The Art of Concurrency

Clay Breshears

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

The Art of Concurrency
by Clay Breshears

Copyright 2009 Clay Breshears. All rights reserved.
Printed in the United States of America.

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

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

Editor Mike Loukides
Production Editor Sarah Schneider
Copyeditor Amy Thomson
Proofreader Sarah Schneider
Indexer Ellen Troutman Yaig
Cover Designer Karen Montgomery
Interior Designer David Futato
Illustrator Robert Romano

Printing History
May 2009

O'Reilly and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *The Art of Concurrency* , the image of wheat-harvesting combines, and related trade dress are trademarks of O'Reilly Media, Inc.

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

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

ISBN

ZV\

1241201585

CONTENTS

	PREFACE	vii
1	WANT TO GO FASTER? RAISE YOUR HANDS IF YOU WANT TO GO FASTER!	1
	Some Questions You May Have	2
	Four Steps of a Threading Methodology	7
	Background of Parallel Algorithms	12
	Shared-Memory Programming Versus Distributed-Memory Programming	15
	This Book's Approach to Concurrent Programming	19
2	CONCURRENT OR NOT CONCURRENT?	21
	Design Models for Concurrent Algorithms	22
	What's Not Parallel	42
3	PROVING CORRECTNESS AND MEASURING PERFORMANCE	49
	Verification of Parallel Algorithms	50
	Example: The Critical Section Problem	53
	Performance Metrics (How Am I Doing?)	66
	Review of the Evolution for Supporting Parallelism in Hardware	71
4	EIGHT SIMPLE RULES FOR DESIGNING MULTITHREADED APPLICATIONS	73
	Rule 1: Identify Truly Independent Computations	74
	Rule 2: Implement Concurrency at the Highest Level Possible	74
	Rule 3: Plan Early for Scalability to Take Advantage of Increasing Numbers of Cores	75
	Rule 4: Make Use of Thread-Safe Libraries Wherever Possible	76
	Rule 5: Use the Right Threading Model	77
	Rule 6: Never Assume a Particular Order of Execution	77
	Rule 7: Use Thread-Local Storage Whenever Possible or Associate Locks to Specific Data	78
	Rule 8: Dare to Change the Algorithm for a Better Chance of Concurrency	79
	Summary	80
5	THREADING LIBRARIES	81
	Implicit Threading	82
	Explicit Threading	88
	What Else Is Out There?	92
	Domain-Specific Libraries	92
6	PARALLEL SUM AND PREFIX SCAN	95
	Parallel Sum	96
	Prefix Scan	103
	Selection	112
	A Final Thought	123

7 MAPREDUCE 125
Map As a Concurrent Operation 127
Reduce As a Concurrent Operation 129
Applying MapReduce 138
MapReduce As Generic Concurrency 143

8 SORTING 145
Bubblesort 146
Odd-Even Transposition Sort 153
Shellsort 162
Quicksort 169
Radix Sort 182

9 SEARCHING 201
Unsorted Sequence 202
Binary Search 210

10 GRAPH ALGORITHMS 221
Depth-First Search 224
All-Pairs Shortest Path 240
Minimum Spanning Tree 245

11 THREADING TOOLS 257
Debuggers 258
Performance Tools 260
Anything Else Out There? 262
Go Forth and Conquer 263

GLOSSARY 265

PHOTO CREDITS 275

INDEX 277

PREFACE

Why Should You Read This Book?

MULTICORE PROCESSORS MADE A BIG SPLASH WHEN THEY WERE FIRST INTRODUCED. Bowing to the physics of heat and power, processor clock speeds could not keep doubling every 18 months as they had been doing for the past three decades or more. In order to keep increasing the processing power of the next generation over the current generation, processor manufacturers began producing chips with multiple processor cores. More processors running at a reduced speed generate less heat and consume less power than single-processor chips continuing on the path of simply doubling clock speeds.

But how can we use those extra cores? We can run more than one application at a time, and each program could have a separate processor core devoted to the execution. This would give us truly parallel execution. However, there are only so many apps that we can run simultaneously. If those apps aren't very compute-intensive, we're probably wasting compute cycles, but now we're doing it in more than one processor.

Another option is to write applications that will utilize the additional cores to execute portions of the code that have a need to perform lots of calculations and whose computations are independent of each other. Writing such programs is known as *concurrent programming*. With any programming language or methodology, there are techniques, tricks, traps, and tools to design and implement such programs. I've always found that there is more ®art¯ than ®science¯ to programming. So, this book is going to give you the knowledge and one or two of the ®secret handshakes¯ you need to successfully practice the art of concurrent programming.

In the past, parallel and concurrent programming was the domain of a very small set of programmers who were typically involved in scientific and technical computing arenas. From now on, concurrent programming is going to be mainstream. Parallel programming will eventually become synonymous with ®programming.¯ Now is *your* time to get in on the ground floor, or at least somewhere near the start of the concurrent programming evolution.

Who Is This Book For?

This book is for programmers everywhere.

I work for a computer technology company, but I'm the only computer science degree-holder on my team. There is only one other person in the office within the sound of my voice who would know what I was talking about if I said I wanted to parse an LR(1) grammar with a deterministic pushdown automata. So, CS students and graduates aren't likely to make up the bulk of the interested readership for this text. For that reason, I've tried to keep the geeky CS material to a minimum. I assume that readers have some basic knowledge of data structures and algorithms and asymptotic efficiency of algorithms (Big-Oh notation) that is typically taught in an undergraduate computer science curriculum. For whatever else I've covered, I've tried to include enough of an explanation to get the idea across. If you've been coding for more than a year, you should do just fine.

What's in This Book?

Chapter 1, *Want to Go Faster? Raise Your Hands if You Want to Go Faster*, anticipates and answers some of the questions you might have about concurrent programming. This chapter explains the differences between parallel and concurrent, and describes the four-step threading methodology. The chapter ends with a bit of background on concurrent programming and some of the differences and similarities between distributed-memory and shared-memory programming and execution models.

Chapter 2, *Concurrent or Not Concurrent?*, contains a lot of information about designing concurrent solutions from serial algorithms. Two concurrent design models—task decomposition and data decomposition—are each given a thorough elucidation. This chapter gives examples of serial coding that you may not be able to make concurrent. In cases where there is a way around this, I've given some hints and tricks to find ways to transform the serial code into a more amenable form.

Chapter 3, *Proving Correctness and Measuring Performance*, first deals with ways to demonstrate that your concurrent algorithms won't encounter common threading errors and

Conventions Used in This Book

The following typographical conventions are used in this book

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`

`+PFKECVGU≥EQOOCPFU≥QRVKQPU≥UYKVEJGU≥XCTKCDNGU≥CVVTKDWVGU≥MG[U≥HWPEVKQPU≥V[RGU ENCUUGU≥PCOGURCEGU≥OGVJQFU≥OQFWNGU≥RTQRGTVKGU≥RCTCOGVGTU≥XCNWGU≥QDLGEVU≥GXGPVU`

Constant width bold

[illegible]

Constant width italic

[illegible]

Using Code Examples

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example *The Art of Concurrency* by Clay Breshears. Copyright 2009 Clay Breshears, 978-0-596-52153-0.

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

Comments and Questions

Please address comments and questions concerning this book to the publisher

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at

http

To comment or ask technical questions about this book, send email to

bookquestions?oreilly.com

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http my.safaribooksonline.com/* .

Acknowledgments

I want to give my thanks to the following people for their influences on my career and support in the writing of this book. Without all of them, you wouldn't be reading this and I'd probably be flipping burgers for a living.

To JOSEPH SARGENT and STANLEY CHASE for bringing *Colossus* to the big screen in 1970. This movie was probably the biggest influence in my early years in getting me interested in computer programming and instilling within me the curiosity to figure out what cool and wondrous things computers could do.

To ROGER WINK for fanning the flame of my interest in computers, and for his 30-plus years of friendship and technical knowledge. He taught me Bubblesort in COBOL and is always working on something new and interesting that he can show off whenever we get the chance to meet up.

To BILL MAGRO and TOM CORTESE for being my first manager at Intel and one of my first teammates at the Intel Parallel Applications Center. Working at the PAC gave me the chance to get my ®hands dirty¯ with lots of different parallel codes, to interact with applications and customers from many different technical and commercial areas, and to learn new methods and new threading libraries. It was a ®dream come true¯ job for me.

To JERRY BAUGH, BOB CHESEBROUGH, JEFF GALLAGHER, RAVI MANOHAR, MIKE PEARCE, MICHAEL WRINN, and HUA (SELWYN) YOU for being fantastic colleagues at Intel, past and present, and for reviewing chapters of my book for technical content. I've relied on every one of these guys for their wide range of technical expertise : for their support, patience, and willingness to help me with my projects and goals : for their informed opinions : and for their continuing camaraderie throughout my years at Intel.

MIKE LOUKIDES, and the rest of the staff at O'Reilly who had a finger in this project. I couldn't have done anything like this without their help and advice and nagging me about my deadlines.

To GERGANA SLAVOVA, who posed as my ®target audience¯ and reviewed the book from cover to cover. Besides keeping me honest to my readers by making me explain complex ideas in simple terms and adding examples when I'd put too many details in a single paragraph, she peppered her comments with humorous asides that broke up the monotony of the tedium of the revision process (and she throws a slammin' tea party, too).

To HENRY GABB for his knowledge of parallel and multithreaded programming, for convincing me to apply for a PAC job and join him at Intel back in 2000, and for his devotion to SEC football and the Chicago Cubs. During the almost 15 years we've known each other, we've worked together on many different projects and we've each been able to consult with the other on technical questions. His knowledge and proficiency as a technical reviewer of this text, and many other papers of mine he has so kindly agreed to review over the years, have improved my written communication skills by an order of magnitude.

And finally, a big heartfelt ®thank you¯ to my patient and loving wife, LORNA, who now has her husband back.

CHAPTER ONE

Want to Go Faster? Raise Your Hands if You Want to Go Faster!

"[A]nd in this precious phial is the power to think twice as fast, move twice as quickly, do twice as much work in a given time as you could otherwise do."

—H. G. Wells, "The New Accelerator" (1901)

WITH THIS BOOK I WANT TO PEEL BACK THE VEILS OF MYSTERY, MISERY, AND misunderstanding that surround concurrent programming. I want to pass along to you some of the tricks, secrets, and skills that I've learned over my last two decades of concurrent and parallel programming experience.

I will demonstrate these tricks, secrets, and skills[2] and the art of concurrent programming[2] by developing and implementing concurrent algorithms from serial code. I will explain the thought processes I went through for each example in order to give you insight into how concurrent code can be developed. I will be using threads as the model of concurrency in a shared-memory environment for all algorithms devised and implemented. Since this isn't a book on one specific threading library, I've used several of the common libraries throughout and included some hints on how implementations might differ, in case your preferred method wasn't used.

Like any programming skill, there is a level of mechanics involved in being ready and able to attempt concurrent or multithreaded programming. You can learn these things (such as syntax, methods for mutual exclusion, and sharing data) through study and practice. There is also a necessary component of logical thinking skills and intuition needed to tackle or avoid even simple concurrent programming problems successfully. Being able to apply that logical thinking and having some intuition, or being able to think about threads executing in parallel with each other, is the art of concurrent and multithreaded programming. You can learn some of this through demonstration by experts, but that only works if the innate ability is already there and you can apply the lessons learned to other situations. Since you've picked up this volume, I'm sure that you, my fine reader, already possess such innate skills. This book will help you shape and aim those skills at concurrent and multithreaded programming.

Some Questions You May Have

Before we get started, there are some questions you may have thought up while reading those first few paragraphs or even when you saw this book on the shelves before picking it up. Let's take a look at some of those questions now.

What Is a Thread Monkey?

A *thread monkey* is a programmer capable of designing multithreaded, concurrent, and parallel software, as well as grinding out correct and efficient code to implement those designs. Much like a ®grease monkey¯ is someone who can work magic on automobiles, a thread monkey is

Why Do I Need to Know This? What's in It for Me?

I'm tempted to be a tad flippant and tell you that there's no way to avoid this topic: multicore processors are here now and here to stay, and if you want to remain a vital and employable programmer, you have no choice but to learn and master this material. Of course, I'd be waving my hands around manically for emphasis and trying to put you into a frightened state of mind. While all that is true to some degree, a kinder and gentler approach is more likely to gain your trust and get you on board with the concurrent programming revolution.

Whether you're a faceless corporate drone for a large software conglomerate, writing code for a small in-house programming shop, doing open source development, or just dabbling with writing software as a hobby, you are going to be touched by multicore processors to one degree or another. In the past, to get a burst of increased performance out of your applications, you simply needed to wait for the next generation of processor that had a faster clock speed than the previous model. A colleague of mine once postulated that you could take nine months off to play the drums or surf, come back after the new chips had been released, run some benchmarks, and declare success. In his seminal (and by now, legendary) article, ®The Free Lunch Is Over *Dr. Dobb's Journal* , March 2005), Herb Sutter explains that this situation is no longer viable. Programmers will need to start writing concurrent code in order to take full advantage of multicore processors and achieve future performance improvements.

What kinds of performance improvements can you expect with concurrent programming on multicore processors? As an upper bound, you could expect applications to run in half the time using two cores, one quarter of the time running on four cores, one eighth of the time running on eight cores, and so on. This sounds much better than the 20±30$ decrease in runtime when using a new, faster processor. Unfortunately, it takes some work to get code whipped into shape and capable of taking advantage of multiple cores. Plus, in general, very few codes will be able to achieve these upper bound levels of increased performance. In fact, as the number of cores increases, you may find that the relative performance actually decreases. However, if you can write good concurrent and multithreaded applications, you will be able to achieve respectable performance increases (or be able to explain why you can't). Better yet, if you can develop your concurrent algorithms in such a way that the same relative performance increases seen on two and four cores remains when executing on 8, 16, or more cores, you may be able to devote some time to your drumming and surfing. A major focus of this book will be pointing out when and how to develop such *scalable* algorithms.

Isn't Concurrent Programming Hard?

Concurrent programming is no walk in the park, that's for sure. However, I don't think it is as scary or as difficult as others may have led you to think. If approached in a logical and informed fashion, learning and practicing concurrent programming is no more difficult than learning another programming language.

Aren't Threads Dangerous?

Yes and no. In the years since multicore processors became mainstream, a lot of learned folks have come out with criticisms of the threading model. These people focus on the dangers inherent in using shared memory to communicate between threads and how nonscalable the standard synchronization objects are when pushed beyond a few threads. I won't lie to you: these criticisms do have merit.

THE TWO-MINUTE PRIMER ON CONCURRENT PROGRAMMING

Concurrent programming is all about independent computations that the machine can execute in any order. Iterations of loops and function calls within the code that can be executed autonomously are two instances of computations that can be independent. Whatever concurrent work you can pull out of the serial code can be assigned to threads (or cooperating processes) and run on any one of the multiple cores that are available (or run on a single processor by swapping the computations in and out of the processor to give the illusion of parallel execution). Not everything within an application will be independent, so you will still need to deal with serial execution amongst the concurrency.

To create the situation where concurrent work can be assigned to threads, you will need to add calls to library routines that implement threading. These additional function calls add to the *overhead* of the concurrent execution, since they were not in the original serial code. Any additional code that is needed to control and coordinate threads, especially calls to threading library functions, is overhead. Code that you add for threads to determine if the computation should continue or to get more work or to signal other threads when desired conditions have been met is all considered overhead, too. Some of that code may be devoted to ensuring that there are equal amounts of work assigned to each thread. This balancing of the workload between threads will make sure threads aren't sitting idle and wasting system resources, which is considered another form of overhead. Overhead is something that concurrent code must keep to a minimum as much as possible. In order to attain the maximum performance gains and keep your concurrent code as scalable as possible, the amount of work that is assigned to a thread must be large enough to minimize or mask the detrimental effects of overhead.

Since threads will be working together in shared memory, there may be times when two or more threads need to access the same memory location. If one or more of these threads is looking to update that memory location, you will have a *storage conflict* or *data race*. The operating system schedules threads for execution. Because the scheduling algorithm relies on many factors about the current status of the system, that scheduling appears to be asynchronous. Data races may or may not show up, depending on the order of thread executions. If the correct execution of your concurrent code depends on a particular order of memory updates (so that other threads will be sure to get the proper saved value), it is the responsibility of the program to ensure this order is guaranteed. For example, in an airline reservation system, if two travel agents see the same empty seat on a flight, they could both put the name of a client into that seat and generate a ticket. When the passengers show up at the airport, who will get the seat? To avoid fisticuffs and to enforce the correct ratio of butts to seats, there must be some means of controlling the updates of shared resources.

There are several different methods of synchronizing threads to ensure mutually exclusive access to shared memory. While synchronization is a necessary evil, use of synchronization objects is considered overhead (just like thread creation and other coordination functions) and their use should be reserved for situations that cannot be resolved in any other way.

The goal of all of this, of course, is to improve the performance of your application by reducing the amount of time it takes to execute, or to be able to process more data within a fixed amount of time. You will need an awareness of the perils and pitfalls of concurrent programming and how to avoid or correct them in order to create a correctly executing application with satisfactory performance.

(QWT 5VGRU QH C 6JTGCFKPI /GVJQFQ

9JGP FGXGNQRKPI UQHVYCTG GURGEKCNN[NCTIG EQOOGTEKCN
VQ GPUWTG VJCV GXGT[VJKPI KU FQPG VQ OGGV VJG IQCNU QH

Step 1. Analysis

Since the code is already designed and written, the functionality of the application is known. You should also know which outputs are generated for given inputs. Now you need to find the parts of the code that can be threaded: that is, those parts of the application that contain independent computations.

If you know the application well, you should be able to home in on these parts of the code rather quickly. If you are less familiar with all aspects of the application, you can use a profile of the execution to identify *hotspots* that might yield independent computations. A hotspot is any portion of the code that has a significant amount of activity. With a profiler, time spent in the computation is going to be the most obvious measurable activity. Once you have found points in the program that take the most execution time, you can begin to investigate these for concurrent execution.

Just because an application spends a majority of the execution time in a segment of code, that does not mean that the code is a candidate for concurrency. You must perform some algorithmic analysis to determine if there is sufficient independence in the code segment to justify concurrency. Still, searching through those parts of the application that take the most time will give you the chance to achieve the most ®bang for the buck¯ (i.e., be the most beneficial to the overall outcome). It will be much better for you (and your career) to spend a month writing, testing, and tuning a concurrent solution that reduces the execution time of some code segment that accounts for 75$ of the serial execution time than it would be to take the same number of hours to slave over a segment that may only account for 2$.

Step 2. Design and Implementation

Once you have identified independent computations, you need to design and implement a concurrent version of the serial code. This step is what this book is all about. I won't spend any more time here on this topic, since the details and methods will unfold as you go through the pages ahead.

Step 3. Test for Correctness

Whenever you make code changes to an application, you open the door to the possibility of introducing bugs. Adding code to a serial application in order to generate and control multiple threads is no exception. As I alluded to before, the execution of threaded applications may or may not reveal any problems during testing. You might be able to run the application correctly hundreds of times, but when you try it out on another system, errors might show up on the new system or they might not. Even if you can get a run that demonstrates an error, running the code through a debugger (even one that is thread-aware) may not pinpoint the problem, since the stepwise execution may mask the error when you are actively looking for it. Using a print statement[2] that most-used of all debugging tools[2] to track values assigned to variables can modify the timing of thread interleavings, and that can also hide the error.

The more common threading errors, such as data races and deadlock, may be avoided completely if you know about the causes of these errors and plan well enough in the Design and Implementation step to avoid them. However, with the use of pointers and other such indirect references within programming languages, these problems can be virtually impossible to foresee. In fact, you may have cases in which the input data will determine if an error might manifest itself. Luckily, there are tools that can assist in tracking down threading errors. I've listed some of these in Chapter 11.

Even after you have removed all of the known threading bugs introduced by your modifications, the code may still not give the same answers as the serial version. If the answers are just slightly off, you may be experiencing *round-off error*, since the order of combining results generated by separate threads may not match the combination order of values that were generated in the serial code.

More egregious errors are likely due to the introduction of some logic error when threading. Perhaps you have a loop where some iteration is executed multiple times or where some loop iterations are not executed at all. You won't be able to find these kinds of errors with any tool that looks for threading errors, but you may be able to home in on the problem with the use of some sort of debugging tool. One of the minor themes of this book is the typical logic errors that can be introduced around threaded code and how to avoid these errors in the first place. With a good solid design, you should be able to keep the number of threading or logic errors to a minimum, so not much verbiage is spent on finding or correcting errors in code.

Step 4. Tune for Performance

After making sure that you have removed all the threading (and new logic) errors from your code, the final step is to make sure the code is running at its best level of performance. Before threading a serial application, be sure you start with a tuned code. Making serial tuning modifications to threaded code may change the whole dynamic of the threaded portions such that the additional threading material can actually degrade performance. If you have started

The testing and tuning cycle

When you modify your code to correct an identified performance bug, you may inadvertently add a threading error. This can be especially true if you need to revise the use of synchronization objects. Once you've made changes for performance tuning, you should go back to the Test for Correctness step and ensure that your changes to fix the performance bugs have not introduced any new threading or logic errors. If you find any problems and modify code to repair them, be sure to again examine the code for any new performance problems that may have been inserted when fixing your correctness issues.

Sometimes it may be worse than that. If you are unable to achieve the expected performance speed from your application, you may need to return to the Design and Implementation step and start all over. Obviously, if you have multiple sites within your application that have been made concurrent, you may need to start at the design step for each code segment once you have finished with the previous code segment. If some threaded code sections can be shown to improve performance, these might be left as is, unless modifications to algorithms or global data structures will affect those previously threaded segments. It can all be a vicious circle and can make you dizzy if you think about it too hard.

What About Concurrency from Scratch?

Up to this point (and for the rest of the book, too), I've been assuming that you are starting with a correctly executing serial code to be transformed into a concurrent equivalent. Can you design a concurrent solution without an intermediate step of implementing a serial code? Yes, but I can't recommend it. The biggest reason is that debugging freshly written parallel code has

Background of Parallel Algorithms

If you're unfamiliar with parallel algorithms or parallel programming, this section is for you[2] it serves as a brief guide to some of what has gone before to reach the current state of concurrent programming on multicore processors.

Theoretical Models

All my academic degrees are in computer science. During my academic career, I've had to learn about and use many different models of computation. One of the basic processor architecture models used in computer science for studying algorithms is the *Random Access Machine* (RAM) model. This is a simplified model based on the von Neumann architecture model. It has all the right pieces
memory. See Figure 1-1 for a pictorial view of the components of the RAM and how data flows between components.

FIGURE 1-1. RAM configuration with data flow indicated by arrows

You can add hierarchies to the memory in order to describe levels of cache, you can attach a random access disk as a single device with both input and output, you can control the complexity and architecture of the CPU, or you can make dozens of other changes and modifications to create a model as close to reality as you desire. Whatever bits and pieces and doodads you think to add, the basics of the model remain the same and are useful in designing serial algorithms.

Memory access parameters	Description
Concurrent Read, Exclusive Write (CREW)	Multiple threads may read from the same memory location at the same time and one thread may write to a given memory location at any time.
Exclusive Read, Concurrent Write (ERCW)	One thread may read from a given memory location at any time and multiple threads may write to the same memory location at the same time.
Exclusive Read, Exclusive Write (EREW)	One thread may read from a given memory location at any time and one thread may write to a given memory location at any time.

On top of these restrictions, it is up to the PRAM algorithm to enforce the exclusive read and exclusive write behavior of the chosen model. In the case of a concurrent write model, the model further specifies what happens when two threads attempt to store values into the same memory location at the same time. Popular variations of this type of PRAM are to have the algorithm ensure that the value being written will be the same value, to simply select a random value from the two or more processors attempting to write, or to store the sum (or some other combining operation) of the multiple values. Since all processors are executing in lockstep fashion, writes to memory are all executed simultaneously, which makes it easy to enforce the designated policy.

Not only must you specify the memory access behavior of the PRAM and design your algorithm to conform to that model, you must also denote the number of processors that your algorithm will use. Since this is a theoretical model, an unlimited number of processors are available. The number is typically based on the size of the input. For example, if you are designing an algorithm to work on N input items, you can specify that the PRAM must have N^2 processors and threads, all with access to the shared memory.

With an inexhaustible supply of processors and infinite memory, the PRAM is obviously a theoretical model for parallel algorithm design. Implementing PRAM algorithms on finite resourced platforms may simply be a matter of simulating the computations of N "logical" processors on the cores available to us. When we get to the algorithm design and implementation chapters, some of the designs will take a PRAM algorithm as the basic starting point, and I'll show you how you might convert it to execute correctly on a multicore processor.

Distributed-Memory Programming

Due to shared bus contention issues, shared-memory parallel computers hit an upper limit of approximately 32 processors in the late '80s and early '90s. Distributed-memory configurations came on strong in order to scale the number of processors higher. Parallel algorithms require some sharing of data at some point. However, since each node in a distributed-memory machine is separated from all the other nodes, with no direct sharing mechanism, developers used libraries of functions to pass messages between nodes.

Parallel Algorithms Literature

Many books have been written about parallel algorithms. A vast majority of these have focused on message-passing as the method of parallelization. Some of the earlier texts detail algorithms where the network configuration (e.g., mesh or hypercube) is an integral part of the algorithm design: later texts tend not to focus so much on developing algorithms for specific network configurations, but rather, think of the execution platform as a cluster of processor nodes. In the algorithms section of this book (Chapters 6 through 10), some of the designs will take a distributed-memory algorithm as the basic starting point, and I'll show you how you might convert it to execute correctly in a multithreaded implementation on a multicore processor.

Shared-Memory Programming Versus Distributed-Memory Programming

Some of you may be coming from a distributed-memory programming background and want to get into threaded programming for multicore processors. For you, I've put together a list that compares and contrasts shared-memory programming with distributed-memory programming. If you don't know anything about distributed-memory programming, this will give you some insight into the differences between the two programming methods. Even if you've only ever done serial programming to this point, the following details are still going to give you an introduction to some of the features of concurrent programming on shared-memory that you never encounter using a single execution thread.

Common Features

The following features are common to shared-memory parallelism and distributed-memory parallelism.

Redundant work

No matter how concurrent an algorithm is, there are still parts that will have to be carried out serially. When the results of those serial computations must be shared, you can perform the computation in one process and broadcast the results out to the other processes that require the data. Sending this information will add overhead to the execution time. On the other hand, if the data used in the computation is already available to other processes, you can have each process perform the computation and generate results locally without the need to send any messages. In shared-memory parallelism, the data for computation is likely already available by default. Even though doing redundant work in threads keeps processing resources busy and eliminates extraneous synchronization, there is a cost in memory space needed to hold multiple copies of the same value.

Dividing work

Work must be assigned to threads and processes alike. This may be done by assigning a chunk of the data and having each thread/process execute the same computations on the assigned block, or it may be some method of assigning a computation that involves executing a different portion of the code within the application.

Sharing data

There will be times when applications must share data. It may be the value of a counter or a vector of floating-point values or a list of graph vertices. Whatever it happens to be, threads and processes alike will need to have access to it during the course of the computation. Obviously, the methods of sharing data will vary: shared-memory programs simply access a designated location in memory, while distributed-memory programs must actively send and receive the data to be shared.

Static/dynamic allocation of work

Depending on the nature of the serial algorithm, the resulting concurrent version, and the number of threads/processes, you may assign all the work at one time (typically at the outset of the computation) or over time as the code executes. The former method is known as a *static allocation* since the original assignments do not change once they have been made. The latter is known as *dynamic allocation* since work is doled out when it is needed. Under dynamic allocation, you may find that the same threads do not execute the same pieces of work from one run to another, while static allocation will always assign the same work to the same threads (if the number of threads is the same) each and every time.

Features Unique to Shared Memory

These next few items are where distributed-memory and shared-memory programming differ. If you're familiar with distributed-memory parallelism, you should be able to see the differences. For those readers not familiar with distributed-memory parallelism, these points and ideas are still going to be important to understand.

Local declarations and thread-local storage

Since everything is shared in shared memory, there are times it will be useful to have a private or local variable that is accessed by only one thread. Once threads have been spawned, any declarations within the path of code execution (e.g., declarations within function calls) will be automatically allocated as local to the thread executing the declarative code. Processes executing on a node within a distributed-memory machine will have all local memory within the node.

A thread-local storage (TLS) API is available in Windows threads and POSIX threads. Though the syntax is different in the different threaded libraries, the API allocates some memory per executing thread and allows the thread to store and retrieve a value that is accessible to only that thread. The difference between TLS and local declarations is that the TLS values will persist from one function call to another. This is much like *static* variables, except that in TLS, each thread gets an individually addressable copy.

Memory effects

Since threads are sharing the memory available to the cores on which they are executing, there can be performance issues due to that sharing. I've already mentioned storage conflicts and data races. Processor architecture will determine if threads share or have access to separate caches. Sharing caches between two cores can effectively cut in half the size of the cache available to a thread, while separate caches can make sharing of common data less efficient. On the good side, sharing caches with commonly accessed, read-only data can be very effective, since only a single copy is needed.

False sharing is a situation where threads are not accessing the same variables, but they are sharing a cache line that contains different variables. Due to cache coherency protocols, when one thread updates a variable in the cache line and another thread wants to access something

Communication in memory

Distributed-memory programs share data by sending and receiving messages between processes. In order to share data within shared memory, one thread simply writes a value into a memory location and the other thread reads the value out of that memory location. Of course, to ensure that the data is transferred correctly, the writing thread must deposit the value to be shared into memory before the reading thread examines the location. Thus, the threads must synchronize the order of writing and reading between the threads. The send-receive exchange is an implicit synchronization between distributed processes.

Mutual exclusion

In order to communicate in memory, threads must sometimes protect access to shared memory locations. The means for doing this is to allow only one thread at a time to have access to shared variables. This is known as *mutual exclusion*. Several different synchronization mechanisms are available (usually dependent on the threading method you are using) to provide mutual exclusion.

Both reading and writing of data must be protected. Multiple threads reading the same data won't cause any problems. When you have multiple threads writing to the same location, the order of the updates to the memory location will determine the value that is ultimately stored there and the value that will be read out of the location by another thread (recall the airline reservation system that put two passengers in the same seat). When you have one thread reading and one thread writing to the same memory location, the value that is being read can be one of two values (the old value or the new value). It is likely that only one of those will be the expected value, since the original serial code expects only one value to be possible. If the correct execution of your threaded algorithm depends upon getting a specific value from a variable that is being updated by multiple threads, you must have logic that guarantees the right value is written at the correct time: this will involve mutual exclusion and other synchronization.

Producer/consumer

One algorithmic method you can use to distribute data or tasks to the processes in distributed-memory programs is *boss/worker*. Worker processes send a message to the boss process requesting a new task: upon receipt of the request, the boss sends back a message/task/data to the worker process. You can write a boss/worker task distribution mechanism in threads, but it requires a lot of synchronization.

To take advantage of the shared memory protocols, you can use a variation of boss/worker that uses a shared queue to distribute tasks. This method is known as *producer/consumer*. The

Readers/writer locks

Since it is not a problem to have multiple threads reading the same shared variables, using mutual exclusion to prevent multiple reader threads can create a performance bottleneck. However, if there is any chance that another thread could update the shared variable, mutual exclusion must be used. For situations where shared variables are to be updated much less frequently than they are to be read, a *readers/writer* lock would be the appropriate synchronization object.

Readers/writer locks allow multiple reader threads to enter the protected area of code accessing the shared variable. Whenever a thread wishes to update (write) the value to the shared variable, the lock will ensure that any prior reader threads have finished before allowing the single writer to make the updates. When any writer thread has been allowed access to the shared variable by the readers/writer lock, new readers or other threads wanting write access are prevented from proceeding until the current writing thread has finished.

This Book's Approach to Concurrent Programming

While writing this book, I was reading *Feynman Lectures on Computation* (Perseus Publishing, 1996). In Chapter 3, Feynman lectures on the theory of computation. He starts by describing finite state machines (automata) and then makes the leap to Turing machines. At first I was a bit aghast that there was nothing at all about push-down automata or context-free languages, nothing about nondeterministic finite-state machines, and nothing about how this all tied into grammars or recognizing strings from languages. A nice progression covering this whole range of topics was how I was taught all this stuff in my years studying computer science.

I quickly realized that Feynman had only one lecture to get through the topic of Turing machines and the ideas of computability, so he obviously couldn't cover all the details that I learned over the course of eight weeks or so. A bit later, I realized that the target audience for his lecture series wasn't computer science students, but students in physics and mathematics. So he only needed to cover those topics that gave his students the right background and enough of a taste to get some insight into the vast field of computability theory.

This is what I'm hoping to do with this book. I don't want to give you all the history or theory about concurrent and parallel programming. I want to give you a taste of it and some practical examples so that you (the brilliant person and programmer that I know you to be) can take them and start modifying your own codes and applications to run in parallel on multicore processors. The algorithms in the later chapters are algorithms you would find in an

CHAPTER TWO

Concurrent or Not Concurrent?

TO GET THINGS STARTED, I WANT TO FIRST TALK ABOUT TWO DESIGN METHODS FOR concurrent algorithms, but I want to do it abstractly. Now, before you roll your eyes too far back and hurt yourself, let me say that there will be plenty of code examples in later chapters to give concreteness to the ideas that are presented here. This is a book on the design of concurrent algorithms, and in this chapter I've collected a lot of the wisdom on initial approaches that apply to a large percentage of code you're likely to encounter (it can get pretty dry without code to look at, so be sure you're well hydrated before you start).

In addition, I want to let you know that not every bit of computation can be made concurrent, no matter how hard you try. To save you the time of trying to take on too many impossible things in the course of your day, I have examples of the kinds of algorithms and computations that are not very amenable to concurrency in the section ®What's Not Parallel¯ on page 42. When any of those examples can be modified to allow for concurrent execution, I've included hints and tips about how to do that.

Design Models for Concurrent Algorithms

If you've got a sequential code that you want to transform into a concurrent version, you need to identify the independent computations that can be executed concurrently. The way you approach your serial code will influence how you reorganize the computations into a concurrent equivalent. One way is *task decomposition*, in which the computations are a set of independent tasks that threads can execute in any order. Another way is *data decomposition*, in which the application processes a large collection of data and can compute every element of the data independently.

The next two sections will describe these approaches in more detail and give an example of a problem that falls into each category. These two models are not the only possibilities, but I've found them to be the two most common. For other patterns of computation and how to transform them into concurrent algorithms, read *Patterns for Parallel Programming* by Timothy G. Mattson et al. (Addison-Wesley, 2004). Many of the ideas presented in the next two sections are rooted in material from that book.

Task Decomposition

When you get right down to it, any concurrent algorithm is going to turn out to be nothing more than a collection of concurrent tasks. Some may be obvious independent function calls within the code. Others may turn out to be loop iterations that can be executed in any order or simultaneously. Still others might turn out to be groups of sequential source lines that can be divided and grouped into independent computations. For all of these, you must be able to identify the tasks and decompose the serial code into concurrently executable work. If you're familiar enough with the source code and the computations that it performs, you may be able to identify those independent computations via code inspection.

What are the tasks and how are they defined?

The ease of identifying independent computations within an application is in direct proportion to your understanding of the code and computations being performed by that code. There isn't any procedure, formula, or magic incantation that I know of where the code is input and out pops a big neon sign pointing to the independent computations. You need to be able to mentally simulate the execution of two parallel streams on suspected parts of the application to determine whether those suspected parts are independent of each other (or might have manageable dependencies).

Simulating the parallel or concurrent execution of multiple threads on given source code is a skill that has been extremely beneficial to me in both designing concurrent algorithms and in proving them to be error-free (as we shall see in Chapter 3). It takes some practice, but like everything else that takes practice, the more you do it, the better you will get at doing it. While you're reading my book, I'll show you how I approach the art of concurrent design, and then you'll be better equipped to start doing this on your own.

