 130–131
transactional memory, 407
variable updates, 238–240

Attributes

mutex, 156–157
POSIX threads, 148–150

Automatic parallelization

codes containing calls, 251–253
compiler assistance, 254–256
overview, 245–250
reductions, 250–251

Availability, 32

B

Bandwidth

memory, 20–21, 354–355
MPI, 403
sharing between cores, 353–355

Bank with multiple branches, 340–345

barrier directive, 279–280

Barriers, 130

atomic operations, 301–303
memory, 25–26
POSIX threads, 162–163

_beginthread routine, 201–203, 238
_beginthreadex routine, 201–204, 206, 235, 238

Binary searches, 59

bind routine, 194

Binding

processors, 371–379
sockets, 194
threads, 362, 365–366, 373

Block literals, 394

blockDim structure, 386
blockIdx structure, 386

Blocking operations in MPI, 405

Blocking, threads, 128, 155

Blocks

GPU, 385
TBB, 386–389

Breaking dependencies, 113–117
BSD Sockets API, 234
Bubble sort, 35–37, 350
Buffers, circular
  atomic operations, 326–328
  overview, 315–318
  scaling, 318–326
Build structures, 39–42
Bypass stage in pipelines, 10–11

C
C and C++ feature proposals, 394–397
C++/CLI, 397–399
Cache coherence, 18
Cache coherent nonuniform memory architecture (ccNUMA), 19
Caches, 5–6, 9
  alignment, 12, 56
  cache line invalidations, 322–325
  conflicts and capacity, 359–363
  data structures, 53–56
  superlinear scaling, 336–337
  working with, 12–15
Callee trees, 412
Calls and calling convention, 22
  and automatic parallelization, 251–253
  library code, 45, 49–51
  stack-based, 23
Calls, wake-up, 136–137, 173–174
Capacity of caches, 359–363
Carets (^)
  block literals, 394
  handle variables, 398
CAS (compare and swap) operations
  atomic operations, 131, 297–301
  spin locks, 325–326
CAS routine, 298, 330–331

ccNUMA (cache coherent nonuniform memory architecture), 19
Chaining signal handlers, 190–191
Child threads
  C++/CLI, 397–398
  passing data to and from, 145–147
Chip multithreading (CMT), 6–8
  latency costs, 55
  memory bandwidth, 21
Chips. See Processors
Chrome browser, 29
Chunks in OpenMP schedules, 266
cilk_for routine, 389
Cilk++ language extensions, 389–392
cilk_main routine, 389
cilk_spawn routine, 390, 392
cilk_sync routine, 390
cilkview tool, 391
cilkscreen tool, 391–392
Circular buffers
  atomic operations, 326–328
  overview, 315–318
  scaling, 318–326
CISCs (complex instruction set computers), 22
CLI (Common Language Infrastructure), 397
Client/server systems
  communication, 140
  division of work in, 108–109
  OpenMP example, 270–273
  sockets, 194–197, 235–237
clock_gettime routine, 161
Clock speed
  historical increases, 3
  pipelines, 10
close socket routine, 194
closeHandle routine

- events, 220
- memory, 226
- mutex locks, 213
- pipes, 233
- semaphores, 216
- Windows sockets, 238
- Windows threads, 201–208

Cloud computing, 92, 103

CLR (Common Language Runtime), 397

Clustering technologies, 103, 402
- grids, 407
- MapReduce algorithm, 406
- MPI, 402–405

CMT (chip multithreading), 6–8
- latency costs, 55
- memory bandwidth, 21

CodeAnalyst tool, 75–77

Coherence, cache, 18

collapse clause, 286–287, 348

Column-major array order, 58

Common execution paths, 61

Common Language Infrastructure (CLI), 397

Common Language Runtime (CLR), 397

Communicating costs in MPI, 403

Compilation of source code
- 32-bit vs. 64-bit code, 23–24
- memory operation order, 24–26
- overview, 21–23
- processes vs. threads, 26–29

Compiler role in performance, 60–62
- cross-file optimization, 65–68
- optimization types, 62–64
- options, 64–65
- pointer aliasing, 70–74
- profile feedback, 68–70
- profiling, 74–80

Compilers and compiling
- automatically parallelizing code, 254
- lazy loading, 52–53
- libraries, 44–45
- memory-ordering directives, 303
- multithreaded code, 151–152
- operation ordering, 304–308
- POSIX threads flags, 197–198

Complex instruction set computers (CISCs), 22

Complexity, algorithmic, 33
- considerations, 38–39
- examples, 33–37
- importance, 37–38

Computational costs in OpenMP, 263–265

Compute Unified Device Architecture (CUDA), 383–384

Computer components, 1–2

Concurrent queues, 387–388

Condition variables, 135–137
- POSIX threads, 170–175
- Windows threads, 209, 218–219

Conditional code, 63–64

Conditional execution in OpenMP, 284

Configuration isolation, 91
Conflicts, cache, 359–363

connect socket routine, 196

Consolidation
  efficiency through, 88–92
  virtualization for, 92

Consumers. See Producer-consumer systems

Containers for isolating applications, 89

Contended mutexes, 127, 340–345

Context switches, 4

copyin clause, 275

copyprivate directive, 275–276

Cores
  bandwidth sharing between, 353–355
  interleaving, 365–366
  multicore. See Multicore processors
  pipelined, 9–12
  processor, 3

Costs
  development, 39
  libraries, 43–44, 47
  MPI, 403
  OpenMP, 263–265
  scaling, 98–100

