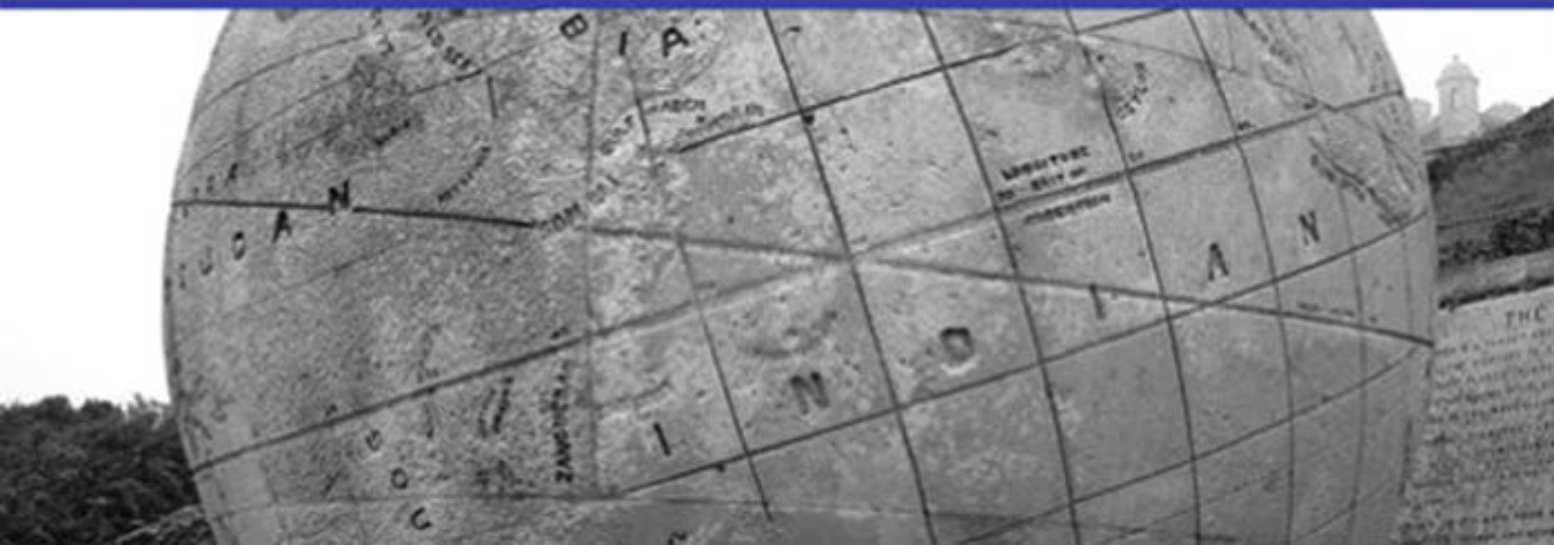
Darryl Gove

Multicore Application Programming

For Windows, Linux, and Oracle® Solaris

Developer's Library



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.
ISBN 978-0-321-71137-3 (pbk. : alk. paper)
1. Parallel programming (Computer science) I. Title.
QA76.642.G68 2011
005.2'75–dc22

2010033284

ISBN-13: 978-0-321-71137-3
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

Editor-in-Chief
Mark Taub

Acquisitions Editor
Greg Doench

Managing Editor
John Fuller

Project Editor
Anna Popick

Copy Editor
Kim Wimpsett

Indexer
Ted Laux

Proofreader
Lori Newhouse

Editorial Assistant
Michelle Housley

Cover Designer
Gary Adair

Cover Photograph
Jenny Gove

Compositor
Rob Mauhar

Contents at a Glance

This page intentionally left blank

Contents

Preface

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.

3

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.