> **NOTE**
>
> **There is one tiny exception for not having a "magic bullet" that can identify potentially independent computations with loop iterations. If you suspect a loop has independent iterations (those that can be run in any order), try executing the code with the loop iterations running in reverse of their original order. If the application still gets the same results, there is a strong chance that the iterations are independent and can be decomposed into tasks. Beware that there might still be a "hidden" dependency waiting to come out and bite you when the iterations are run concurrently—for example, the intermediate sequence of values stored in a variable that is harmless when the loop iterations were run in serial, even when run backward.**

To get the biggest return on investment, you should initially focus on computationally intense portions of the application. That is, look at those sections of code that do the most computation or account for the largest percentage of the execution time. You want the ratio of the

NOTE

Granularity, defined another way, is the amount of computation done before synchronization is needed. The longer the time between synchronizations, the coarser the granularity will be. Fine-grained concurrency runs the danger of not having enough work assigned to threads to overcome the overhead costs (synchronizations) of using threads. Adding more threads, when the amount of computation doesn't change, only exacerbates the problem. Coarse-grained concurrency has lower relative overhead costs and tends to be more readily scalable to an increase in the number of threads.

Consider the case where the time for overhead computations per task is the same for two different divisions of tasks. If one task divides the total work into 16 tasks, and the other uses

FIGURE 2-1. Task granularity example

You can imagine that the height of each box in Figure 2-1 represents the amount of time required to execute the associated computation. With many small tasks—each requiring a separate overhead computation—Figure 2-1 (a), the fine-grained solution, will take longer to run than the same work divided into fewer larger tasks, as shown in Figure 2-1 (b), the coarse-grained decomposition. Larger tasks also provide better opportunities for other performance benefits (e.g., reuse of cache, more efficient memory access patterns) that are already part of the serial algorithm.

You will need to strike a balance between these two criteria. For a fixed amount of work, the larger you define your tasks to be, the fewer tasks there will be to assign to threads. You may find that in order to satisfy the second criterion, which is the more important of the two, you will need to define a number of tasks that is fewer than the number of threads. Because reducing execution time is the main reason for writing and running concurrent code, having an application that might not utilize all the cores available is more desirable than having an application that performs worse (takes longer to execute) when using all the cores in the platform.

Finally, don't be afraid to go back and rework your task decomposition. If your initial decomposition does not meet the criteria, you should consider alternate decompositions. Also, if you find that you are not achieving the performance levels you expect, you may need to go back and redefine the tasks, the number of threads to be used, or how those tasks are assigned to threads.

What are the dependencies between tasks and how can they be satisfied?

Two types of dependencies can occur between tasks. The first is order dependency, where some task relies on the completed results of the computations from another task. This reliance can be a direct need to use the computed values as input to the succeeding task, or it may simply be the case that the task that follows will be updating the same memory locations as the previous task and you must ensure that all of the previous updates have been completed before proceeding. Both of these cases describe a potential data race, which we need to avoid.

For example, if you are building a house, putting up the roof involves attaching the rafters to the walls, laying down the decking, and applying the shingles. The dependence between these three tasks is one of execution order. You can't put down shingles until the decking is there, and you can't nail down the decking unless you have the rafters in place. So, instead of hiring three teams to do these three tasks in parallel, you can hire one roofing crew to do all three in the order required (there is parallelism within each of the roofing steps, plus the act of putting on the roof is independent of installing the electrical wiring, the plumbing, and putting up drywall).

To satisfy an execution order constraint, you can schedule tasks that have an order dependency onto the same thread and ensure that the thread executes the tasks in the proper sequence. The serial code was written with the order dependency already taken care of. So, the serial algorithm should guide the correct decomposition of the computations into tasks and assignment of those tasks to threads. Still, even after grouping tasks to execute on threads, there may be order constraints between threads. If regrouping tasks to threads is not an option or will severely hurt performance, you will be forced to insert some form of synchronization to ensure correct execution order.

The second type of dependency is data dependency. Identifying potential data dependencies can be straightforward
assignment operator. Data races require that the variable in question have at least one thread that is writing to that variable. Check for any assignment of values to the same variable that might be done concurrently as well as any updates to a variable that could be read concurrently. Of course, using pointers to reference memory locations can make the identification process trickier. There are tools (covered in Chapter 11) that can assist in finding nonobvious data dependencies in your code.

Solving data dependencies can be more complicated than solving execution order dependencies. In the latter, the sequence of execution within the serial code gives us the solution : in the former, the serial code being written with the assumption of a single-threaded execution leads to the problem.

If you're fortunate enough to have code with no dependencies, sometimes called *enchantingly parallel*, you can skip down to the next section about scheduling. If you're like the rest of us and aren't so fortunate, examine your dependencies to see if they might be removable (recurrences and induction variables) or separable (reduction computations). There are

EXAMPLE 2-1. Pseudocode with shared variable

```
popDiff = abs(Population[MyTown] - Population[NewYork]);
DoSomething(popDiff, MyTown, NewYork);
popDiff = abs(Population[MyTown] - Population[Chicago]);
DoSomething(popDiff, MyTown, Chicago);
```

```
+H≥YG≥MPQY≥VCIV≥EQPEWTTGPV≥ECNNU≥VQ≥VCI≥DoSomething()≥HWPEVKQP≥CTG≥VJGCFUCHG≥KG≥VCITG≥CTG
PQ≥UKFG≥GHHGEVU≥QT≥FGRGPFGPEKGU≥YGIP≥VCITG≥KU≥OQTG≥VCIP≥QPG≥EQPEWTTGPV≥ECNN≥VQ≥VCI≥HWPEVKQP
YG≥ECP≥CUUKIP≥VCI≥HKTUV≥VYQ≥NKPGU≥QH≥VCI≥GZCORNG≥VQ≥QPG≥VJGCF≥CPF≥VCI≥NCUV≥VYQ≥NKPGU≥VQ≥C≥UGEQPF
VJGCF≥6KU≥YKNN≥ETGCVG≥C≥FCVC≥TCEG≥QP≥VCI≥popDiff≥XCTKCDNG≥5KPEG≥VKU≥XCTKCDNG≥KU≥WUGF≥QPN[≥CU≥C
VGORQTCT[≥QT≥pYQTMq≥XCTKCDNG≥CNNQECVKPI≥C≥NQECN≥EQR[≥VQ≥GCE3VJGCF≥YKNN≥GNKOKPCVG≥VCI≥RTQDNGO

&GRGPFKPI≥QP≥VCI≥VJGCFKPI≥OQFGN≥VCIV≥[QW≥CTG≥WUKPI≥VQ≥KORNGOGPV≥[QWT≥EQPEWTTGPV
CNIQTKVO≥VCITG≥ECP≥DG≥UGXGTCN≥YC[U≥VQ≥ETGCVG≥XCTKCDNGU≥VCIV≥CTG≥CEEGUUKDNG≥QPN[≥VQ≥C≥IKXGP
VJGCF≥+P≥CNN≥ECUGU≥KH≥[QW≥FGENCTG≥C≥XCTKCDNG≥YKVKP≥C≥HWPEVKQP≥VCIV≥KU≥GZGEWVGF≥D[≥VJGCFU≥VQUG
XCTKCDNGU≥YKNN≥DG≥NQECN≥VQ≥VCI≥ECNNKPI≥VJGCF≥YGIP≥VCI[≥CTG≥CNNQECVGF≥7RNKEKVN[≥CNNQECVKPI≥URCEG
HTQO≥C≥VJGCFoU≥UVCEM≥UC[≥YKV3VCI≥alloca()≥HWPEVKQP≥KU≥CPQVCIT≥YC[≥1RGP/2≥CIU≥VCI
private≥ENCWUG≥VQ≥IGPGTCVG≥NQECN≥EQRKGU≥QH≥XCTKCDNGU≥HQT≥GCE3VJGCF≥YKVKP≥VCI≥RCTCNNGN≥TGIKQP≥VQ
YKE3VCI≥ENCWUG≥KU≥CVVCECIF≥@V39KPFQYU≥CPF≥215+:≥VJGCFU≥KPENWFG≥C≥VJGCFNQECN≥UVQTCIG
6.5≥#2+≥VCIV≥YKNN≥CNNQECVG≥OGOQT[≥VQ≥QNF≥EQRKGU≥QH≥XCTKCDNGU≥QPG≥RGT≥VJGCF≥CPF≥CNNQY
VJGCFU≥VQ≥CIXG≥CEEGUU≥QPN[≥VQ≥VCI≥EQR[≥VCIV≥KU≥GCTOCTMGF≥HQT≥VCIV≥VJGCF
```

> **NOTE**
>
> **The TLS API is pretty "heavy." I wouldn't recommend using it for things like local work variables within a single routine. Variables that are allocated and accessed via the TLS are persistent across function scopes. If you need local copies of a variable and that variable and its contents need to be available to different functions or disparate calls to the same function executed by the thread, TLS is the mechanism that can give you the private copy and the persistence of value required.**

When all else fails, when you don't have the option to make a local copy of a shared variable, when none of the transformations given in "What's Not Parallel" on page 42 can eliminate the data dependency, the only option left is to explicitly provide mutually exclusive access to the shared variable. In most cases a simple lock, or *mutex*, will suffice. In some instances, you can use a less onerous atomic operation. Different threading models will have different options

How are the tasks assigned to threads?

Tasks must be assigned to threads for execution. Perhaps the more correct way to say this is that threads must know which tasks to execute. In either case, you always want to assure that the amount of computation done by threads is roughly equivalent. That is, the load (of computation) is balanced per thread. We can allocate tasks to threads in two different ways static scheduling or dynamic scheduling.

> **NOTE**
>
> **Under worksharing constructs in OpenMP and the parallel algorithms of Intel Threading Building Blocks (TBB), the actual assignment of tasks to threads is done "under the covers." The programmer can influence that assignment to some degree, though. Even if you use only one of these two threading libraries for your concurrent coding, you should still read through the advice in this section to help you better influence the task assignments in your applications.**

In static scheduling, the division of labor is known at the outset of the computation and doesn't change during the computation. If at all possible, when developing your own concurrent code, try to use a static schedule. This is the easiest method to program and will incur the least amount of overhead.

The mechanics and logic of code needed to implement a static schedule will involve each thread having a unique identification number in the range of 0, N 1 for *N* threads. This number can be easily assigned at the time a thread is created in the order that threads are created (code that can generate unique identification numbers to threads will be part of several implementation examples in later chapters). If tasks are collections of separate, independent function calls or groups of sequential source lines, you can group those calls or code lines into tasks that are assigned to threads through a thread's ID number (e.g., through a `switch`

UVCVGOGPV≥+H≥VCUMU≥CTG≥NQQR≥KVGTCVKQPU≥[QW≥ECP≥FKXKFG≥VG≥VQVCN≥PWODGT≥QH≥KVGTCVKQPU≥D[≥VG
PWODGT≥QH≥VJGCFU≥CPF≥CUUKIP≥DNQEMU≥QH≥KVGTCVKQPU≥VQ≥VJGCFU≥CICKP≥VJQWI3VG≥VJGCFoU≥+&
PWODGT≥;QW≥YKNN≥GXG≥VQ≥CFF≥CFFKVKQPCN≥NQIKE≥VQ≥EQORWVG≥VG≥UVCTV≥CPF≥GPF≥XCNWGU≥QH≥VG≥NQQR
DQWPFU≥KP≥QTFGT≥HQT≥GCE3VJGCF≥VQ≥FGVGTOKPG≥VG≥DNQEM≥VGV≥UQWNF≥DG≥GZGEWVGF

```
  area = width * sum;
  printf("Computed pi = %f\n",area);
}
```

```
9CV≥CTG≥VC≥KPFGRGPFGPV≥VCUMU≥KP≥VKU≥UKORNG≥CRRNKECVKQP!≥6C≥EQORWVCVKQP≥QH≥VC≥CKIV≥HQT
C≥TGEVCPING≥KU≥KPFGRGPFGPV≥QH≥VC≥EQORWVCVKQP≥QH≥CP[≥QVCT≥TGEVCPING≥+P≥CFFKVKQP≥PQVKEG≥VCV
VC≥NQQR≥VCV≥RGTHQTOU≥VCUG≥ECNEWNCVKQPU≥QNFU≥VC≥DWNM≥QH≥VC≥EQORWVCVKQP≥HQT≥VC≥GPVKTG
CRRNKECVKQP≥6C≥NQQR≥KU≥CP≥GZGEWVKQP≥QVURQV≥VCV≥[QW≥UQWNF≥GZCOKPG≥HQT≥KPFGRGPFGPV
EQORWVCVKQPU
```

```
#TG≥VCTG≥CP[≥FGRGPFGPEKGU≥DGVYGGP≥VCUG≥VCUMU≥CPF≥KH≥UQ≥QY≥ECP≥YG≥UCVKUH[≥VCO!≥6C≥VYQ
YQTM≥XCTKCDNGUmid≥CPFheight≥CTG≥CUUKIPGF≥XCNWGU≥KP≥GCE3KVGTCVKQP≥+H≥GCE3VTGCF≥CF≥C≥NQECN
EQR[≥VCV≥EQR[≥EQWNF≥DG≥WUGF≥FWTKPI≥GZGEWVKQP≥QH≥KVGTCVKQPU≥VCV≥CTG≥CUUKIPGF≥VQ≥C≥VTGCF≥#NUQ
VC≥KVGTCVKQP≥XCTKCDNG≥i≥KU≥WRFCVGF≥HQT≥GCE3KVGTCVKQP≥CE3VTGCF≥YKNN≥PGGF≥C≥NQECN≥EQR[≥KP
QTFGT≥VQ≥CXQKF≥KPVGTHGTKPI≥YKV3VC≥GZGEWVKQP≥QH≥KVGTCVKQPU≥YKVKP≥QVCT≥VTGCFUmid≥XCTKCDNG
KU≥WRFCVGF≥KP≥GCE3KVGTCVKQP≥DWV≥UKPEG≥VKU≥XCNWG≥KU≥WUGF≥QWVUKFG≥QH≥VC≥NQQR≥YG≥ECPoV≥CXG≥GCEJ
VTGCF≥YQTM≥YKV3C≥NQECN≥EQR[≥VCV≥YQWNF≥DG≥FKUECTFGF≥CHVGT≥VC≥VTGCF≥YCU≥FQPG≥6KU≥KU≥C
TGFWEVKQP2CPF≥[QWoNN≥HKPF≥VKRU≥QP≥UQNXKPI≥UWE3UKVWCVKQPU≥NCVGT≥KP≥VC≥UGEVKQP≥p9CVoU≥oQV
2CTCNNGNq≥QP≥RCIG≥≥6C≥SWKEM≥CPF≥FKTV[≥UQNWVKQP≥YQWNF≥DG≥VQ≥RWV≥C≥U[PETJQPK\CVKQP≥QDLGEV
CTQWPF≥VC≥NKPG≥QH≥EQFG≥WRFCVKPSum≥UQ≥VCV≥QPN[≥QPG≥VTGCF≥CV≥C≥VKOG≥YKNN≥YTKVG≥VQ≥VC≥XCTKCDNG
```

```
*QY≥UQWNF≥[QW≥CUUKIP≥VCUMU≥VQ≥VTGCFU!≥9KV3NQQR≥KVGTCVKQPU≥CU≥VCUMU≥YG≥ECP≥CUUKIP≥DNQEMU≥QH
KVGTCVKQPU≥VQ≥VTGCFU≥DCUGF≥QP≥CP≥CUUKIPGF≥+&≥PWODGT≥#NVGTPCVKXGN[≥YG≥ECP≥CXG≥VTGCFU
GZGEWVG≥CNVGTPCVKPI≥KVGTCVKQPU≥DCUGF≥QP≥VC≥PWODGT≥QH≥VTGCFU≥$ECWUG≥VCTG≥CTG≥PQ≥KPFGZGF
CTTC[≥GNGOGPVU≥KP≥VC≥NQQR≥CPF≥VKU≥PQ≥ECEC≥KUUWGU≥+≥YQWNF≥TGEQOOGPF≥VC≥NCVVGT≥CRRTQCEJ
```

```
6C≥HKPCN≥RKGEG≥VQ≥EQPUKFGT≥KU≥CFFKPI≥VC≥TGUWNVU≥QH≥CNN≥VC≥KPFKXKFWCN≥NQQR≥EQORWVCVKQPU≥VQIGVCT
CPF≥UVQTKPI≥VCO≥KP≥C≥NQECVKQP≥VCV≥[QW≥ECP≥RTKPV≥HTQO≥6KU≥YKNN≥FGRGPF≥FKTGEVN[≥QP≥QY≥VC
TGFWEVKQP≥QRGTCVKQP≥QP≥sum≥KU≥CPFNGF
```

Data Decomposition

Before I get started on data decomposition, I want to make sure that you haven't skipped down to this section without reading the previous section on task decomposition. There is a lot of good stuff in that section and I want to make sure you've covered it and absorbed it. Many of the things that are covered there will apply directly to data decomposition, and I won't likely repeat them here. So, even if you think you will only ever be working on data decomposition solutions for the rest of your programming career, be sure to read the section on task decomposition. You'll get a better understanding of the problems that are shared between the two decomposition methods.

When you start to examine a serial application for transformation into an equivalent concurrent solution, the first feature of the computations you might identify is that the execution is dominated by a sequence of update operations on all elements of one or more

How should you divide the data into chunks?

Partitioning the global data structure into chunks is at the heart of a data decomposition design. The mechanics for doing this will depend mainly on the type of data structure you are decomposing. For concreteness, I'll deal with one- and two-dimensional arrays as examples. The ideas here can be applied to other structures with a little ingenuity on your part.

Since each chunk of data will have an associated task, many of the same criteria that we had when defining tasks for task decomposition can be applied to the chunks of data decomposition. Specifically, make sure you have at least one chunk per thread (more is probably better) and ensure that the amount of computation that goes along with that chunk is sufficient to warrant breaking out that data as a separate chunk (now, aren't you glad you read the task decomposition section before starting into this section?).

With arrays of elements, you can divide the data into chunks at the individual element level, at the row or column level, as groups of rows or columns, or blocks of nonoverlapping subranges of rows and columns. Figure 2-2 shows a 4ñ4 array divided into chunks in several different ways.

FIGURE 2-2. Array decomposition examples

The amount of computation required by the associated task will be in direct proportion to the number of elements in a chunk. As stated before, this is known as the granularity of the computation and is exactly like what we had when we were considering how to define tasks.

FIGURE 2-3. Volume-to-surface ratio examples

Irregular shapes may be necessary due to the irregular organization of the data. You need to be more vigilant with chunks of irregular shapes to ensure that a good load balance can be maintained, as well as a high enough granularity within the chunk to lessen the impact of unavoidable overheads.

You may need to revise your decomposition strategy after considering how the granularity and shape of the chunks affect the exchange of data between tasks. The division of data structures into chunks influences the need to access data that resides in another chunk. The next section develops ideas about accessing neighboring data that you should consider when deciding how to best decompose a data structure into chunks.

FIGURE 2-4. Using ghost cells to hold copied data from a neighboring chunk

Accessing data as needed takes full advantage of shared memory communication between threads and the logic of the original serial algorithm. You also have the advantage of being able to delay any coordination between threads until the data is needed. The downside is that you must be able to guarantee that the correct data will be available at the right time. Data elements that are required but located within a neighboring chunk may be in the process of receiving updates concurrently with the rest of the elements in the neighboring chunk. If the local chunk requires the "old" values of nonlocal data elements, how can your code ensure that those values are not the "new" values? To answer this question or to know whether we must even deal with such a situation, we must look at the possible interactions between the exchange of data from neighboring chunks and the update operation of local chunk elements.

If all data is available at the beginning of tasks and that data will not change during the update computation, the solution will be easier to program and more likely to execute efficiently. You can either copy relatively small amounts of data into ghost cells or access the unchanging data through shared memory. In order to perform the copy of nonlocal data, add a data gathering (exchange) phase before the start of the update computations. Try to minimize the execution time of the data-gathering phase, since this is pure overhead that was not part of the original serial code.

If nonlocal data will be accessed (or copied) during update computations, you will need to add code to ensure that the correct data will be found. Mixing exchange and update computations can complicate the logic of your application, especially to ensure correct data is retrieved. However, the serial application likely had this requirement, too, and the solution to the need for accessing correct data concurrently should simply follow the serial algorithm as much as possible.

For example, if you are modeling the distribution of heat from a source through a metal plate, you can simulate the plate by a two-dimensional array of current temperatures at discrete spatial locations. At each time step of the computation, the new value of each discrete location is the average of the current temperature and the temperature of some set of neighboring cells. Since this calculation will update the current temperature of a cell and skew the results of other

FIGURE 2-5. Game of Life example; black dots represent "live" cells

The state of cells can change from one generation to the next according to the following rules:

> *The neighbors of a cell are the eight cells that border the given cell horizontally, vertically, and diagonally.*
>
> *If a cell is alive but has one or fewer neighboring cells alive, or has four or more neighboring cells alive, the cell will die of starvation or overcrowding, respectively, in the next generation.*
>
> *Living cells with two or three living neighbors will remain alive in the next generation.*
>
> *If a cell is dead and has exactly three neighboring cells that are alive, there will be a birth and the dead cell will become alive in the next generation. All other dead cells will remain dead in the next generation.*
>
> *All births and deaths take place at exactly the same time. This means that a cell that will be killed is still counted to help give birth to a neighboring cell.*

In theory, the grid should be unbounded, but with only finite memory available in computer platforms, the size of the grid "universe" is usually restricted. Still, a very large two-dimensional array can hold current and successive generations of a simulation. The computations to decide whether any cell lives, dies, or is born is independent of the computations of any other cell. Thus, with the very large two-dimensional array and the update of elements in that array, the Game of Life simulation is an excellent candidate for a data decomposition design.

Example 2-3 shows serial code for updating the status of each cell into the next generation from the current generation. I assume that `Grid≥KU≥C≥FCVC≥UVTWEVWTG≥VCIV≥KU≥GUUGPVKCNN[≥C≥VYQ`
`FKOGPUKQPCN≥CTTC[≥VCIV≥ECP≥QNF≥VCI≥XCNWGU≥QH≥VCI≥WUGTFGHKPGF≥EQPUVCPVU≥ALIVE≥CPF≥DEAD≥6CI`
`ITKF≥CTTC[≥CIU≥TQYU≥KPFGZGF≥D[≥o≥VQ≥N+1≥CPF≥EQNWOPU≥KPFGZGF≥D[≥o≥VQ≥M+1≥6CI≥GZVTC≥TQYU≥CPF`
`EQNWOPU≥YQUG≥EGNNU≥CTG≥PQV≥EQPUKFGTGF≥HQT≥WRFCVG≥KP≥VCI≥IKXGP≥TQWVKPG≥ECP≥DG≥VQWIV≥QH≥CU`
`DQWPFCT[≥EGNNU≥QH≥VCI≥pWPKXGTUGq≥VCIV≥CTG≥CNYC[U≥FGCF≥VKIPM≥QH≥KV≥CU≥C≥YCNN≥QH≥RQKUQP≥VCIV≥ECPoV`
`DG≥DTGCECIF≥$≥CFFKPI≥VCIUG≥DQWPFCT[≥EGNNU≥+≥TGOQXG≥VCI≥PGGF≥VQ≥FGCN≥YKVI≥VCI≥ITKF≥GFIGU`
`CU≥C≥URGEKCN≥ECUG≥#HVGT≥EQORWVKPI≥VCI≥PGKIDQT≥EQWPV≥HQT≥C≥EGNN≥VCI≥FKURQUKVKQP≥QH≥VCIV≥EGNN≥KP`
`VCI≥PGZV≥IGPGTCVKQP≥KU≥FGEKFGF≥DCUGF≥QP≥VCI≥TWNGU≥LWUV≥IKXGP≥#U≥YKVI≥VCI≥VCUM≥FGEQORQUKVKQP`
`GZCORNG≥+≥YQPoV≥ETGCVG≥C≥VTJGCFGF≥XGTUKQP≥QH≥VCI≥EQFG≥DWV≥[QWoTG≥YGNEQOG≥VQ≥IKXG≥KV≥C≥VT[≥QP`
`[QWT≥QYP≥DCUGF≥QP≥VCI≥FKUEWUUKQP≥VCIV≥HQNNQYU`

EXAMPLE 2-3. Game of Life routine to compute next generation configuration from current

```
void computeNextGen (Grid curr, Grid next, int N, int M)
{
  int count;
  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
      count = 0;
      if (curr[i-1][j-1] == ALIVE) count++; // NW neighbor
      if (curr[i-1][j]   == ALIVE) count++; // N neighbor
      if (curr[i-1][j+1] == ALIVE) count++; // NE neighbor
      if (curr[i][j-1]   == ALIVE) count++; // W neighbor
      if (curr[i][j+1]   == ALIVE) count++; // E neighbor
      if (curr[i+1][j-1] == ALIVE) count++; // SW neighbor
      if (curr[i+1][j]   == ALIVE) count++; // S neighbor
      if (curr[i+1][j+1] == ALIVE) count++; // SE neighbor

      if (count <= 1 || count >= 4)
         next[i][j] = DEAD;
      else if (curr[i][j] == ALIVE && (count == 2 || count == 3))
         next[i][j] = ALIVE;
      else if (curr[i][j] == DEAD && count == 3)
         next[i][j] = ALIVE;
      else next[i][j] = DEAD;
    }
  }
  return;
}
```

```
9CV≥KU≥VG≥NCTIG≥FCVC≥UVTWEVWTG≥KP≥VKU≥CRRNKECVKQP≥CPF≥QY≥ECP≥[QW≥FKXKFG≥KV≥KPVQ≥EWPMU!≥+oXG
CNTGCF[≥IKXGP≥CYC[≥VG≥HCEV≥VCV≥YG≥ECP≥FKXKFG≥VG≥ITKF≥KPVQ≥EWPMU≥YQUG≥VCUMU≥ECP≥DG≥CUUKIPGF
VQ≥VJGCFU≥9CV≥KU≥VG≥DGUV≥YC[≥VQ≥RGTHQTO≥VG≥FKXKUKQP!≥4GHGT≥VQ≥(KIWTG≥≥DWV≥VKPM≥DKIIGT
)TQWRU≥QH≥TQYU≥QT≥ITQWRU≥QH≥EQNWOPU≥QT≥DNQEMU≥UGGO≥VG≥OQUV≥PCVWTCN≥FKXKUKQPU≥;QW≥UQWNF
CPVKEKRCVG≥VG≥SWGUVKQP≥QH≥YKE3GZECIPIGU≥QH≥FCVC≥GCE3RQVGPVKCN≥FKXKUKQP≥UVTCVGI[≥YKNN≥GPVCKN
CPF≥NGV≥VCV≥KPHNWGPEG≥[QWT≥CPUYGT

1VGT≥HCEVQTU≥VQ≥EQPUKFGT≥CTG≥VG≥NC[QWV≥QH≥VG≥ITKF≥KP≥OGOQT[≥YKE3OC[≥DG≥CPVKEKRCVGF≥YKVKP
VG≥UGTKCN≥EQFG≥CNTGCF[≥CPF≥QY≥GCE3FKXKUKQP≥UECOG≥OKIV≥CFXGTUGN[≥CHHGEV≥VG≥IKXGP≥NC[QWV
6G≥COQWPV≥QH≥PGY≥EQFG≥CPF≥NKPG≥ECIPIGU≥PGGFGF≥VQ≥VTCPUHQTO≥VG≥UQWTEG≥EQFG≥ECP≥DG≥C≥HCEVQT
KPHNWGPEKPI≥[QWT≥FGEKUKQP≥&KXKFKPI≥D[≥TQYU≥YQWNF≥TGSWKTG≥OQFKHKECVKQPU≥VQ≥YQWNF≥FKXKFKPI
D[≥EQNWOPU≥YQWNF≥TGSWKTG≥OQFKHKECVKQPU≥VQ≥YQWNF≥TGSWKTG≥CPF≥WUKPI≥C≥DNQEM≥FKXKUKQP≥YQWNF≥TGSWKTG
[QW≥VQ≥OQFKH[≥DQV3VG≥i≥CPF≥j≥NQQRU

(QT≥C≥EQUGP≥FCVC≥FGEQORQUKVKQP≥YCV≥GZECIPIG≥QH≥FCVC≥DGVYGGP≥VCUMU≥KU≥TGSWKTGF≥CPF≥QY
YKNN≥VKU≥DG≥CEEQORNKUGF!≥5KPEG≥[QW≥PGGF≥CEEGUU≥VQ≥CNN≥GKIV≥PGKIDQTU≥QH≥C≥IKXGP≥EGNN≥KH≥CP[
PGKIDQT≥EGNNU≥CTG≥KP≥C≥FKHHGTGPV≥EWPM≥VGTG≥YKNN≥PGGF≥VQ≥DG≥UQOG≥HQTO≥QH≥FCVC≥GZECIPIG
*QYGXGT≥VG≥EQFG≥WUGU≥VG≥curr≥ITKF≥HQT≥EQWPVKPI≥PGKIDQTU≥CPF≥KU≥TGCFQPN[≥CE3VCUM≥ECP
UKORN[≥CEEGUU≥VG≥FCVC≥YCP≥KV≥KU≥PGGFGF≥YKVQWV≥HGCT≥QH≥IGVVKPI≥VG≥YTQPI≥XCNWG≥QT≥ECWUKPI≥C
FCVC≥TCEG≥YKV3CPQVGT≥VCUM≥1P≥VG≥QVGT≥CPF≥KH≥VG≥TWNGU≥QH≥EQORWVKPI≥VG≥PGZV≥IGPGTCVKQP
CNNQYGF≥WU≥VQ≥OCMG≥ECIPIGU≥YKVKP≥VG≥UCOG≥ITKF≥VCV≥YCU≥DGKPI≥WUGF≥VQ≥EQWPV≥PGKIDQTU≥KV
```

```
count≥KU≥C≥VGORQTCT[≥YQTM≥XCTKCDNG≥UQ≥C
NQECN≥EQR[≥RGT≥VJGCF≥VCMGU≥ECTG≥QH≥VCV≥CE≥VJGCF≥YKNN≥CNUQ≥PGGF≥C≥EQR[≥QH≥VC≥VYQ≥NQQR
KVGTCVKQP≥XCTKCDNGU≥6C≥INQDCN≥CTTC[U≥CTG≥UCHG≥UKPEG≥curr≥KU≥QPN[≥TGCF≥CPF≥VC≥WRFCVGU≥VQ≥VC
GNGOGPVU≥KP≥VC≥next≥ITKF≥YKNN≥PQV≥QXGTNCR
```

Concurrent Design Models Wrap-Up

There often aren't clear, discrete steps to follow when developing a task or data decomposition solution. When considering how to answer the design element questions for your chosen design model, some of the decisions you make will be based on answers to questions that follow and answers that have come before. I hope you got that sense from the discussion of each example given. Even though I had to write down the questions sequentially, you may need to consider more than one thing at a time to devise an efficient concurrent solution.

DESIGN FACTOR SCORECARD

For most of the algorithms that are discussed and analyzed in Chapters 6 through 10, I will include a "Design Factor Scorecard" to discuss the concurrent algorithm and its implementation on four key factors. These factors are taken from Mattson's *Patterns for Parallel Programming* where they are used as criteria concurrent programmers need to keep in mind when designing and implementing parallel applications through the parallel programming patterns that are presented. For our purposes, I've redefined these terms slightly from how Mattson et al. originally used them. My interpretations of these terms are given shortly.

It is my contention that programmers of concurrent algorithms should keep each of these factors in mind, along with their relative importance to each other and the tradeoffs between stressing one factor over another when designing and implementing concurrent algorithms. The merits and tradeoffs possible for the algorithms and code given in later chapters will also be presented in the Design Factor Scorecard section after the descriptions of each algorithm presented.

Efficiency

Your concurrent applications must run quickly and make good use of processing resources. With regard to a concurrent algorithm, efficiency will examine the overhead computations that

you must add to ensure a correct execution, how alternative arrangements of threads or organizations of tasks might work better or worse, and what other problems there could be with the performance of the threaded application.

Simplicity

The simpler your concurrent algorithms are, the easier they will be to develop, debug, verify, and maintain. In terms of concurrent code based on a serial version, discussions of simplicity will focus on how much extra code you have to add to achieve a concurrent solution and how much of the original structure of the serial algorithm remains.

Portability

One of the goals of this book is to be as agnostic as possible with regard to available threading models and which models are used to implement the solution algorithms presented. Portability discussions will examine the tradeoffs that you could encounter if you used a different threading model from the one used in this text. While this book is primarily dedicated to the design and exploration of multithreaded codes, one of the options discussed under portability will be distributed-memory variations of the algorithms.

Scalability

Because the number of cores will only increase as time passes, your concurrent applications should be effective on a wide range of numbers of threads and cores, and sizes of data sets. Scalability refers to what you should expect with regard to how a given concurrent algorithm will behave with changes in the number of cores or size of data sets.

My Two Cents' Worth on the Factors

For me, scalability is the most important of these four factors, with efficiency a close second. This means that I will typically try to design a concurrent algorithm that will maintain its level of performance as the number of cores and threads increases to the detriment of a simpler or more portable algorithm. In order to gain that scalability, the algorithm and its implementation must be efficient, so that is my secondary goal.

There have been many times when the scalability of a concurrent application I've written has peaked and flattened out. This was usually due to the requirements of the algorithm and the paraphernalia provided by the threading model being used. In these cases, I would ask myself if it is worth the extra work to try to discover an alternative (which may not exist) that might scale better, or to rewrite the whole thing in terms of a different threading or design model. These are some of the tradeoffs you face when tackling concurrent algorithms.

What's Not Parallel

In the chapters and pages that follow, we're going to explore quite a few things that can be executed concurrently. Before we get to those, though, I want to impress upon you that not

NOTE

One of the more famous illustrations of a situation that cannot be made parallel is cited in Fred Brooks's 1995 book, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley Professional). The nine-month gestation period for a human is a serial process—you can't get a baby in one month by assigning nine women to the job.

On the other hand, if you wanted to raise a baseball team of players from cradle to Wrigley Field in the shortest amount of time, you could employ 9 women (10 if you want a designated hitter). This would give you a newborn starting lineup after nine months. Asking one woman to do all the work would require about nine years, barring twins and such, and the pitcher (firstborn) would be starting her Little League career when the right fielder (ninth-born) was just arriving.

Algorithms with State

The first example of code that cannot be executed concurrently is algorithms, functions, or procedures that contain a *state*. That is, something kept around from one execution to the next. For example, the seed to a random number generator or the file pointer for I/O would be considered state. Algorithms with state cannot be made directly concurrent, and whenever you encounter such code, a red flag should go up when you are considering concurrency. However, you may be able to take steps to render the code thread-safe, which may be sufficient.

You can make state-filled code thread-safe by adding some form of synchronization or writing the code to be reentrant (i.e., it can be reentered without detrimental side effects while it is already running). The former option will serialize all concurrent executions of the synchronized code (and add unnecessary overhead when not called concurrently), while the latter option may not be possible if the update of global variables is part of the code.

If the variable(s) holding the state does not have to be shared between threads, you can use TLS to overcome this dependence. Using TLS, each thread would have a separate copy of the state variable(s) (accessed in exactly the same way across all threads) to ensure there are no data races on the variable(s). Thus, each thread can have a different random number seed and use the same code to generate a separate stream of numbers that will not interfere with any other thread's seed.

Recurrences

Recurrence relations within loops feed information forward from one iteration to the next. Prime examples of this are time-stepping loops and convergence loops. No matter how many tea leaves we read, tarot cards we consult, or Magic Eight Ball apps we write, we can't parse out future time steps to multiple threads for concurrent execution.