Counters
  increment operations, 309–310
  for semaphores, 128

CPU_SET macro, 376

CPU_ZERO macro, 376

CPUs. See Processors

cputrack tool, 362

CreateEvent routine, 220, 235

CreateFileMapping routine, 225–226

CreateMutex routine, 213, 221, 229–231

CreateMutexA routine, 221

CreateMutexEx routine, 213

CreateMutexW routine, 221

CreateNamedPipe routine, 231–232

CreatePipe routine, 231, 233

CreateProcess routine, 222–223, 229

CreateProcessW routine, 223

CreateSemaphore routine, 216

CreateSemaphoreEx routine, 216

CreateThread routine, 199–201

critical directive, 282

Critical paths, 117–118

Critical sections
  mutex locks, 126–128
  OpenMP, 282–283
  Windows threads, 208, 210–213

Cross-file optimization, 40–42, 62, 65–68

CUDA (Compute Unified Device Architecture), 383–384

cudaMalloc routine, 385

cudaMemcpy routine, 385–386

dataMlock routine, 385

dataMunmap routine, 385–386

cudaMemcpy routine, 385–386

D

-D_REENTRANT flag, 151–152

Data-carried dependencies, 413

Data padding
  caches, 56
  for false sharing, 357

Data races, 295
  avoiding, 126
  CAS operations, 299
  detecting, 123–125
  mutex locks for, 154–155, 413
  overview, 121–123
  transactional memory, 407–408

Data sharing
  storing thread-private data, 141–142
  synchronization. See Synchronization

Data structures
  array access patterns, 58–59
  choosing, 59–60
Data structures (continued)
  density and locality, 55–57
  performance, 53–60
Data TLB (DTLB), 16
Deadlocks
  circular buffers, 320
  overview, 132–133
Debug level optimization, 64–65
  _declspec specifier, 240–241
Decode stage in pipelines, 9
  decrement atomic operations, 239
  Decrement routine, 127
Default size
  pages, 17
  stack, 149
Dekker’s algorithm, 312–315
Delays in spin code, 368–369
DeleteCriticalSection routine, 210
Deleting POSIX thread shared memory, 181
Density, data, 55–57
Dependencies
  antidependencies and output, 111–113
  breaking, 113–117
  compiling for, 52–53
  critical paths, 117–118
  determining, 413
  parallelism, 110–111
Design, performance by, 82–83
deructor function, 178
Detached threads, 147–148
Detecting data races, 123–125
Developer time and cost in algorithm complexities, 39
Device drivers in libraries, 51
Direct mapped caches, 13
Directives, 74, 256–257
dirent structure, 198
dispatch_apply routine, 393–394
dispatch_async routine, 393
Divide-and-conquer approach, 106
Division of work in client-server configuration, 108–109
Division with reductions, 250, 261
Domains, logical, 90
Doors, 141
down method, 129
Downtime, 32
DTLB (data TLB), 16
Dual-core processors, 5
Dynamic scheduling in OpenMP, 264–266
  example, 291–293
  impact, 286
Dynamically defined parallel tasks, 269–273
  E
  Echo threads
    POSIX, 194–195
    Windows sockets, 237–238
  Efficiency through consolidation, 88–92
  Empty loops, 62
  EMT64 instruction set, 23
    _endthread routine, 205
    _endthreadex routine, 205
  EnterCriticalSection routine, 211–212
  er_src tool, 259–260
  errno variable, 151–152, 193
  Error-handling code, time spent in, 75
  Events
    automatically reset, 220
    and signals, 137–138
    Windows sockets, 235
    Windows synchronization, 209, 219–221
  Exceptional conditions, time spent in, 75
  exec routine, 179–180
  execl routine, 179
Execute stage in pipelines, 10
Execution duration in algorithm complexities, 37–38
Execution order in OpenMP, 285–286
Execution paths, common, 61
ExitThread routine, 205
Experimentation, virtualization for, 91

Factorial sums, 34–35
False dependencies, 413
False sharing, 355–359, 380
-fansi_alias flag, 73
Feedback-directed optimization, 69
Fences, memory, 25–26, 393
Fetch stage in pipelines, 9–10
Fibonacci numbers, 399–401
FIFO (first-in, first-out) queues, 185
Filling, register, 23
Firefox, 86
First-in, first-out (FIFO) queues, 185
First-touch placement, 373
firstprivate clause, 262, 273
Flags, compiler, 64–65
Floating-point values
  incrementing, 299
  loops, 62–63
  pipelines, 11
  reductions, 250
flush directive, 287–288
-fno-inline-functions flag, 247
for loops in OpenMP, 258
Fork-Exec model, 179–180
Fork-join pattern
  OpenMP, 258
  task splitting, 106
fork routine, 179–180

Fortress language, 399
free routine, 345–347
free_spinlock routine, 302–303, 319
FreeBSD jails, 89
Freeing locks, 25
ftruncate routine, 180
Fully associative caches, 14
Function address tables, 44
Function calls in loops, 62
Future of parallelization, 416

gcc asm statement, 307–308
GCD (Grand Central Dispatch), 392–394
gcnew routine, 398
General-purpose registers, 23
getchar routine, 207, 224
GetCurrentThreadId routine, 200
GetThreadPriority routine, 243
GetTickCount routine, 376–377
gettid routine, 376
gmtimeofday routine, 375–376
Global indexes, 241
__global__ keyword, 386
Global variables in POSIX threads, 175–178
Go language, 399
GPU-based computing, 383–386
GPUs (graphics processing units), 383–384
Grand Central Dispatch (GCD), 392–394
Granularity, locking, 413–414
Graphics processing units (GPUs), 383–384
Grid computing, 103
Grid sorting method, 94
Grids for tasks, 407
Groups, locality, 8, 372–373
Guided schedules, 266
Hadoop, 406