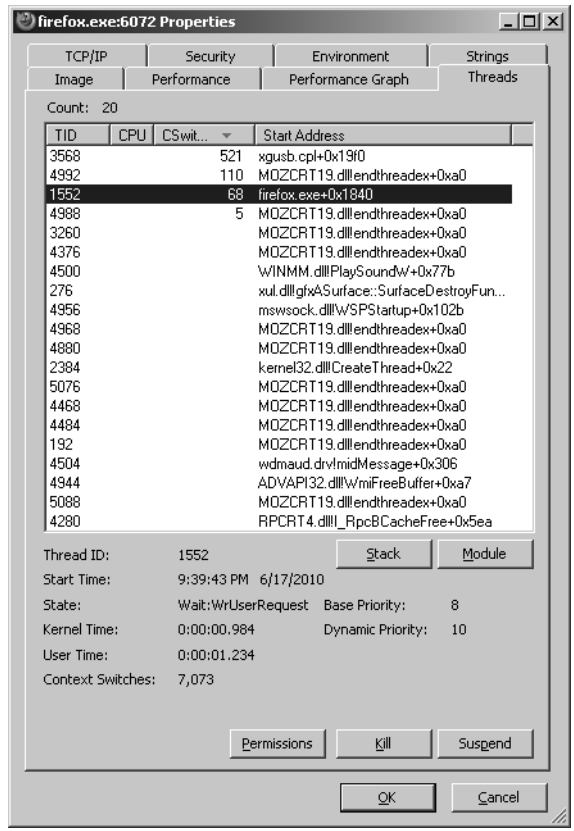


Figure 3.1 Windows Process Explorer showing thread activity in Mozilla Firefox

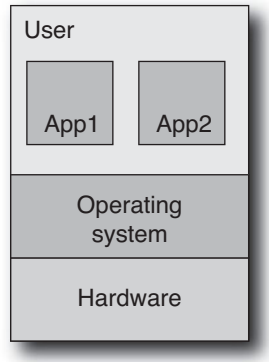


Figure 3.2 Single user on system

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.

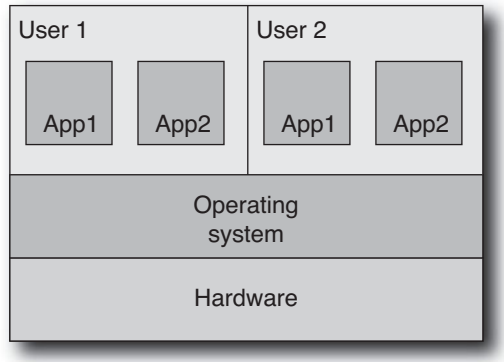


Figure 3.3 A single system supporting multiple users

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.

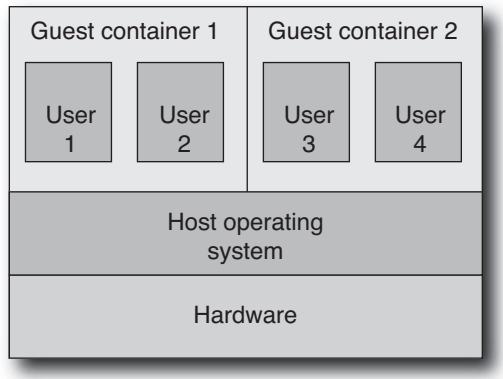


Figure 3.4 Using containers to host multiple guest operating systems in one system

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.

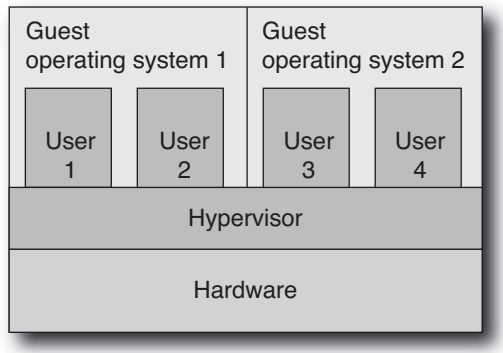


Figure 3.5 Type 1 hypervisor

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.