A simple code example of a recurrence is given in Example 2-4. The recurrence shown is the read access of the `a[i-1]`≥GNGOGPV≥VQ≥V≥YCU≥EQORWVGF≥KP≥VG≥RTGXKQWU≥KVGTCVKQP

EXAMPLE 2-4. Recurrence relation on array access

```
for (i = 1; i < N; i++)
  a[i] = a[i-1] + b[i];
```

7PHQTVWPCVGN[≥OQUV≥TGEWTTGPEGU≥ECPPQV≥DG≥OCFG≥EQPEWTTGPV≥2TGHKZ≥UWO≥KU≥C≥URGEKCN≥ECUG≥QH≥C
TGEWTTGPEG≥VQ≥V≥ECP≥DG≥OCFG≥VQ≥TWP≥EQPEWTTGPVN[≥UGG≥%QRVGT≥≥HQT≥OQTG≥FGVCKNU≥QP≥EQPEWTTGPV
CNIQTKVOU≥HQT≥RTGHKZ≥UECP

+H≥[QWoXG≥IQV≥C≥TGEWTTGPEG≥TGNCVKQPUHR≥VQ≥V≥KU≥C≥QVURQV≥KP≥[QWT≥EQFG≥NQQM≥HQT≥C≥RQKPV≥pKIGTq
KP≥VG≥ECNN≥VTGG≥VQ≥V≥YQWNF≥KPENWFG≥GZGEWVKQP≥QH≥VG≥TGEWTTGPEG≥6JGCF≥VGTG≥YGTG≥RQUUKDNG

Induction Variables

Induction variables are variables that are incremented on each trip through a loop. Most likely, these are index variables that do not have a one-to-one relation with the value of the loop iterator variable. Example 2-5 shows a code segment with two induction variables, `i1`≥CPF≥`i2`

EXAMPLE 2-5. Induction variables

```
i1 = 4;
i2 = 0;
for (k = 1; k < N; k++) {
  B[i1++] = function1(k,q,r);
  i2 += k;
  A[i2] = function2(k,r,q);
}
```

#U≥VHU≥EQFG≥UVCPFU≥GXGP≥KH≥VG≥ECNNU≥VQ≥`function1()`≥CPF≥`function2()`≥CTG≥KPFGRGPFGPV≥VGTGoU≥PQ
YC[≥VQ≥VTCPUHQTO≥VHU≥EQFG≥HQT≥EQPEWTTGPE[≥YKVQWV≥C≥HGY≥TCFKECN≥CNVGTCVKQPU≥VQ≥VG≥UGTKCN
UQWTEG≥5RGEKHKECNN[≥[QW≥YQWNF≥PGGF≥VQ≥TGRNCEG≥VG≥CTTC[≥KPFGZ≥KPETGOGPV≥GZRTGUUKQPU≥YKV3C
ECNEWNCVKQP≥DCUGF≥UQNGN[≥QP≥VG≥XCNWG≥QH≥VG≥NQQR≥KVGTCVQT≥XCTKCDNG

9KVQWV≥OWE3UVTCKP≥QP≥[QWT≥DTCKP≥+oO≥UWTG≥[QW≥ECP≥UGG≥VQ≥V≥[QW≥EQWNF≥TGYTKVG≥VG≥HKTUV
UVCVGOGPV≥KP≥VG≥NQQR≥CU≥UQ≥YP≥KP≥ZCORNG≥

EXAMPLE 2-6. Solution for first induction variable increment

```
B[k+4] = function1(k,q,r);
```

k≥6ØU≥[QW≥ECP≥WUG≥VØ≥EQFG
KP≥ZCORNG≥≥KP≥C≥EQPEWTTGPV≥XGTUKQP≥QH≥VØ≥NQQR

EXAMPLE 2-7. Solution for second induction variable increment

```
i2 = (k*k + k)/2;
A[i2] = function2(k,r,q);
```

6ØU≥KU≥C≥EQPVTKXGF≥GZCORNG≥QH≥EQWTUG≥;QWT≥UKVWCVKQPU≥OC[≥PQV≥DG≥UQ≥PGCV≥CPF≥ENGCP≥VQ≥WIJ
+≥QRG≥VØ[≥CTG

Reduction

Reductions take a collection (typically an array) of data and reduce it to a single scalar value through some combining operation. To remove this dependency, the operation must be both associative and commutative, such as addition, multiplication, or some logical operations. The loop in the code fragment shown in Example 2-8 will find the sum of all elements of the c
CTTC[≥CU≥YGNN≥CU≥VØ≥NCTIGUV≥OCZKOWO≥GNGOGPV≥KP≥VØ≥CTTC[

EXAMPLE 2-8. Reduction code

```
sum = 0;
big = c[0];
for (i = 0; i < N; i++) {
  sum += c[i];
  big = (c[i] > big ? c[i] : big); // maximum element
}
```

#V≥HKTUV≥INCPEG≥VØ≥NQQR≥ECPPQV≥DG≥OCFG≥VQ≥GZGEWVG≥EQPEWTTGPVN[≥ÇE3GNGOGPV≥KP≥VWTP≥KU
CFFGF≥VQ≥VØ≥TWPPKPI≥VQVCN≥CPF≥EQORCTGF≥VQ≥VØ≥NCTIGUV≥GNGOGPV≥HQWPF≥UQ≥HCT≥TGRNCEKPI≥VØ≥V
NCTIGUV≥HQWPF≥GNGOGPV≥CU≥PGGFGF≥6ØU≥KU≥CNN≥FQPG≥KP≥VØ≥QTFGT≥QH≥VØ≥KPETGOGPVKPI≥KPFGZ≥XCTKCDNG
i≥*QYGXGT≥PQVKEG≥VØV≥VØ≥TGUWNVU≥YQWNF≥DG≥GZCEVN[≥VØ≥UCOG≥YKVØP≥NKOKVU≥QH≥TQWPFKPI≥CPF

sum≥CPF≥big≥KP≥VQ≥RTGEGFKPI≥GZCORNG≥KP≥NQECN
UVQTCIG≥OGZV≥EQODKPG≥GCE3RCTVKCN≥TGUWNV≥KPVQ≥C≥INQDCN≥TGUWNV≥VCMKPI≥ECTG≥VQ≥U[PEJTQPK\G≥CEEGUU
VQ≥VQ≥UQTGF≥XCTKCDNGU≥1H≥EQWTUG≥KH≥[QWoTG≥VJGCFKPI≥[QWT≥NQQR≥YKV31RGP/2≥QT≥+PVGN≥6$
[QW≥LWUV≥PGGF≥VQ≥OCMG≥WUG≥QH≥VQ≥reduction≥ENCWUG≥QT≥parallel_reduce≥CNIQTKVO

Loop-Carried Dependence

The final example of code that poses a problem to our efforts to write concurrent applications is known as loop-carried dependence. This dependence occurs when results of some previous iteration are used in the current iteration. Typically, this situation will be evidenced by references to the same array element on both the left- and righthand sides of assignments and a backward reference in some righthand side use of the array element. Obviously, the iterations of such loops are not completely independent. Example 2-9 shows a code fragment that updates corresponding elements of the a≥CPF≥b≥CTTC[U≥DWV≥VQ≥WRFCVG≥QH≥a≥GNGOGPVU≥TGNKGU≥QP
RTGXKQWUN[≥WRFCVGF≥GNGOGPVU≥HTQO≥VQ≥b≥CTTC[

EXAMPLE 2-9. Loop-carried dependence code

```
for (k = 5; k < N; k++) {
  b[k] = DoSomething(k);
  a[k] = b[k-5] + MoreStuff(k);
}
```

&KXKFKPI≥UWE3NQQR≥KVGTCVKQPU≥KPVQ≥VCUMU≥RTGUGPVU≥VQ≥RTQDNGO≥QH≥TGSWKTKPI≥GZVTC≥U[PEJTQPK\CVKQP
VQ≥GPUWTG≥VQV≥VQ≥DCEMYCTF≥TGHGTGPEGU≥QXG≥DGGP≥EQORWVGF≥DGHQTG≥VQ[≥CTG≥WUGF≥KP
EQORWVCVKQP≥QH≥VQ≥EWTTGPV≥KVGTCVKQP≥4GEWTTGPEG≥KU≥C≥URGEKCN≥ECUG≥QH≥C≥NQQRECTTKGF≥FGRGPFGPEG
YQTG≥VQ≥DCEMYCTF≥TGHGTGPEG≥KU≥VQ≥KOOGFKCVG≥RTGXKQWU≥KVGTCVKQP≥6QTG≥KU≥PQ≥YC[≥VQ
GHHKEKGPVN[≥GZGEWVG≥UWE3NQQR≥KVGTCVKQPU≥EQPEWTTGPVN[≥UKPEG≥YCKVKPI≥HQT≥VQ≥DCEMYCTF≥TGHGTGPEGU
VQ≥DG≥TGUQNXGF≥ECP≥TGSWKTG≥C≥QHV[≥COQWPV≥QH≥U[PEJTQPK\CVKQP

(QT≥GZCORNG≥KH≥[QWT≥DCEMYCTF≥TGHGTGPEG≥URCPU≥HKXG≥KVGTCVKQPU≥CU≥KP≥ZCORNG≥≥[QW≥ECP
FKXKFG≥WR≥VQ≥NQQR≥KVGTCVKQPU≥KPVQ≥EWPMU≥YKV3HKXG≥KVGTCVKQPU≥KP≥GCE3EWPM≥1PEG≥VQ≥HKTUV
KVGTCVKQP≥QH≥VQ≥HKTUV≥EWPM≥QU≥EQORNGVGF≥VQ≥HKTUV≥KVGTCVKQP≥KP≥VQ≥UGEQPF≥EWPM≥ECP≥UVCTV
UKPEG≥VQ≥FGRGPFGPEG≥QH≥VQ≥HKTUV≥KVGTCVKQP≥QH≥VQ≥UGEQPF≥EWPM≥KVGTCVKQP≥≥QU≥DGGP≥UCVKUHKGF
;QW≥ECP≥FCKU[EQKP≥NQQR≥EWPMU≥CPF≥VJGCFU≥NKMG≥VKU≥DWV≥KV≥YKNN≥CNN≥PGGF≥UQOG≥OCLQT≥EQFG
OQFKHKECVKQPU≥CPF≥U[PEJTQPK\CVKQP≥VQ≥GPUWTG≥VQV≥CNN≥RTGTGSWKUKVG≥FGRGPFGPEGU≥QXG≥DGGP
UCVKUHKGF≥DGHQTG≥GZGEWVKQP≥ECP≥UVCTV≥QP≥GCE3UGRCTCVG≥KVGTCVKQP

Not-so-typical loop-carried dependence

The loop in the code fragment given in Example 2-10 cannot be made parallel because wrap≥KU
ECTTKGF≥HTQO≥QPG≥KVGTCVKQP≥VQ≥VC≥PGZV≥6K≥KU≥KU≥NQQRECTTKGF≥FGRGPFGPEG≥VC≥V≥FQGUPoV≥HQNNQY≥VC≥
V[RKECN≥HQTOCV≥KPXQNXKPI≥QDXKQWU≥DCEMYCTF≥TGHGTGPEGU≥UKPEG≥VC≥DCEMYCTF≥TGHGTGPEGU≥CTG
pK≥FFGPq≥KP≥VC≥wrap≥XCTKCDNG

EXAMPLE 2-10. Atypical loop-carried dependence

```
wrap = a[0] * b[0];
for (i = 1; i < N; i++) {
  c[i] = wrap;
  wrap = a[i] * b[i];
  d[i] = 2 * wrap;
}
```

(QTVWPCVGN[≥[QW≥ECP≥TGUVTWEVWTG≥VK≥U≥UKORNG≥ECUG≥VQ≥FGHKPG≥wrap≥DGHQTG≥WUG≥KP≥GCE≥KVGTCVKQP≥CPF
ETGCVG≥C≥NQQR≥YQ≥UG≥KVGTCVKQPU≥ECP≥DG≥GZGEWVGF≥EQPEWTTGPVN[≥6K≥U≥KU≥RQUUKDNG≥DGECWUG≥[QW≥ECP
CUUKIP≥VC≥RTQRGT≥XCNWG≥QH≥wrap≥DCUGF≥UQNGN[≥QP≥VC≥XCNWG≥QH≥VC≥NQQR≥KVGTCVQT≥XCTKCDNG
7CORNG≥≥UQ≥YU≥VC≥TGUWNVU≥QH≥VK≥U≥EQFG≥TGUVTWEVWTKPI

EXAMPLE 2-11. Modified loop-carried dependence

```
for (i = 1; i < N; i++) {
  wrap = a[i-1] * b[i-1];
  c[i] = wrap;
  wrap = a[i] * b[i];
  d[i] = 2 * wrap;
}
```

&KUTGICTFKPI≥VC≥EQFG≥WUGF≥VQ≥KORNGOGPV≥VC≥NQQR≥TGRGVKVKQP≥KPKVKCNK\KPI≥KPETGOGPVKPI≥CPF
VGUVKPI≥VC≥NQQR≥KVGTCVQT≥[QW≥YKNN≥PQVKEG≥VC≥V≥VC≥OQFKHKGF≥EQFG≥KU≥PQY≥GZGEW≥VKPI≥
UVCVGOGPVU≥CU≥QRRQUGF≥VQ≥VC≥2 ≥PGGFGF≥KP≥7CORNG≥

4GYTKVKPI≥CP≥GZKUVKPI≥CNIQTKVO≥VQ≥UQOGVK≥PI≥NGUU≥GHHKEKGPV≥KP≥QTFGT≥VQ≥IGV≥C≥DGVVGT≥EC≥PEG≥QH
EQPEWTTGPE[≥OC[≥DG≥PGEGUUCT[≥&QPoV≥WUG≥UQOGVK≥PI≥VC≥V≥KU≥VQQ≥HCT≥CHKGNF≥QH≥VC≥QTKIKPCN
VQ≥WI≥.GUU≥GHHKEKGPV≥UGTKCN≥CNIQTKVOU≥YKNN≥VGPF≥VQ≥CFF≥QXGTC≥CF≥CU≥UGGP≥YC≥P≥EQORCTKPI≥VC≥
PWODGT≥QH≥UVCVGOGPVU≥KP≥VC≥NQQR≥DQFKGU≥KP≥VC≥RTGXKQWU≥GZCORNGU

CHAPTER THREE

Proving Correctness and Measuring Performance

THIS CHAPTER TAKES A LOOK AT TOPICS RELATED TO THE FINAL TWO STEPS of the threading methodology. The first is knowing when your concurrent algorithms will run correctly or at least have a good idea that you've done a good job of designing an error-free concurrent algorithm. The second topic covers some of the ways you can measure how well your concurrent code is executing in parallel. Finally, I've put in a little history review (don't worry – it's short, it's related to the topic of this book, and it never hurts to know where you've been to have a clue about where you might be going).

Verification of Parallel Algorithms

In his 2006 book *Principles of Concurrent and Distributed Programming*, Second Edition (Addison-Wesley), M. Ben-Ari defines an abstraction for formally verifying the correctness and other properties of concurrent algorithms. Unlike other theoretical abstractions in computer science that deal with hardware (e.g., PRAM), Ben-Ari's abstraction deals with how concurrent programs execute. I don't want to get into all the details and justifications for this abstraction here. I recommend that you read Ben-Ari's book for that and for another good resource in concurrent algorithm design. I want to cover just enough for you to understand the principles and the basic idea of the concurrency abstraction and how to prove concurrent algorithms are correct.

For those of you who haven't immediately curled up into a fetal position or flopped onto the floor kicking and screaming at the thought of formal proofs of algorithms (and for those of you who did but have now picked the book up again), let me assure you that I consider this a vital part of concurrent algorithm design and it's probably not as bad as you think. The more time you spend creating correct concurrent algorithms from the get-go, the less time you will spend chasing down errors that only show up on those Thursdays when the dates are prime numbers. This is a great example of the phrase, ®An ounce of prevention equals a pound of cure.¯

Some potential concurrency errors are going to be obvious and you can avoid them easily. Other errors are going to be subtler and may only show up under very rare and very particular circumstances. Do you need to deal with those rare ®corner¯ cases? Yes, you do. Having a familiarity with Ben-Ari's methods and practicing them whenever you are designing concurrent algorithms will sharpen your skills to identify the potential corner cases that your colleagues overlook. Besides, I make use of Ben-Ari's technique throughout the later chapters of this book and if you haven't read the following sections, you won't know what I'm talking about.

The first part of the concurrent abstraction is that *programs are the execution of atomic statements*. An atomic statement is one that cannot be divided into smaller instructions or that cannot be interrupted until the statement is completed. The bottom line is that there is some granularity of executable code at which even the operating system is unable to interrupt and must wait before it can affect the process that is executing that atomic statement. For purposes of the concurrent abstraction, there are different levels of atomicity we might choose to work

FIGURE 3-1. Interleaving two statements within each of two threads

With two threads, three statements per thread yields 20 distinct interleavings, and four statements each would require verification of 70 interleavings. And this is with only two threads. Imagine the number of interleavings using three, four, or eight threads. What is a programmer to do?

Typically, you will find that of the many different possible interleavings, only a few are relevant in proving that the algorithm under consideration has the desired properties. In the set of interleavings enumerated in Figure 3-1, it may be that we only have to be concerned in those

Example

A *critical section* is a portion of code from a concurrent algorithm where shared variables are accessed and an update of these variables is involved. Thus, to avoid a data race on the shared variables, all accesses, read or write, must be restricted to allow only a single thread to execute code in the critical sections at a time. The Critical Section Problem is an exercise to develop a means for mutual exclusion that doesn't use any synchronization objects or primitives that might be defined within a threading model. I prefer the term *critical region* to denote such regions of code, especially since there is a Windows threading object called `CRITICAL_SECTION`

VCV≥ECP≥DG≥WUGF≥HQT≥GPHQTEKPI≥OWVWCN≥GZENWUKQP≥KP≥C≥ETKVKECN≥TGIKQP≥+≥NKMG≥VQ≥CXQKF≥EQPHWUKQP DGVYGGP≥VGTOU≥YCPGXGT≥+≥ECP≥9KNG≥+≥WUG≥VC≥ENCUUKE≥PCOG≥HQT≥VC≥RTQDNGO≥+≥YKNN≥TGHGT≥VQ VCUG≥EQFG≥UGIOGPVU≥CU≥ETKVKECN≥TGIKQPU≥HQT≥VC≥TGUV≥QH≥VC≥DQQM

2TQRQUGF≥UQNWVKQPU≥VQ≥VC≥RTQDNGO≥OWUV≥RTQXG≥VCV≥VC≥HQNNQYKPI≥VYQ≥RTQRGTVKGU≥QNF≥KP≥CNN EKTEWOUVCPEGU

6C≥EQFG≥GPHQTEGU≥OWVWCN≥GZENWUKQP≥6JGCFU≥CTG≥FKUCNNQYGF≥HTQO≥GPVGTKPI≥VC≥ETKVKECN TGIKQP≥YCP≥CPQVCT≥VJGCF≥KU≥QEEWR[KPI≥VC≥ETKVKECN≥TGIKQP≥2NWU≥KH≥VCTG≥CTG≥OWNVKRNG EQPEWTTGPV≥TGSWGUVU≥VQ≥GPVGT≥VC≥ETKVKECN≥TGIKQP≥CPF≥PQ≥QVCT≥VJGCF≥KU≥EWTTGPVN[≥GZGEWVKPI YKVKP≥VC≥TGIKQP≥QPN[≥QPG≥TGSWGUVKPI≥VJGCF≥OWUV≥DG≥CNNQYGF≥VQ≥GPVGT≥VC≥ETKVKECN≥TGIKQP

+H≥C≥VJGCF≥KU≥PQV≥GZGEWVKPI≥YKVKP≥VC≥ETKVKECN≥TGIKQP≥VCV≥VJGCF≥ECPPQV≥RTGXGPV≥CPQVCT VJGCF≥UGGMKPI≥GPVT[≥HTQO≥GPVGTKPI≥VC≥TGIKQP

9CV≥HQNNQYU≥KU≥C≥UVGRD[UVGR≥FGXGNQROGPV≥QH≥UWEEGUUKXG≥CNIQTKVOU≥VQ≥C≥UQNWVKQP≥MPQYP≥CU &GMMGToU≥#NIQTKVO≥$P#TK≥FGXQVGU≥C≥YQNG≥ECRVGT≥VQ≥FGOQPUVTCVG≥VC≥WUG≥QH≥KU EQPEWTTGPE[≥CDUVTCEVKQP≥D[≥TGYQTMKPI≥CP≥KPKVKCN≥CVVGORV≥VQ≥FGTKXG≥&GMMGToU≥#NIQTKVO≥6CTG CTG≥UQOG≥FKHHGTGPEGU≥DGVYGGP≥VC≥CVVGORVGF≥UQNWVKQP≥CNIQTKVOU≥C≥FGUETKDGU≥CPF≥VC≥QPGU≥+ IKXG≥CTG

+≥YKNN≥QPN[≥EQPUKFGT≥VC≥ECUG≥YKV3VYQ≥VJGCFU≥EQORGVKPI≥HQT≥VC≥ETKVKECN≥TGIKQP≥UKPEG≥&GMMGToU #NIQTKVO≥KU≥FGHKPGF≥QPN[≥HQT≥VYQ≥VJGCFU≥CPF≥KV≥MGGRU≥VKPIU≥UKORNG≥1PG≥VJGCF≥YKNN≥DG GZGEWVKPI≥VC≥ThreadZero()≥HWPEVKQP≥CPF≥VC≥QVCT≥YKNN≥GZGEWVG≥VC≥ThreadOne()≥HWPEVKQP≥+P O[≥CPCN[UKU≥QH≥VC≥GZCORNGU≥+≥UCNN≥TGHGT≥VQ≥VCUG≥VJGCFU≥CU≥6≥CPF≥6≥TGURGEVKXGN[

+P≥VC≥UQNWVKQP≥RUGWFQEQFG≥GZCORNGU≥VC≥ETKVKECN≥TGIKQPU≥YKNN≥DG≥TGRTGUGPVGF≥CU≥ECNNU≥VQ≥VC HWPEVKQPU≥CriticalRegionZero≥CPF≥CriticalRegionOne≥+≥UCNN≥CUUWOG≥VCV≥DQV3TGIKQPU≥CEEGUU≥VC UCOG≥UCTGF≥XCTKCDNGU≥CPF≥OWUV≥DG≥GZGEWVGF≥KP≥C≥OWVWCNN[≥GZENWUKXG≥YC[≥9CP≥C≥VJGCF≥KU PQV≥YKVKP≥VC≥ETKVKECN≥TGIKQP≥VCTG≥CTG≥QVCT≥EQORWVCVKQPU≥VQ≥DG≥FQPG≥OtherStuffZero OtherStuffOne

6C≥RWTRQUG≥QH≥VKU≥GZGTEKUG≥KU≥PQV≥VQ≥FGXGNQR≥C≥UQNWVKQP≥VQ≥VC≥%TKVKECN≥5GEVKQP≥2TQDNGO≥9G ECP≥GCUKN[≥GORNQ[≥CP≥CRRTQRTKCVG≥U[PEJQPK\CVKQP≥QDLGEV≥VQ≥NKOKV≥VC≥PWODGT≥QH≥VJGCFU GPVGTKPI≥C≥ETKVKECN≥TGIKQP≥QH≥QWT≥EQFG≥6C≥FGXGNQROGPV≥QH≥EQFG≥VCV≥GXGPVWCNN[≥NGCFU≥VQ≥&GMMGToU #NIQTKVO≥KU≥WUGF≥VQ≥KNNWUVTCVG≥/≥$P#TKoU≥OGVQFU≥QH≥XGTKH[KPI≥VC≥EQTTGEVPGUU≥QH≥EQPEWTTGPV CNIQTKVOU

First Attempt

Example 3-1 shows the first attempt at an algorithm that will enforce mutual exclusion on entering a critical region.

EXAMPLE 3-1. First attempt

```
int threadNumber = 0;

void ThreadZero()
{
  while (TRUE) do {
    while (threadNumber == 1) do {} //spin-wait
    CriticalRegionZero;
    threadNumber = 1;
    OtherStuffZero;
  }
}

void ThreadOne()
{
  while (TRUE) do {
    while (threadNumber == 0) do {} //spin-wait
    CriticalRegionOne;
    threadNumber = 0;
    OtherStuffOne;
  }
}
```

```
6C≥HKTUV≥CVVGORV≥OCMGU≥WUG≥QH≥C≥INQDCN≥XCTKCDNG≥threadNumber≥VQ≥CPPQWPEG≥YKVJ≥VJGCF≥OC[
GPVGT≥VC≥ETKVKECN≥TGIKQP≥6C≥XCNWG≥QH≥VJU≥INQDCN≥KU≥KPKVKCNK\GF≥VQ≥≥VQ≥CNNQY≥6≥HKTUV≥CEEGUU≥VQ≥VC
ETKVKECN≥TGIKQP≥$HQTG≥GPVGTKPI≥VC≥TGIKQP≥C≥VJGCF≥HKTUV≥EC≥EMU≥VQ≥UGG≥YC≥VCT≥KV≥KU≥CNNQYGF≥VQ
GPVGT≥DCUGF≥QP≥VC≥XCNWG≥UVQTGF≥KP≥threadNumber≥+H≥KV≥KU≥VC≥VJGCF≥RTQEGGFU≥KPVQ≥VC≥ETKVKECN
TGIKQP≥CPF≥WRQP≥GZKV≥WRFCVGU≥VC≥threadNumber≥XCTKCDNG≥YKVJ≥VC≥XCNWG≥VCV≥CNNQYU≥VC≥QVCT
VJGCF≥GPVT[≥+H≥VC≥XCNWG≥QH≥threadNumber≥FQGUPoV≥OCVEJ≥VC≥VJGCF≥GPVGTU≥C≥URKPYCKV≥NQQR
WPVKN≥VC≥QVCT≥VJGCF≥CU≥GZGEWVGF≥VC≥ETKVKECN≥TGIKQP≥CPF≥EC≥PIGF≥VC≥XCNWG≥QH≥threadNumber

#UUWOKPI≥C≥UKPING≥EQTG≥YKVJ≥VJGCFU≥VCV≥CTG≥UYCRRGF≥KPVQ≥VC≥RTQEGUUQT≥YG≥ECP≥UGG≥VCV
CP≥GZGEWVKQP≥VTCEG≥HQT≥VJU≥GZCORNG≥EQWNF≥RTQEGGF≥CU≥UQ≥YP≥CTG
```

```
6≥CTTKXGU≥CV≥while≥NQQR

6≥YCKVU≥UKPEG≥threadNumber≥≥

6≥CTTKXGU≥CV≥while≥NQQR

6≥RTQEGGFU≥UKPEG≥threadNumber≥≥

6≥GPVGTU≥ETKVKECN≥TGIKQP

6≥GZKVU≥ETKVKECN≥TGIKQP≥CPF≥UGVU≥threadNumber≥≥

6≥GPVGTU≥ETKVKECN≥TGIKQP
```

```
                                                        while≥NQQR≥DGHQTG≥6≥KU≥CNNQYGF≥VQ≥GZGEWVG
KP≥VQ≥RTQEGUUQT≥6≥YKNN≥KV≥VQ≥URKPYCKV≥NQQR≥+P≥HCEV≥VKU≥KU≥VQ≥UEGPCTKQ≥HQT≥CP[≥KPVGTNGCXKPI
VQV≥UVCTVU≥YKV36≥CPF≥6≥KU≥CNNQYGF≥VQ≥GZGEWVG≥DGHQTG≥6≥QU≥UGW≥threadNumber≥#HVGT≥VQ≥UQTGF
XCTKCDNG≥KU≥UGV≥6≥YKNN≥DG≥IKXGP≥CEEGUU≥VQ≥VQ≥ETKVKECN≥TGIKQP≥CPF≥6≥YKNN≥DG≥DNQEMGF

/WVWCN≥GZENWUKQP≥KU≥IWCTCPVGGF≥1PN[≥QPG≥VJGCF≥CV≥C≥VKOG≥KU≥CNNQYGF≥KPVQ≥VQ≥ETKVKECN≥TGIKQP
YKNG≥VQ≥QVQT≥VJGCF≥YKNN≥PQV≥DG≥CNNQYGF≥KPVQ≥VQ≥TGIKQP≥WPVKN≥VQ≥VJGCF≥CNTGCF[≥GZGEWVKPI
YKVKP≥VQ≥TGIKQP≥QU≥HKPKUGF≥CPF≥TGUGV≥VQ≥threadNumber≥HNCI≥VQ≥VQ≥CRRTQRTKCVG≥XCNWG

9QV≥CDQWV≥VQ≥ECUG≥YQTG≥6≥GPVGTU≥VQ≥ETKVKECN≥TGIKQP≥FQGU≥VQ≥CUUKIPGF≥EQORWVCVKQP≥GZKVU
VQ≥ETKVKECN≥TGIKQP≥CPF≥VQP≥PGGFU≥VQ≥IGV≥DCEM≥KPVQ≥VQ≥ETKVKECN≥TGIKQP≥DGHQTG≥6≥QU≥QF≥VQ
EQPEG≥VQ≥OCMG≥C≥RCUU≥VJQWI3VQ≥ETKVKECN≥TGIKQP!≥6≥YKNN≥PQV≥DG≥CDNG≥VQ≥TGGPVGT≥VQ≥ETKVKECN
TGIKQP≥WPVKN≥6≥QU≥EQORNGVGF≥VQ≥GZGEWVKQP≥QH≥OtherStuffZero≥IQPG≥VJQWI3VQ≥ETKVKECN≥TGIKQP
CPF≥TGUGV≥VQ≥threadNumber≥HNCI≥9G≥PQY≥QXG≥VQ≥ECUG≥YQTG≥QPG≥VJGCF≥VQV≥KU≥PQV≥KP≥VQ
ETKVKECN≥TGIKQP≥ECP≥RTGXGPV≥CPQVQT≥VJGCF≥HTQO≥GPVGTKPI≥6KU≥XKQNCVGU≥VQ≥UGEQPF≥RTQRGTV[≥C
EQTTGEV≥UQNWVKQP≥OWUV≥QXG

+P≥HCEV≥VQ≥EKTEWOUVCPEGU≥ETGCVGF≥D[≥VQ≥UKORNG≥UQNWVKQP≥RTQRQUGF≥KP≥ZCORNG≥≥HQTEG≥C
NQEMUVGR≥CNVGTPCVKQP≥QH≥CNNQYKPI≥VJGCFU≥VQ≥GPVGT≥CPF≥GZKV≥VQ≥ETKVKECN≥TGIKQP≥#P[≥VJGCF≥VQV
CVVGORVU≥VQ≥FGXKCVG≥HTQO≥VQV≥TKIKF≥CNVGTPCVKQP≥KU≥UV[OKGF≥CPF≥OWUV≥GZGEWVG≥VQ≥URKPYCKV≥WPVKN
VQ≥QVQT≥VJGCF≥QU≥EQPIGF≥VQ≥XCNWG≥QH≥VQ≥  threadNumber≥HNCI≥%QORQWPFKPI≥VKU≥RTQDNGO≥KU
VQ≥RQUUKDKNKV[≥QH≥QPG≥VJGCF≥VGTOKPCVKPI≥YQVQT≥D[≥FGUKIP≥D[≥CEEKFGPV≥QT≥D[≥HQWN≥RNC[
DGHQTG≥VQ≥QVQT≥+H≥VQ≥UWTXKXKPI≥VJGCF≥CVVGORVU≥VQ≥GPVGT≥VQ≥ETKVKECN≥TGIKQP≥YKVQWV≥VQ≥FGCF
VJGCF≥QXKPI≥C≥EQPEG≥VQ≥EQPIG≥VQ≥XCNWG≥QH≥VQ≥ threadNumber≥HNCI≥VKU≥YKNN≥ETGCVG≥C≥FGCFNQEM
QP≥VQ≥UWTXKXKPI≥VJGCF

6KU≥KU≥PQV≥CP≥CEEGRVCDNG≥UVCVG≥QH≥CHHCKTU≥.GVoU≥UGG≥KH≥YG≥ECP≥FQ≥DGVVGT≥KP≥VQ≥PGZV≥CVVGORV
```

Second Attempt

To get around the problem of the lockstep execution requirement of the first proposed solution to the Critical Section Problem, the second solution uses a global flag, one per thread, to denote when a thread is executing within the critical region. Example 3-2 shows the pseudocode for the second attempt.

EXAMPLE 3-2. Second attempt

```
int Thread0inside = 0;
int Thread1inside = 0;
```

while≥NQQR≥EQPFKVKQPCN≥GXCNWCVKQP≥YGPGXGT≥VG≥QVGT≥VJGCF≥KU≥TWPPKPI≥VG≥while≥NQQR≥VGUV
YKVKP≥KVU≥QYP≥HWPEVKQP

Third Attempt

The previous attempt used what I would call ®selfish¯ threads. Before entering the critical region, a thread sets its status flag announcing that it is proceeding into the critical region. The only cooperation between the threads is checking the status of the other selfish thread before barreling into the critical region. The third attempt uses more genteel threads that take into account the other thread's intention to enter the critical region. Example 3-3 shows the pseudocode for the third attempt.