Hand-coded synchronization, 295
  atomic memory operations, 295–297
  compare and swap, 297–301
  memory ordering, 301–303
  operating system-provided, 309–311
  operation ordering, 304–308
  overview, 295–297
  volatile variables, 308
  lockless. See Lockless algorithms

Handles
  kernel resources, 207–208
  processes, 224, 228–229
  Windows threads, 200–204

Hardware constraints to scaling, 352–353
  bandwidth sharing between cores, 353–355
  cache conflict and capacity, 359–363
  false sharing, 355–359
  pipeline resource starvation, 363–369

Hardware isolation, 91

Hardware prefetching, 55

Hardware threads, 4

Hardware transactional memory, 408

Hashing in hardware, 360

Haskell language, 399–401

Header files
  multithreaded code, 151–152
  Windows sockets, 234–235

Heap
  data sharing through, 175
  POSIX threads, 150

Helgrind tool, 124

Hierarchy, memory, 13

Hot mutex locks, 340–345

Hypervisors, 89–92

I

Identifying
  reductions, 250–251
  tasks, 411–412

IDs for Windows threads, 200

inc instruction, 296

Inclusive time, 412

Increment operations
  array values, 254
  atomic operations, 239
  C and C++ proposals, 395–396
  Dekker’s algorithm, 312–313
  floating-point values, 299
  mutex locks, 127
  variables, 21–22, 25

Incremental parallelization, 257

Independent tasks, 102–103

Infinite loops, 175

Inheriting handles in child processes, 228–229

InitializeConditionVariable routine, 218

InitializeCriticalSection routine, 210

InitializeCriticalSectionAndSpinCount routine, 212–213

Inlining
  accessor functions, 40
  compiler role, 62, 176
  cross-file optimization, 41, 66–69
  disabling, 247
  loops, 253

Instruction issue rate, pipelining for, 9–12

Instruction TLB (ITLB), 16

Instruments tool, 80

Insufficient work constraints, 347–350

Integer pipelines, 11

Integration
  OpenMP, 349–350
  trapezium rule, 348–349
Interleaving cores, 365–366
Interlocked functions, 238
InterlockedBitTestAndReset routine, 239
InterlockedBitTestAndSet routine, 239
InterlockedCompareExchange routine, 239
InterlockedExchangeAdd routine, 238–239, 309
InterlockedIncrement routine, 239
Inversion, priority, 244, 379–380
Isolating applications, 89
Items per unit time metric, 31
ITLB (instruction TLB), 16

J

Jails, 89
Joinable threads, 147–148

K

Kernel resources, handles to, 207–208
kill routine, 188

L

Language extensions, 386
    C and C++ feature proposals, 394–397
    Cilk++, 389–392
    GCD, 392–394
    Microsoft C++/CLI, 397–399
    TBB, 386–389
lastprivate clause, 262–263
Latency
    caches, 13
    CMT processors, 55
    memory, 18–21, 54, 373–379, 415
    metrics, 32
    page access, 18
    producer-consumer model, 109
    queuing, 355
    between threads, 99
Lazy loading, 47, 51–52
LD_DEBUG environment variable, 46–47
LeaveCriticalSection routine, 211
Levels, caches, 13–14
lfence, 303
lgrpinfo tool, 372
libfast library, 152
Libraries
    application structure, 42–53
    benefits, 42–43
    build process, 41–42
    calling code, 45, 51
    compiling, 44–45
    costs, 43–44, 47
    defining, 44
    guidelines, 50
    lazy loading, 47, 51–52
    linking, 47
    memory maps, 45–46
    multithreaded code, 151
    scaling code, 345–347
    stepping through calls, 49–50
    TBB, 386–389
Linkers, 47
listen socket routine, 194
Lists, circular
    atomic operations, 326–328
    overview, 315–318
    scaling, 318–326
Literals, block, 394
Livelocks, 132–133
Loading, lazy, 47, 51–52
Locality
    data, 55–57
    memory, 371–379
Locality groups, 8, 372–373

Lockless algorithms, 312

ABA problem, 329–332
atomic operations, 130–131, 300

circular buffers. See Circular buffers
Dekker’s algorithm, 312–315

Locks
freeing, 25
granularity, 413–414
mutex. See Mutexes and mutex locks
read-write, 159–162
readers-writer, 129
spin. See Spin locks
synchronization, 126

Logical domains, 90

Logical operations
for conditional code, 63–64
for reductions, 250, 261

Loops
algorithmic complexity, 34
arrays, 58
automatic parallelization, 246
collapsing, 286–287
empty, 62
floating-point arithmetic, 62–63
function calls, 62, 251–253
infinite, 175
merging, 347–348
OpenMP, 258, 278
potential compiler aliasing, 71
with reductions, 250–251
vector addition, 101–102
versioning, 255

Loosely coupled tasks, 103–105

Lost wake-up calls, 136–137, 173–174
-lpthread flag, 152

Main threads, 143

malloc routine
critical regions, 127–128
library code, 345–347
memory placement, 373

Mandelbrot set
cilk_for, 389–390
dependencies, 110
determining if a point is in the, 348
GCD, 393
loop-carried dependencies, 110
MPI, 403–404
OpenMP, 288–292
TBB, 388–390

Manually reset events, 219

MapReduce algorithm, 406

Maps, 134–135
memory to applications and libraries,
45–46, 49–50
memory to caches, 13–14
virtual CPU numbers to cores, 365
virtual memory to physical memory, 15