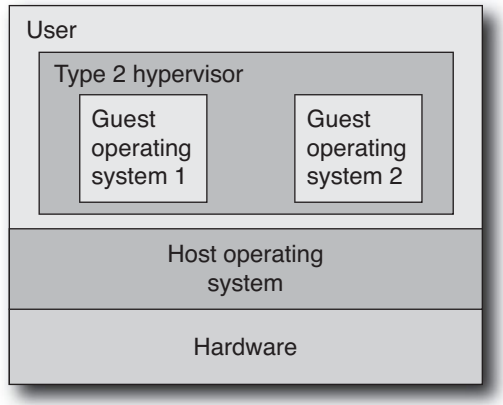


Figure 3.6 Type 2 hypervisor

- **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 multithreaded 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*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*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:

$$\text{Runtime} = S + \frac{P}{N}$$

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

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.

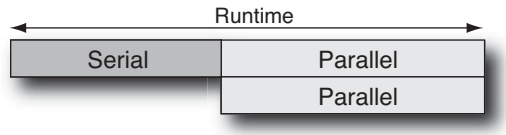


Figure 3.8 Runtime with two threads

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

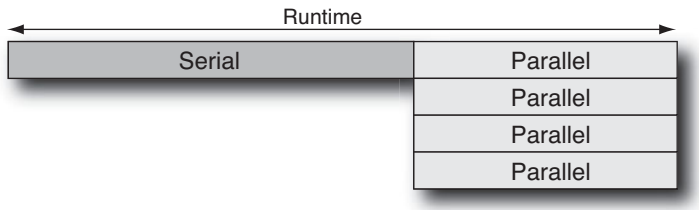


Figure 3.9 Runtime with four threads

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.

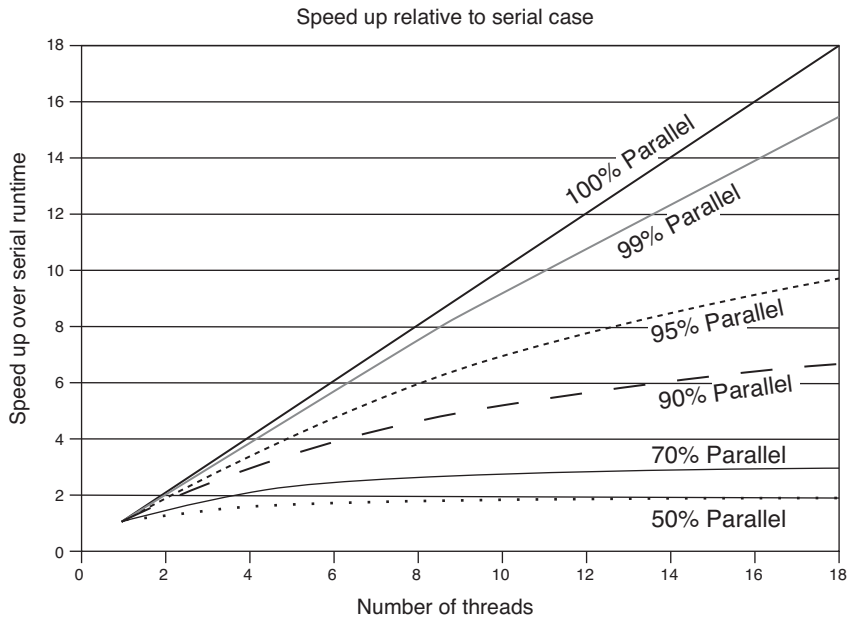


Figure 3.10 Scaling with diminishing parallel regions

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:

$$\text{Acceptable runtime} = S * (1 + T)$$

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

$$S * (1 + T) = \left(S + \frac{P}{N} \right)$$

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

$$N = \frac{P}{ST} = \frac{P}{(1 - P)T}$$

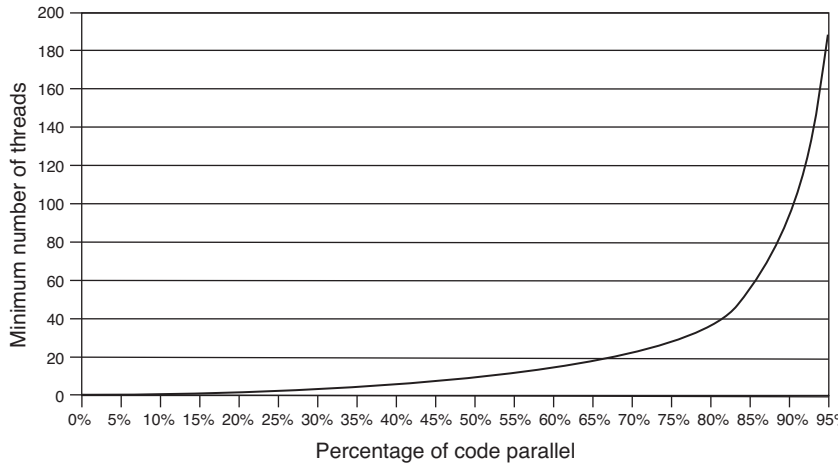


Figure 3.11 Minimum number of threads required to get 90% of peak performance

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*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.

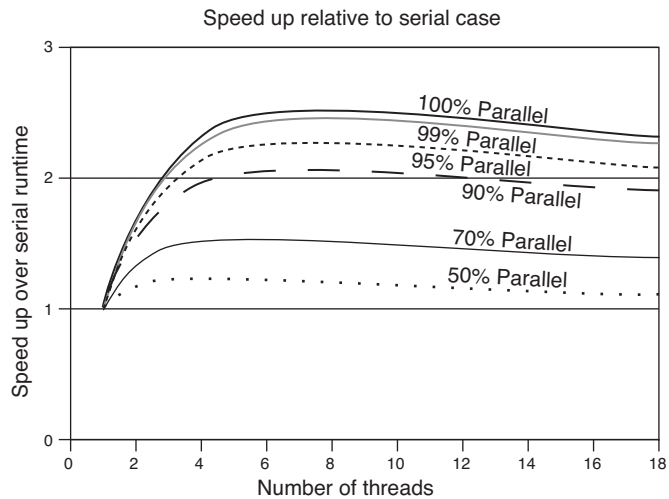


Figure 3.12 Scaling with exaggerated synchronization overheads

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 cooperating 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**

```
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**

```
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.

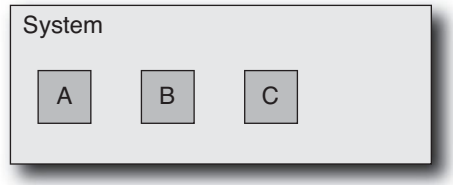


Figure 3.13 Three independent tasks

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.

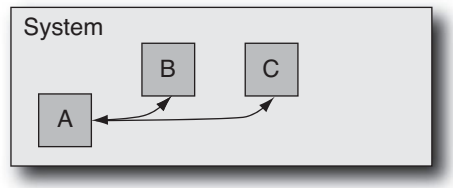


Figure 3.14 Loosely coupled tasks

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.

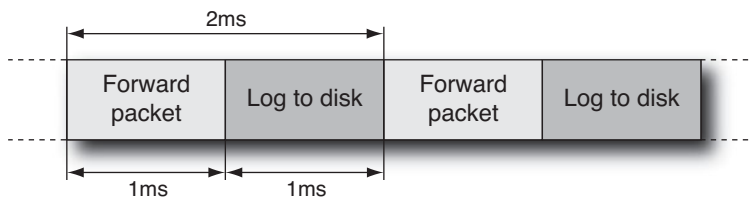


Figure 3.15 Single thread performing packet forwarding and log writing

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.