EXAMPLE 3-3. Third attempt

```
int Thread0WantsToEnter = 0;
int Thread1WantsToEnter = 0;

void ThreadZero()
{
  while (TRUE) do {
    Thread0WantsToEnter = 1;
    while (Thread1WantsToEnter) do {}  // spin-wait
    CriticalRegionZero;
    Thread0WantsToEnter = 0;
    OtherStuffZero;
  }
}

void ThreadOne()
{
  while (TRUE) do {
    Thread1WantsToEnter = 1;
    while (Thread0WantsToEnter) do {}
    CriticalRegionOne;
    Thread1WantsToEnter = 0;
    OtherStuffOne;
  }
}
```

+P≥VKU≥CVVGORV≥GCE≥VJGCF≥FGENCTGU≥VG≥FGUKTG≥VQ≥GPVGT≥VG≥ETKVKECN≥TGIKQP≥D[≥UGVVKPI≥VG
CRRTQRTKCVG≥FGUKTG≥HNCI≥*QYGXGT≥DGHQTG≥RTQEGGFKPI≥KPVQ≥VG≥ETKVKECN≥TGIKQP≥VG≥VJGCF≥HKTUV
EGEMU≥QP≥YGVGT≥VG≥QVGT≥VJGCF≥YCPVU≥VQ≥GPVGT≥VG≥ETKVKECN≥TGIKQP≥+H≥UQ≥VG≥HKTUV≥VJGCF≥YKNN
GPVGT≥C≥URKPYCKV≥NQQR≥WPVKN≥VG≥UGEQPF≥VJGCF≥CU≥GZGEWVGF≥VG≥ETKVKECN≥TGIKQPtUCVKUH[KPI≥VG
FGUKTG≥VQ≥GPVGT≥VG≥TGIKQPtCPF≥TGUGVU≥KVU≥FGUKTG≥HNCI≥1VGTYKUG≥HKPFKPI≥VGV≥VG≥QVGT≥VJGCF

while≥VGUV≥ECOG≥DGHQTG≥VQ≥UGVVKPI≥QH≥C≥VJGCFoU≥HNCI≥KP≥VQ≥VJTF
CVVGORV≥VQ≥UVCVWU≥HNCI≥KU≥UGV≥DGHQTG≥VQ≥URKPYCKV≥while≥EQPFKVKQPCN≥EQ≥EMU≥VQ≥QVQT≥VJGCFoU
HNCI

6Q≥KNNWUVTCVG≥QY≥VJU≥CNIQTKVQ≥KU≥KPVGPFGF≥VQ≥GZGEWVG≥NGVoU≥CUUWOG≥YG≥QXG≥6≥GZGEWVKPI
OtherStuffOne≥9QP≥6≥YKUQU≥VQ≥GPVGT≥VQ≥ETKVKECN≥TGIKQP≥KV≥HKTUV≥UGVU ThreadOWantsToEnter≥VQ≥
CPF≥VQP≥GZCOKPGU≥VQ≥XCNWG≥QH≥Thread1WantsToEnter≥VQ≥HKPF≥VQV≥6≥QU≥PQV≥CPPQWPEGF≥CP[
FGUKTG≥VQ≥GPVGT≥VQ≥ETKVKECN≥TGIKQP≥6≥YKNN≥RTQEGGF≥KPVQ≥VQ≥ETKVKECN≥TGIKQP≥5QWNF≥6≥HKPKU3YKVJ
VQ≥QVQT≥EQORWVCVKQPU≥CPF≥TGSWKTG≥CEEGUU≥VQ≥VQ≥ETKVKECN≥TGIKQP≥KV≥YKNN≥UGV
Thread1WantsToEnter≥VQ≥≥CPF≥VQP≥GZCOKPG≥ ThreadOWantsToEnter≥YKE3KV≥HKPFU≥VQ≥CNUQ≥QXG≥VQ
XCNWG≥≥6KU≥YKNN≥RWV≥6≥KPVQ≥VQ≥URKPYCKV≥NQQR≥WPVKN≥6≥GZKVU≥VQ≥ETKVKECN≥TGIKQP≥CPF≥TGUGVU
ThreadOWantsToEnter≥VQ≥

> **NOTE**
>
> **The desire flag will be set to 1 not only when a thread wishes to enter the critical region, but also while the thread is executing in the critical region. Since threads cannot know whether a desire flag is merely the announcement of intent or denotes that the other thread is currently executing in the critical region, they must assume that it means the latter and that the other thread is executing in the critical region.**

Mutual exclusion is enforced when a thread is in the critical region, and threads are free to enter and leave the critical region in any order that their execution will take them. As you should have guessed by now, there is a problem and I hope you've already seen it. If not, the following interleaving shows how this attempt can run into trouble:

1. T0 sets ThreadOWantsToEnter≥≥

   6≥UGVU≥Thread1WantsToEnter≥≥

   6≥VGUVU≥Thread1WantsToEnter≥KP≥while≥EQPFKVKQPCN

   6≥HKPFU≥Thread1WantsToEnter≥≥≥EQPFKVKQPCN≥KU≥647'

   6≥GZGEWVGU≥URKPYCKV

   6≥VGUVU≥ThreadOWantsToEnter≥KP≥while≥EQPFKVKQPCN

   6≥HKPFU≥6hreadOWantsToEnter≥≥≥EQPFKVKQPCN≥KU≥647'

   6≥GZGEWVGU≥URKPYCKV

FOUR NECESSARY CONDITIONS REQUIRED TO ALLOW DEADLOCK

To have the potential for deadlock between two or more threads and the set of resources that are required by each, four conditions must be met. In a paper titled, "Sequencing tasks in multiprocess systems to avoid deadlocks" (in *Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory*), E. G. Coffman, Jr. and A. Shoshani first enumerated these four conditions as follows:

Mutual exclusion condition

Individual resources are either available or they are held by no more than one thread at a time.

Hold and wait condition

Threads that are already holding some resources may attempt to hold new resources.

No preemption condition

Once a thread is holding a resource, that resource can only be removed when the holding thread voluntarily releases the resource.

Circular wait condition

A circular chain of threads requesting resources that are held by the next thread in the chain can exist.

To prevent the possibility of deadlock from occurring, one of these conditions must not be allowed to exist. There are many different ways to prevent one or more of these conditions. A text on operating systems theory should have some discussion about deadlock and how you can prevent it.

```
  Thread1WantsToEnter≥≥
    6≥HKPFU≥6≥KP≥ETKVKECN≥TGIKQP
    6≥TGUGVU≥Thread1WantsToEnter≥≥≥CPF≥FGNC[U
    6≥GZKVU≥ETKVKECN≥TGIKQP
    6≥SWKEMN[≥RGTHQTOU≥OtherStuffZero
    6≥ICKPU≥CEEGUU≥VQ≥ETKVKECN≥TGIKQP
    6≥UGVU≥Thread1WantsToEnter≥≥
    6≥VGUVU≥ThreadOWantsToEnter≥KP≥while≥EQPFKVKQPCN≥CPF≥HKPFU≥6≥KP≥ETKVKECN≥TGIKQP
  6≥TGUGVU≥Thread1WantsToEnter≥≥≥CPF≥FGNC[U
  6≥GZKVU≥ETKVKECN≥TGIKQP
  6≥SWKEMN[≥RGTHQTOU≥OtherStuffZero
  6≥ICKPU≥CEEGUU≥VQ≥ETKVKECN≥TGIKQP
#TG≥[QW≥UVCTVKPI≥VQ≥IGV≥VC≥CPI≥QH≥VKU≥KPVGTNGCXKPI≥CPCN[UKU!≥$HQTG≥[QW≥UCY≥O[≥KPVGTNGCXKPI
FKF≥[QW≥HKIWTG≥QWV≥VCV≥KV≥KU≥RQUUKDNG≥HQT≥QPG≥VJTGCF≥VQ≥MGGR≥VC≥QVCT≥QWV≥QH≥VC≥ETKVKECN≥TGIKQP
KPFGHKPKVGN[!≥5WE3DGCXKQT≥KU≥MPQYP≥CW4VCTXCVKQ2≥6KU≥KPVGTNGCXKPI≥ECP≥IQ≥QP≥KPFGHKPKVGN[
UVTCPFKPI≥VC≥WPNWEM[≥6≥KP≥C≥RGTRGVWCN≥UVCVG≥QH≥UVCTXCVKQP

&QGU≥VKU≥XKQNCVG≥VC≥TGSWKTGF≥RTQRGTVKGU≥QH≥C≥EQTTGEV≥UQNWVKQP!≥oQV≥TGCNN[≥6KU≥UVCTXCVKQP≥CPF
OKUVTGCVOGPV≥QH≥6≥KU≥PQV≥VC≥HCWNV≥QH≥6≥DWV≥KU≥C≥FKTGEV≥TGUWNV≥QH≥VC≥CFXGTUG≥UEC1FWNKPI≥D[≥VC
QRGTCVKPI≥U[UVGO≥9KNG≥+≥YQWNF≥CITGG≥VCV≥UWE3UVCTXCVKQP≥QH≥QPG≥VJTGCF≥D[≥VKU≥CNIQTKVO≥KU
GZVTGOGN[≥WPNKMGN[≥KV≥KU≥UVKNN≥RQUUKDNG≥CPF≥KV≥KU≥C≥UKVWCVKQP≥VCV≥YG≥UQWNF≥VT[≥VQ≥CXQKF≥KH≥CV≥CN
RQUUKDNG
```

&GMMGToU #NIQTKVJO

1WT HKPCN CVVGORV KU MPQYP CU &GMMGToU #NIQTKVJO 4CVJG VJTGCFU QT C ECRTKEKQWU TCPFQO HWPEVKQP VQ DTGCM VJG V ETKVKECN TGIKQP VJG CNIQTKVJO DGUVQYU C pHCXQTGFq UVCVV CEVU CU C VKGDTGCMGT NKMG VJG RQUUGUUKQP CTTQY KP DCU 'ZCORNG IKXGU VJG RUGWFQEQFG HQT VJKU UQNWVKQP VQ VJ

EXAMPLE 3-5. Dekker's Algorithm

```
int favored;
int Thread0WantsToEnter, Thread1WantsToEnter;

void ThreadZero()
{
  while (TRUE) do {
    Thread0WantsToEnter = 1;
    while (Thread1WantsToEnter) do {
      if (favored == 1) {
        Thread0WantsToEnter = 0;
        while (favored == 1) do {}  // spin-wait
        Thread0WantsToEnter = 1;
      }
    }
    CriticalRegionZero;
    favored = 1;
    Thread0WantsToEnter = 0;
    OtherStuffZero;
  }
}

void ThreadOne()
{
  while (TRUE) do {
    Thread1WantsToEnter = 1;
    while (Thread0WantsToEnter) do {
      if (favored == 0) {
        Thread1WantsToEnter = 0;
        while (favored == 0) do {}
        Thread1WantsToEnter = 1;
      }
    }
    CriticalRegionOne;
```

```
    favored = 0;
    Thread1WantsToEnter = 0;
    OtherStuffOne;
  }
}
```

(QT≥CNN≥VG≥RTGXKQWU≥CVVGORVU≥YG≥QPN[≥DF≥VQ≥UQY≥VDV≥QPG≥ECUG≥NGF≥VQ≥C≥FKNGOOC≥KP≥QTFGT≥VQ
UQY≥VGTG≥YCU≥C≥RTQDNGO≥YKVJVG≥CNIQTKVO≥6Q≥UQY≥VDV≥&GMMGToU≥#NIQTKVO≥IKXGU≥C≥EQTTGEV
UQNWVKQP≥YG≥OWUV≥GZCOKPG≥CNN≥RQUUKDNG≥KPVGTNGCXKPIU≥QH≥VJGCF≥GZGEWVKQP≥CPF≥UQY≥VDV≥GCEJ
QPG≥YKNN≥NGCF≥VQ≥C≥FGUKTGF≥QWVEQOG≥(QT≥VKU≥CNIQTKVO≥YG≥ECP≥VCMG≥CNN≥QH≥VG≥RQUUKDNG≥UKVWCVKQPU
DGVYGGP≥VJGCFU≥CPF≥EQPFGPUG≥VGO≥KPVQ≥VG≥HQWT≥ECUGU≥NKUVGF≥PGZV≥5KPEG≥VG≥EQFG≥HQT≥GCEJ
VJGCF≥KU≥CNIQTKVOKECNN[≥VG≥UCOG≥YG≥ECP≥TGUVTKEV≥QWT≥CPCN[UKU≥VQ≥QPG≥VJGCF≥FGCNKPI≥YKVJVG
UVCVWU≥QH≥VG≥QVGT≥VJGCF≥CPF≥MPQY≥VDV≥VG≥CPCN[UKU≥QH≥ECUGU≥YGTG≥VG≥TQNGU≥CTG≥TGXGTUGF≥YKNN
DG≥VG≥UCOG

Case 1

If there is no conflict for entry into the critical region, Dekker's Algorithm will allow a thread to enter as needed, just as the three previous attempts did. For example, if T0 is ready for access to the critical region, it sets `ThreadOWantsToEnter`≥VQ≥≥CPF≥EGEMU≥`Thread1WantsToEnter`≥+H≥VG≥FGUKTG
HNCI≥HQT≥6≥KU≥≥FGVGTOKPGF≥D[≥VG≥OKFFNG`while`≥EQPFKVKQP≥VGUV≥6≥UMKRU≥VG≥DQF[≥QH≥VG≥OKFFNG
`while`≥NQQR≥CPF≥GPVGTU≥VG≥ETKVKECN≥TGIKQP

Case 2a

For this case, we assume that T1 is executing in the critical region and T0 wishes entry. T0 first sets `ThreadOWantsToEnter`≥VQ≥≥9GP≥KV≥HKPFU≥VDV≥VG≥XCNWG≥QH≥`Thread1WantsToEnter`≥KU≥≥6≥YKNN
EGEM≥VG≥ `favored`≥VQIING≥9GP≥VG≥XCNWG≥KU≥≥6≥KU≥VG≥HCXQTGF≥VJGCF≥6≥YKNN≥PQV≥GZGEWVG≥VG
DQF[≥QH≥VG≥`if`≥UVCVGOGPV≥DWV≥YKNN≥YCKV≥HQT≥6≥VQ≥GZKV≥VG≥ETKVKECN≥TGIKQP≥CPF≥TGUGV≥KVU≥FGUKTG≥HNCI
#HVGT≥VDV≥DU≥DRRGPGF≥6≥YKNN≥GZKV≥VG≥OKFFNG≥`while`≥NQQR≥CPF≥RTQEGGF≥KPVQ≥VG≥ETKVKECN≥TGIKQP
5KPEG≥`ThreadOWantsToEnter`≥DU≥DGGP≥UGV≥CPF≥TGOCKPU≥UGV≥YKNG≥6≥FQGU≥C≥URKPYCKV≥6≥YKNN≥DG
RTGXGPVGF≥HTQO≥GPVGTKPI≥VG≥ETKVKECN≥TGIKQP≥WPVKN≥CHVGT≥6≥DU≥VCMGP≥CFXCPVCIG≥QH≥VG≥HCXQTGF
UVCVWU≥CPF≥EQORNGVGU≥GZGEWVKQP≥KP≥VG≥ETKVKECN≥TGIKQP

Case 2b

This case also assumes that T1 is executing in the critical region and T0 wishes entry. T0 first sets `ThreadOWantsToEnter`≥VQ≥≥9GP≥KV≥HKPFU≥VDV≥VG≥XCNWG≥QH≥`Thread1WantsToEnter`≥KU≥≥6≥YKNN
VGP≥EGEM≥VG≥`favored`≥VQIING≥9GP≥VG≥XCNWG≥KU≥≥6≥KU≥VG≥HCXQTGF≥VJGCF≥CPF≥6≥TGUGVU≥KVU
FGUKTG≥HNCI≥DGHQTG≥GPVGTKPI≥VG≥URKPYCKV≥NQQR≥YCKVKPI≥HQT≥6≥VQ≥6GV≥VG≥`favored`≥VQIING≥WRQP≥GZKV
HTQO≥VG≥ETKVKECN≥TGIKQP≥1PEG≥6≥DU≥GZKVGF≥VG≥URKPYCKV≥CPF≥UGV≥`ThreadOWantsToEnter`≥VQ≥≥6
YQWNF≥PQV≥DG≥CNNQYGF≥VQ≥GPVGT≥VG≥ETKVKECN≥TGIKQP≥WPVKN≥6≥DU≥TGUGV≥KVU≥FGUKTG≥HNCI≥CHVGT≥VCMKPI
CFXCPVCIG≥QH≥VG≥HCXQTGF≥UVCVWU≥CPF≥GZGEWVGF≥VG≥ETKVKECN≥TGIKQP≥EQFG

Case 3

If both threads wish to enter the critical region at the same time, both threads will set their respective desire flags. If T1 is the favored thread, upon entering the middle while≥NQQR≥6≥YKNN GZGEWVG≥VQ≥DQF[≥QH≥VQ≥if≥UVCVGOGPV≥CPF≥TGUGV≥KVU≥FGUKTG≥HNCI≥{}≥VWTPKPI≥QHH≥KVU≥QYP≥HNCI≥6 CNNQYU≥6≥VQ≥HCXQTGF≥VJGCF≥VQ≥UMKR≥VQ≥ if≥UVCVGOGPV≥CPF≥GZKV≥VQ≥OKFFNG≥while≥NQQR≥KPVQ≥VQ ETKVKECN≥TGIKQP≥9QP≥KV≥NGCXGU≥VQ≥ETKVKECN≥TGIKQP≥6≥UGVU≥VQ≥favored≥VQIING≥VQ≥≥6≥OC[≥VQP RTQEGGF≥HTQO≥VQ≥URKPYCKV≥QP≥favored≥CPF≥KPVQ≥VQ≥ETKVKECN≥TGIKQP≥YQP≥Thread1WantsToEnter QU≥DGGP≥TGUGV

What about indefinite postponement?

The fourth solution attempt suffered from indefinite postponement when a thread, having exited the critical region, quickly executed other computations and reentered the critical region before the random delay of the other thread had elapsed. Could T1 keep T0 from entering the critical region? The only way that this can happen is if T1 is allowed to execute and T0 is not. However, this situation violates the fairness criteria of Ben-Ari's concurrent abstraction. Thus, at some point we must allow T0 to execute the next atomic statement. When this next statement is the evaluation of the while≥EQPFKVKQPCN≥GZRTGUUKQP≥QH≥(favored == 1)≥VQ≥TGUWNV≥YKNN DG≥HCNUG≥CPF≥6≥YKNN≥VQP≥DTGCM≥QWV≥QH≥VQ≥URKPYCKV≥CPF≥UGV≥KVU≥FGUKTG≥HNCI≥1PEG≥VKU≥QRRGPU 6≥YKNN≥PQV≥DG≥CNNQYGF≥DCEM≥KPVQ≥VQ≥ETKVKECN≥TGIKQP≥UKPEG≥6≥QU≥UQYP≥C≥FGUKTG≥VQ≥FQ≥UQ≥CPF 6≥KU≥PQV≥VQ≥HCXQTGF≥VJGCF≥6Q≥HQNNQYKPI≥KPVGTNGCXKPI≥UQYU≥QPG≥RQUUKDNG≥GZGEWVKQP≥VQV FGOQPUVTCVGU≥VKU

%WTTGPV≥UVCVWU

```
ThreadOWantsToEnter≥≥
Thread1WantsToEnter≥≥
favored≥≥
6≥YCU≥NCUV≥GZGEWVKPI≥while (favored == 1) do {}
6≥KU≥GZGEWVKPI≥CriticalRegionOne
```

6≥GZKVU≥ETKVKECN≥TGIKQP

6≥UGVU≥favored≥≥≥CPF≥TGUGVU≥Thread1WantsToEnter≥≥

6≥GZGEWVGU≥1VQT5VWHH1PG

6≥UGVU≥Thread1WantsToEnter≥≥

6≥VGUVU≥ThreadOWantsToEnter≥KP≥while≥EQPFKVKQPCN

6≥GPVGTU≥ETKVKECN≥TGIKQP

6≥VGUVU≥favored≥VQIING≥KP≥while≥EQPFKVKQPCN

6≥HKPFU≥favored≥≥

6≥UGVU≥ThreadOWantsToEnter≥≥

6≥GZKVU≥ETKVKECN≥TGIKQP

6≥UGVU≥favored≥≥

```
Thread1WantsToEnter≥≥
  6≥GZGEWVGU≥1VQT5VWHH1PG
  6≥UGVU≥Thread1WantsToEnter≥≥
  6≥VGUVU≥Thread0WantsToEnter≥CPF≥HKPFU≥6≥YKUKPI≥VQ≥GPVGT≥VQ≥ETKVKECN≥TGIKQP
  6≥TGUGVU≥Thread1WantsToEnter≥≥
  6≥NQQRU≥KP≥URKPYCKV≥WPVKN≥favored≥≥
  6≥GPVGTU≥VQ≥ETKVKECN≥TGIKQP
#V≥UQOG≥RQKPV≥6≥OWUV≥DG≥IKXGP≥VQ≥EQPEG≥VQ≥TGUWOG≥GZGEWVKQP≥QH≥CV≥NGCUV≥QPG≥CVQOKE
UVCVGOGPV≥9QP≥6≥KU≥CDNG≥VQ≥GZGEWVG≥KV≥HKPFU≥VQ≥favored≥VQIING≥QU≥DGGP≥UGV≥KP≥KVU≥HCXQT≥CPF
KV≥ECP≥UGV≥KVU≥FGUKTG≥HNCI≥VQ≥KPHQTO≥6≥VQV≥6≥KU≥TGCF[≥VQ≥GPVGT≥VQ≥ETKVKECN≥TGIKQP≥(TQO≥VKU
KPVGTNGCXKPI≥YG≥ECP≥EQPENWFG≥VQV≥C≥VJGCF≥ECPPQV≥KPFGHKPKVGN[≥RQUVRQPG≥CPQVQT≥VJGCF≥HTQO
GPVGTKPI≥VQ≥ETKVKECN≥TGIKQP
```

What Did You Learn?

Dekker's Algorithm was used here to solve the Critical Section Problem. One of the obvious drawbacks to Dekker's Algorithm is that it is only defined for two threads. Plus, there are better algorithms to solve this problem, such as Lamport's Bakery Algorithm and Peterson's Algorithm. The latter is simpler than Dekker's Algorithm and even has a variation that works for multiple threads. Besides, all threading models and libraries have synchronization objects that can enforce mutual exclusion between any number of threads. Thus, Dekker's Algorithm was not the main point of the preceding exercise.

The point of all of this was to show you how to identify concurrency errors or demonstrate correctness using the concurrency abstraction of Ben-Ari and the interleaving of atomic statement executions between the two threads. I hope that you tried the interleaving analysis on your own for each solution attempt before reading about the faults that each attempt contained. Some of these were subtle, particularly in the starvation case. This is not something that a software tool would be able to identify, and this is why the interleaving analysis is important and can assist you in designing correct concurrent solutions before we ever attempt to alter a line of code.

I'll be using the interleaving analysis during the discussions and implementation of several algorithms in the later chapters of this book. You might want to practice your analysis skills as you cover a new algorithm, especially those codes for which I don't have analysis already.

There Are No Evil Threads, Just Threads Programmed for Evil

It is tempting to anthropomorphize (ascribe human attributes to) threads when trying to determine how they will interact with each other. I do it with both serial and parallel codes and data structures. It can make analysis a little easier to wrap your mind around. Just don't

Performance Metrics (How Am I Doing?)

Once you have coded an algorithm, you will want to know how well that code will execute. The faster an application runs, the less time a user will need to wait for her results. Also, shorter execution time gives the user the chance to run larger data sets (e.g., a larger number of data records, more pixels, or a bigger physical model) in an acceptable amount of time. For serial applications, you can measure an application's speed using a stopwatch. Simply time the run from start to finish.

After you've made optimizations to the code (for example, rearranged the execution order of statements, used a more efficient memory access scheme, or replaced a critical algorithm with a more efficient one) and rebuilt the application, by simply comparing execution times from the two versions you can see whether there was any serial execution improvement. It is the same with concurrent programs.

The time taken to execute is your paramount concern in developing parallel solutions. When doing comparisons, elapsed time is always the final judge of whether the concurrent code is better than the serial one. If you've taken the time to modify your company's flagship application to run on multiple cores and the execution time is slower than the original, you shouldn't expect too much in the way of a Christmas bonus. However, after reading this book, if you are able to make a positive change to the application and the parallel version runs faster, how can you communicate how much faster your concurrent application now performs (and enhance your bonus prospects)? You could report two execution figures ² serial execution time and parallel execution time on the same input data set ² to your manager. Managers, being the busy people that they are, will be happier if you give them one number instead of two. One such number that offers a tangible comparison of serial and parallel execution time is *speedup*.

Speedup

Simply stated, speedup is the ratio of serial execution time to parallel execution time. In the past I've seen this ratio expressed as both a percentage and as a multiplier. I prefer stating speedup figures as a multiplier, since using percentages can lead to confusion (and your goal is to make clear to your manager the superiority of your programming skills and accomplishments, not confuse him). For example, if you stated that your parallel execution is 200$ faster than the serial code, does it run in half the time of the serial version or one third of the time? Is 105$ speedup almost the same time as the serial execution or twice as fast? Is the baseline serial time 0$ speedup or 100$ speedup? On the other hand, if you said your

FIGURE 3-2. Example speedup curves

In very rare circumstances, you may find that the speedup of your application exceeds the number of cores. This phenomenon is known as ***superlinear speedup****. If you run into this, suspect that you have made some error. First, double-check the timings of both the serial and*

Amdahl's Law

Before starting any parallelization project, you may wish to estimate the amount of performance increase (speedup) that you can realize. Without actually writing any concurrent code, you can use Amdahl's Law to give you an upper bound on the speedup you can attempt to achieve. To use Amdahl's Law, you will need to know what percentage of execution will be able to run in parallel and what amount of code must run in serial. Since this is only an estimate, you don't need to have exact figures. If you have an idea about which functions should execute mostly concurrent, you can use a profiler report (from a typical data set) with a breakdown of percentages of execution time per function. Once you have the percentage of parallel execution time and, consequently, serial execution time, just plug the values into the formula.

There are several formulations of Amdahl's Law, but I prefer this one

$$Speedup \leq \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}$$

where *pctPar* is the percentage of execution time that will be run in parallel, and *p* is the number of cores on which to run the parallel application. To compute speedup, the formula has taken the serial execution time and normalized it to 1. The time of the parallel execution is estimated in the denominator to be the percentage of serial time (*1 ± pctPar*) and the percentage of execution that can be run in parallel divided by the number of cores to be used (*pctPar/p*). Figure 3-3 shows several speedup curves using this formula on different numbers of cores with varying percentage amounts of parallel execution. Notice that the curve for 75$ parallel execution is only approaching 3ñ speedup at 8 cores.

By arbitrarily increasing the number of cores available, you can lower the amount of execution time required for the parallel sections of your code as much as you want. If we assume infinite numbers of cores are available, the parallel execution time could be essentially zero. With this

FIGURE 3-3. Estimated speedup curves for different amounts of percentage of parallel execution time using Amdahl's Law

Amdahl's Law has received quite a bit of criticism for the way it ignores real-world circumstances like concurrency overhead (communication, synchronization, and other thread management) and not having processors with infinite numbers of cores available (yet). Other parallel execution models have been proposed that attempt to make reasonable assumptions for the discrepancies in the simple model of Amdahl's Law. Still, for its simplicity and the understanding by the user that this is an upper bound, which is very unlikely to be achieved or surpassed, Amdahl's Law is a pretty good indication of the potential for speedup in a serial application.

Gustafson-Barsis's Law

Besides not taking into account the overheads inherent in concurrent algorithms, one of the strongest criticisms of Amdahl's Law is that as the number of cores increase, the amount of data handled is likely to increase as well. Amdahl's Law assumes a fixed data set size for any and all numbers of cores used. This is reflected in the assumption of the serial percentage remaining the same. But what if you had eight cores and were able to compute a data set that was eight times the size of the original? Does the serial execution time increase? Even if it does, will the time of the serial portion in the concurrent code be the same fraction of overall

NOTE

Could you have used Amdahl's Law to compute this speedup estimate? If we take the serial execution percentage of 1.3 , the equation for Amdahl's Law yields 22.808 1/(0.013 (0.987/32)). However, this is a false computation, since the percentage of serial time is relative to the parallel time of the 32-core execution, not the potential parallel time.

If you multiply the number of seconds (1,026) for parallel execution on 32 cores, you find that the total amount of work done by the application takes 1,026 32 14 32,846 seconds. The nonparallel time (14 seconds) is 0.0426 of that total work time. Using that figure, Amdahl's Law calculates a speedup of 1/(0.000426 (0.999574/32)) 31.582.

If you want to see the derivation of the Gustafson-Barsis Law, you can go to John L. Gustafson's original paper, "Reevaluating Amdahl's Law" (*Communications of the ACM* , 1988) or Michael J. Quinn's book, *Parallel Programming in C with MPI and OpenMP* (McGraw-Hill, 2004).

Efficiency

Related to speedup is the metric of *efficiency* . Whereas speedup gives us a metric to determine how much faster our parallel applications are versus their serial brothers, efficiency tells us how well we are utilizing the computational resources of the system. To calculate the efficiency of your parallel execution, take the observed speedup and divide by the number of cores used. This number is then expressed as a percentage.

One Final Note on Speedup and Efficiency

Throughout this discussion of metrics, I assume that you are using one thread per core. If your application overloads the system with more threads than cores, you need to use the number of threads (in place of ®cores¯) in all the previous formulas. There can be performance benefits (though usually minor) when you overload the system with threads. Of course, Intel processors with Hyper-Threading (HT) technology are designed to support multiple threads per core.

Your measurements of speedup and efficiency will tell you whether the utilization of more threads than cores is worthwhile. If your execution time remains almost the same with two threads per core and no change in the data set, your speedup will also stay about the same. Your efficiency will be halved, though. This is telling you that even if the physical cores are cranking at nearly 100$, the threads are only being used half the time (where threads may have been nearly fully utilized when there was only one thread per core). If hardware utilization is more important to you, and the execution time is not suffering, not having all threads as busy as possible may not be a detriment.

Review of the Evolution for Supporting Parallelism in Hardware

The following paragraphs take a roughly chronological tour of processor and platform innovations that have led us to multicore processors. Being a self-described ®software guy,¯ I tend to drift off when programming books or presentations go on about hardware details for long. So, I won't take much time to cover this topic. It won't hurt my feelings if you skip it.

CHAPTER FOUR

Eight Simple Rules for Designing Multithreaded Applications

SINCE IT IS RIGHT THERE IN THE TITLE OF THIS BOOK, THE FOLLOWING SENTENCE shouldn't come as any surprise *Concurrent programming is still more art than science*. This chapter gives eight simple rules that you can add to your toolkit of threading design methods. I've tried to organize the rules in a semichronological way, but there's no hard and fast order to the rules. It's like being confronted with, ®No running by the pool,¯ and, ®No diving in the shallow end.¯ Both good ideas, but not diving can come before not running or vice versa.

By following these rules, you will have more success in writing the best and most efficient threaded implementation of your applications. You may recognize some of these, since I've mentioned a few of them in previous chapters. In upcoming chapters, when discussing the design and implementation of specific algorithms, I'll try to drop in a relevant reference to one or more of these eight rules to show that they're not just here to fill out an extra chapter.

Rule 1

I've already covered this first rule seven ways to Sunday, but since it's the crux of the whole matter, it bears repeating at least one more time. You can't execute anything concurrently unless the operations that would be executed can be run independently of each other. I can easily think of different real-world instances of independent actions being performed to satisfy a single goal. Consider, for example, a DVD rental warehouse. Orders for movies are collected and then distributed to the workers, who go out to where all the disks are stored and find copies to satisfy their assigned orders. When one worker pulls out a classic musical comedy, it does not interfere with another worker who is looking for the latest science fiction masterpiece, nor will it interfere with a worker trying to locate episodes from the second season of a popular crime drama series (I assume that any conflicts resulting from unavailable inventory have been dealt with before orders were transmitted to the warehouse). Also, the packaging and mailing of each order will not interfere with disk searches or the shipping and handling of any other order.

There are cases in which you will have exclusively sequential computations that cannot be made concurrent: many of these will be dependencies between loop iterations or steps that must be carried out in a specific order. A list of common situations was covered earlier in ®What's Not Parallel¯ on page 42.

Rule 2

There are two directions you can take when approaching the threading of a serial code. These are *bottom-up* and *top-down*. When initially analyzing your code, you are looking for the computational hotspots that account for the most execution time. Running those portions in parallel will give you the best chance of achieving the maximum performance possible.

In a bottom-up approach, you consider threading the hotspots in your code directly. If this is not possible, search up the call stack of the application to determine whether there is another

Rule 3
Numbers of Cores

As I'm writing this, quad-core processors are becoming the default multicore chip. The number of cores available in future processors will only increase. Thus, you should plan for such processor increases within your software. *Scalability* is the measure of an application's ability to handle changes, typically increases, in system resources (e.g., number of cores, memory size, bus speed) or data set sizes. In the face of more cores being available, you must write flexible code that can take advantage of different numbers of cores.

Rule 5

If threaded libraries are insufficient to cover all the concurrency of an application and you must employ user-controlled threads, don't use explicit threads if an implicit threading model (e.g., OpenMP or Intel Threading Building Blocks) has all the functionality you need. Explicit threads do allow for finer control of the threading implementation. However, if you are only parallelizing compute-intensive loops or don't need the extra flexibility you can get with explicit threads, there's probably no reason to do more work than necessary. The more complex the implementation, the easier it will be to make a mistake and the harder it will be to maintain such code later.

OpenMP is focused on data decomposition methods, especially targeted to threading loops that range over large data sets. Even if this is the only type of parallelism that you can introduce into an application, there may be external requirements (such as engineering practices dictated by your employer or management preferences) that will prohibit your use of OpenMP. In that case, you will need to implement your threading with an approved (explicit) model. In such a situation, I recommend that you use OpenMP to prototype the planned concurrency and estimate the potential performance gains, possible scalability, and how much effort will be needed to thread the serial code with explicit threads.

Rule 6

With serial computations, it is easy to predict the statement that will be executed following any other statement in a program. On the other hand, execution order of threads is *nondeterministic* and controlled by the OS scheduler. This means that there is no reliable way of predicting the order of threads running from one execution to another, or even which thread will be scheduled to run next. This is done primarily to hide execution latency within an application, especially when run on a system with fewer cores than threads. If a thread blocks because it needs memory that is not located in cache or to process an I/O request, the scheduler will swap out the blocked thread and swap in a thread that is ready to run.

Data races are a direct result of this scheduling nondeterminism. If you assume that one thread will write a value into a shared variable before another thread will read that value, you may be right all of the time, you may be right some of the time, or you may be right none of the time. Sometimes, if you're lucky, the order of thread execution remains unchanged on a specific platform each and every time you run an application. Every difference between systems (bit locations on the disk or memory speed or frequency of the AC power coming out of the wall sockets) has the potential to alter the thread schedule. Code that relies on a particular

Rule 7
Associate Locks to Specific Data

Synchronization is overhead that does not contribute to the furtherance of the computation, except to guarantee the correct answers are produced from the parallel execution of an application. Synchronization is a necessary evil. Even so, you should actively seek to keep the amount of synchronization to a minimum. You can do this by using storage that is local to threads or using exclusive memory locations (such as an array element indexed by thread ID).

Temporary work variables are rarely shared between threads, and should be declared or allocated locally to each thread. Variables that hold partial results for each thread should also be local to threads. Combining the partial results into a shared location will require some synchronization. Ensuring that the shared updates are done as infrequently as possible will keep the amount of overhead to a minimum. If you are using explicit threads, you can use the available thread-local storage APIs to enable the persistence of data local to threads from one threaded region to another or from one threaded function call to the next execution of the same function.

If local storage for each thread is not a valid option and you must coordinate access to shared resources through synchronization objects (such as a lock), be sure to properly associate (or ®attach¯) locks to data items. The easiest way to do this is to have a one to one (1 of locks to data items. If you have multiple shared variables that are always accessed together, use a single lock to allow exclusive access to all critical regions involving these variables. In later chapters, I'll discuss some of the tradeoffs and alternative synchronization techniques that you can employ, especially if you have to protect access to a large collection of data (for example, an array of 10,000 items).

However you decide to associate locks with data items, never associate more than one lock to a single data object. Segal's Law states, ®A man with a watch knows what time it is. A man

```
lockA≥CPF≥lockBtRTQVGEV
CEEGUU≥VQ≥VQ≥UCOG≥XCTKCDNG≥QPG≥RCTV≥QH≥VQ≥EQFG≥OC[≥WUGG≥A≥HQT≥CEEGUU≥YKNNG≥CPQVQT≥UGEVKQP
QH≥EQFG≥ECP≥WUG≥lockB≥6TGCFU≥GZGEWVKPI≥KP≥VQUG≥VYQ≥EQFG≥RQTVKQPU≥YKNN≥ETGCVG≥C≥FCVC≥TCEG
UKPEG≥GCEJ≥YKNN≥CUUWOG≥KV≥QU≥GZENWUKXG≥CEEGUU≥VQ≥VQ≥EQPVGUVGF≥FCVC
```

Rule 8
Concurrency

For comparing performance of applications, both serial and concurrent, the bottom line is wall clock execution time. When choosing between two or more algorithms, programmers may rely on the asymptotic order of execution. This metric will almost always correlate with an application's relative performance to another. That is, with everything else held constant, an application that uses an $O(n \log n)$ algorithm (like Quicksort) will run faster than an $O(n^2)$ algorithm (such as selection sort) and will generate the same results.

In concurrent applications, algorithms with a better asymptotic order of execution will run faster, too. Nonetheless, there will be times when the best serial algorithm will not be amenable to parallelization. If you cannot easily turn a hotspot into threaded code (and you can't find a point higher in the call stack of the hotspot that can be made concurrent), you should consider using a suboptimal serial algorithm to transform, rather than the algorithm currently in the code.