MapViewOfFile routine, 225–226

master directive, 279, 282

Master threads, 143
MPI, 404–405
OpenMP, 258
in regions of code, 282

Matrices, multiplying by vectors, 247–248

MAX operations, 250, 261

Maximum practical threads, 97–98

membar instructions, 25–26, 303, 314

Memory, 1–3
atomic operations, 295–308
bandwidth, 20–21, 354–355
caches. See Caches
communication through, 134–135
consistency, 287–288
hierarchy, 13
latency, 18–21, 54, 373–379, 415
locality, 8, 371–379
maps, 45–46, 49–50, 134–135
multiprocessor systems, 18–20
ordering, 24–26, 301–303
POSIX threads, 175–178, 180–183
sharing, 134–135, 180–183, 225–228
in superlinear scaling, 337
transactional, 407
virtual, 9, 15–18
Windows threads, 225–228

Memory barriers, 25–26
atomic operations, 301–303
circular buffers, 316
Dekker’s algorithm, 314–315
Memory-carried dependencies, 111, 413
MemoryBarrier macro, 303
memset routine, 353
Merging loops, 347–348
Message Passing Interface (MPI), 402–405
Messages
POSIX threads, 184–186
queues, 138
signals and events, 137–138
Metrics, performance, 31–32
mfence operations, 25–26, 303, 314
Microsoft C++/CLI, 397–399
Microsoft Windows threads. See Windows threads
Migration of threads, 371–372
MIN operations, 250, 261
Mispredicted branches, 10–11
Miss rates in TLB, 17
mknod routine, 187–188
mmap routine, 180
Motherboards, 1
Mozilla Firefox, 86
MPI (Message Passing Interface), 402–405
MPI_Comm_rank routine, 402
MPI_Comm_size routine, 402
MPI_Finalize routine, 402
MPI_Init routine, 402
MPI_Recv routine, 403–404
MPI_Send routine, 403
mq_attr structure, 184
mq_close routine, 184
mq_open routine, 184
mq_receive routine, 185
mq_reltimedreceive_np routine, 185
mq_reltimedsend_np routine, 185
mq_send routine, 185
mq_timedreceive routine, 185
mq_timedsend routine, 185
mq_unlink routine, 184
mtx_init routine, 396
Multicore processors, 414–415
caches, 12–15
instruction issue rate, 9–12
motivation, 3–4
multiple thread support, 4–9
optimizing programs, 415–416
scaling, 380–381
virtual addresses, 16–18
virtual memory, 15–16
Multiple barriers, 130
Multiple-reader locks, 129
Multiple tasks
copies, 105–106
independent, 102–103
loosely coupled, 103–105
Multiple users on single systems, 87–88

Multiplication
matrices by vectors, 247–248
with reductions, 250, 261

Multiprocessor systems
characteristics, 18–20
latency and bandwidth, 20–21
POSIX threads, 179–193
for productivity, 85–87

Multithreaded code, compiling, 151–152
munmap routine, 180

Mutexes and mutex locks
addition of values, 301
atomic operations, 301–303, 310
attributes, 156–157
in C and C++, 396–397
condition variables, 135–137
contended, 340–345
critical regions, 126–128
data races, 126
OpenMP, 283–284
for ordering, 398–399
POSIX threads, 154–157
scaling limitations, 413
semaphores as, 165
vs. spin locks, 128
Windows threads, 208, 213–214, 229–231

Mutual exclusion
circular buffers, 319–322
Dekker’s algorithm for, 312–313
queue access, 167

Named pipes, 139
POSIX threads, 186–188
Windows threads, 231–232

Named semaphores, 164–165
Native Windows threads, 199–204

Nested loops
algorithmic complexity, 34
memory access, 58

Nested parallelism, 268–269, 273
Network stack, 139–140

no-op instruction, 369
Nodes in MPI, 402
Noncontiguous memory access patterns, 58
Noncritical code, time spent in, 75
now routine, 353, 356–357, 370, 373–375, 377
nowait clause, 279
num_threads clause, 277
numactl tool, 372
Number of threads in OpenMP, 276–277
Numerical integration
OpenMP, 349–350
trapezium rule, 348–349

O

O_CREAT flag, 164, 180, 184, 186
O_EXCL flag, 164, 180, 184
O_NONBLOCK flag, 185
O_RDONLY flag, 180, 184
O_RDWR flag, 180, 184
O_RDWR flag, 180, 184
Odd-even sort, 350–352
omp_destroy_lock routine, 283
OMP_DYNAMIC environment variable, 277
omp_get_dynamic routine, 277
omp_get_max_threads routine, 276
omp_get_nested routine, 268

N

Named critical sections, 282
Named mutexes, 229–231
omp_get_schedule routine, 278
omp_get_thread_limit routine, 277
omp_get_thread_num routine, 276
omp_init_lock routine, 283
omp_lock_t type, 283
OMP_NUM_THREADS environment variable, 247, 258–259, 276
omp parallel directive, 273
omp_sched_auto routine, 278
omp_sched_dynamic routine, 278
omp_sched_guided routine, 278
omp_sched_static routine, 278
OMP_SCHEDULE environment variable, 278
omp_set_dynamic routine, 277
omp_set_lock routine, 283
omp_set_nested routine, 268–270
omp_set_num_threads routine, 276
omp_set_schedule routine, 278
omp single directive, 273
OMP_STACKSIZE environment variable, 278
omp task directive, 273
omp_test_lock routine, 283
OMP_THREAD_LIMIT environment variable, 277
omp_unset_lock routine, 283
OoO (out-of-order) execution, 20
Open Computing Language (OpenCL), 383–384
open routine, 187
OpenEvent routine, 220
OpenMP API, 245, 256–257
collapse clause, 348
collapsing loops, 286–287, 348
dynamically defined parallel tasks, 269–273
dissample, 288–293
example, 288–293
execution order, 285–286
memory consistency, 287–288
nested parallelism, 268–269
numerical integration, 349–350
parallel sections, 267–268
parallelizing loops, 258
parallelizing reductions, 260–261
private data, 274–276
QuickSort using, 350–352
restricting threads, 281–282
runtime behavior, 258
runtime environment, 276–278
thread restriction in, 281–284
waiting for work to complete, 278–281
work distribution scheduling, 263–267
OpenMP scheduling modes, 266–267
OpenMutex routine, 229
OpenSemaphore routine, 216
Operating system constraints to scaling
oversubscription, 369–371
priority inversion, 379–380
processor binding, 371–379
Operating system-provided atomics, 309–311
Operating systems, hypervisors for, 89–92
Operation count in algorithm complexities, 39
Operation ordering, 304–308
operator routine, 388–389, 397
oprofile tool, 75
OR operations for reductions, 250, 261
Order of N computations (O(N)), 34
ordered directive, 285–286
Ordering
execution, 285–286
memory, 24–26, 301–303
mutexes for, 398–399
operations, 304–308
POSIX threads, 165–166
OSMemoryBarrier macro, 303
Out-of-order (OoO) execution, 20, 54–55
Output dependencies, 111–113
output routine, 390
output-dependency routine, 112–113
OverFileMapping routine, 225–226
Oversubscription, 369–371