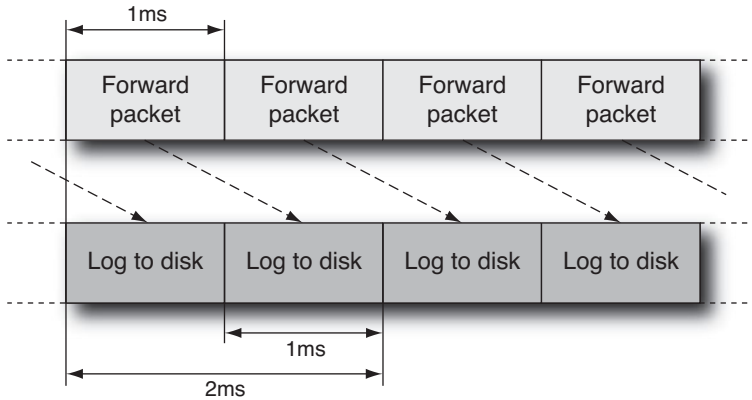


Figure 3.16 Using two threads to perform packet forwarding and log writing

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.

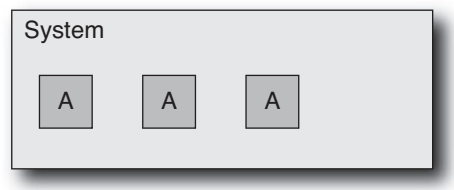


Figure 3.17 Multiple copies of a single task

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.

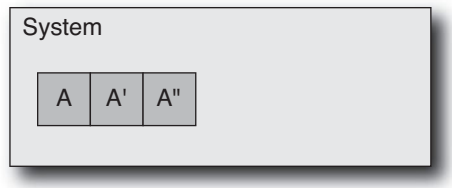


Figure 3.18 Multiple threads working on a single task

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.

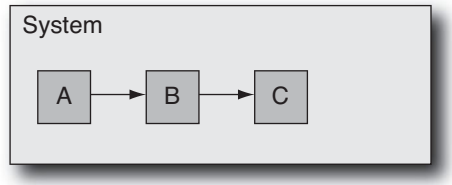


Figure 3.19 Pipeline of tasks

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.

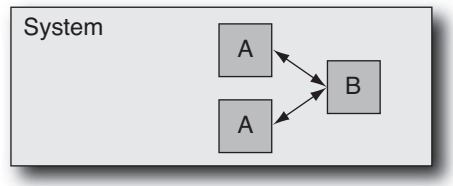


Figure 3.20 Client-server division of work

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.

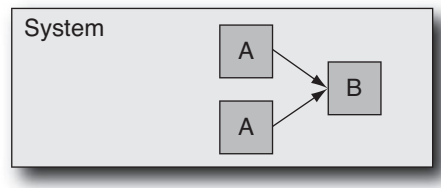


Figure 3.21 Producer-consumer division of work

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-* or *data-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**

```
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**

```
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**

		Second task	
		Read	**Write**
First task	**Read**	Read after read (RAR) No dependency	Write after read (WAR) Antidependency
	Write	Read after write (RAW) True dependency	Write after write (WAW) Output dependency

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 `data1`. The first statement needs to complete before the second statement because the second statement reuses the variable `data1`.

Listing 3.6 **An Example of an Antidependency**

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

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

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
}
```

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

Listing 3.8 **An Output Dependency**

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

If the first target variable was renamed `data1_prime`, then both statements could proceed in any order. Listing 3.9 shows this fix.

Listing 3.9 **Fixing an Output Dependency**

```
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**

```
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**

```
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 T1 seconds and the second calculation takes T2 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 T1 + P * T2.

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 (T1,T2).

Figure 3.22 shows the two situations.