For example, consider the linear algebra operation for the multiplication of two square matrixes. Strassen's Algorithm has one of the best asymptotic orders of execution, $O(n^{2.81})$. This is better than the $O(n^3)$ of the traditional triple-nested loop algorithm. Strassen's method divides up each of the matrixes into four chunks (or submatrixes) and uses seven recursive calls to multiply the n/2 ñ n/2 submatrixes. To parallelize these recursive calls, you could create a new thread to execute each of the seven independent submatrix multiplications. The number of threads will increase exponentially (much like the wives, sacks, cats, and kittens coming from St. Ives). As the submatrixes get smaller and smaller, the granularity of the assigned work given to a newly created thread will get finer and finer. When the submatrixes achieve a given size, switch to a serial algorithm.

A much easier means to parallelize matrix multiplication is to use the asymptotically inferior triple-nested loop algorithm. There are several ways to perform a data decomposition on the matrixes (divide by rows, divide by columns, or divide by blocks) and assign the necessary computations to threads. You can do this using OpenMP pragmas at one of the loop levels or by using explicit threads that implement the division of the loop indexes as needed. Less code modification is required for the simpler serial algorithm, and the structure of the code would likely be left more intact than it would be if you attempted to thread Strassen's Algorithm. Better yet, follow Simple Rule 4 and use a concurrent library function that performs the matrix-matrix multiplication.

Summary

I've given you eight simple rules that you should keep in mind when designing the threading that will transform a serial application into a concurrent version. By following the rules presented here, I've been able to more easily create concurrent solutions that are more robust, less likely to contain threading problems, and that move toward optimal performance with less development time. I'm sure you will, too.

CHAPTER SIX

Parallel Sum and Prefix Scan

SUMMING THE ELEMENTS OF AN ARRAY OR FINDING ALL PARTIAL SUMS OF THE ELEMENTS in an array are basic algorithmic problems. The solution to these problems is easy to describe in a single sentence or two. The concurrent versions of these algorithms, known as *parallel sum* and *prefix scan* (or *parallel scan*), are simple and easy to understand. Since they are so simple, these problems have been extensively analyzed and are used as bellwether algorithms within the parallel programming community. Description, design, analysis, and implementation of these two algorithms will get our feet wet for the rest of the algorithms contained in the text.

Study of these two concurrent algorithms is all well and good, but if you can't find a use for them, you might think that reading through this chapter could be a waste of time. I'm sure you can imagine cases in which you might need to find the sum of an array of items or figure out the largest item within an array. These are examples of parallel sum. Prefix scan is a bit more abstract and its use as part of another algorithm is less obvious. So, after going over the design and implementation of these two concurrent algorithms, I'll point out some other algorithms where prefix scan is used.

Parallel Sum

You may need to compute the sum of all values within a given array. Example 6-1 shows a serial code that will perform such a summation of the N≥GNGOGPVU≥YKVH≥VQ≥KPVGIGT≥CTTC[≥A
#HVGT≥GZGEWVKQP≥QH≥VQ≥EQFG≥VQ≥sum≥XCTKCDNG≥YKNN≥QXG≥VQ≥VQVCN≥UWO≥QH≥CNN≥GNGOGPVU≥HTQO≥VQ
A≥CTTC[≥CUUWOKPI≥PQ≥QXGTHNQY≥QT≥QVGT≥GZEGRVKQP≥YCU≥GPEQWPVGTGF

EXAMPLE 6-1. Serial summation of integer array

```
int sum = 0;
for (int i = 0; i < N; i++)
  sum += A[i];
```

6Q≥RCTCNNGN≥UWO≥QRGTCVKQP≥ECP≥YQTM≥YKVH≥CP[≥CUUQEKCVKXG≥CPF≥EQOOWVCVKXG≥EQODKPKPI
QRGTCVKQP≥(QT≥GZCORNG≥VQ≥QRGTCVKQPU≥QH≥OWNVKRNKECVKQP≥OCZKOWO≥OKPKOWO≥CPF≥UQOG
NQIKECN≥QRGTCVKQPU≥YQWNF≥CNN≥DG≥CRRTQRTKCVG≥+P≥C≥UGTKCN≥RTQITCO≥VQ≥EQOOWVCVKXG≥RTQRGTV[≥QH
VQ≥QRGTCVKQP≥KU≥PQV≥KORQTVCPV≥UKPEG≥VQ≥QTFGT≥YKVH≥YKEH≥VQ≥UGTKCN≥GZGEWVKQP≥EQODKPGU≥VQ
GNGOGPVU≥KU≥VQ≥UCOG≥GCEH≥CPF≥GXGT[≥VKOG≥OGXGTVQNGUU≥VQ≥QXG≥CP[≥EQPEG≥QH≥RGTHQTOKPI≥VQ
EQORWVCVKQP≥EQPEWTTGPVN[≥YG≥TGN[≥QP≥VKU≥RTQRGTV[≥QH≥VQ≥QRGTCVKQP≥UKPEG≥EQPEWTTGPV
GZGEWVKQP≥FQGU≥PQV≥KORQUG≥C≥HKZGF≥UEQFWNG≥QP≥VQ≥QTFGT≥QH≥KPFGRGPFGPV≥EQORWVCVKQPU
6TQWIQWV≥VKU≥EQRVGT≥+oNN≥VCNM≥CDQWV≥CFFKVKQP≥QT≥VQ≥UWOOKPI≥QH≥GNGOGPVU≥CU≥VQ≥EQODKPKPI
QRGTCVKQP≥$≥CYCTG≥VQV≥[QW≥ECP≥TGRNCEG≥VKU≥YKVH≥CPQVQT≥CRRTQRTKCVG≥QRGTCVKQP≥YQP
PGGFGF

.QQMKPI≥ENQUGT≥CV≥VQ≥UGTKCN≥EQFG≥UKPEG≥VQ≥XCNWG≥QH≥sum≥CV≥CP[≥UKPING≥KVGTCVKQP≥FGRGPFU≥QP≥CNN
VQ≥RTGXKQWU≥CFFKVKQPU≥VQ≥sum≥VKU≥OKIV≥NQQM≥NKMG≥C≥UKVWCVKQP≥VQV≥ECPPQV≥DG≥OCFG≥EQPEWTTGPV
.WEMKN[≥VKU≥KU≥C≥URGEKCN≥ECUG≥QH≥CP≥KPFWEVKQP≥XCTKCDNG≥YQTG≥VQ≥RWTRQUG≥QH≥VQ≥EQORWVCVKQP≥KU

PRAM Algorithm

Imagine a complete binary tree with the same number of leaf nodes as there are elements in the vector to be summed. If we assign a vector value to individual leaf nodes and make the internal nodes of the tree correspond to the addition of child nodes, we can find the sum of all the leaf node values by working up the tree and carrying out the addition operations. Figure 6-1 shows an inverted binary tree with eight leaf nodes (squares) and assigned values. The internal nodes (circles) hold the sums of the node's children, and the root will have the sum of all vector values.

FIGURE 6-1. Binary tree illustration of parallel sum computation

Each addition within the same level of the tree is independent of all the other additions within that level. This is where the concurrency of the algorithm is to be found.

For a PRAM model, if you have a processor for each element of the array (leaf node), you can compute the parallel sum in a logarithmic number of steps. Example 6-2 shows the pseudocode for this computation adapted from the Daniel Hillis and Guy Steele paper, "Data Parallel Algorithms" (Communications of the ACM, 1986).

EXAMPLE 6-2. PRAM algorithm for parallel sum

```
for j := 1 to log_2(n) do
  for all k in parallel do
    if (((k + 1) mod 2^j) = 0) then
      X[k] := X[k - 2^(j-1)] + X[k]
    fi
```

NUM_THREADS≥&GVCKNU≥CDQWV≥QY≥VC≥FCVC≥KU≥KPKVKCNK\GF≥InitializeArray()≥CTG≥NGHV≥QWV≥QH≥VC≥U
EQFG≥VQQ

':#/2.' 2CTCNNGN UWO WUKPI 215+: VJTGCFU CPF INQDCN RCTVKCN UW

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 4

int N;  // number of elements in array X
int *X;
int gSum[NUM_THREADS];  // global storage for partial results

void *Summation (void *pArg)
{
  int tNum = *((int *) pArg);
  int lSum = 0;
  int start, end;

  start = (N/NUM_THREADS) * tNum;
  end = (N/NUM_THREADS) * (tNum+1);
  if (tNum == (NUM_THREADS-1)) end = N;
  for (int i = start; i < end; i++)
    lSum += X[i];
  gSum[tNum] = lSum;
  free(pArg);
}

int main(int argc, char* argv[])
{
  int j, sum = 0;
  pthread_t tHandles[NUM_THREADS];

  InitializeArray(X, &N);  // get values into X array; not shown
  for (j = 0; j < NUM_THREADS; j++) {
    int *threadNum = new(int);
    *threadNum = j;
    pthread_create(&tHandles[j], NULL, Summation, (void *)threadNum);
  }
  for (j = 0; j < NUM_THREADS; j++) {
    pthread_join(tHandles[j], NULL);
    sum += gSum[j];
  }
  printf("The sum of array elements is %d\n", sum);
  return 0;
}
```

Summation()≥HWPEVKQP≥ĆE∃VJGCF≥HKTUV
HKPFU≥VQ≥DQWPFCTKGU≥QH≥VQ≥CTTC[≥EØPM≥VQV≥KV≥YKNN≥DG≥CUUKIPGF≥6Ø UG≥DQWPFCTKGU≥CTG≥UVQTGF
KP≥NQECN≥EQRKGU≥QH≥start≥CPF≥end≥6Ø ≥DQWPFCT[≥EQORWVCVKQP≥TGSWKTGU≥VQV≥GCE∃VJGCF≥DG
CUUKIPGF≥C≥WPKSWG≥EQPUGEWVKXG≥KPVGIGT≥+&≥PWODGT≥UVCTVKPI≥YKV∃≥6Ø ≥VJGCF≥+&≥KU≥RCUUGF≥VQ
VQ≥VJGCF≥XKC≥VQ≥UKPING≥RCTCOGVGT≥CNNQYGF≥6Ø ≥NCUV≥VJGCF≥YQUG≥+&≥PWODGT≥KU≥NUM_THREADS
– 1≥OWUV≥DG≥UWTG≥VQ≥RKEM≥WR≥VQ≥HKPCN≥GNGOGPVU≥QH≥VQ≥X≥CTTC[≥UKPEG≥VQ≥EQFG≥WUGU≥KPVGIGT
FKXKUKQP≥6Ø ≥if≥UVCVGOGPV≥CHVGT≥VQ≥CUUKIPOGPV≥QH≥end≥YKNN≥FQ≥VØU

1PEG≥VQ≥DQWPFCTKGU≥CTG≥MPQYP≥GCE∃VJGCF≥NQQRU≥QXGT≥VQ≥EØPM≥QH≥ X≥CPF≥UWOU≥VQUG≥XCNWGU
KPVQ≥C≥NQECN≥KPVGIGT≥#HVGT≥RTQEGUUKPI≥C≥EØPM≥GCE∃VJGCF≥YKNN≥EQR[≥VQ≥XCNWG≥QH≥VQ≥NQECNN[
EQORWVGF≥RCTVKCN≥UWO≥KPVQ≥C≥WPKSWG≥INQDCN≥CTTC[≥GNGOGPV≥HQT≥VQ≥OCKP≥VJGCF≥VQ≥UWO≥WR≥KPVQ
VQ≥HKPCN≥VQVCN

#U≥HQT≥VQ≥OCKP≥VJGCF≥CHVGT≥KPKVKCNK\KPI≥VQ≥CTTC[≥C≥PWODGT≥QH≥VJGCFU≥CTG≥ETGCVGF≥+oXG≥WUGF
VQ≥FGHKPGF≥EQPUVCPV≥NUM_THREADS≥VQ≥UGV≥VQ≥PWODGT≥QH≥VJGCFU≥KP≥ŽCORNG≥≥ĆE∃VJGCFoU
+&≥PWODGT≥KU≥UVQTGF≥KP≥C≥PGYN[≥CNNQECVGF≥KPVGIGT≥CPF≥UGPV≥VQ≥VQ≥ETGCVGF≥VJGCF≥VJQWI∃VQ
UKPING≥RCTCOGVGT≥$HQTG≥VGTOKPCVKQP≥VJGCFU≥OWUV≥DG≥UWTG≥VQ≥HTGG≥VQ≥OGOQT[≥CNNQECVGF≥HQT
VQ≥VJGCF≥+&≥VQ≥CXQKF≥C≥OGOQT[≥NGCM

6Ø ≥OCKP≥VJGCF≥YCKVU≥HQT≥GCE∃QH≥VQ≥ETGCVGF≥VJGCFU≥VQ≥EQORNGVG≥GZGEWVKQP≥1PEG≥C≥VJGCF
ØU≥VGTOKPCVGF≥VQ≥OCKP≥VJGCF≥YKNN≥MPQY≥VQV≥VQ≥CUUQEKCVGF≥RCTVKCN≥UWO≥HQT≥VQV≥VJGCF≥KU
CXCKNCDNG≥KP≥VQ≥INQDCN≥CTTC[≥CPF≥ECP≥DG≥CFFGF≥VQ≥VQ≥QXGTCNN≥sum≥XCTKCDNG≥6Ø ≥VQVCN≥UWO≥KU
RTKPVGF≥CHVGT≥CNN≥RCTVKCN≥UWOU≥ØXG≥DGGP≥EQODKPGF

#UVWVG≥TGCFGTU≥YKNN≥TGCNK\G≥VQV≥QPEG≥VQ≥FCVC≥ØU≥DGGP≥TGFWEGF≥VQ≥C≥PWODGT≥QH≥RCTVKCN≥UWOU
GSWCN≥VQ≥VQ≥PWODGT≥QH≥VJGCFU≥[QW≥EQWNF≥WUG≥C≥EQPEWTTGPV≥CNIQTKVØ≥DCUGF≥QP≥VQ≥24#/
CNIQTKVØ≥9GoNN≥VCMG≥C≥NQQM≥CV≥CPQVØT≥OGVØF≥HQT≥C≥TGFWEVKQP≥EQORWVCVKQP≥DCUGF≥OQTG≥ENQUGN[
QP≥VQ≥24#/≥CNIQTKVØ≥KP≥%ØRVGT≥

Design Factor Scorecard

How efficient, simple, portable and scalable is the parallel sum code described earlier? Let's examine the algorithm with respect to each of these categories.

Efficiency

The addition of code to compute chunk boundaries is overhead to the concurrent execution. However, if the addition of two assignments and an if≥UVCVGOGPV≥ECWUGU≥C≥PQVKEGCDNG
FGITCFCVKQP≥QH≥RGTHQTOCPEG≥VQ≥FCVC≥EØPMU≥OC[≥PQV≥DG≥DKI≥GPQWI∃VQ≥YCTTCPV≥EQPEWTTGPV
GZGEWVKQP≥+P≥VØU≥ECUG≥[QW≥PGGF≥VQ≥KPETGCUG≥VQ≥ITCPWNCTKV[≥QH≥VQ≥EØPMU≥D[≥TGFWEKPI≥QT
EQORNGVGN[≥GNKOKPCVKPI≥VQ≥PWODGT≥QH≥VJGCFU≥WUGF

;QW≥ECP≥EQORNGVGN[≥GNKOKPCVG≥VQ≥PGGF≥HQT≥EQORWVKPI≥EØPM≥DQWPFCTKGU≥D[≥UVCTVKPI≥GCEJ
VJGCF≥CV≥C≥FKHHGTGPV≥GNGOGPV≥QH≥VQ≥CTTC[≥DCUGF≥QP≥VJGCF≥+&≥CPF≥CEEGUUKPI≥GXGT[≥HQWTV∃QT
NUM_THREADS^VJ≥GNGOGPV≥6Ø ≥EQFG≥EØPIG≥HQT≥VØU≥UEØOG≥KU≥IKXGP≥KP≥ŽCORNG≥

EXAMPLE 6-5. Computation loop modification to eliminate need for start and end

```
for (int i = tNum; i < N; i += NUM_THREADS)
  lSum += A[i];
```

6[KU≥E[CPIG≥KPVTQFWEGU≥C≥PGGF≥VQ≥U[CTG≥ECE[G≥NKPGU≥DGVYGGP≥EQTGU≥YKV3VJGCFU≥CEEGUUKPI≥GXGT[
HQWTV3IGNGOGPV≥Z[CORNG≥≥*QYGXGT≥UKPEG≥V[KU≥YKNN≥QPN[≥KPXQNXG≥TGCFCEEGUU≥QH≥V[G≥ECE[G
FCVC≥V[CTG≥U[QWNF≥DG≥PQ≥HCNUG≥U[CTKPI≥RGPCNV[≥5VKNN≥V[KU≥OQFKHKECVKQP≥YKNN≥OCMG≥WUG≥QH≥QPN[≥C
UOCNN≥UGV≥QH≥KVGOU≥YKV[KP≥C≥YQNG≥ECE[G≥NKPG≥6[WU≥V[G≥VKOG≥VQ≥TGCF≥C≥HWNN≥ECE[G≥NKPG≥YKNN≥DG
RTCEVKECNN[≥YCUVGF≥FWG≥VQ≥V[G≥HCEV≥V[V≥PQV≥CNN≥QH≥V[G≥FCVC≥KP≥GCE3NKPG≥KU≥WUGF≥D[≥C≥VJGCF

6[CTG≥YKNN≥DG≥UQOG≥HCNUG≥U[CTKPI≥RQVGPVKCN≥YKV3OWNVKRNG≥VJGCFU≥WRFCVKPI≥FKHHGTGPV≥GNGOGPVU
QH≥V[G≥gSum≥CTTC[≥*QYGXGT≥KV≥KU≥CNYC[U≥DGVVGT≥VQ≥[XG≥QPG≥RQUUKDNG≥HCNUG≥U[CTKPI≥KPEKFGPV≥RGT
VJGCF≥V[P≥VQ≥[XG≥QPG≥RGT≥FCVC≥KVGO≥KH≥V[G≥VJGCFU≥CTG≥WRFCVKPI≥C≥UKPING≥U[CTGF≥INQDCN≥XCTKCDNG
HQT≥GCE3CFFKVKQP≥PQV≥VQ≥OGPVKQP≥V[G≥OWNVKRNG≥VJGCFU≥EQPVGPFKPI≥HQT≥V[G≥UCOG
U[PEJQPK\CVKQP≥QDLGEV≥#U≥CP≥CNVGTPCVKXG≥[QW≥EQWNF≥QHHUGV≥GCE3IGNGOGPV≥KP≥gSum≥WUGF≥D[≥V[G
PWODGT≥QH≥D[VGU≥KP≥C≥ECE[G≥NKPG
```

## Simplicity

The two concurrent solutions here are very simple data decompositions of the original serial code. It is straightforward to compute partial sums of chunks from the array and then add those partial sums together into a final total.

While there appears to be a huge code explosion between the simple three-line serial version in Example 6-1 and the Pthreads version, the ratio in this case is a bit lopsided due to the very tiny amount of serial code that was used initially. Other algorithms that we will examine will start with more code, and, in my experience, the absolute number of added lines for implementing explicit threading will typically be close to the number of additional lines shown in Example 6-4.

## Portability

The Pthreads algorithm translates pretty easily to other explicit threading models. There is nothing unique to Pthreads used here. The explicit threads algorithm uses the same idea as a distributed-memory algorithm. Processes are assigned a chunk of data, the data is summed locally, and the partial results are sent to a single process for final summation. If all processes require the answer, the final answer must be broadcast to all the other processes.

## Scalability

With increases in the amount of data, the algorithm will scale well. If the data size remains fixed but the number of cores and threads increases, there will be a point (of diminishing returns) where dividing up the data into smaller chunks for more threads will begin to yield worse performance. TBB attempts to control the chunk size behind the scenes, while OpenMP has the schedule≥ENCWUG≥VQ≥IKXG≥V[G≥RTQITCOOGT≥UQOG≥EQPVTQN≥(QT≥CP≥GZRNKEKVN[≥VJGCFGF
```

Prefix Scan

Prefix scan computes all partial sums of a vector of values. That is, the results of a prefix scan will be a vector[2] the same size as the original vector[2] where each element is the sum of the preceding elements in the original vector up to the corresponding position. Two uses of the results of this algorithm are to find the longest sequence of 1s in a binary array (sequence) and packing only the desired elements from an array (which I'll demonstrate later in this chapter). See *Principles of Parallel Programming* by Calvin Lin and Lawrence Snyder (Pearson Education, 2008) or Selim G. Akl's *Parallel Computation* (Prentice Hall, 1997) for other uses of prefix scan.

Figure 6-3 shows an eight-element vector and the results of the prefix scan operation on that vector.

Original vector	3	5	2	5	7	9	4	6
Inclusive prefix scan	3	8	10	15	22	31	35	41
Exclusive prefix scan	0	3	8	10	15	22	31	35

FIGURE 6-3. Prefix scan examples

There are two versions of prefix scan: inclusive and exclusive. For an inclusive scan, the result is the sum of all preceding values, as well as the value of the element in the position under consideration. To compute the inclusive prefix scan of the fifth position in the vector, you need to add 3 5 2 5 7 = 22. The exclusive version does not include the value of the vector element at the position of interest. Thus, the exclusive scan result for the fifth position is 3 5 2 5 = 15.

As with parallel sum, the prefix scan operation can work with any associative combining operation. Besides addition, you can also use the operations of multiplication, maximum, minimum, and such logical operations as AND, OR, and eXclusive OR (but I'll continue to make reference to prefix scan using the addition operation). Unlike parallel sum, the combining operation does not need to be commutative because of the fixed order used to combine elements. Example 6-6 shows serial code to implement the inclusive prefix scan of an array of
N≥KPVGIGTU≥UVQTKPI≥VQ≥TGUWNVU≥KP≥EQTTGURQPFKPI≥GNGOGPVU≥QH≥VQ≥CTTC[≥prefixScan

EXAMPLE 6-6. Serial prefix scan computation of integer array

```
prefixScan[0] = A[0];
for (int i = 1; i < N; i++)
  prefixScan[i] = prefixScan[i-1] + A[i];
```

6[illegible]≥EQFG≥HKTUV≥UVQTGU≥V[illegible]≥A[0]≥XCNWG≥KPVQ≥V[illegible]≥prefixScan[0]≥UNQV≥VQ≥KPKVKCNK\G≥V[illegible]≥TGUWNVU≥CTTC[
5WDUGSWGPV≥KVGTCVKQPU≥QH≥V[illegible]≥NQQR≥WUG≥V[illegible]≥i-1≥XCNWG≥QH≥V[illegible]≥TGUWNVU≥CTTC[≥CFFGF≥VQ≥V[illegible]≥A[i]
XCNWG≥VQ≥EQORWVG≥V[illegible]≥XCNWG≥VQ≥DG≥UVQTGF≥KP≥V[illegible]≥prefixScan[i]≥GNGOGPV

> **NOTE**
>
> **If you want to have an in-place version of the serial inclusive prefix scan, simply replace the** `prefixScan` **CTTC[[illegible]**

[illegible]

PRAM Algorithm

For those of you still reading, let's first take a look at how you can implement the prefix scan operation on the PRAM model. In ®Data Parallel Algorithms¯ (*Communications of the ACM* , 1986), Hillis and Steele recognized that in their parallel sum algorithm (Example 6-2), most of the processors would be idle for most of the time. In fact, half of the processors would never be used at all. Making good use of the idle processors enables machines to carry out the computation of all partial sums in the same execution time as the parallel sum algorithm.

To illustrate this better utilization of processors, Figure 6-4 shows how data will be accessed (arrows) and combined in-place for computing the prefix scan on an eight-element array. Rather than showing the data, the figure shows the range of indexes that have been summed (using bracket notation) within the array element.

FIGURE 6-4. Example of PRAM computation for prefix scan

Before I give you the code for this operation, let's examine the operation of the PRAM algorithm by focusing on the slot of the array with index 5 and see how the prefix scan of the preceding elements is computed in that slot. The initial value is simply the data that is stored there (denoted by [5]≥+P≥VG≥HKTUV≥UVGR≥VG≥RTQEGUUQT≥CUUKIPGF≥VQ≥VGV≥RQUKVKQP≥QH≥VG≥CTTC[≥YKNN
TGCF≥VG≥EWTTGPV≥EQPVGPVU≥QH≥VG≥KPFGZ≥≥GNGOGPV≥CPF≥YKNN≥CFF≥VGV≥XCNWG≥VQ≥VG≥EQPVGPVU≥QH≥VG
KPFGZ≥≥UNQV≥[KGNFKPI≥VG≥UWO≥QH≥VG≥KPFGZ≥≥CPF≥≥GNGOGPVU≥[4,5]≥6G≥RTQEGUUQT≥YKNN≥VGP
TGCF≥VG≥EWTTGPV≥EQPVGPVU≥QH≥VG≥KPFGZ≥≥GNGOGPV≥CPF≥YKNN≥CFF≥VGV≥XCNWG≥VG≥UWO≥QH≥VG≥QTKIKPCN
KPFGZ≥≥CPF≥≥GNGOGPVU≥VQ≥VG≥EQPVGPVU≥QH≥VG≥KPFGZ≥≥UNQV≥6KU≥YKNN≥DG≥VG≥UWO≥QH≥VG≥QTKIKPCN
GNGOGPVU≥HTQO≥KPFGZ≥≥VJQWI3CPF≥KPENWFKPI≥KPFGZ≥≥[2,5]≥+P≥VG≥PGZV≥UVGR≥VG≥HKPCN≥UVGR
HQT≥GKIV≥GNGOGPVU≥VG≥RTQEGUUQT≥YKNN≥TGCF≥VG≥EWTTGPV≥EQPVGPVU≥QH≥VG≥KPFGZ≥≥UNQV≥CPF≥YKNN
CFF≥VKU≥XCNWG≥VQ≥VG≥EQPVGPVU≥QH≥VG≥KPFGZ≥≥UNQV≥YKE3IKXGU≥WU≥VG≥UWO≥QH≥CNN≥GNGOGPVU≥KPFGZGF
HTQO≥≥VJQWI3≥[0,5]≥;QW≥ECP≥HQNNQY≥VG≥CTTQYU≥KP≥VG≥HKIWTG≥VQ≥UGG≥QY≥GCE3KPFKXKFWCN
UWO≥KU≥EQORWVGF≥HTQO≥QPG≥UVGR≥VQ≥VG≥PGZV

9G≥MPQY≥VGV≥GCE3UVGR≥KNNWUVTCVGF≥KP≥(KIWTG≥≥KU≥EQORNGVGF≥DGHQTG≥VG≥PGZV≥UVGR≥DGIKPU
DGECWUG≥VG≥24#/≥OQFGN≥YQTMU≥KP≥NQEMUVGR≥GZGEWVKQP≥6G≥CNIQTKVO≥CUUWOGU≥VGTG≥KU≥C
UGRCTCVG≥RTQEGUUQT≥CUUKIPGF≥VQ≥EQORWVG≥GCE3GNGOGPV≥YKVKP≥VG≥CTTC[≥#NN≥RTQEGUUQTU≥YKNN≥VCMG
VG≥UCOG≥VKOG≥VQ≥TGCF≥VG≥EWTTGPV≥EQPVGPVU≥QH≥C≥NQYGTKPFGZGF≥UNQV≥CFF≥VG≥XCNWG≥HQWPF≥VQ≥VG
EWTTGPV≥EQPVGPVU≥QH≥VG≥CUUKIPGF≥CTTC[≥GNGOGPV≥CPF≥UVQTG≥VG≥TGUWNV≥DCEM≥KPVQ≥VG≥CUUKIPGF≥CTTC[
RQUKVKQP≥6KU≥VGTG≥KU≥PQ≥PGGF≥HQT≥GZRNKEKV≥U[PEJQPK\CVKQP≥CPF≥YG≥MPQY≥VGTG≥KU≥PQ≥EGPEG
QH≥FCVC≥TCEGU≥KP≥YKE3QPG≥RTQEGUUQT≥CVVGORVU≥VQ≥TGCF≥C≥XCNWG≥CV≥VG≥UCOG≥VKOG≥VG≥CUUKIPGF
RTQEGUUQT≥KU≥WRFCVKPI≥VG≥EQPVGPVU≥QH≥CP≥CTTC[≥GNGOGPV

6G≥RUGWFQEQFG≥HQT≥VG≥24#/≥KPENWUKXG≥RTGHKZ≥UECP≥CNIQTKVO≥IKXGP≥KP≥ZCORNG≥≥KU≥CFCRVGF
HTQO≥*KNNKU≥CPF≥5VGGNGoU≥p&CVC≥2CTCNNGN≥#NIQTKVOUq≥6G≥QPN[≥FKHHGTGPEG≥DGVYGGP≥VKU≥EQFG≥CPF

FIGURE 6-5. Prefix scan algorithm example

The first step (local prefix scan) and third step (adding the preceding chunk's sum to local chunk) can be done independently. The second step requires a prefix scan computation with a number of elements equal to the number of threads. We could implement an adaptation of the PRAM algorithm for this situation or take a hint from the parallel sum implementation and do this second step in serial.

If we do the second step in serial, we don't want to terminate the threads after completion of the first step, since we would need to recreate them for the third step. Starting and stopping threads is too much overhead and should be avoided if at all possible. Still, the serial thread executing the second step needs to know when all the chunk totals have been computed before it can start the prefix scan of the chunk totals. And, on that same side of the coin, the computational threads must wait until the serial prefix scan of chunk totals is complete before starting on the third step. Thus, we need some way to signal the completion of these two events.

Example 6-8 shows threaded code using Windows Threads and event synchronization objects to implement an inclusive prefix scan. As before, to keep things simple, I've written code that handles integers as the data on which to find the prefix scans, and set the number of threads to a fixed value (`NUM_THREADS≥&GVCKNU≥CDQWV≥QY≥VC≥FCVC≥KU≥KPKVKCNK\GF≥InitializeArray()≥CTGQOKVVGF`

EXAMPLE 6-8. Prefix scan using Windows Threads

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
```

```
#define NUM_THREADS 4

int N, *A;
int inTotals[NUM_THREADS], outTotals[NUM_THREADS];
HANDLE doneStep1[NUM_THREADS];
HANDLE doneStep2;

unsigned __stdcall prefixScan(LPVOID pArg)
{
  int tNum = *((int *) pArg);
  int start, end, i;
  int lPrefixTotal;

  free(pArg);
  start = (N / NUM_THREADS) * tNum;
  end = (N / NUM_THREADS) * (tNum + 1);
  if (tNum == (NUM_THREADS-1)) end = N;

// Step 1
  for (i = start+1; i < end; i++)
    A[i] = A[i-1] + A[i];

  inTotals[tNum] = A[end-1];
  SetEvent(doneStep1[tNum]); //signal completion of Step 1

// wait for completion of Step 2
  WaitForSingleObject(doneStep2, INFINITE);

// Step 3
  lPrefixTotal = outTotals[tNum];
  for (i = start; i < end; i++)
    A[i] = lPrefixTotal + A[i];

  return 0;
}

int main(int argc, char* argv[])
{
  int i, j;
  HANDLE tH[NUM_THREADS];

  InitializeArray(A,&N);  // get values into A array; not shown
// Create manual reset events initially unsignaled
  for (i = 0; i < NUM_THREADS; i++)
    doneStep1[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
  doneStep2 = CreateEvent(NULL, TRUE, FALSE, NULL);

  for (i = 0; i < NUM_THREADS; i++) {
    int *tnum = new int;
    *tnum = i;
    tHandles[i] = (HANDLE) _beginthreadex(NULL, 0, prefixScan, (LPVOID) tnum, 0, NULL);
  }
// wait for Step 1 completion
  WaitForMultipleObjects(NUM_THREADS, doneStep1, TRUE, INFINITE);
```

```
// Step 2
  outTotals[0] = 0;
  for (j = 1; j < NUM_THREADS; j++)
    outTotals[j] = outTotals[j-1] + inTotals[j-1];

  SetEvent(doneStep2);  //signal completion of Step 2

// wait for completion of Step 3
  WaitForMultipleObjects(NUM_THREADS, tHandles, TRUE, INFINITE);

  return 0;
}
```

What the main thread does

The main thread creates an array of event objects (doneStep1≥HQT≥VG≥EQORWVCVKQP≥VJGCFU≥VQ
WUG≥KP≥UKIPCNKPI≥YGP≥VG[≥GXG≥EQORNGVGF≥VG≥YQTM≥KP≥UVGR≥≥#NUQ≥C≥UKPING≥GXGPV≥QDLGEV
doneStep2≥KU≥UGV≥WR≥HQT≥VG≥OCKP≥VJGCF≥VQ≥UKIPCN≥VGV≥KV≥GU≥EQORNGVGF≥VG≥GZENWUKXG≥RTGHKZ
UECP≥EQORWVCVKQP≥QH≥VG≥EWPM≥VQVCNU≥#HVGT≥VG≥GXGPVU≥CTG≥UGV≥WR≥VG≥EQORWVCVKQP≥VJGCFU≥CTG
ETGCVGF≥CPF≥VG≥RTQEGUU≥VJGCF≥YCKVU≥HQT≥CNN≥QH≥VG≥doneStep1≥GXGPVU≥VQ≥DG≥KP≥VG≥UKIPCNGF≥UVCVG

6G≥GZENWUKXG≥RTGHKZ≥UECP≥EQORWVCVKQP≥HQT≥UVGR≥≥WUGU≥VYQ≥CTTC[U≥QPG≥VQ≥QNF≥VG≥QTKIKPCN≥FCVC
CPF≥QPG≥VQ≥QNF≥VG≥UECP≥TGUWNVU≥6KU≥OCMGU≥VG≥KORNGOGPVCVKQP≥UKORNG≥UKPEG≥KV≥QPN[≥TGSWKTGU
QPG≥NKPG≥QH≥EQFG≥6KU≥QPG≥NKPG≥QH≥EQFG≥YQTMU≥DGECWUG≥YG≥RTGUGTXG≥VG≥QTKIKPCN≥FCVC≥GXGP≥YGP
VG≥EQTTGURQPFKPI≥TGUWNV≥GNGOGPV≥KU≥WRFCVGF≥+oXG≥KPENWFGF≥CP≥KPRNCEG≥XCTKCVKQP≥QH≥VKU
GZENWUKXG≥RTGHKZ≥UECP≥KP≥ZCORNG≥

1PEG≥VG≥GZENWUKXG≥UGTKCN≥RTGHKZ≥UECP≥KU≥EQORNGVG≥VG≥OCKP≥VJGCF≥UKIPCNU≥VG≥EQORWVCVKQPCN
VJGCFU≥VGV≥KV≥KU≥UCHG≥VQ≥DGIKP≥UVGR≥≥6G≥QPN[≥VKPI≥NGHV≥HQT≥VG≥OCKP≥VJGCF≥VQ≥FQ≥KU≥YCKV≥HQT
VG≥VGTOKPCVKQP≥QH≥VG≥EQORWVCVKQPCN≥VJGCFU≥VKU≥KPFKECVGU≥VGV≥VG≥UECP≥QH≥VG≥QTKIKPCN≥CTTC[
KU≥EQORNGVG

What the spawned threads are doing

Each computational thread first figures out which chunk is assigned to it using the thread ID
number sent as a parameter to the prefixScan()≥HWPEVKQP≥CPF≥UVQTGF≥NQECNN{≥GU≥6KU≥KU≥FQPG
KP≥VG≥UCOG≥OCPPGT≥CU≥VG≥RCTCNNGN≥UWO≥EQFG≥KP≥ZCORNG≥≥6G≥VJGCFU≥EQORWVG≥VG≥RTGHKZ
UECP≥QP≥VG≥EWPM≥QH≥CUUKIPGF≥FCVC≥6G≥XCNWG≥HTQO≥VG≥HKPCN≥EWPM≥GNGOGPV≥VG≥VQVCN≥QH≥CNN
CTTC[≥GNGOGPVU≥YKVKP≥VG≥EWPM≥KU≥EQRKGF≥VQ≥VG≥UCTGF≥inTotals[tNum]≥UNQV≥5KPEG≥VKU
EQORNGVGU≥RTQEGUUKPI≥HQT≥UVGR≥≥VG≥VJGCF≥UGVU≥VG≥GXGPV≥QDLGEV≥doneStep1[tNum]≥CPF≥YCKVU≥HQT
VG≥UKIPCN≥VGV≥VG≥OCKP≥VJGCF≥GU≥EQORNGVGF≥UVGR≥

#HVGT≥TGEGKXKPI≥VG≥doneStep2≥UKIPCN≥GCE3EQORWVCVKQPCN≥VJGCF≥EQRKGU≥VG≥XCNWG≥HTQO≥VG
outTotals[tNum]≥UNQV≥CPF≥UVQTGU≥VKU≥XCNWG≥KP≥VG≥NQEEN≥xTotal≥XCTKCDNG≥CE3QH≥VG≥RCTVKCN
RTGHKZ≥UECP≥XCNWGU≥HTQO≥UVGR≥≥KP≥VG≥CUUKIPGF≥EWPM≥KU≥WRFCVGF≥YKV3VG≥CFFKVKQP≥QH
lPrefixTotal≥VQ≥EQORWVG≥VG≥HKPCN≥UECP≥TGUWNVU≥6GTOKPCVKQP≥QH≥CNN≥VJGCFU≥ENWGU≥KP≥VG≥OCKP
VJGCF≥VQ≥VG≥HCEV≥VGV≥VG≥EQORWVCVKQP≥KU≥EQORNGVG≥CPF≥VGV≥KV≥ECP≥WUG≥VG≥TGUWNVU≥CU≥PGGFGF

N O T E

If you think back to the beginning of the prefix scan discussion, I stated that the combining operation must be associative but does not have to be commutative. This is due to the unambiguous order in which the results are combined. In "Data Parallel Algorithms," Hillis and Steele make note of this and point out that the code that does the combination within their PRAM pseudocode is written in a specific order, namely:

```
X[k] := X[k - 2^(j-1)] + X[k]
```

YGTG≥VG≥XCNWGU≥CPF≥RTGXKQWU≥TGUWNVU≥VGV≥QEEWT≥KP≥NQYGT≥KPFGZGF≥GNGOGPVU≥CNYC[U≥CRRGCT QP≥VG≥NGHV≥QH≥VG≥EQODKPKPI≥QRGTCVQT≥YKVKP≥VG≥TKIVPF≥UKFG≥QH≥VG≥CUUKIPOGPV≥UVCVGOGPV (CKVHWNPGUU≥VQ≥VKU≥HQTOCV≥YKNN≥RTGUGTXG≥VG≥EQTTGEVPGUU≥QH≥VG≥CNIQTKVO≥KH≥[QW≥GXGT≥HKPF [QWTUGNH≥WUKPI≥C≥PQPEQOOWVCVKXG≥QRGTCVQT≥KP≥RNCEG≥QH≥VG≥CFFKVKQP≥QRGTCVKQP≥UQYP

+oXG≥CFGTGF≥VQ≥VKU≥QTFGTKPI≥KP≥CNN≥QH≥VG≥EQFG≥RTGUGPVGF≥KP≥VKU≥ECRVGT≥TCVGT≥VGP≥WUKPI≥VG UQTVEWV≥QRGTCVQT≥+=

Design Factor Scorecard

How efficient, simple, portable, and scalable is the prefix scan code described earlier? Let's examine the implementation with respect to each of these categories.