Padding
caches, 56
for false sharing, 357

Paging from disk, 15–18
-par-report flag, 247
-par-threshold flag, 247, 256
parallel directive, 284
-parallel flag, 247
parallel for directive, 279, 286
parallel_for routine, 388
Parallel sections, 267–268
parallel sections directive, 268, 279

Parallelism
and algorithm choice, 93–94
Amdahl’s law, 94–96
automatic. See Automatic parallelization through consolidation, 88–92
dependencies, 110–118
in Haskell, 401
maximum practical threads, 97–98
multiple processes, 85–87
multiple users on single system, 87–88
opportunities, 118–119
patterns. See Patterns in parallelization for single task performance, 92–100
synchronization costs, 98–100
visualizing, 92–93
Parallels software, 90

Passing
data to and from POSIX child threads, 145–147
values by pointer, 254–255

Patterns in parallelization, 100–101
client–server configuration, 108–109
combining strategies, 109–110
data parallelism using SIMD instructions, 101–102
multiple copies of same task, 105–106
multiple independent tasks, 102–103
multiple loosely coupled tasks, 103–105
pipelines, 106–108
processes and threads, 102
producer-consumer model, 109
split tasks, 106

pause instruction, 369

Performance
algorithmic complexity, 33–39
application structure. See Applications
compiler role. See Compiler role in performance
defining, 31–33
by design, 82–83
gain estimates, 412
optimization guidelines, 80–82

Peripherals, 3

Physical addresses, translating virtual addresses to, 16–18
pipe routine, 186

Pipelines
disadvantages, 107–108
resource starvation, 363–369
tasks, 106–108

Pipes
named, 139
POSIX threads, 186–188
Windows threads, 231–234
plockstat tool, 343
PLTs (procedure linkage tables), 48–49
pmap utility, 45
Pointers
  64-bit, 24
  aliasing, 61, 70–74
  restrict-qualified, 249, 254–255
POSIX threads, 123, 143
  attributes, 148–150
  barriers, 162–163
  compiling multithreaded code, 151–152
  concurrent queues, 387–388
  condition variables, 170–175
  creating, 143–144
  detached, 147
  memory, 175–178, 180–183
  message queues, 184–186
  multiprocess programming, 179–193
  mutex attributes, 156–157
  mutex locks, 154–156
  passing data to and from child threads, 145–147
  pipes, 186–188
  process termination, 153–154
  read-write locks, 159–162
  reentrant code and compiler flags, 197–198
  semaphores, 163–170, 183
  signals, 188–193
  sockets, 193–197
  spin locks, 157–159
  termination, 144–145
  variables, 175–178
post method, 129
#pragma directives, 114
#pragma omp directive, 256–257
#pragma omp critical directive, 282
#pragma omp parallel directive, 268
#pragma omp section directive, 268
Pragmas, 74
Prefetching, 55
Prime number testing, 209–210, 239–240
printf routine
  safety of, 189
  wide strings, 222
Printing
  signals for, 189–190
  stack addresses, 359–360
  Windows threads for, 204–205
Priorities
  inversion, 244, 379–380
  Windows threads, 242–244
Private data in OpenMP, 259–263, 274–276
Procedure linkage tables (PLTs), 48–49
PROCESS_INFORMATION structure, 222–223
Processes
  communication with threads. See Communication between threads and processes
  creating, 179–180, 222–225
  inheriting handles, 228–229
  memory sharing, 225–228
  multiple. See Multiprocessor systems
  mutexes, 229–231
  pipes, 231–234
  sockets, 234–238
  termination, 153–154
  vs. threads, 26–29
  Windows threads. See Windows threads
processor_bind routine, 362, 365, 375
Processors, 1–3
  binding, 371–379
  multicore. See Multicore processors
Producer-consumer systems
atomics, 326–328
with circular buffers, 315–318
concurrent queues, 387–388
condition variables, 135–137
overview, 109
scaling, 318–326
semaphores, 168–169
Profiling
feedback, 68–70
importance, 74–75
performance gain estimates, 412
tools, 75–80
Protocol families for sockets, 194
pthread_attr_destroy routine, 148
pthread_barrier_destroy routine, 162
pthread_barrier_init routine, 162
pthread_barrier_wait routine, 162
pthread_cond_broadcast routine, 172
pthread_cond_destroy routine, 170
pthread_cond_init routine, 170
pthread_cond_signal routine, 172
pthread_cond_timedwait routine, 174–175
pthread_cond_wait routine, 173
pthread_create routine, 143–145, 147–149
pthread_detach routine, 147, 194
pthread_exit routine, 145, 153
pthread_getspecific routine, 177
pthread_join routine, 144–147, 194, 305
pthread_key_create routine, 177–178
pthread_key_delete routine, 177
pthread_mutex_attr_destroy routine, 157
pthread_mutex_destroy routine, 154
pthread_mutex_init routine, 154, 156–157
pthread_mutex_lock routine, 155
pthread_mutex_setpshared routine, 156
pthread_mutex_trylock routine, 155
pthread_mutex_unlock routine, 155
pthread_mutexattr_init routine, 156–157
pthread_mutexattr_t structure, 156
pthread_rwlock_destroy routine, 160
pthread_rwlock_rdlock routine, 160
pthread_rwlock_rdunlock routine, 160
pthread_rwlock_timedrdlock routine, 161
pthread_rwlock_timedrdlock_np routine, 161
pthread_rwlock_timedwrlock routine, 161
pthread_rwlock_timedwrlock_np routine, 161
pthread_rwlock_tryrwlock routine, 161
pthread_rwlock_trywrlock routine, 161
pthread_rwlock_wlock routine, 160
pthread_rwlock_wrlock routine, 160
pthread_rwlock_wrunlock routine, 160
pthread_rwlockattr_destroy routine, 160
pthread_rwlockattr_init routine, 159
pthread_rwlockattr_setpshared routine, 159
pthread_self routine, 147
pthread_setspecific routine, 177
pthread_spin_destroy routine, 157
pthread_spin_init routine, 157
pthread_spin_lock routine, 157
pthread_spin_trylock routine, 158
pthread_spin_unlock routine, 157
PTHREAD_STACK_MIN variable, 150
pthread_t structure, 143–144