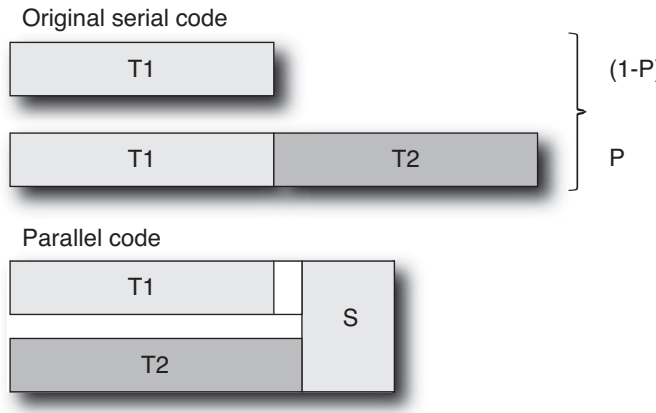


Figure 3.22 Parallelization using speculative execution

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

- If T1 > T2, then for the speculation to be profitable, S+T1 < T1+P*T2, or S < P*T2. 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 T2 > T1 (as shown in Figure 3.21), then for speculation to be profitable, S+T2 < T1+P*T2 or P > (T2 +S -T1)/T2. 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 T2 > T1, T2 + S -T1 is always >0. T2 + S -T1 represents the overhead introduced by parallelization. For the parallel code to be profitable, this has to be lower than the cost contributed by executing T2. Hence, the probability of executing T2 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 T2, then T2 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**

```
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**

```
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

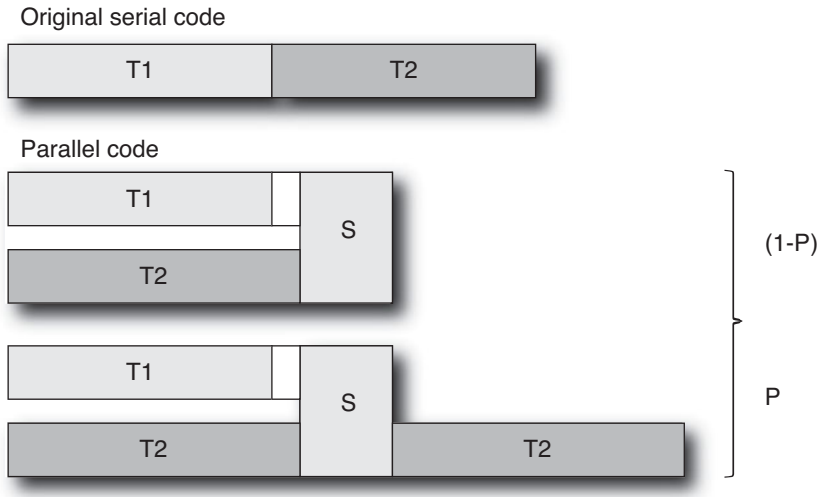


Figure 3.23 Parallelization using value speculation

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 T1+T2 seconds to complete. The parallel code takes max(T1,T2)+S+P*T2. For the parallelization to be profitable, one of the following conditions needs to be true:

- If T1 > T2, then for the speculation to be profitable, T1 + S + P*T2 < T1 +T2. So, S < (1-P) * T2. If the speculation is mostly correct, the synchronization costs just need to be less than the costs of performing T2. If the synchronization is often wrong, then the synchronization costs need to be much smaller than T2 since T2 will be frequently executed to correct the misspeculation.
- If T2 > T1, then for the speculation to be profitable, T2 + S + P*T2 < T1 +T2. So, S <T1 – P*T2. The synchronization costs need to be less than the cost of T1 after the overhead of recomputing T2 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

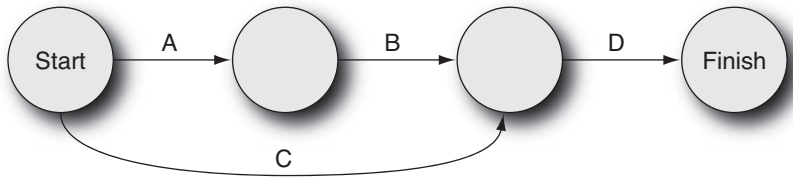


Figure 3.24 Critical paths

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.

Listing 3.14 **Opportunities for Parallelization at Different Granularities**

```
void handlePacket(packet_t *packet)
{
   doOneTask(packet);
   doSecondTask(packet);
}

void handleStream( stream_t* stream )
{
  for( int i=0; i < stream->number_of_packets; i++)
  {
    handlePacket( stream->packets[i] );
  }
}
```

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.

Index

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z