Efficiency

As with the parallel sum implementation, the code to compute chunk boundaries is overhead. The same analysis of chunk size applies here. Unlike the parallel sum implementation, the loop code modification to assign alternate iterations to threads shown in Example 6-5 cannot be used since the prefix scan algorithm relies on having contiguous chunks of data elements.

With multiple threads updating different elements of the inTotals≥CTTC[≥FWTKPI≥VG≥NCUV≥RCTV≥QH UVGR≥≥VGTG≥KU≥C≥ECPEG≥HQT≥HCNUG≥UGTKPI≥%QPEWTTGPV≥CEEGUU≥VQ≥VG≥outTotals≥KU≥TGCFQPN[≥UQ VGTG≥UQWNF≥DG≥PQ≥FGVTKOGPV≥HTQO≥HCNUG≥UGTKPI≥QP≥VGV≥CTTC[≥KP≥RTGRCTCVKQP≥HQT≥UVGR≥

+H≥OGOQT[≥KU≥VKIV≥CPF≥CP≥CFFKVKQPCN≥VQVCNU≥CTTC[≥ TotalS≥OC[≥DG≥VQQ≥OWEH≥[QW≥ECP≥WUG≥CP KPRNCEG≥UGTKCN≥XGTUKQP≥QH≥GZENWUKXG≥RTGHKZ≥UECP≥%QFG≥HQT≥VKU≥CNIQTKVO≥KU≥IKXGP≥KP≥ZCORNG≥

EXAMPLE 6-9. In-place exclusive prefix scan in serial

```
int nexT = Totals[0], curr;
for (j = 1; j < NUM_THREADS; j++) {
  curr = Totals[j];
  Totals[j] = nexT;
  nexT = nexT + curr;
}
Totals[0] = 0;
```

1PN[≥VYQ≥GZVTC≥XCTKCDNGU≥TCVGT≥VGP≥CP≥GPVKTG≥CTTC[≥CTG≥PGGFGF≥VQ≥MGGR≥VTCEM≥QH≥VG≥PGZV RTGHKZ≥UECP≥XCNWG≥nexT≥VQ≥DG≥UVQTGF≥KP≥VG≥Totals≥CTTC[≥CPF≥VQ≥QNF≥VG≥EWTTGPV≥FCVC≥XCNWG≥HTQO

```
curr≥DGHQTG≥VCIV≥XCNWG≥KU≥QXGTYTKVVGP≥6CI≥HKPCN≥UVGR≥QH≥VCI≥EQFG≥UGVU≥VCI≥HKTUV≥GNGOGPV
KP≥VCI≥CTTC[≥VQ≥
```

Simplicity

Like parallel sum before, the concurrent solution in Example 6-8 is a data decomposition. This is not a straightforward decomposition of the serial code, since there are three steps, with one of those steps actually being a prefix scan computation across threads. Implementing something closer in structure to the PRAM algorithm for step 2 might be possible, but, due to the required synchronization to keep execution order, the code wouldn't be as simple as the serial version.

Even so, I think once you understand the three steps of the concurrent algorithm and what they accomplish, the code is easy to understand and to modify for some other operation besides addition. Lin and Snyder, in *Principles of Parallel Programming*, give an implementation for an abstract parallel prefix scan method (as well as one for reduction). Their algorithm isolates the combining operation of the scan and would allow you to easily reuse the code with a different combining operation.

Portability

The Windows Threads algorithm translates pretty easily to other explicit threading models. There is nothing uniquely Windows used here. You could implement this algorithm in a distributed-memory environment, too, with some message-passing between steps to gather data for the exclusive prefix scan of totals. The results of the exclusive scan are then scattered back out to processes for local processing.

You could also use this algorithm under OpenMP. The API finds the number of threads (thread ID number are required) and carries out the serial prefix scan of the `Totals`≥CTTC[≥WPFGT≥VCI

```
single≥RTCIOC≥+P≥O[≥QRKPKQP≥KH≥[QW≥CIXG≥VQ≥IQ≥VQ≥VCI≥1RGP/2≥#2+≥HWPEVKQPU≥CPF≥WUG≥VQQUG
XCNWGU≥VQ≥FKXKFG≥VCI≥FCVC≥CTTC[≥KPVQ≥EWPMU≥[QW≥CIXG≥IQPG≥QWVUKFG≥VCI≥TGCNO≥QH≥QY≥DGUV≥VQ
WUG≥1RGP/2≥+H≥[QWoTG≥IQKPI≥VQ≥IQ≥VQ≥VCIV≥VTQWDNG≥YJ≥PQV≥LWUV≥YTKVG≥VCI≥EQFG≥YKVJCP≥GZRNKEKV
VJGCFKPI≥NKDTCT[!≥+H≥[QWoXG≥IQV≥QVCIT≥RCTVU≥QH≥[QWT≥CRRNKECVKQP≥KP≥1RGP/2≥CPF≥YCPV≥VQ≥MGGR
VCI≥GPVKTG≥CRRNKECVKQP≥KP≥1RGP/2≥VCIP≥+oO≥1-≥YKVJYTKVKPI≥C≥UOCNN≥RCTV≥QH≥VCI≥EQFG≥WUKPI
1RGP/2≥CU≥KH≥KV≥YGTG≥GZRNKEKV≥VJGCFU≥;QWoTG≥HTGG≥VQ≥FQ≥YCIV≥[QW≥YCPV≥QH≥EQWTUG≥+≥LWUV≥CUM
VCIV≥[QW≥FQPoV≥VGNN≥OG≥CDQWV≥KV
```

Scalability

The prefix scan code given here is going to have the same scaling characteristics as the parallel sum. With an explicit threads implementation, you'll need to find where too little data in each chunk causes the implementation performance to degrade below acceptable bounds.

Selection

Studying parallel sum and prefix scan is instructive in its own right, but there are actually larger algorithms that incorporate these operations. Selecting the kth-largest element from an unsorted list is a good example. In this section I'll describe serial and concurrent solutions for this example algorithm and show how you can apply the parallel sum and prefix scan operations.

One obvious solution is to first sort the elements of the list and then pick out the item in the kth position. If you need to pull out different k^{th} items multiple times, it might be worthwhile to do the sorting. If, however, you only need to do this once, or if the list is updated between selections, you can use an algorithm with a better asymptotic complexity than sorting, namely $O(n)$.

The Serial Algorithm

When I was first shown this algorithm, early in my computer science academic years, I remember being flabbergasted at the simplicity of the idea and the insight it took to think of this problem in the way that led to this solution. The crux of the algorithm is in finding the median of a list. You will recall that the median of a data set is the item in the middle of the set: half the data is smaller than the median element, and half the data is greater than the median value. For a data set of n items, this median would be in the

$$k = \left\lceil \frac{n}{2} \right\rceil$$

position if the data were sorted.

The serial algorithm for selection is recursive and can be described with five algorithmic steps

1. If the size of the data set to be used is less than some constant size, Q, sort the data and return the kth element: otherwise, subdivide the data set into chunks of size Q and whatever is left over.
2. Sort each chunk and find the median of each.
3. Recursively call the selection routine to find the median of the medians found in the previous step.
4. Partition the data set into three subsequences median of medians, those that are equal to the median of medians, and those that are greater than the median of medians.
5. Determine which subsequence contains the kth element, from the sizes of the three subsequences, and recursively call the selection routine on that subsequence. If the kth element is not in the subsequence of smaller or larger items, it must be in the subsequence equal to the median of medians, so just return the median of medians value.

SequentialSelect()≥HWPEVKQP≥6G≥EQFG≥HQT≥VYQ≥QH≥VG≥TGSWKTGF≥UWRRQTV≥HWPEVKQPU
CountAndMark()≥CPF≥ArrayPack()≥CTG≥IKXGP≥NCVGT≥KP≥7CORNG≥≥(TQO≥VG≥CPCN[UKU≥QH≥5GNKO≥)
#MN≥KP≥6G≥&GUKIP≥CPF≥#PCN[UKU≥QH≥2CTCNNGN≥#NIQTKVJOU≥TGPVKEG≥*CNN≥≥VG≥XCNWG≥QH≥ Q≥ECP
DG≥CP[≥KPVGIGT≥ITGCVGT≥VG≥P≥QT≥GSWCN≥VQ≥≥+oXG≥EQUGP≥VQ≥FGHKPG≥≥HQT≥O[≥KORNGOGPVCVKQP

EXAMPLE 6-10. Serial code to implement the selection algorithm

```
int SequentialSelect(int *S, int num, int k)
{
  if (num <= Q) return SortLessThanQ(S, num, k);

  int cNum = num/Q + 1;
  int *Medians = new int[cNum];
  int i = 0;
  for (int j = 0; j < num/Q; j++) {
    Medians[j] = SortSelect5(&S[i], 3);  // find medians of subsequences
    i += Q;
  }
  int lastNum = num - (Q * (num / Q));
  if (lastNum) {
    int lastQ = Q * (num / Q);
    Medians[cNum-1] = SortLessThanQ(&S[lastQ], lastNum, (lastNum+1)/2);
  }
  else cNum--;

  int M = SequentialSelect(Medians, cNum, (cNum+1)/2);

  int leg[3] = {0,0,0};
  int *markS = new int[num];
  CountAndMark(S, markS, num, M, leg);

  if (leg[0] >= k) {
    int *sPack = new int[leg[0]];
    ArrayPack(S, sPack, num, markS, 0);
    return SequentialSelect(sPack, leg[0], k);
  }
  else if ((leg[0] + leg[1]) >= k) return M;
  else {
    int *sPack = new int[leg[2]];
    ArrayPack(S, sPack, num, markS, 2);
    return SequentialSelect(sPack, leg[2], k-(leg[0]+leg[1]));
  }
}
```

6G≥RCTCOGVGTU≥VQ≥VG≥SequentialSelect()≥TQWVKPG≥CTG≥VG≥CTTC[≥QH≥FCVC≥VQ≥DG≥UGNGEVGF≥HTQO≥
VG≥PWODGT≥QH≥GNGOGPVU≥KP≥VG≥CTTC[≥num≥CPF≥VG≥UGNGEVKQP≥KPFGZ≥k≥(QT≥UKORNKEKV[≥+oXG≥OCFG
S≥CP≥CTTC[≥QH≥KPVGIGTU≥6G≥HKTUV≥UVGR≥KU≥VQ≥FGVGTOKPG≥YG≥VG≥T≥VG≥TG≥CTG≥OQTG≥VCNNGF≥VG≥OU≥VQ≥DG
UGCTEG≥F≥+H≥PQV≥VG≥EQFG≥ECNNU≥SortLessThanQ()≥PQV≥UQ≥YP≥YG≥E3UKORN[≥EQPVCKPU≥C≥switch
UVCVGOGPV≥VQ≥UGNGEV≥VG≥CPFEQFGF≥UQTV≥TQWVKPG≥HQT≥VG≥URGEKHKE≥XCNWGU≥NGUU≥VG≥P≥QT≥G≥SWCN≥VQ≥
VG≥PWODGT≥QH≥GNGOGPVU≥KP≥VG≥NKUV≥KU≥NCTIGT≥VG≥PWODGT≥QH≥ QUK\GF≥EQ≥PMU≥KU≥EQORWVGF

median≥KP≥VG≥Marks≥CTTC[≥CPF≥KPETGOGPVU≥VG
CRRTQRTKCVG≥GNGOGPV≥QH≥VG≥leg≥CTTC[≥oQVKEG≥VGV≥VG≥HWPEVKQP≥EQORWVGU≥VJGG≥UWOU≥DCUGF≥QP
VG≥EQPVGPVU≥QH≥CP≥CTTC[

6G≥ArrayPack()≥HWPEVKQP≥VCMGU≥C≥NKUV≥QH≥GNGOGPVU≥S≥CPF≥CP≥CTTC[≥QH≥KVGOU≥PQVKPI≥YKEJ
RCTVKVKQP≥VG≥GNGOGPVU≥HTQO≥S≥UQWNF≥DG≥CUUKIPGF≥Marks≥XGT[YGTG≥VG≥Marks≥GNGOGPV≥CPF
VG≥ scanSym≥OCVEJ≥VG≥EQTTGURQPFKPI≥GNGOGPV≥HTQO≥ S≥KU≥RCEMGF≥KPVQ≥ArrayPack≥9KNG≥KV≥OC[≥PQV≥DG
QDXKQWU≥[QW≥ECP≥FQ≥VKU≥QRGTCVKQP≥YKVJ≥C≥RTGHKZ≥UECP≥QH≥VG≥Marks≥CTTC[≥+oNN≥UQY≥[QW≥QY
VGVoU≥FQPG≥KP≥VG≥PGZV≥UGEVKQP

+≥QRG≥[QW≥ECP≥UGG≥VGV≥VKU≥CNIQTKVO≥YQTMU≥D[≥WUKPI≥VG≥OGFKCP≥QH≥OGFKCPU≥VQ≥RCTVKVKQP≥VG
FCVC≥KPVQ≥VJGG≥ITQWRU≥CPF≥VGP≥KFGPVKH[KPI≥VG≥RCTVKVKQP≥KP≥YKEJ≥VGV≥MGNGOGPV≥YQWNF≥DG
HQWPF≥6Q≥RTQXG≥VGV≥VG≥CNIQTKVO≥TWPU≥KP≥NKPGCT≥VKOG≥YG≥PGGF≥VQ≥FGOQPUVTCVG≥VGV≥CV≥NGCUV
≥QH≥VG≥FCVC≥YKNN≥DG≥KP≥VG≥RCTVKVKQP≥VGV≥FQGU≥PQV≥EQPVCKP≥VG≥KVGO≥QH≥KPVGTGUV≥6KU≥KU
KNNWUVTCVGF≥KP≥(KIWTG≥

FIGURE 6-6. Selection diagram

Figure 6-6 contains 11 subsequences of 5 elements each. The 5 elements in each subsequence have been sorted. For the sake of the figure, assume that I've arranged these subsequences in sorted order by their median value. We can quickly identify which element would be the median of medians (shown as a star in the figure). The circled collection of elements is the set of items known to be less than or equal to the median of medians. Thus, if the element being sought through selection is greater than the median of medians, we are guaranteed to be able to eliminate at least 3 5 of half (or 30) of the data for each recursive call in the final step of the algorithm. We can make the same claim if the item is smaller than the median of medians by shifting the gray box to surround the items in the lower-right corner of the elements.

If you're interested in the details of a more rigorous proof of runtime for this serial algorithm, see Akl's *The Design and Analysis of Parallel Algorithms* *or* *Fundamentals of Algorithmics* *(Prentice Hall, 1996) by Gilles Brassard and Paul Bratley.*

The Concurrent Algorithm

Three points within the serial algorithm contain independent operations of the medians from each of the [redacted]

sPack≥CTTC[≥YKV≥VG≥KVGOU≥HTQO≥VG≥RCTVKVKQP≥EQPVCKPKPI≥VG≥NGNGOGPV
+oNN≥FKUEWUU≥GCE≥QH≥VG≥UG≥KP≥VWTP≥CPF≥UQY≥[QW≥QY≥VQ≥VTCPUHQTO≥VG≥UGTKCN≥EQFG≥VQ≥C≥EQPEWTTGPV
GSWKXCNGPV≥WUKPI≥6$

+oXG≥EQUGP≥VQ≥WUG≥6$DGECWUG≥KV≥EQPVCKPU≥VG≥parallel_scan≥CNIQTKVO≥YKE≥OCMGU≥KV≥GCUKGT
VQ≥KORNGOGPV≥VG≥CTTC[≥RCEMKPI≥#NUQ≥VG≥RQKPVU≥QH≥EQPEWTTGPE[≥CTG≥UGRCTCVGF≥D[≥UQOG≥UGTKCN
EQFG≥6W≥WUKPI≥6$CNNQYU≥WU≥VQ≥EQPUVTWEV≥VG≥EQPEWTTGPE[≥CV≥VQUG≥RQKPVU≥VGV≥ECP≥DG≥OCFG
EQPEWTTGPV≥CPF≥VQ≥GPUWTG≥EQPEWTTGPV≥QRGTCVKQPU≥CTG≥EQORNGVGF≥DGHQTG≥RTQEGGFKPI≥VQ≥VG≥PGZV
UVGR≥QH≥VG≥CNIQTKVO≥1RGP/2≥YQWNF≥IKXG≥WU≥VG≥UCOG≥U[PEJTQPK\CVKQP≥QH≥VJTGCFU≥DGVYGGP
RCTCNNGN≥TGIKQPU≥DWV≥YKVQWV≥C≥RTGHKZ≥UECP≥QRGTCVKQP≥YGoF≥CXG≥CF≥VQ≥YTKVG≥QPG≥D[≥CPF≥#P
GZRNKEKV≥VJTGCFU≥UQNWVKQP≥YQWNF≥TGSWKTG≥WU≥VQ≥KPEQTRQTCVG≥VG≥NQIKE≥VQ≥FQ≥VG≥GZGEWVKQP≥QTFGT
U[PEJTQPK\CVKQP≥+oO≥IQKPI≥VQ≥YCKV≥WPVKN≥%CRVGT≥≥DGHQTG≥IKXKPI≥FGVCKNU≥CPF≥CP≥GZCORNG≥VGV
KNNWUVTCVGU≥YGV≥ECP≥DG≥FQPG≥YKV≥GZRNKEKV≥VJTGCFU≥VGV≥PGGF≥VQ≥EQQTFKPCVG≥GZGEWVKQP≥QTFGT≥QH
VCUMU≥YKVKP≥VJTGCFU

$HQTG≥YG≥GZCOKPG≥GCE≥QH≥VG≥ECPIGU≥HQT≥EQPEWTTGPE[≥KP≥FGRV≥NGVoU≥NQQM≥CV≥VG≥ECPIGU≥VQ
VG≥QXGTCNN≥UGNGEVKQP≥HWPEVKQP≥ParallelSelect()≥6KU≥EQFG≥UGIOGPV≥KU≥UQYP≥KP
ZCORNG≥≥+oXG≥NGHV≥QWV≥VG≥TGSWKTGF≥6$include≥NKPGU≥;QW≥ECP≥WUG≥VKU≥CU≥CP≥GZGTEKUG≥VQ
FGVGTOKPG≥YGV≥T≥[QW≥ECP≥KFGPVKH[≥CNN≥VG≥KPENWFG≥HKNGU≥VGV≥CTG≥PGGFGF≥YGP≥[QW≥KORNGOGPV
EQPEWTTGPV≥EQFGU≥YKV≥6$

EXAMPLE 6-12. ParallelSelect() code

```
int ParallelSelect(int *S, int num, int k)
{
  if (num <= Q) return SortLessThanQ(S, num, k);

  int cNum = num/Q + 1;
  int *Medians = new int[cNum];

  parallel_for(blocked_range<int>(0, num/Q), FindMedians(S, Medians), auto_partitioner());

  int lastNum = num - (Q * (num / Q));
  if (lastNum) {
    int lastQ = Q * (num / Q);
    Medians[cNum-1] = SortLessThanQ(&S[lastQ], lastNum, (lastNum+1)/2);
  }
  else cNum--;

  int M = ParallelSelect(Medians, cNum, (cNum+1)/2);

  int *markS = new int[num];
  CountAndMark camBody(S, markS, M);
  parallel_reduce(blocked_range<size_t>(0,num), camBody, auto_partitioner());

  int numLessEqual = camBody.leg.less + camBody.leg.equal;
  if (camBody.leg.less >= k) {
    int *sPack = new int[camBody.leg.less];
    ArrayPack(S, sPack, num, markS, 0);
```

```
    return ParallelSelect(sPack, camBody.leg.less, k);

  }
  else if ((numLessEqual) >= k) return M;
  else {
    int *sPack = new int[camBody.leg.greater];
    ArrayPack(S, sPack, num, markS, 2);
    return ParallelSelect(sPack, camBody.leg.greater, k-(numLessEqual));
  }
}
```

6G≥DCUKE≥UVTWEVWTG≥QH≥VG≥EQPEWTTGPV≥EQFG≥KU≥VG≥UCOG≥CU≥VG≥UGTKCN≥XGTUKQP≥HTQO
ZCORNG≥≥6G≥ for≥NQQR≥VGV≥HKPFU≥VG≥UGV≥QH≥OGFKCPU≥QH≥VG≥UWDUGSWGPEGU≥GU≥DGGP≥TGRNCEGF
D[≥ECNNKPI≥VG≥parallel_for≥CNIQTKVO≥QP≥VG≥FindMedians≥ENCUU≥FGHKPGF≥KP≥ZCORNG≥≥#HVGT
VG≥OGFKCP≥QH≥OGFKCPU≥ M≥KU≥FGVGTOKPGF≥C parallel_reduce≥CNIQTKVO≥QP≥VG≥CountAndMark≥ENCUU
FGHKPGF≥NCVGT≥KP≥ZCORNG≥≥EQWPVU≥VG≥PWODGT≥QH≥GNGOGPVU≥KP≥VG≥VTGG≥RCTVKVKQPU≥KP
TGNCVKQP≥VQ≥VG≥XCNWG≥QH≥CPF≥OCTMU≥YHE≥RCTVKVKQP≥GCE≥GNGOGPV≥DGNQPIU≥VQ≥KP≥VG markS≥CTTC[
6G≥RCTVKVKQP≥QH≥KPVGTGUV≥KU≥RCEMGF≥KPVQ≥VG≥sPack≥CTTC[≥D[≥ECNNKPI≥VG≥ArrayPack()≥HWPEVKQP
FGHKPGF≥NCVGT≥KP≥ZCORNG≥≥6H≥KU≥HWPEVKQP≥YKNN≥WUG≥VG≥ parallel_scan≥CNIQTKVO≥VQ≥KFGPVKH[
VG≥GNGOGPVU≥VGV≥PGGF≥VQ≥DG≥ICVGTGF≥KPVQ≥VG≥sPack≥CTTC[≥CPF≥C≥parallel_for≥ECNN≥YKNN≥RCEM
VG≥TGNGXCPV≥KVGOU≥KPVQ≥VG≥CTTC[≥+H≥VG≥XKVGO≥KU≥GSWCN≥VQ≥VG≥OGFKCP≥QH≥OGFKCPU≥VHU≥XCNWG
KU≥KOOGFKCVGN[≥TGVWTPGF≥KPUVGCF

Finding the medians of subsequences

The code for the FindMedians≥ENCUU≥KU≥UQYP≥KP≥ZCORNG≥≥6G≥ENCUU≥KPENWFGU≥VG≥TGSWKTGF
OGVQFU≥HQT≥6$VQ≥FKXKFG≥WR≥VG≥VCUMU≥CPF≥CUUKIP≥VGO≥VQ≥VTGCFU≥VTQWI3VG≥parallel_for
CNIQTKVO

EXAMPLE 6-13. FindMedians class definition

```
class FindMedians {
  int *S;
  int *M;

public:
  void operator()(const blocked_range<int>& r) const {
    int i, j;
    for (j = r.begin(); j < r.end(); j++) {
      i = j * Q;
      M[j] = SortSelect5(&S[i], 3);  // find medians of subsequences
    }
  }

  FindMedians(int *_S, int *_Medians): S(_S), M(_Medians){}

  FindMedians(FindMedians& x, split): S(x.S), M(x.M) {}
};
```

SortSelect5()
HWPEVKQP≥VQ≥UQTV≥VC≥HKXG≥HTQO≥Q≥GNGOGPVU≥QH≥VC≥UWDUGSWGPEG≥+H≥VC≥HKPCN≥EWPM≥CU≥HGYGT
VCP≥3≥GNGOGPVU≥VC≥OGFKCP≥UGNGEVKQP≥QH≥VCV≥NCUV≥EWPM≥KU≥FQPG≥KP≥VC≥parallelSelect()≥TQWVKPG
KOOGFKCVGN[≥HQNNQYKPI≥parallel_for

1PG≥FKHHGTGPEG≥DGVYGGP≥VKU≥NQQR≥CPF≥VC≥EQTTGURQPFKPI≥NQQR≥HTQO≥VC≥UGTKCN≥EQFG≥KU≥VCV≥VC
UGTKCN≥EQFG≥NQQR≥WUGU≥i≥CU≥CP≥KPFWEVKQP≥XCTKCDNG≥KPETGOGPVGF≥D[≥Q≥GCE3KVGTCVKQP≥(QT≥VC
EQPEWTTGPV≥XGTUKQP≥+≥TGEQPHKIWTGF≥VC≥EQFG≥VQ≥EQORWVG≥VC≥XCNWG≥QH≥i≥DCUGF≥QP≥VC≥XCNWG≥QH
j≥KP≥GCE3KVGTCVKQP≥YKE3GNKOKPCVGF≥VC≥KPFWEVKQP≥FGRGPFGPEG≥&KF≥[QW≥PQVKEG≥VC≥TCPIG≥IKXGP
VQ≥VC≥parallel_for≥QH≥VKU≥UVGR≥KP≥7CORNG≥≥YCU≥o, num/Q!≥6KU≥OCMGU≥CP≥KVGTCVKQP
KPFGRGPFGPV≥QH≥VC≥QVCT≥KVGTCVKQPU≥CPF≥CNNQYU≥6$VQ≥FKXKFG≥VCO≥WR≥KPVQ≥KPFGRGPFGPV≥VCUMU

Counting and marking elements for partitions

The LEG≥CPF≥CountAndMark≥ENCUUGU≥CTG≥UQYP≥KP≥7CORNG≥≥6C≥ LEG≥ENCUU≥WUGF≥KP≥RNCEG≥QH≥VC
VJGGGNGOGPV≥ leg≥CTTC[≥HTQO≥VC≥UGTKCN≥EQFG≥QNFU≥VC≥PWODGT≥QH≥GNGOGPVU≥VCV≥CTG≥NGUU≥VCP
GSWCN≥VQ≥CPF≥ITGCVGT≥VCP≥VC≥median≥XCNWG≥6C≥ CountAndMark≥ENCUU≥FGHKPGU≥VC≥OGVQFU≥VQ≥HKPF
VC≥PWODGT≥QH≥GNGOGPVU≥YKVKP≥VC≥VJGG≥RCTVKVKQPU≥CPF≥VQ≥FGVGTOKPG≥VC≥RCTVKVKQP≥QH≥GCEJ
GNGOGPV≥6C≥6$parallel_reduce≥CNIQTKVO≥KU≥WUGF≥QP≥VC≥CountAndMark≥ENCUU

EXAMPLE 6-14. LEG and CountAndMark classes

```
class LEG {

public:
  int less, equal, greater;

  LEG(): less(0), equal(0), greater(0) {}
};

class CountAndMark {
  int *S;
  int *Marks;
  int median;

public:
  LEG leg;

  void operator()(const blocked_range<size_t>& r) {
    for (size_t i = r.begin(); i != r.end(); ++i) {
      if (S[i] < median) {Marks[i] = 0; leg.less++;}
      else if (S[i] == median) {Marks[i] = 1; leg.equal++;}
      else {Marks[i] = 2; leg.greater++;}
    }
  }

  void join (const CountAndMark& y) {
    leg.less += y.leg.less;
    leg.equal += y.leg.equal;
    leg.greater += y.leg.greater;
  }
```

```
  CountAndMark(int *_S, int *_Marks, int _median):
    S(_S), Marks(_Marks), median(_median) {}

  CountAndMark(CountAndMark& x, split):
    S(x.S), Marks(x.Marks), median(x.median) {}

};
```

OQV≥QPG≥PQV≥VYQ≥DWV≥VJGG≥UGRCTCVG≥RCTCNNGN≥UWOU≥CTG≥EQORWVGF≥YKVJ3VG≥CountAndMark≥ENCUU
6G≥UUG≥UWOU≥CTG≥MGRV≥KP≥VG≥VJGG≥HKGNFU≥QH≥VG≥LEG≥QDLGEV≥;QW≥ECP≥UGG≥VGV≥VG≥join≥OGVQF≥KP
ZCORNG≥≥CFFU≥VG≥less≥equal≥CPF≥greater≥RCTVKCN≥UWOU≥VQIGVGT≥CHVGT≥VJGCFU≥GXG≥IQPG
VJQWI3CNN≥GNGOGPVU≥KP≥CUUKIPGF≥EQPMU

6G≥ operator()≥EQORWVGU≥VG≥RCTVKCN≥UWOU≥HQT≥GCE3QH≥VG≥VJGG≥RCTVKVKQP≥EQKEGU≥6G≥if-then-
else≥VGUVU≥FGVGTOKPG≥YKE3RCTVKCN≥UWO≥KU≥KPETGOGPVGF≥KP≥GCE3NQQR≥KVGTCVKQP≥+P≥CFFKVKQP≥VG
EQTTGURQPFKPI≥Marks≥CTTC[≥GNGOGPVU≥CTG≥CUUKIPGF≥C≥PQVCVKQP≥XCNWG≥VQ≥KPFKECVG≥KP≥YKE3RCTVKVKQP
VG≥GNGOGPV≥WPFGT≥UETWVKP[≥DGNQPIU≥6G≥Marks≥CTTC[oU≥CEVWCN≥RCTCOGVGT≥YKNN≥DG≥WUGF≥KP≥VG
ArrayPack()≥HWPEVKQP≥VQ≥KFGPVKH[≥YKE3GNGOGPVU≥HTQO≥VG≥QTKIKPCN≥CTTC[≥CTG≥KP≥VG≥FGUKTGF
RCTVKVKQP

The ArrayPack() function

From the point of view of the ParallelSelect()≥HWPEVKQP≥VG≥ArrayPack()≥HWPEVKQP≥FQGU≥PQV6KPI
FKHHGTGPV≥VGP≥KV≥FKF≥KP≥VG≥UGTKCN≥XGTUKQP≥6G≥EQPEWTTGPV≥XGTUKQP≥QH≥ArrayPack()≥KU≥UQYP≥KP
ZCORNG≥≥6G≥UWRRQTV≥ENCUUGU≥PackingScan≥CPF≥PackingMove≥CTG≥UQYP≥NCVGT≥KP≥ZCORNGU
≥CPF≥≥TGURGEVKXGN[

EXAMPLE 6-15. ArrayPack() function

```
void ArrayPack(int S[], int sPack[], int num, int Marks[], int scanSym)
{
  int *scanIdx = new int[num];
  PackingScan body(scanIdx, Marks, scanSym);

  parallel_scan(blocked_range<int>(0,num), body, auto_partitioner());

  if (scanIdx[0]) sPack[0] = S[0];  // move if first element is marked
  parallel_for(blocked_range<int>(1,num), PackingMove(scanIdx,sPack,S),
               auto_partitioner());
}
```

6G≥ Marks≥CPF≥scanSym≥RCTCOGVGTU≥KFGPVKH[≥VG≥GNGOGPVU≥VGV≥CTG≥VQ≥DG≥OQXGF≥HTQO≥VG≥CTTC[
VQ≥VG≥sPack≥CTTC[≥#≥RTGHKZ≥UECP≥TGEQIPK\GU≥VG≥GNGOGPVU≥VQ≥DG≥OQXGF≥CPF≥EQORWVGU≥VG≥KPFGZ
YGTG≥VG[≥CTG≥VQ≥DG≥TGNQECVGF≥YKVJ4P≥sPack≥+oXG≥WUGF≥VG≥6$parallel_scan≥CNIQTKVO≥VQ
CEEQORNKUJ3VKU≥6G≥CEVWCN≥OQXGOGPV≥KU≥FQPG≥EQPEWTTGPVN[≥VJQWI3C≥parallel_for≥CHVGT≥VG
HKTUV≥GNGOGPV≥KH≥EGEMGF≥CPF≥OQXGF≥KH≥PGGFGF≥5GG≥VG≥UKFGDCT≥p#TTC[≥2CEMKPI≥YKVJ2TGHKZ
5ECPq≥HQT≥KPHQTOCVKQP≥QP≥WUKPI≥RTGHKZ≥UECP≥VQ≥RCEM≥CTTC[U

ARRAY PACKING WITH PREFIX SCAN

Given an array of data elements and a corresponding "mark" array that indicates some elements as being of interest (assuming 1 indicates interest in the data element and 0 indicates no interest), the task is to gather all the marked elements into a contiguous range of elements within an array. A prefix scan of the binary array of marks is first computed. From the results of this computation, the set of originally marked elements can be identified by the change in value of a mark element and the immediate prior element (i.e., the value in the `[i]` position is one greater than the value in the `[i-1]` position). The value of the scanned mark array—for each marked element—directly determines the index within the packed array to which the marked element will be copied. Figure 6-7 shows an example of the computation.