Quality of service (QoS) metric, 32
Queues
concurrent, 387–388
latencies, 355
messages, 138, 184–186
Quicksort
algorithmic complexity, 35–37
Cilk++, 390–392
OpenMP, 350–352
Read routine, 194
Read task dependencies, 111
Read-write locks, 159–162
_ReadBarrier routine, 308
readdir routine, 197–198
readdir_r routine, 198
Readers-writer locks
overview, 129
  Windows threads, 214–216
ReadFile routine, 232
ReadWriteBarrier routine, 308
recv routine, 194
Reduced instruction set computers (RISCs), 22
reduction clause, 261
Reductions
  identifying and parallelizing, 250–251
  MapReduce for, 406
  OpenMP, 282–283
_REENTRANT flag, 197
Reentrant code
  POSIX threads, 197–198
  strength, 63
References, 417–418
Regions, critical. See Critical sections
Registers
  spilling and filling, 23
tick, 368–369
Relative timeouts, 162
Release barriers, 302
release method, 129
ReleaseMutex routine, 213, 399
ReleaseSemaphore routine, 217
ReleaseSRWLockExclusive routine, 214
ReleaseSRWLockShared routine, 214
Reloading variables, 176, 304–305, 308
Replication, virtualization for, 91
ResetEvent routine, 220
Resources
  handles, 207–208
  sharing. See Synchronization starvation in pipelines, 363–369
Response time metrics, 32
restrict keyword, 74, 254–255
Restrict-qualified pointers, 249, 254–255
Restrictions
  regions of code threads, 281–284
  virtualization for, 91
ResumeThread routine, 207
Retire stage in pipelines, 10
RISCs (reduced instruction set computers), 22
Robustness, hypervisors for, 90
Row-major array order, 58
Runtime environment in OpenMP, 276–278
  behavior, 258
  scheduling modes, 266–267