FIGURE 6-7. Prefix scan to pack an array

Figure 6-7 (a) shows an array of 16 integers (labeled "Original data array") and an array of "Marks" identifying those elements whose value is less than five (1) and greater than or equal to five (0). Figure 6-7 (b) is the Marks array after an inclusive prefix scan has been run on the binary values. You can see from Figure 6-7 (c) that the elements from the original array that are less than five in value are copied into contiguous elements of the Packed array via the results of the prefix scan. If the first element of the Marks array is 1, the first element (3 in this example) was an originally marked item and needs to be moved into the Packed array. For every other element of the Marks array, if the value is different from the value in the element to the immediate left, the corresponding item from the Data array is moved into the Packed array (shown via the arrows in the figure). The index of the slot in the Packed array to receive a data item is the value of the Marks array (minus 1 for 0-based array indexing).

If you need to pack the marked elements and unmarked elements in-place, you can use the following steps after running the prefix scan of the Marks array:

1. Allocate a separate integer array that is equal in size to the number of data elements, initialized to 1 (I am assuming the size of the data elements is large enough that this allocation is relatively small). This array will hold the destinations for each data element. The first element from the Marks array that corresponds to the lowest indexed nonmarked data element is assigned a value that is one greater than the number of marked elements.
2. Copy the negative of the destination indexes for all marked elements from the scanned array into the destination array.
3. Run a prefix scan of only the positive elements (the lowest-indexed nonmarked element's value and all the initial 1 values) within the destination array. That is, treat the negative values as 0, but don't change them.
4. Use the absolute values of destination array elements to move each element from its current location to the index location in the array given in the corresponding element of the destination array. Be careful, though; this mass movement of data is easy to do concurrently using a PRAM with a number of processors equal to the number of items, but will be much trickier with a finite number of threads and cores.

If you can get away with a serial scan, you can just use the original Marks array, but you will need to identify the unmarked data elements and treat these as if the value 1 were stored there in order to compute destination index values.

Once you know the destination indexes, you can move data elements from the original array to the packed array concurrently. Copying data from one memory location to another doesn't rely on any other copy operation, unless there is a potential for conflict at the receiving locations. The indexes for receiving a data item are precomputed, and you can see that they are all going to be unique, thus, no data races.

The PackingScan≥ENCUU≥KP≥ZCORNG≥≥UGVU≥WR≥VQ≥6$ENCUU≥TGSWKTGOGPVU≥VQ≥EQORWVG≥VQ≥RTGHKZ
UECP≥QP≥VQ≥Marks≥CTTC[≥6Q≥PackingMove≥ENCUU≥UQYP≥NCVGT≥KP≥ZCORNG≥≥IKXGU≥VQ≥ENCUU
FGENCTCVKQPU≥VQ≥RCEM≥VQ≥FCVC≥KVGOU≥EQPEWTTGPVN[≥HTQO≥VQ≥UGNGEVGF≥RCTVKVKQP

EXAMPLE 6-16. PackingScan class

```
class PackingScan {
  int sum;
  int* const y;
  const int* const x;
  const int symbol;

public:

  template<typename Tag>
```

```
  void operator()(const blocked_range<int>& r, Tag) {
    int temp = sum;
    for (int i = r.begin(); i < r.end(); ++i) {
      if (x[i] == symbol) temp++;
      if(Tag::is_final_scan())
        y[i] = temp;
    }
    sum = temp;
  }

  PackingScan(int y_[], const int x_[], const int sym_) :
              sum(0), x(x_), y(y_), symbol(sym_) {}

  PackingScan(PackingScan& b, split) :
              sum(0), x(b.x), y(b.y), symbol(b.symbol) {}

  void reverse_join(PackingScan& a) {sum = a.sum + sum;}

  void assign(PackingScan& b) {sum = b.sum;}
};
```

```
6G≥PackingScan≥ENCUU≥KU≥DCUGF≥FKTGEVN[≥QP≥VG≥parallel_scan≥GZCORNG≥HQWPF≥KP≥VG≥+PVGN≥6$
GZCORNG≥EQFGU≥6G≥QPN[≥EGPIG≥DGUKFGU≥VG≥NQIKECN≥RCTCOGVGT≥NKUV≥KU≥VGV≥C≥VGUV≥HQT≥YGVGT
QT≥PQV≥VG≥U[ODQN≥KP≥VG≥x≥CTTC[≥Marks≥KP≥ZCORNG≥≥KU≥GSWCN≥VQ≥VG≥U[ODQN≥KFGPVKH[KPI≥VG
FGUKTGF≥RCTVKVKQP≥6KU≥VGUV≥KU≥KP≥VG≥operator()≥FGHKPKVKQP≥5KPEG≥VG≥x≥CTTC[≥EQPVCKPU≥VJGG
FKHHGTGPV≥XCNWGU≥YG≥PGGF≥VQ≥WUG≥VKU≥CU≥C≥NQIKECN≥DKPCT[≥CTTC[≥6GV≥KU≥KH≥VG≥XCNWGU≥KP≥VG≥
OCVEGU≥VG≥U[ODQN≥DGKPI≥GZCOKPGF≥VG≥XCNWG≥KU≥VQ≥DG≥VTGCVGF≥CU≥KH≥KV≥YGTG≥C≥≥QVGTYKUG≥KH
VG≥U[ODQN≥FQGUPoV≥OCVE3VG≥EQPVGPVU≥QH≥VG≥CTTC[≥DGKPI≥UECPPGF≥VGV≥GNGOGPV≥KU≥VTGCVGF≥CU
C≥≥9GP≥VGTG≥KU≥C≥OCVE3VG≥temp≥XCNWG≥KU≥KPETGOGPVGF≥CU≥KH≥CFFKPI≥≥VQ≥VG≥TWPPKPI≥UECP
VQVCN≥+H≥VGTG≥KU≥PQV≥C≥OCVE3VG≥temp≥XCNWG≥KU≥NGHV≥WPEGPIGF≥CU≥KH≥CFFKPI≥≥VQ≥VG≥TWPPKPI
VQVCN≥&WTKPI≥VG≥HKPCN≥UECP≥RGUG≥YKE3KU≥FGPQVGF≥D[≥VG≥XCNWG≥QH≥Tag≥VKU≥TWPPKPI≥UWO
XCNWG≥KU≥UVQTGF≥KP≥VG≥UECPoU≥QWVRWV≥CTTC[≥y
```

EXAMPLE 6-17. PackingMove class

```
class PackingMove {
  const int *scanIdx;
  int *sPack;
  const int *S;

public:
  void operator()(const blocked_range<int>& r) const {
    for (int i = r.begin(); i < r.end(); i++) {
      if (scanIdx[i-1] != scanIdx[i])
        sPack[scanIdx[i]-1] = S[i];
    }
  }
  PackingMove(const int sIdx_[], int sPack_[], const int *S_) :
              scanIdx(sIdx_), sPack(sPack_), S(S_) {}
  PackingMove(const int sIdx_[], int sPack_[], const int *S_, split) :
              scanIdx(sIdx_), sPack(sPack_), S(S_) {}
};
```

```
                    PackingMove≥ENCUU≥YKV3VG≥parallel_for≥CNIQTKVO≥ECNN≥HTQO≥YKVXKP≥VG
ArrayPack()≥HWPEVKQP≥ZCORNG≥≥;QW≥ECP≥WUG≥VG≥TGUWNVU≥QH≥VG≥RTGHKZ≥UECP≥VQ≥KFGPVKH[
YXE3GNGOGPVU≥CTG≥VQ≥DG≥OQXGF≥CPF≥YGTG≥VQ≥EQR[≥VG≥KVGOU≥KPVQ≥VG≥RCEMGF≥CTTC[≥+P≥VG
operator()≥EQFG≥KP≥ZCORNG≥≥VG≥XCNWG≥QH≥GCE3GNGOGPV≥KP≥VG≥UECPPGF≥CTTC[≥scanIdx≥KU
EQORCTGF≥VQ≥VG≥XCNWG≥QH≥VG≥RTGEGFKPI≥GNGOGPV≥+H≥VGUG≥VYQ≥GNGOGPVU≥CTG≥FKHHGTGPV≥VG≥XCNWG
QH≥VG≥GNGOGPV≥OKPWU≥≥KU≥WUGF≥VQ≥KPFGZ≥VG≥RCEMGF≥CTTC[≥Pack≥UNQV≥VGV≥YKNN≥TGEGKXG≥VG≥XCNWG
DGKPI≥EQRKGF≥HTQO≥S≥9G≥MPQY≥VGV≥GCE3FGUVKPCVKQP≥KU≥IKXGP≥VQ≥QPN[≥QPG≥GNGOGPV≥HTQO≥VG
S≥CTTC[≥UQ≥VGTG≥CTG≥PQ≥EQPHNKEVU≥QT≥FCVC≥TCEGU

1PEG≥VG≥GNGOGPVU≥HTQO≥VG≥CRRTQRTKCVG≥RCTVKVKQP≥CTG≥RCEMGF≥VG≥ArrayPack()≥HWPEVKQP≥TGVWTPU
CPF≥C≥TGEWTUKXG≥ECNN≥VQ≥parallelSelect()≥YKV3VKU≥PGY≥RCTVKVKQP≥CPF≥VG≥XCNWG≥QH≥CFLWUVGF≥CU
PGGFGF≥KU≥GZGEWVGF
```

Some Design Notes

Rather than breaking down the Selection code in terms of efficiency, simplicity, portability, and scalability, I just want to conclude with some general design comments. All that I've said so far about these topics in reference to the parallel sum and prefix scan algorithms will apply to the use of those algorithms within the examples.

The simple algorithms for parallel sum and prefix scan are merely the basis for the steps needed in the selection algorithm. Rather than a single sum, the `CountAndMark`≥ENCUU≥TCP≥VJGG≥UGRCTCVG

```
UWOU≥UKOWNVCPGQWUN[≥#NUQ≥VG≥UECP≥QRGTCVKQP≥PGGFGF≥VQ≥KPVGTRTGV≥VG≥FCVC≥XCNWGU≥CPF≥EQORWVG
YKV3NQIKECN≥DKPCT[≥GSWKXCNGPVU≥DCUGF≥WRQP≥VG≥XCNWG≥QH≥GCE3GNGOGPV≥6KU≥UQTV≥QH≥CNIQTKVO
VTCPUOQITKHKECVKQP≥KU≥PQVXKPI≥PGY≥2TQITCOOGTU≥TQWVKPGN[≥VCMG≥VG≥EQPEGRV≥QH≥CP≥CNIQTKVO
CPF≥TGUGRG≥KV≥VQ≥HKV≥VG≥EWTTGPV≥PGGFU≥QH≥VG≥CRRNKECVKQP≥VG[≥CTG≥YQTMKPI≥QP≥6G≥VTKEM≥QH
EQWTUG≥KU≥DGKPI≥CDNG≥VQ≥FGVGTOKPG≥YXE3DCUG≥CNIQTKVO≥KU≥VG≥DGUV≥VQ≥OQFKH[
```

> **NOTE**
>
> ***Transmogrification*** **is the act of changing into a different form or appearance. I've liked this word ever since seeing the misadventures caused by writing "Transmogrifier" on a cardboard box in the "Calvin and Hobbes" comic strip.**

A Final Thought

I don't mean to use PRAM algorithms as the basis for all future algorithms in this book. I'm trying to present you with information that is more practical than theoretical. The parallel sum and prefix scan algorithms are quite easy to implement on a PRAM, though. I wanted to give you a taste of what you might face if you pick up another parallel algorithms book that uses PRAM implementations of the algorithms. Assuming you're not a computer science graduate, this exposure will better equip you as far as how to interpret those algorithms and what steps you may need to go through to adapt them into practical code. While it should be possible to

CHAPTER SEVEN

MapReduce

MapReduce is an algorithmic framework, like divide-and-conquer or backtracking, rather than a specific algorithm. The pair of operations, *map* and *reduce*, is found in LISP and other functional languages. MapReduce has been getting a lot of buzz as an algorithmic framework that can be executed concurrently. Google has made its fortune on the application of MapReduce within a distributed network of thousands of servers (see ®MapReduce Simplified Data Processing on Large Clusters¯ in *Communications of the ACM* Z2008\ by Jeffrey Dean and Sanjay Ghemawat), which has only served to heighten awareness and exploration of this method.

The idea behind *map* is to take a collection of data items and associate a value with each item in the collection. That is, to match up the elements of the input data with some relevant value to produce a collection of key-value pairs. The number of results from a map operation should be equal to the number of input data items within the original collection. In terms of concurrency, the operation of pairing up keys and values should be completely independent for each element in the collection.

The *reduce* operation takes all the pairs resulting from the map operation and does a reduction computation on the collection. As I've said before, the purpose of a reduction is to take in a collection of data items and return a value derived from those items. Parallel sum (from Chapter 6) is an example of a reduction computation. In more general terms, we can allow the reduce operation to return with zero, one, or any number of results. This will all depend on what the reduction operation is computing and the input data from the map operation.

Before looking at the implementation and other details, let's look at an example of MapReduce in action. Consider the task of counting the number of vowels and consonants in the following sentence

The quick brown fox jumps over the lazy dog.

In my head, I run through the sentence, character by character, once to count the consonants and then again to count the vowels. The results would then be two integers vowels (12) and the number of consonants (23). Did you only count 11 vowels? Since ®lazy¯ has two syllables, which requires two vowel sounds, there must be at least two vowels. Thus, ®y¯ is doing duty as a vowel in this sentence. Example 7-1 has pseudocode of the MapReduce operation that would compute these values. For this example, S≥KU≥VQ≥UVTKPI≥QH≥EQ≥TCEVGTU CTTC[≥QNFKPI≥VQ≥UGPVGPEG

EXAMPLE 7-1. MapReduce pseudocode example

```
// Map
  for i = 1 to length(S) {
    if (S[i] is a consonant)
      generate_pair(key[i]=S[i], value[i]=1);
    else
      generate_pair(key[i]=S[i], value[i]=-1);
  }
```

```
// Reduce
  cCount = 0; vCount = 0;
  for i = 1 to length(S) {
    if (value[i] > 0) cCount += value[i];
    else              vCount += abs(value[i]);
  }
```

```
(QT≥VG≥IKXGP≥FCVC≥UGV≥S≥VG≥/CR4GFWEG≥UQNWVKQP≥HKTUV≥OCRU≥C≥XCNWG≥VQ≥GCE3NGVVGT≥KP≥QTFGT≥VQ
ETGCVG≥≥MG[XCNWG≥NGVVGTKPVGIGT≥RCKTU≥6G≥EQ0KEG≥QH≥XCNWGU≥FGRGPFU≥QP≥YGVVGT≥VG≥NGVVGT
KU≥C≥EQPUQPCPV≥1≥QT≥C≥XQYGN≥-1≥6G≥TGFWEVKQP≥QRGTCVKQP≥YKNN≥VCMG≥GCE3MG[XCNWG≥RCKT≥CPF
CFF≥VG≥XCNWG≥KPVQ≥QPG≥QH≥VYQ≥EQWPVGTU≥DCUGF≥QP≥VG≥V[RG≥QH≥NGVVGT≥EQPVCKPGF≥KP≥VG≥MG[
```

> **NOTE**
>
> **We will need some way to identify "y" as a vowel. This identification will need to influence the key-value pair created in the map phase so that the reduce computation can add the associated value to the correct counter.**

If we wrote code for this letter-counting operation in serial, we could have simply examined each letter in turn and incremented the proper counter. In the (serial) MapReduce variation (Example 7-1), we're required by the framework to use the values associated with the keys in the reduction computation. While it may seem to be more work, this difference creates the situation where the data is divorced from the algorithm and makes the reduction computations more independent.

Map As a Concurrent Operation

Look back at the pseudocode of the vowel/consonant counting algorithm in Example 7-1. Can you see that the creation of each key-value pair in the map phase would be independent of every other pair creation? Simply divide the letters among threads and create key-value pairs for each letter. One goal I would urge you to keep in mind when using MapReduce is to make the reduction computation as simple as possible. This is why the algorithm in Example 7-1 decided the category of each letter in the map phase.

An alternate mapping would assign a value of 1 to each key (character) and let the reduce phase decide whether the key is a vowel or consonant. The context of the keys that exists in the map phase may not be available when the reduction computation is executed. Or, if not unavailable, it may require extra data to preserve the context and correctly process values in the reduction. Determining whether to label ®y¯ as a consonant or a vowel requires the context of the word itself. If the map and reduce operations were in different functions and we used the alternate mapping of 1≥HQT≥CNN≥MG[U≥YG≥YQWNF≥PGGF≥VQ≥UGPF≥VG≥TGFWEG≥HWPEVKQP≥VG≥EQPVGZV
VG≥UGPVGPEG≥KP≥QTFGT≥VQ≥ENCUUKH[≥VG≥p[q≥RTQRGTN[

Reduce As a Concurrent Operation

As I recommended in the previous chapter, if you have the chance, use either OpenMP or Intel TBB to do reduction computations. All of the grunt work and coordination of threads and partial values goes on ®behind the scenes.¯ Why would you want to do any heavy lifting if you don't have to? Plus, making use of code that is already written and debugged will give you more of a warm fuzzy feeling about your concurrent application.

If you don't have the option of using the reduction algorithms built into TBB or OpenMP, you will need an explicit threads solution. Example 6-4 includes a handcoded reduction for summing up all the items within an integer array. You can use this as a model for implementing reduce when there are few threads.

For this chapter, though, I want to present an alternative that doesn't use the serial processing at the final step to combine the partial results from the previous independent computations. Let's again take the summing of all elements from an array as the specific problem to be solved. As with the code in Example 6-4, we'll use Pthreads as the explicit threading library for implementation.

Handcoded Reduction

To get started, each thread working on the reduction is assigned a nonoverlapping chunk of the overall data set. Any of the previously discussed methods that ensures all of the data are assigned will work. A static division of data is best. Next, each thread calculates the sum of all the items assigned. This will yield a number of partial sums equal to the number of threads. At this point, we can use the PRAM version of the parallel sum algorithm presented in Example 6-2 (one thread per data item to be summed). However, since we probably don't have lockstep execution of threads built into the hardware, we will need to coordinate the threads' executions on the data and with each other.

As in Example 6-4, we will store the partial sums in separate elements of a global array. Threads use the assigned ID for the index into the global array gSum≥9G≥YKNN≥EQORWVG≥VQ≥HKPCN≥UWO
EQPEWTTGPVN[≥WUKPI≥VQ≥INQDCN≥CTTC[≥FCVC≥6Q≥HKTUV≥GNGOGPV≥YKVH≥gSum≥VQ≥CTTC[≥KPFGZ≥≥YKNN
QNF≥VQ≥EQORWVGF≥VQVCN≥HQT≥VQ≥GNGOGPVU≥QH≥VQ≥QTKIKPCN≥CTTC[≥(KIWTG≥≥UQYU≥C
TGRTGUGPVCVKQP≥QH≥VQ≥EQPEWTTGPV≥EQORWVCVKQPU≥VQV≥YG≥YKNN≥PGGF≥VQ≥VCMG≥VQ≥RCTVKCN≥UWOU≥CPF
CFF≥VQO≥VQIGVQT≥HQT≥VQ≥HKPCN≥CPUYGT

FIGURE 7-1. Reduction computation

Each array displayed in Figure 7-1 represents the contents of the gSum≥CTTC[≥CHVGT≥C≥TQWPF≥QH
EQORWVCVKQP≥6G≥CTTQYU≥KPFKECVG≥YGTG≥FCVC≥KU≥TGCF≥HTQO≥CPF≥YGTG≥VG≥TGUWNVU≥CTG≥UVQTGF≥KP
GCE≥TQWPF≥6G≥RNWU≥UKIP≥≥YKVKP≥VG≥EKTENG≥KU≥VG≥EQODKPKPI≥QRGTCVKQP≥CFFKVKQP≥HQT≥VKU
GZCORNG≥WUGF≥KP≥VG≥TGFWEVKQP≥6G≥ITC[GF≥PWODGTU≥UKORN[≥KPFKECVG≥VGV≥VG≥FCVC≥YKVKP≥VGV
GNGOGPV≥QH≥VG≥CTTC[≥KU≥PQ≥NQPIGT≥PGGFGF≥KP≥HWTVGT≥TQWPFU≥QH≥EQORWVCVKQP≥(QT≥DTGXKV[≥+oXG
NKOKVGF≥VG≥GZCORNG≥VQ≥GKIV≥VTGCFU≥+≥VKPM≥VGV≥QPEG≥[QWoXG≥UGGP≥QY≥VYQ≥QT≥VTGG≥TQWPFU
CTG≥EQORWVGF≥[QWoNN≥GCUKN[≥DG≥CDNG≥VQ≥GZVTCRQNCVG≥VKU≥CNIQTKVO≥VQ≥CP[≥PWODGT≥QH≥VTGCFU

+P≥VG≥HKTUV≥TQWPF≥#FF≥EQPUGEWVKXG≥≥GNGOGPV≥RCKTU≥GCE≥VTGCF≥YQUG≥+&≥KU≥C≥OWNVKRNG≥QH≥
≥TGCFU≥VG≥XCNWG≥UVQTGF≥KP≥VG≥GNGOGPV≥KPFGZGF≥D[≥=VTGCF≥+&?≥≥≥ tKH≥UWE≥CP≥GNGOGPV
KU≥RCTV≥QH≥VG≥CTTC[tCPF≥CFFU≥VKU≥XCNWG≥VQ≥VG≥XCNWG≥UVQTGF≥KP≥VG≥GNGOGPV≥KPFGZGF≥YKV≥VG
VTGCFoU≥+&≥+P≥VG≥UGEQPF≥TQWPF≥#FF≥EQPUGEWVKXG≥ ≥GNGOGPV≥RCKTU≥GCE≥VTGCF≥YQUG≥+&≥KU
C≥OWNVKRNG≥QH≥≥ ≥TGCFU≥VG≥XCNWG≥UVQTGF≥KP≥VG≥GNGOGPV≥KPFGZGF≥D[≥=VTGCF≥+&?≥≥≥ tKH
UWE≥CP≥GNGOGPV≥KU≥RCTV≥QH≥VG≥CTTC[tCPF≥CFFU≥VKU≥XCNWG≥VQ≥VG≥XCNWG≥UVQTGF≥KP≥VG≥GNGOGPV
KPFGZGF≥YKV≥VG≥VTGCFoU≥+&≥+P≥VG≥VKTF≥TQWPF≥#FF≥EQPUGEWVKXG≥ ≥GNGOGPV≥RCKTU≥GCEJ
VTGCF≥YQUG≥+&≥KU≥C≥OWNVKRNG≥QH≥≥ ≥TGCFU≥VG≥XCNWG≥UVQTGF≥KP≥VG≥GNGOGPV≥KPFGZGF≥D[≥=VTGCF
+&?≥≥≥ tKH≥UWE≥CP≥GNGOGPV≥KU≥RCTV≥QH≥VG≥CTTC[tCPF≥CFFU≥VKU≥XCNWG≥VQ≥VG≥XCNWG≥UVQTGF
KP≥VG≥GNGOGPV≥KPFGZGF≥YKV≥VG≥VTGCFoU≥+&≥ *WPF≥UQ≥XGKVG≥*QT≥pCPF≥UQ≥QPq≥HQT≥O[≥PINKUJ
TGCFGTU≥6G≥RCVVGTP≥HQT≥GCE≥UWEEGUUKXG≥TQWPF≥UKORN[≥TGRGCVU≥YKV≥VG≥KPFGZGU≥KPXQNXGF
ITQYKPI≥D[≥C≥HCEVQT≥QH≥VYQ

6G≥EQFG≥VQ≥KORNGOGPV≥C≥TGFWEVKQP≥UWOOCVKQP≥QP≥CP≥CTTC[≥QH≥KPVGIGTU≥KU≥IKXGP≥KP
ZCORNG≥≥6KU≥EQFG≥KU≥OQFKHKGF≥HTQO≥VG≥EQFG≥IKXGP≥KP≥ZCORNG≥≥YKV≥VG≥FKHHGTGPEGU
KIN KIVGF≥KP≥DQNF≥VGZV

EXAMPLE 7-2. Reduction code to sum elements of an array

```
#include <pthread.h>
#include <stdio.h>
#include "pth_barrier.h"

#define NUM_THREADS 128

int N;  // number of elements in array A
int *A;
int gSum[NUM_THREADS];  // global storage for partial results
pth_barrier_t B;

void *SumByReduction (void *pArg)
{
  int tNum = *((int *) pArg);
  int lSum = 0;
  int start, end, i;

  start = ((float)N/NUM_THREADS) * tNum;
  end =   ((float)N/NUM_THREADS) *(tNum+1);
  if (tNum == (NUM_THREADS-1)) end = N;
  for (i = start; i < end; i++)
    lSum += A[i];
  gSum[tNum] = lSum;

  pth_barrier(&B);
  int p2 = 2;
  for (i = 1; i <= NUM_THREADS; i *= 2) {
    if ((tNum % p2) == 0)
      if (tNum+i < NUM_THREADS) gSum[tNum] += gSum[tNum+i];
    p2 *= 2;
    pth_barrier(&B);
  }
  free(pArg);
}

int main(int argc, char* argv[])
{
  int j, sum = 0;
  pthread_t tHandles[NUM_THREADS];

  InitializeArray(A,&N);  // get values into A array; not shown
  pth_barrier_init(&B, NUM_THREADS);
  for (j = 0; j < NUM_THREADS; j++) {
    int *threadNum = new(int);
    *threadNum = j;
    pthread_create(&tHandles[j], NULL, SumByReduction, (void *)threadNum);
  }
// just wait for thread  with id=0 to terminate; others will follow
  pthread_join(&tHandles[0], NULL);
  printf("The sum of array elements is %d\n", gSum[0]);
  return 0;
}
```

pth_barrier.h≥CPF≥C≥DCTTKGT≥QDLGEV≥INQDCNN[≥FGENCTGF≥D[≥pth_barrier_DCTTKGT≥KU≥C
U[PEJQPK\CVKQP≥VQV≥YKNN≥RCWUG≥VJGCFU≥CV≥VQ≥DCTTKGT≥RQKPV≥KP≥VQ≥EQFG≥WPVKN≥CNN≥VJGCFU≥YQTMKPI
KP≥VQ≥EQORWVCVKQP≥QXG≥TGCEQF≥VQV≥UCOG≥RQKPV≥1PEG≥CNN≥VJGCFU≥QXG≥CTTKXGF≥VQ≥DCTTKGT
TGNGCUGU≥VQ≥VJGCFU≥VQ≥DGIKP≥GZGEWVKQP≥QH≥VQ≥UWEEGGFKPI≥EQFG≥6KU≥KU≥NKMG≥C≥UVCTVKPI≥NKPG≥CV≥C
TCEG≥VQV≥FQGUPoV≥CNNQY≥VQ≥EQORGVKVQTU≥VQ≥UVCTV≥WPVKN≥CNN≥TCEGTU≥QXG≥TGCEQF≥VQ≥UVCTV≥NKPG≥QPEG
CNN≥VQ≥TCEGTU≥CTG≥TGCF[≥VQ[≥CTG≥CNNQYGF≥VQ≥DGIKP≥VQ≥GXGPV≥5GG≥p#≥$TTKGT≥1DLGEV
+ORNGOGPVCVKQPq≥QP≥RCIG≥≥HQT≥FGVCKNU≥CDQWV≥KORNGOGPVKPI≥UWE3C≥DCTTKGT≥QDLGEV≥YKV3VQ
215+:≥VJGCFU≥NKDTCT[

+H≥[QW≥NQQM≥CV≥VQ≥main()≥HWPEVKQP≥[QW≥YKNN≥UGG≥VQ≥KPKVKCNK\CVKQP≥QH≥VQ≥CTTC[≥VQ≥KPKVKCNK\CVKQP
QH≥VQ≥DCTTKGT≥QDLGEV≥CPF≥VQ≥ETGCVKQP≥QH≥VJGCFU≥VQ≥GZGEWVC≥VQ≥SumByReduction()≥HWPEVKQP≥6Q
OCKP≥VJGCF≥VQP≥YCKVU≥HQT≥QPN[≥VQ≥HKTUV≥VJGCF≥ETGCVGF≥6KU≥VJGCF≥YKNN≥QXG≥CP≥+&
threadNum≥QH≥≥CPF≥VQ≥ pthread_t≥XCNWG≥TGVWTPGF≥HTQO≥pthread_create()≥KU≥UVQTGF≥KP≥VQ≥KPFGZ
≥GNGOGPV≥QH≥VQ≥tHandles≥CTTC[≥5KPEG≥VQ≥HKPCN≥TGUWNV≥YKNN≥DG≥KP≥VQ≥gSum[0]≥NQECVKQP≥CPF≥VKU
pq≥VJGCF≥EQORWVGU≥VQV≥HKPCN≥XCNWG≥CPF≥UVQTGU≥KV≥KP≥VQ≥NQECVKQP≥QPEG≥VQ≥≥VJGCF≥QU
VGTOKPCVGF≥VQ≥HKPCN≥UWO≥QU≥DGGP≥UVQTGF≥CPF≥KU≥TGCF[≥VQ≥DG≥WUGF

6Q≥HKTUV≥QNH≥QH≥VQ≥SumByReduction()≥HWPEVKQP≥EQFG≥KU≥VCMGP≥XGTDCVKO≥HTQO≥VQ≥EQTTGURQPFKPI
HWPEVKQP≥HQWPF≥KP≥ZCORNG≥≥6Q≥UGEQPF≥QNH≥KP≥VQ≥DQNF≥VGZV≥KU≥VQ≥TGFWEVKQP≥QH≥VQ≥RCTVKCN
UWOU≥IGPGTCVGF≥D[≥VQ≥VJGCFU≥KPVQ≥C≥UKPING≥UWOOCVKQP≥XCNWG≥$HQTG≥TGFWEVKQP≥QH≥VQ≥RCTVKCN
UWOU≥ECP≥DGIKP≥CNN≥QH≥VQ≥RCTVKCN≥UWOU≥OWUV≥DG≥EQORWVGF≥XGP≥VQWI3YG≥EQWNF≥CUUWOG≥VQV
CP≥GSWCN≥FKUVTKDWVKQP≥QH≥EQPMU≥QH≥FCVC≥CTTC[≥YKNN≥DG≥CUUKIPGF≥VQ≥GCE3VJGCF≥YG≥ECPPQV≥CUUWOG
5KORNG≥4WNG≥≥VQV≥CNN≥VJGCFU≥YKNN≥HKPKU3CV≥VQ≥GZCEV≥UCOG≥VKOG≥VQ≥UVQTG≥VQ≥RCTVKCN≥UWO≥TGUWNV
KP≥VQ≥RTQRGT≥gSum≥UNQV≥9KNG≥VQTG≥EQWNF≥DG≥OQTG≥VQP≥GPQWI3TGUQWTEGU≥VQ≥CUUKIP≥QPG≥VJGCF
VQ≥C≥EQTG≥GZENWUKXGN[≥VQTG≥CTG≥O[TKCFU≥QH≥QVQT≥HCEVQTU≥YKVKP≥VQ≥QRGTCVKPI≥U[UVGO≥QT≥VQ
RTQEGUUQT≥QTFYCTG≥VQV≥ECP≥UNQY≥FQYP≥QT≥KPKDKV≥VQ≥EQORWVCVKQP≥QH≥QPG≥QT≥OQTG≥VJGCFU
6KU≥YG≥PGGF≥VQ≥RNCEG≥C≥DCTTKGT≥CHVGT≥VQ≥CUUKIPOGPV≥QH≥GCE3VJGCFoU≥RCTVKCN≥UWO≥KPVQ≥VQ
gSum≥CTTC[≥#HVGT≥VQ≥NCUV≥VJGCF≥QU≥TGCEQF≥VQ≥DCTTKGT≥YG≥MPQY≥VQV≥CNN≥VQ≥RCTVKCN≥UWOU≥QXG
DGGP≥UVQTGF≥CPF≥VQ≥EQORWVCVKQP≥ECP≥UCHGN[≥RTQEGGF≥VQ≥VQ≥TGFWEVKQP≥6Q≥YQNG≥MG[≥VQ≥VQ
TGFWEVKQP≥CNIQTKVO≥YQTMKPI≥EQTTGEVN[≥CPF≥DGKPI≥CDNG≥VQ≥RTQXG≥VQV≥VKU≥KU≥C≥EQTTGEV≥CNIQTKVO
KU≥VQ≥DCTTKGT≥U[PEJQPK\CVKQP

6Q≥TGFWEVKQP≥EQORWVCVKQP≥OKOKEU≥VQ≥EQODKPKPI≥QH≥FCVC≥KP≥(KIWTG≥≥6Q≥for≥NQQR≥EQWPVU
QHH≥VQ≥TQWPFU≥YKV3VQ≥ i≥XCTKCDNG≥UGTXKPI≥CU≥VQ≥QHHUGV≥KPVQ≥gSum≥CTTC[≥HTQO≥YKE3C≥VJGCF
YKNN≥TGCF≥FCVC≥FWTKPI≥VQ≥EWTTGPV≥TQWPF≥6Q≥NQQR≥XCTKCDNG≥KU≥OWNVKRNKGF≥D[≥≥KP≥GCE3KVGTCVKQP

6Q≥ p2≥XCTKCDNG≥YKNN≥DG≥RQYGTU≥QH≥≥VQV≥CTG≥WUGF≥VQ≥FGVGTOKPG≥YKE3VJGCFU≥PGGF≥VQ≥TGCF≥FCVC
HTQO≥VQ≥gSum≥CTTC[≥CPF≥CFF≥VQV≥XCNWG≥VQ≥VQ≥XCNWG≥HQWPF≥KP≥VQV≥VJGCFoU≥CUUKIPGF≥gSum≥GNGOGPV
6Q≥QWVGT≥if-then≥UVCVGOGPV≥KP≥VQ≥DQF[≥QH≥VQ≥NQQR≥OCMGU≥VKU≥FGVGTOKPCVKQP≥D[≥FKXKFKPI≥C
VJGCFoU≥+&≥PWODGT≥D[≥p2≥CPF≥CNNQYKPI≥VQUG≥VJGCFU≥VQV≥CTG≥GXGPN[≥FKXKUKDNG≥D[≥VQ≥EWTTGPV

p2≥VQ≥RTQEGGF≥VQ≥VG≥KPPGT≥if-then≥UVCVGOGPV≥6G≥KPPGT≥EQPFKVKQPCN≥GZRTGUUKQP≥tNum+i <
NUM_THREADS≥YKNN≥GPUWTG≥VGV≥VG≥RTQRQUGF≥GNGOGPV≥QH≥VG≥gSum≥CTTC[≥VQ≥DG≥TGCF≥CEVWCNN[≥GZKUVU
YKVKP≥VG≥CTTC[≥DQWPFU≥CPF≥KH≥UQ≥KPETGOGPVU≥VG≥XCNWG≥QH≥VG≥VJGCFogSum≥UNQV≥YKVJVG≥XCNWG
CEEGUUGF