S
SA_SIGINFO flag, 193
Saturated memory chips, 355
Scalability in cilkview, 392
Scaling
  algorithm complexities, 39
  Amdahl’s law, 94–96
  applications. See Application scaling constraints
collapsing loops for, 287
dynamically scheduled code, 293
hardware. See Hardware constraints to scaling
MapReduce for, 406
multicore processors, 380–381
Scaling (continued)
mutex locks, 413
operating system constraints. See
Operating system constraints to scaling
producer-consumer systems, 318–326
sockets for, 193
synchronization costs, 98–100
task identification for, 411–412
virtualization for, 91–92
sched_setaffinity routine, 376
schedule clause, 265, 267
schedules, OpenMP
example, 291–293
impact, 286
runtime loops, 278
work distribution, 263–267
Scoping in OpenMP, 259–263
Searches, 59–60
sections directive, 268
Security, hypervisors for, 90
SECURITY_ATTRIBUTES structure, 223, 229
sem_close routine, 164
sem_destroy routine, 163
sem_getvalue routine, 165
sem_init routine, 163
sem_open routine, 164
sem_post routine, 165
sem_trywait routine, 165
sem_unlink routine, 164
sem_wait routine, 165, 170
SemaphoreCreate routine, 216
SemaphoreCreateEx routine, 216
Semaphores, 128–129
POSIX threads, 163–170, 183
Windows threads, 208–209, 216–218
send routine, 194
Serial code
in Haskell, 400
performance, 31, 334–336
Serializing programs, 128
Servers. See Client/server systems
SetCriticalSectionSpinCount routine, 212–213
setdata routine, 241
SetEvent routine, 220
SetThreadAffinityMask routine, 376
SetThreadPriority routine, 243
sfence, 303
Shared variables
C and C++, 396–397
OpenMP, 259–260
Sharing
between cores, 353–355
false, 355–359, 380
heap for, 175
memory, 134–135, 180–183, 225–228
between POSIX threads. See POSIX threads
resources. See Synchronization
Shark tool, 78
shm_open routine, 180–181
shm_unlink routine, 181
Side effects from static libraries, 42
sigaction routine, 190, 192
siginfo_t data, 193
SIGKILL signal, 137, 188
signal routine
POSIX threads, 188
semaphores, 129
signalHandler routine, 138
Signals
and events, 137–138
POSIX threads, 188–193
semaphores, 128, 165
Windows threads, 219–221
SIGPROF signal, 190
sigqueue routine, 192–193
SIGRTMAX signal, 189
SIGRTMIN signal, 189, 193
SIMD (single instruction multiple data) instructions
  data parallelism using, 101–102
  vectorization, 408
sin routine, 252
single directive, 275, 279–282
Single instruction multiple data (SIMD) instructions
  data parallelism using, 101–102
  vectorization, 408
Single instruction single data (SISD) instructions, 408
64-bit code performance, 23–24
Size
caches, 13
page, 17
stack, 149, 278
sleep command, 179
SleepConditionVariableCS routine, 218
SleepConditionVariableSRW routine, 218
Slim reader/writer locks, 208, 214–216
Snooping for cache coherence, 18
sockaddr_in structure, 269
socket routine, 194
Sockets,
  POSIX threads, 193–197
  setting up, 140
  Windows threads, 234–238
Software prefetching, 55
Software threads, 4
Software transactional memory, 408
Solaris operating system
doors, 141
locality groups, 8
zones, 89
Solaris Studio Performance Analyzer, 76, 78–79, 291
Sorting
  algorithmic complexity, 35–37
  Cilk++, 390–392
  OpenMP, 350–352
  parallelism for, 93–94
Source code
  build structure trade-offs, 39–42
  translation to assembly language. See Translating source code to assembly language
Source-level profiles, 77
SPARC architecture, 3
  assembly language, 21–22
  memory barriers, 25
  page size, 17
SPEC Java Application Server benchmark, 31
Speculation to break dependencies, 113–117
Speculative execution, 115
Spilling, register, 23
Spin locks, 128
  with barriers, 302–303
  using CAS, 298–299
  circular buffers, 319–326
  POSIX threads, 157–159
spin routine, 356–358, 367–368, 370
Spinning threads, 366–369
Split tasks, 106
Splitting structures, 57
Stack, network, 139–140
Stack addresses, printing, 359–360
Stack-based calling convention, 23
Stack-based data, 141
Stack size
default, 149
OpenMP worker threads, 278
Start method, 398
STARTUPINFO structure, 222–223
Static libraries
build process, 41–42
side effects, 42
Static scheduling, 263–265
__stdcall calling convention, 201
Stepping through library calls, 49–50
Storing thread-private data, 141–142
Strands, 4
Strategies in Haskell, 401
Strength reduction, 63
String handling, 221–222
strlen routine, 353
Structure of applications
build, 39–42
libraries, 42–53
Structures
64-bit, 24
data. See Data structures
passing, 254–255
Studio Performance Analyzer, 76, 78–79, 291
Subtraction with reductions, 250, 261
Super-scalar execution, 11
Superlinear scaling, 336–337
suspended Windows threads, 207
SuspendThread routine, 207
__sync_fetch__ routine, 309
Synchronization, 121
atomic operations, 130–131
barriers, 130
communication. See Communication between threads and processes
critical regions, 126–128
data races, 121–126
deadlocks and livelocks, 132–133
hand-coded. See Hand-coded synchronization
multicore processors, 380–381
primitives, 126–131
readers-writer locks, 129
scaling costs, 98–100
semaphores, 128–129
spin locks, 128
Windows threads. See Windows threads
System-wide profiling, 75

task directive, 269
Tasks
dynamically defined parallel, 269–273
identifying, 411–412
multiple copies, 105–106
multiple loosely coupled, 103–105
pipelines, 106–108
split, 106
taskwait directive, 280–281
TBB (Threading Building Blocks) library, 386–389
TerminateThread routine, 205
Termination
POSIX threads, 144–145
processes, 153–154
Windows threads, 204–206
TEXT macro, 221–222
Thrashing, 359
Thread Analyzer, 124–125
Thread code routine, 144, 153–154
Thread-local data
  allocating, 240–242
  arrays for, 141
  declaring, 142, 177
Thread object, 398
Thread-private data, 141–142, 176
thread-safe malloc routines, 127–128
__thread specifier, 142, 177
threadbind routine, 373, 376–377
Threading Building Blocks (TBB) library, 386–389
threadprivate directive, 274–275
Threads
  binding, 362, 365–366, 373
  C and C++ proposals, 394–395
  C++/CLI, 397–398
  communication with processes. See Communication between threads and processes
defined, 4
  maximum practical, 97–98
  migration, 371–372
  oversubscription, 369–371
  POSIX. See POSIX threads vs. processes, 26–29
  spinning, 366–369
  support for, 4–9
  tasks split over, 106
  Windows. See Windows threads
ThreadStart object, 398–399
Thundering herd problem, 323
Tick registers, 368–369
Time per item metric, 32
Timeouts
  POSIX thread condition variables, 174–175
  read–write locks with, 161–162
  TLBs (translation look-aside buffers), 9, 16–17, 27–28
  TlsAlloc routine, 241
  TlsGetValue routine, 241
  TlsSetValue routine, 241
_tprintf routine, 222
Transactional memory, 407
Transactions per second metric, 31
Translating source code to assembly language, 21–23
  memory ordering, 24–26
  performance of 32-bit vs. 64-bit code, 23–24
  processes vs. threads, 26–29
Translating virtual addresses to physical addresses, 16–18
Translation look-aside buffers (TLBs), 9, 16–17, 27–28
Trapezium rule, 348–349
True dependencies, 413
TryAcquireSRWLockExclusive routine, 216
TryAcquireSRWLockShared routine, 216
TryEnterCriticalSection routine, 212
Type 1 hypervisors, 90
Type 2 hypervisors, 90–92

U

uintptr_t type, 201
ulimit command, 150, 278
UltraSPARC T2 processors
  floorplan, 6–8
  pipelines, 10–11
UMA (uniform memory architecture), 19
Unicode
  processes, 223
  wide string handling, 221–222
Uniform memory architecture (UMA), 19
unlink routine, 187
UnmapViewOfFile routine, 226
Unnamed semaphores, 163
up method, 129
Updates
atomic, 238–240
data races, 122
UTF-16 format, 221
POSIX threads, 175–176
reordering operations, 304–305
VTune tool, 75, 78–79

Wait-free implementation, 131
wait method for semaphore, 129
WaitForMultipleObjects routine, 205–206
WaitForSingleObject routine
    events, 220
    mutex locks, 213
    processes, 224
    semaphores, 216
    Windows threads, 202–203, 205
waiting for work to complete, 278–281
WaitOne routine, 399
waitpid routine, 183
WaitForMultipleObjects routine
    events, 220
    mutex locks, 213
    processes, 224
    semaphores, 216
    Windows threads, 202–203, 205
waiting for work to complete, 278–281
WaitOne routine, 399

Valgrind tool, 124
Valgrind tool, 124
Value speculation, 115–117
Vectorization, 408–409
Vectors
    adding, 101–102
double-precision, 246
    multiplying matrices by, 247–248
Versions
    ABA problem, 329–330
    loops, 255
Virtual CPUs, 4
    ABA problem, 329
    mapping to cores, 365
Virtual memory
    address translation to physical
    addresses, 16–18
    benefits, 15–16
    TLBs for, 9
VirtualBox software, 90
Virtualization benefits, 90–92
Visualizing parallel applications, 92–93
VMware software, 90
volatile keyword and variables
    atomic operations, 296, 308
    CAS operations, 298
    Dekker’s algorithm, 313
    atomic updates of variables, 238–240
    creating, 199–204
    kernel resources, 207–208
    priorities, 242–244
    processes. See Processes
    suspended, 207
<table>
<thead>
<tr>
<th>Elements</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronizing</td>
<td>208–209</td>
</tr>
<tr>
<td>Condition variables</td>
<td>218–219</td>
</tr>
<tr>
<td>Critical regions</td>
<td>210–213</td>
</tr>
<tr>
<td>Example</td>
<td>209–210</td>
</tr>
<tr>
<td>Mutex locks</td>
<td>213–214</td>
</tr>
<tr>
<td>Semaphores</td>
<td>216–218</td>
</tr>
<tr>
<td>Signals</td>
<td>219–221</td>
</tr>
<tr>
<td>Slim reader/writer locks</td>
<td>214–216</td>
</tr>
<tr>
<td>Terminating</td>
<td>204–206</td>
</tr>
<tr>
<td>Thread-local variables</td>
<td>240–242</td>
</tr>
<tr>
<td>Wide string handling</td>
<td>221–222</td>
</tr>
<tr>
<td>Work distribution scheduling</td>
<td>263–267</td>
</tr>
<tr>
<td>Worker threads</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>404–405</td>
</tr>
<tr>
<td>OpenMP</td>
<td>258</td>
</tr>
<tr>
<td>Stack size</td>
<td>278</td>
</tr>
<tr>
<td>Workload balance</td>
<td></td>
</tr>
<tr>
<td>Application scaling constraints</td>
<td>338–339</td>
</tr>
<tr>
<td>Collapsing loops for</td>
<td>286–287</td>
</tr>
<tr>
<td>Wprintf routine</td>
<td>222</td>
</tr>
<tr>
<td>Write routine</td>
<td>189, 194</td>
</tr>
<tr>
<td>Write task dependencies</td>
<td>111</td>
</tr>
<tr>
<td>_WriteBarrier routine</td>
<td>308</td>
</tr>
<tr>
<td>WriteFile routine</td>
<td>232, 234</td>
</tr>
<tr>
<td>WriteLine routine</td>
<td>397</td>
</tr>
<tr>
<td>WSACleanup routine</td>
<td>235</td>
</tr>
<tr>
<td>WSADATA structure</td>
<td>235</td>
</tr>
<tr>
<td>WSAStartup routine</td>
<td>235</td>
</tr>
<tr>
<td>X64 instruction set</td>
<td>23</td>
</tr>
<tr>
<td>X86-64 instruction set</td>
<td>23</td>
</tr>
<tr>
<td>Xadd instruction</td>
<td>296–297</td>
</tr>
<tr>
<td>-xalias_level flag</td>
<td>73</td>
</tr>
<tr>
<td>-xautopar flag</td>
<td>246</td>
</tr>
<tr>
<td>-xbuiltin flag</td>
<td>252, 256</td>
</tr>
<tr>
<td>-xloopinfo flag</td>
<td>246, 259</td>
</tr>
<tr>
<td>-xopenmp flag</td>
<td>259</td>
</tr>
<tr>
<td>Xor operation for atomic operations</td>
<td>239</td>
</tr>
<tr>
<td>-xreduction flag</td>
<td>251, 256</td>
</tr>
<tr>
<td>-xrestrict flag</td>
<td>73</td>
</tr>
<tr>
<td>Yieldprocessor macro</td>
<td>369</td>
</tr>
<tr>
<td>Zero-sum performance view</td>
<td>69</td>
</tr>
<tr>
<td>Zones</td>
<td>89</td>
</tr>
</tbody>
</table>