4GICTFNGUU≥QH≥YGVGT≥QT≥PQV≥C≥VJGCF≥YCU≥CNNQYGF≥VQ≥RCTVKEKRCVG≥KP≥VG≥CFFKVKQP≥QRGTCVKQP
GXGT[≥VJGCF≥OWNVKRNKGU≥VG≥NQECN≥EQR[≥QH≥p2≥D[≥≥CPF≥VGP≥YCKVU≥CV≥C≥DCTTKGT≥WPVKN≥CNN≥VJGCFU
GXG≥EQORNGVGF≥YGVGXGT≥EQORWVCVKQP≥YCU≥CNNQYGF≥YKVKP≥VG≥EWTTGPV≥TQWPF≥QH≥VG≥TGFWEVKQP
#NN≥VJGCFU≥YKNN≥GZGEWVG≥UQOGVKPI≥YKVKP≥GCEJ≥TQWPF≥QH≥VG≥TGFWEVKQP≥GXGP≥VQWIJ≥CNH≥CU
OCP[≥CTG≥FQKPI≥WUGHWN≥YQTM≥KP≥C≥IKXGP≥TQWPF≥VGP≥KP≥VG≥RTGXKQWU≥TQWPF≥6KU≥OC[≥UGGO≥NKMG
C≥YCUVG≥QH≥TGUQWTEGU≥*QYGXGT≥UKPEG≥CNN≥VJGCFU≥OWUV≥OGGV≥WR≥CV≥VG≥DCTTKGT≥YG≥MGGR≥GCEJ
VJGCF≥TWPPKPI≥CPF≥FQKPI≥C≥OKPKOCN≥COQWPV≥QH≥YQTM≥FQWDNKPI≥VG≥XCNWG≥QH≥p2≥KP≥GCEJ≥TQWPF
VQ≥MGGR≥KP≥U[PE≥YKVJVQUG≥VJGCFU≥VGV≥CTG≥UVKNN≥FQKPI≥EQPUVTWEVKXG≥YQTM

0 1 6 '

9JKNG +oXG PGXGT YTKVVGP QPG QT UGGP CP KORNGOGPVCVKQP QH YQTM YKVJ C FKHHGTGPV PWODGT QH VJTGCFU GCEJ VKOG KV YCU CPF FCWPVKPI VCUM +oO CHTCKF VJCV VJG GZGEWVKQP QH UWEJ C QH QXGTJGCFtEGTVCKPN[OWEJ OQTG VJCP MGGRKPI UQOG VJTGCFU RCUV VJG VKOG VJG[CTG FQKPI CP[VJKPI RTCEVKECN

1PEG [QWoXG JCF C EJCPEG VQ FKIGUV VJG EQFG KP 'ZCORNG GZGEWVKQP QH VJG EQFG WUKPI VJG GZCORNG IKXGP KP (KIWT VJKU YKNN YQTM YKVJ C PWODGT QH VJTGCFU VJCV KU PQV C R DG RTGFKECVGF QP RQYGTU QH DWV VJGTG OC[EQOG C VKOO VJTGCFU HQT C TGFWEVKQP QRGTCVKQP 4GUV CUUWTGF VJG E VJCV KU PQV C RQYGT QH

6Q RTQXG VJKU VQ [QWTUGNH VT[VTCEKPI VJG CNIQTKVJO YKV
6G≥ gSum≥CTTC[≥YKNN≥DG≥KPFGZGF≥HTQO≥=?≥VQ≥=?≥XKUWCNK\G≥CPQVGT≥GNGOGPV≥CVVCEGF≥VQ≥VG≥TKIV
QH≥VG≥gSum≥CTTC[≥UQYP≥KP≥(KIWTG≥≥6GTG≥YKNN≥DG≥HQWT≥TQWPFU≥VQ≥VG≥TGFWEVKQP≥CNIQTKVO
YGTG≥ i≥YKNN≥DG≥CUUKIPGF≥XCNWGU≥≥≥≥CPF≥≥6G≥HKTUV≥GKIV≥GNGOGPVU≥QH≥VG≥gSum≥CTTC[≥YKNN≥DG
RTQEGUUGF≥CU≥UQYP≥KP≥(KIWTG≥≥FWTKPI≥VG≥HKTUV≥VJGG≥TQWPFU≥QH≥VG≥CNIQTKVO≥6G≥PKPVJ
GNGOGPV≥KPFGZ≥=?≥YKNN≥DG≥WPECIPIGF≥UKPEG≥CNN≥RQVGPVKCN≥GNGOGPVU≥QH≥VGV≥YQWNF≥DG≥TGCF
CPF≥WUGF≥VQ≥CFF≥KPVQ≥VG≥EQPVGPVU≥QH≥VGV≥UNQV≥CTG≥PQV≥YKVKP≥VG≥DQWPFU≥QH≥VG≥CTTC[≥#V≥VG
HQWTV≥TQWPF≥VG≥≥VJGCF≥YKNN≥TGCF≥VG≥EQPVGPVU≥QH≥gSum[8]≥≥≥i≥≥≥CPF≥CFF≥VGV≥XCNWG≥VQ
VG≥EQPVGPVU≥QH≥gSum[o]≥;QW≥ECP≥TGRTQFWEG≥VKU≥KFGC≥VQ≥CP[≥PWODGT≥QH≥VJGCFU≥DGVYGGP≥≥CPF
≥QT≥CP[≥QVGT≥PQPRQYGT≥QH≥≥YGTG≥VG≥WRRGT≥UNQVU≥QH≥ gSum≥YKNN≥DG≥UWOOGF≥XKC≥TGFWEVKQP
KPVQ≥VG≥KPFGZ≥=?≥GNGOGPV≥FWTKPI≥VG≥HKTUV≥VJGG≥TQWPFU≥CPF≥VG≥HQWTV≥YKNN≥DTKPI≥VG≥HKPCN
VQVCN≥KPVQ≥gSum[o]

5RGCMKPI≥QH≥QFF≥PWODGTU≥QH≥VJGCFU≥FKF≥[QW≥PQVKEG≥VG≥WUG≥QH≥VG≥(float)≥ECUV≥KP≥VG
EQORWVCVKQP≥QH≥start≥CPF≥end≥KP≥ZCORNG≥!≥4CVGT≥VGP≥QRKPI≥HQT≥C≥PWODGT≥QH≥KVGTCVKQPU

```
end≥XCNWG≥GCE≥VTJGCF≥YQWNF
DG≥CUUKIPGF≥≥KVGTCVKQPU≥GZEGRV≥VG≥NCUV≥QPG≥YKE≥YQWNF≥IGV≥≥KH≥[QW≥FQPoV≥DXG≥C
ECNEWNCVQT≥DPF[≥LWUV≥VTWUV≥OG≥QP≥VG≥CTKVOGVKE≥9GP≥WUKPI≥HNQCVKPIRQKPV≥FKXKUKQP≥VDV
VTWPECVGU≥HTCEVKQPCN≥RCTVU≥YGP≥TGECUV≥DCEM≥VQ≥(int)≥≥QH≥VG≥VTJGCFU≥YKNN≥DG≥CUUKIPGF≥
KVGTCVKQPU≥CPF≥VG≥QVGT≥≥IGV≥≥+H≥VG≥VKOG≥VQ≥EQORWVG≥QPG≥KVGTCVKQP≥KU≥UQTV≥≥GZVTC
KVGTCVKQPU≥CUUKIPGF≥VQ≥VG≥NCUV≥VTJGCF≥OKIV≥PQV≥DXG≥OWE≥KORCEV≥+H≥CP≥KVGTCVKQP≥VCMGU≥
UGEQPFU≥VQ≥EQORWVG≥YCKVKPI≥UKZ≥CPF≥C≥DNH≥OKPWVGU≥HQT≥QPG≥VTJGCF≥VQ≥HKPKU≥KU≥C≥UGTKQWU≥NQCF
DCNCPEG≥KUUWG≥4GICTFNGUU≥QH≥VG≥VKOG≥RGT≥KVGTCVKQP≥HQT≥UVCVKE≥UEDFWNKPI≥QH≥NQQR≥KVGTCVKQPU≥WUKPI
(float)≥YGP≥EQORWVKPI≥VG≥start≥CPF≥end≥DQWPFU≥YKNN≥CNYC[U≥IGPGTCVG≥C≥DGVVGT≥NQCF≥DCNCPEG
DGVYGGP≥VTJGCFU
```

A Barrier Object Implementation

A barrier object can synchronize thread execution at a specific point within the code. Threads are blocked at a barrier until all threads have reached the barrier point, and then all threads are released. With this description, we can develop the code to implement a barrier object to be used in Pthreads codes.

A Pthreads condition variable will hold threads until they can be released. So, we need a condition variable and the associated mutex object. In addition, the barrier must know the total number of threads that are participating in the barrier and how many threads have arrived at the barrier. When the final thread has come to the barrier, use that thread to release all the other waiting threads and to reset the counters of the barrier for the next use.

In my initial implementations of the barrier object, since I was using the count of threads that have arrived as the condition to keep threads waiting, I had thought to have the last thread exiting the barrier do the reset of all the counters. By keeping the count of threads at the barrier equal to 0 until the last thread was ready to exit, I didn't take into account the chance of another thread reentering the barrier, acquiring the mutex, checking the conditional expression, and passing over the pthread_cond_wait()≥9GP≥C≥VTJGCF≥GZVGTPCN≥VQ≥VG≥while≥NQQR≥WUWTRU≥VG
OWVGZ≥CPF≥KUPoV≥HQTEGF≥VQ≥YCKV≥CU≥KV≥UQWNF≥DG≥KV≥KU≥MPQYP≥KPUVGCFo≥KPVGTEGRVGF≥YCKV≥QH≥VKU≥ECUG
VG≥KPVGTEGRVGF≥YCKV≥QH≥CP≥GZVGTPCN≥VTJGCF≥GPVGTKPI≥VG≥DCTTKGT≥DGHQTG≥CNN≥VTJGCFU≥RTGXKQWUN[
YCKVKPI≥QP≥VG≥DCTTKGT≥DF≥NGHV≥ECP≥NGCF≥VQ≥C≥FGCFNQEM

6WU≥+≥PGGF≥C≥EQPFKVKQPCN≥GZRTGUUKQP≥HQT≥VG≥EQPFKVKQP≥XCTKCDNG≥VDV≥FQGUPoV≥TGN[≥QP≥VG≥EQWPV
QH≥VTJGCFU≥CV≥VG≥DCTTKGT≥+oXG≥EQUGP≥VQ≥pEQNQTq≥GCE≥WUG≥QH≥C≥DCTTKGT≥KP≥C≥E[ENKE≥HCUKQP≥9GP
VTJGCFU≥GPVGT≥C≥DCTTKGT≥VG[≥OWUV≥YCKV≥HQT≥CNN≥VTJGCFU≥VQ≥GPVGT≥VG≥DCTTKGT≥YKNG≥VG≥DCTTKGT≥KU
VG≥UCOG≥EQNQT≥9GP≥VG≥HKPCN≥VTJGCF≥UQYU≥WR≥VG≥EQNQT≥QH≥VG≥HWVWTG≥DCTTKGT≥EDPIGU≥CPF
VG≥EQWPVGT≥TGUGVU≥HQT≥VG≥PGZV≥WUG≥6TJGCFU≥VDV≥CTG≥UKIPCNGF≥VQ≥YCMG≥WR≥EDEM≥VG≥DCTTKGT
EQNQT≥+H≥KV≥KU≥PQV≥VG≥UCOG≥EQNQT≥VDV≥VG[≥HQWPF≥YGP≥VG[≥GPVGTGF≥VG[≥MPQY≥VDV≥VG≥HKPCN
```
```

pth_barrier_t≥V[RG

KU≥IKXGP≥GTG

```
    typedef struct {
      pthread_mutex_t m;
      pthread_cond_t c;
      int count, color, numThreads;
    } pth_barrier_t:
```

(QNNQYKPI≥VG≥EQPXGPVKQP≥QH≥QVGT≥2VJGCFU≥U[PEJQPK\CVKQP≥QDLGEVU≥+≥GXG≥YTKVVGP≥CP
KPKVKCNK\KPI≥HWPEVKQP≥VQ≥UGV≥WR≥C≥DCTTKGT≥6RU≥HWPEVKQP≥UKORN[≥ECNNU≥VG≥KPKVKCNK\CVKQP≥HWPEVKQPU
HQT≥VG≥EQPFKVKQP≥XCTKCDNG≥CPF≥OWVGZ≥6G≥VQVCN≥PWODGT≥QH≥VJGCFU≥VGV≥YKNN≥CNYC[U≥RCTVKEKRCVG
KP≥GCE3WUG≥QH≥VG≥DCTTKGT≥KU≥UGPV≥CU≥C≥RCTCOGVGT≥VQ≥VG≥KPKVKCNK\CVKQP≥HWPEVKQP≥6RU≥XCNWG≥UGVU
VG≥VYQ≥KPVGIGT≥EQWPVGTU≥YKVRP≥VG≥QDLGEV≥6G≥count≥KU≥FGETGOGPVGF≥CU≥VJGCFU≥EQOG≥KPVQ≥VG
DCTTKGT≥CPF≥YKNN≥TGCE3≥YGP≥VG≥HKPCN≥VJGCF≥GU≥CTTKXGF≥6G≥color≥YKNN≥CEVWCNN[≥VQIING
DGVYGGP≥pq≥CPF≥pPQV≥q≥DWV≥+oXG≥CFFGF≥C≥FGHKPKVKQPRQH≥VQ≥DG≥WUGF≥KP≥VG≥KPKVKCNK\CVKQP≥HQT
UQOG≥GZVTC≥HNCKT≥6G≥KPKVKCNK\CVKQP≥EQFG≥KU≥IKXGP≥GTG

```
    #define RED 0

    pth_barrier_init (pth_barrier_t *b, int numT)
    {
      pthread_mutex_init(&b->m, NULL);
      pthread cond_init(&b->c, NULL);
      b->count = b->numThreads = numT;
      b->color = RED;
    }
```

7RQP≥GPVGTKPI≥VG≥pth_barrier()≥HWPEVKQP≥C≥VJGCF≥HKTUV≥ICKPU≥EQPVTQN≥QH≥VG≥QDLGEVoU≥OWVGZ≥CPF
PQVGU≥VG≥EWTTGPV≥EQNQT≥QH≥VG≥DCTTKGT≥6G≥EQNQT≥KU≥GNF≥KP≥C≥XCTK6DNG≥NQECN≥VQ≥GCE3VJGCF
GPVGTKPI≥VG≥DCTTKGT≥HWPEVKQP≥UKPEG≥kolor≥KU≥FGENCTGF≥KP≥C≥HWPEVKQP≥ECNNGF≥D[≥C≥VJGCF≥6G
VJGCF≥VGP≥FGVGTOKPGU≥YGVGT≥KV≥KU≥VG≥NCUV≥VQ≥CTTKXG≥+H≥PQV≥KV≥DNQEMU≥KVUGNH≥QP≥VG≥EQPFKVKQP
XCTKCDNG≥CPF≥TGNGCUGU≥VG≥OWVGZ≥+H≥VG≥VJGCF≥KU≥VG≥NCUV≥VQ≥CTTKXG≥YGE3KV≥YKNN≥MPQY≥HTQO
VG≥DCTTKGToU≥count≥DGKPI≥FGETGOGPVGF≥VQ≥≥KV≥YKNN≥TGUGV≥VG≥color≥QH≥VG≥DCTTKGT≥UGV≥VG≥count
CPF≥YCMG≥WR≥CNN≥VJGCFU≥VGV≥GXG≥DGGP≥YCKVKPI≥6G≥EQFG≥HQT≥VG≥pth_barrier()≥HWPEVKQP≥KU
IKXGP≥GTG

```
    void pth_barrier (pth_barrier_t *b)
    {
      pthread_mutex_lock(&b->m);
      int kolor = b->color;
      if (--(b->count)) {
        while (kolor == b->color) pthread_cond_wait(&b->c, &b->m);
      }
      else {  // last thread
        pthread_cond_broadcast(&b->c);
        b->count = b->numThreads;
        b->color = !b->color;
      }
      pthread_mutex_unlock(&b->m);
    }
```

color≥QH≥VQ≥DCTTKGT≥VQ≥BLUE≥UC[≥UGVU≥VQ
count≥CPF≥DTQCFECUVU≥VQ≥YCMGWR≥UKIPCN≥DGHQTG≥TGNGCUKPI≥VQ≥OWVGZ≥+H≥6≥TCEGU≥VJQWI≥VQ
EQFG≥HQNNQYKPI≥VQ≥DCTTKGT≥CPF≥GPEQWPVGTU≥VQ≥DCTTKGT≥CICKP≥RGTURU≥KP≥C≥NQQR≥KV≥YKNN≥UGG≥VQV
KV≥KU≥PQV≥VQ≥NCUV≥VQ≥CTTKXG≥CV≥C≥BLUE≥DCTTKGT≥6≥FGETGOGPVU≥VQ≥DCTTKGT≥count≥CPF≥YKNN≥ECNN
pthread_cond_wait()≥(TQO≥VQ≥HCKTPGUU≥RTQRGTV[≥QH≥VQ≥KPVGTNGCXKPI≥CDUVTCEVKQP≥YG≥MPQY≥VQV
6≥YKNN≥GXGPVWCNN[≥CESWKTG≥VQ≥OWVGZ≥7RQP≥TGVWTP≥HTQO≥YCKVKPI≥KV≥GXCNWCVGU≥VQ≥while
EQPFKVKQPCN≥GZRTGUUKQP≥5KPEG≥6≥YCU≥YCKVKPI≥CMEC≥kolor≥DCTTKGT≥VQ≥GZRTGUUKQP≥KU≥HCNUG≥VQ
EWTTGPV≥DCTTKGT≥color≥KU≥BLUE≥CPF≥VQ≥VJGCF≥YKNN≥GZKV≥VQ≥DCTTKGT≥#HVGT≥TWPPKPI≥VJQWI≥VQ
EQFG≥HQNNQYKPI≥VQ≥DCTTKGT≥VQ≥PGZV≥KPUVCPEG≥QH≥VQ≥UCOG≥DCTTKGT≥GPEQWPVGTGF≥D[≥6≥YKNN≥DG≥

6JGCFU≥WPFGT≥9KPFQYU≥8KUVC≥QXG≥CFFGF≥C≥CONDITION_VARIABLE≥QDLGEV≥VQV≥YQTMU≥OWE≥NKMG
VQ≥2VJGCFU≥GSWKXCNGPV≥#≥CRITICAL_SECTION≥QDLGEV≥KU≥CUUQEKCVGF≥CPF≥TGNGCUGF≥YQP≥VQ≥VJGCF
YCKVU≥D[≥ECNNKBI≥eepConditionVariableCS()≥.KMG≥OCP[≥QVQT≥9KPFQYU≥6JGCFU≥HWPEVKQPU≥VQV
DNQEM≥VJGCFU≥VQU≥HWPEVKQP≥CNNQYU≥[QW≥VQ≥UGV≥C≥VKOG≥NKOKV≥6Q≥YCMG≥VJGCFU≥UNGGRKPI≥QP≥C
CONDITION_VARIABLE≥WUG≥C≥ECNN≥VQ≥WakeConditionVariable()

+≥ECPoV≥VCMG≥VQQ≥OWE≥ETGFKV≥HQT≥VQ≥DCTTKGT≥KORNGOGPVCVKQP≥QTG≥+≥QF≥VTKGF≥WUKPI≥VYQ
FGETGOGPVKPI≥EQWPVGTU≥VQ≥UGEQPF≥QH≥YKE≥EQWPVGF≥VJGCFU≥NGCXKPI≥VQ≥DCTTKGT≥UQ≥VQV≥VQ
HKPCN≥VJGCF≥EQWNF≥TGUGV≥CNN≥VQ≥EQWPVU≥9QP≥VKU≥OGVQF≥MGRV≥FGCFNQEMKPI≥+≥VWTPGF≥VQ
2TQITCOOKPI≥YKV≥3215+:≥6JGCFU ≥#FFKUQP9GUNG[≥2TQHGUUKQPCN≥≥D[≥&CXKF≥4
$VGPQH≥CPF≥IQV≥VQ≥EQNQT≥KPURKTCVKQP≥6Q≥DCTTKGT≥KORNGOGPVCVKQP≥IKXGP≥QTG≥KU≥C
UKORNKHKECVKQP≥QH≥VQ≥DCTTKGT≥EQFG≥KP≥$VGPQHoU≥DQQM≥1H≥EQWTUG≥KU≥KORNGOGPVCVKQP≥KU≥OWEJ
OQTG≥FGVCKNGF≥CPF≥RQTVCDNG≥VQP≥VQ≥QPG≥+oXG≥EQDDNGF≥VQIGVQT≥+≥WTIG≥[QW≥VQ≥IQ≥QXGT≥KU≥EQFGU
CPF≥EQPUKFGT≥WUKPI≥KU≥HWNN≥KORNGOGPVCVKQP≥KH≥[QW≥PGGF≥VQ≥WUG≥DCTTKGTU≥KP≥OQTG≥EQORNGZ
UKVWCVKQPU

Design Factor Scorecard

How efficient, simple, portable, and scalable is the reduce code described earlier? Let's examine the algorithm with respect to each of these categories.

Efficiency

The code declares a local sum variable for each thread (lSum≥VQ≥QNF≥VQ≥QPIQKPI≥EQORWVCVKQP
QH≥VQ≥NQECN≥RCTVKCN≥UWO≥XGP≥VQWI≥GCE≥VJGCF≥YKNN≥WRFCVG≥C≥WPKSWG≥GNGOGPV≥HTQO≥VQ
gSum≥CTTC[≥WUKPI≥VQ≥NQECN≥UWO≥XCTKCDNG≥CXQKFU≥CNN≥VQ≥HCNUG≥UQTKPI≥EQPHNKEVU≥VQV≥EQWNF≥CTKUG≥HQT
GCE≥CPF≥GXGT[≥KVGO≥YKVKP≥CP≥CUUKIPGF≥EQPM≥QH≥FCVC≥$≥WRFCVKPI≥CgSum≥UNQV≥QPEG≥RGT≥VJGCF
[QW≥ECP≥NKOKV≥VQ≥PWODGT≥QH≥HCNUG≥UQTKPI≥EQPHNKEVU≥VQ≥VQ≥PWODGT≥QH≥VJGCFU≥PQV≥VQ≥PWODGT
QH≥KVGOU≥KP≥VQ≥QTKIKPCN≥FCVC≥CTTC[

Simplicity

With the help of Figure 7-1, the code is pretty simple and straightforward. The use of i≥CPF
p2≥VQ≥FGVGTOKPG≥YKE3VJGCFU≥CTG≥CNNQYGF≥VQ≥RTQEGGF≥CPF≥HTQO≥YCTG≥FCVC≥KU≥ICVCTGF≥YQWNF
DG≥VC≥OQUV≥EQPHWUKPI≥RCTVU≥VQ≥UQOGQPG≥WPHCOKNKCT≥YKV3VC≥CNIQTKVO≥9KNG≥+≥EQUG≥VQ≥CPFNG
VCO≥UGRCTCVGN[≥HQT≥VC≥GZCORNG≥VC≥ i≥CPF≥p2≥XCTKCDNGU≥EQWNF≥DG≥EQODKPGF≥KPVQ≥VC≥NQQR

Portability

OpenMP has an explicit barrier for threads within an OpenMP team. There are also implicit barriers at the end of OpenMP worksharing constructs that you can use (or turn off with the nowait≥ENCWUG≥KH≥PQV≥PGGFGF≥+PUVGCF≥QH≥WUKPI≥reduction≥ENCWUG≥KP≥1RGP/2≥[QW≥EQWNF≥YTKVG
VC≥GZRNKEKV≥CNIQTKVO≥KP≥1RGP/2≥D[≥CVVCEJKPI≥VJGCF≥+&≥PWODGTU≥VQ≥GCE3VJGCF≥KP≥VC≥VGCO
CPF≥WUKPI≥VC≥GZRNKEKV≥DCTTKGT≥OQTOCNN[≥MPQYKPI≥UQOGQPG≥YCU≥GXGP≥EQPVGORNCVKPI≥UWE3C
WUG≥QH≥1RGP/2≥YQWNF≥GNKEKV≥QYNU≥QH≥FGTKUKXG≥NCWIVGT≥$WEG≥KV≥KH≥+≥ECP≥TGRNCEG≥CNN≥VC≥EQFG
HTQO≥p#≥$TTKGT≥1DLGEV≥+ORNGOGPVCVKQPq≥QP≥RCIG≥≥YKV3VC≥UKPING≥NKPG≥#pragma omp
barrier≥+≥YQWNF≥DG≥YKNNKPI≥VQ≥UYCNNQY≥O[≥RTGLWFKEGU≥CPF≥VCMG≥VC≥UKORNGT≥RCVJ

+P≥C≥OGUUCIGRCUUKPI≥U[UVGO≥[QW≥ECP≥YTKVG≥CP≥CNIQTKVOKE≥EQPUVTWEVKQP≥HQT≥TGFWEVKQP≥UKOKNCT
VQ≥VC≥QPG≥FKUEWUUGF≥RTGXKQWUN[≥+P≥VKU≥ECUG≥GCE3RTQEGUU≥CU≥C≥EOPM≥QH≥VC≥FCVC≥VC≥TGFWEVKQP
EQORWVCVKQP≥QP≥VCV≥RQTVKQP≥QH≥VC≥FCVC≥KU≥EQORWVGF≥NQECNN[≥CPF≥VC≥RTQEGUU≥+&≥PWODGTU≥CTG
WUGF≥VQ≥EQQTFKPCVG≥OGUUCIGU≥DGVYGGP≥RTQEGUUGU≥VQ≥RCUU≥VC≥NQECN≥RCTVKCN≥TGUWNVU≥VQ≥QVCT
RTQEGUUGU≥6CTG≥KU≥PQ≥PGGF≥HQT≥GZRNKEKV≥DCTTKGT≥U[PEJQPK\CVKQPU≥UKPEG≥VC≥CEV≥QH≥RCUUKPI
OGUUCIGU≥ECP≥IWCTCPVGG≥VC≥EQTTGEV≥QTFGT≥QH≥FCVC≥KU≥UGPV≥CPF≥UGPV≥QPN[≥YCP≥KV≥KU≥TGCF[
4GEGKXKPI≥RTQEGUUGU≥PGGF≥VQ≥DNQEM≥WPVKN≥VC≥FCVC≥KU≥TGEGKXGF≥6CUG≥RTQEGUUGU≥ECP≥GCUKN[≥EQORWVG
HTQO≥YCTG≥VC≥FCVC≥YKNN≥DG≥UGPV≥9KVKP≥VC≥/2+≥OGUUCIGRCUUKPI≥NKDTCT[≥VCTG≥KU≥C≥TGFWEVKQP
HWPEVKQP≥VCV≥YKNN≥OQTG≥QHVGP≥VCP≥PQV≥IKXG≥DGVVGT≥RGTHQTOCPEG≥VCP≥C≥CPFEQFGF≥TGFWEVKQP
CNIQTKVO

Scalability

Is this algorithm, with the barrier and all, the best way to do a reduction when there are a large number of threads involved? The implementation of the barrier object will be the principal limit to scalability of this reduction algorithm. The synchronization objects used within the implementation of a user-coded barrier object can be a bottleneck as the number of threads increases.

Applying MapReduce

I want to give you an idea about how to determine when MapReduce might be a potential solution to a concurrent programming problem. The task we're going to carry out here is finding all pairs of natural numbers that are mutually *friendly* within the range of positive integers provided to the program at the start of execution (this computation was part of the problem posed during the Intel Threading Challenge contest in July 2008). Two numbers are mutually friendly if the ratio of the sum of all divisors of the number and the number itself is equal to the corresponding ratio of the other number. This ratio is known as the *abundancy* of a number. For example, 30 and 140 are friendly, since the abundancy for these two numbers is equal (see Figure 7-2).

$$\frac{1+2+3+5+6+10+15+30}{30} = \frac{72}{30} = \frac{12}{5}$$

$$\frac{1+2+4+5+7+10+14+20+28+35+70+140}{140} = \frac{336}{140} = \frac{12}{5}$$

FIGURE 7-2. Friendly numbers

The serial algorithm for solving this problem is readily evident from the calculations shown in Figure 7-2. For each (positive) integer in the range, find all the divisors of the number, add them together, and then find the irreducible fractional representation of the ratio of this sum and the original number. After computing all the ratios, compare all pairs of ratios and print out a message of the friendly property found between any two numbers with matching ratios.

To decide whether this problem will fit into the MapReduce mold, you can ask yourself a few questions about the algorithm. Does the algorithm break down into two separate phases? Will the first phase have a data decomposition computation? Are those first phase computations independent of each other? Is there some "mapping" of data to keys involved? Can you "reduce" the results of the first phase to compute the final answer(s)?

This is a two-part computation. We can think of the first phase as a data decomposition of the range of numbers to be investigated, and there is a natural mapping of each number to its abundancy ratio. Factoring a number to compute the divisors of that number is independent

EXAMPLE 7-3. MapReduce solution to finding friendly numbers

```
int gcd(int u, int v)
{
  if (v == 0) return u;
  return gcd(v, u % v);
}

void FriendlyNumbers (int start, int end)
{
  int last = end-start+1;
  int *the_num = new int[last];
  int *num = new int[last];
  int *den = new int[last];

#pragma omp parallel
 {int i, j, factor, ii, sum, done, n;
//  -- MAP --
#pragma omp for schedule (dynamic, 16)
  for (i = start; i <= end; i++) {
    ii = i - start;
    sum = 1 + i;
    the_num[ii] = i;
    done = i;
    factor = 2;
    while (factor < done) {
      if ((i % factor) == 0) {
        sum += (factor + (i/factor));
        if ((done = i/factor) == factor) sum -= factor;
      }
      factor++;
    }
    num[ii] = sum; den[ii] = i;
    n = gcd(num[ii], den[ii]);
    num[ii] /= n;
```

```
    den[ii] /= n;
  }  // end for

//  -- REDUCE --
#pragma omp for schedule (static, 8)
  for (i = 0; i < last; i++) {
    for (j = i+1; j < last; j++) {
      if ((num[i] == num[j]) && (den[i] == den[j]))
        printf ("%d and %d are FRIENDLY \n", the_num[i], the_num[j]);
    }
  }
 }  // end parallel region
}
```

```
+IPQTG≥VG≥1RGP/2≥RTCIOCU≥HQT≥VG≥OQOGPV≥YKNG≥+≥FGUETKDG≥VG≥WPFGTN[KPI≥UGTKCN≥EQFG≥6G
FriendlyNumbers()≥HWPEVKQP≥VCMGU≥VYQ≥KPVGIGTU≥VGV≥FGHKPG≥VG≥TCPIG≥VQ≥DG≥UGCTEGF≥start≥CPF
end≥9G≥ECP≥CUUWOG≥VGV≥GTTQT≥EGEMKPI≥DGHQTG≥ECNNKPI≥VKU≥HWPEVKQP≥GPUWTGU≥VGV≥start≥KU≥NGUU
VGP≥end≥CPF≥VGV≥DQVJ3CTG≥RQUKVKXG≥PWODGTU≥6G≥EQFG≥HKTUV≥EQORWVGU≥VG≥NGPIVJ3QH≥VG≥TCPIG
last≥CPF≥CNNQECVGU≥OGOQT[≥VQ≥QNF≥VG≥PWODGTU≥YKVKP≥VG≥TCPIG≥the_num≥+V≥CNUQ≥CNNQECVGU
OGOQT[≥URCEG≥HQT≥VG≥PWOGTCVQT≥ num≥CPF≥FGPQOKPCVQT≥ den≥QH≥VG≥CDWPFCPE[≥TCVKQ≥HQT≥GCEJ
PWODGT≥9G≥FQPoV≥YCPV≥VQ≥WUG≥VG≥HNQCVKPI≥RQKPV≥XCNWG≥QH≥VG≥CDWPFCPE[≥UKPEG≥YG≥ECPoV
IWCTCPVGG≥VGV≥VYQ≥TCVKQU≥UWEJ3CU≥≥CPF≥≥YKNN≥[KGNF≥VG≥GZCEV≥UCOG≥float
XCNWG
```

```
6G≥HKTUV≥for≥NQQR≥KVGTCVGU≥QXGT≥VG≥PWODGTU≥KP≥VG≥TCPIG≥QH≥KPVGTGUV≥+P≥GCEJ3KVGTCVKQP≥VG≥EQFG
EQORWVGU≥VG≥QHHUGV≥KPVQ≥VG≥CNNQECVGF≥CTTC[U≥ii≥UCXGU≥VG≥PWODGT≥VQ≥DG≥HCEVQTGF≥CPF≥HKPFU
VG≥FKXKUQTU≥QH≥VGV≥PWODGT≥CPF≥CFFU≥VGO≥VQIGVGT≥sum≥6G≥KPVGTPCN≥while≥NQQR≥HKPFU≥VG
FKXKUQTU≥QH≥VG≥PWODGT≥D[≥C≥DTWVG≥HQTEG≥OGVQF≥9GPGXGT≥KV≥HKPFU≥C≥HCEVQT≥KV≥CFFU≥VGV≥HCEVQT
factor≥CPF≥VG≥CUUQEKCVGF≥OWNVKRNKECPF≥i/factor≥VQ≥VG≥TWPPKPI≥sum≥6G≥EQPFKVKQPCN≥VGUV
OCMGU≥UWTG≥VGV≥VG≥KPVGITCN≥USWCTG≥TQQV≥HCEVQT≥KU≥PQV≥CFFGF≥KP≥VYKEG≥6G≥done≥XCTKCDNG≥KU≥VG
NCTIGUV≥XCNWG≥VGV≥RQVGPVKCN≥FKXKUQTU≥factor)≥ECP≥DG≥6KU≥XCNWG≥KU≥UGV≥VQ≥i/factor≥YGPGXGT≥C
HCEVQT≥KU≥HQWPF≥UKPEG≥VGTG≥ECP≥DG≥PQ≥QVGT≥FKXKUQTU≥ITGCVGT≥VGP≥VG≥QPG≥CUUQEKCVGF≥YKVJ3VG
HCEVQT≥XCNWG≥KP≥factor
```

```
#HVGT≥UWOOKPI≥CNN≥VG≥FKXKUQTU≥QH≥C≥PWODGT≥VG≥EQFG≥UVQTGU≥sum≥KP≥VG≥CRRTQRTKCVG≥PWOGTCVQT
UNQV≥num≥CPF≥UVQTGU≥VG≥PWODGT≥KVUGNH≥KP≥VG≥EQTTGURQPFKPI≥FGPQOKPCVQT≥UNQV≥den≥6G
gcd()≥HWPEVKQP≥EQORWVGU≥VG≥ITGCVGUV≥EQOOQP≥FKXKUQT≥)%&≥HQT≥VGUG≥VYQ≥PWODGTU≥XKC≥VG
TGEWTUKXG≥WENKFGCP≥CNIQTKVO≥CPF≥FKXKFGU≥GCEJ3D[≥VG≥)%&≥UVQTGF≥KP≥n≥VQ≥RWV≥VG≥TCVKQ≥QH
VG≥VYQ≥KPVQ≥NQYGUV≥VGTOU≥#U≥PQVGF≥KP≥VG≥EQOOGPVU≥VG≥HCEVQTKPI≥CPF≥TCVKQ≥EQORWVCVKQPU
YKNN≥DG≥VG≥OCR≥RGUG
```

```
6G≥PGUVGF≥for≥NQQRU≥VGV≥HQNNQY≥EQORCTG≥VG≥PWOGTCVQTU≥CPF≥FGPQOKPCVQTU≥DGVYGGP≥WPKSWG
RCKTU≥QH≥PWODGTU≥YKVKP≥VG≥QTKIKPCN≥TCPIG≥6Q≥GPUWTG≥QPN[≥WPKSWG≥RCKTU≥QH≥TCVKQU≥CTG≥EQORCTGF
VG≥KPPGT≥j≥NQQR≥CEEGUUGU≥VG≥PWOGTCVQT≥CPF≥FGPQOKPCVQT≥CTTC[U≥HTQO≥VG≥i+1≥RQUKVKQP≥VQ≥VG
last
```

[i]≥KPFGZGF≥PWOGTCVQT≥CPF≥FGPQOKPCVQT≥XCNWGU≥OCVE≥VQ≥[j]≥KPFGZGF≥PWOGTCVQT≥CPF
FGPQOKPCVQT≥XCNWGU≥C≥HTKGPFN[≥RCKT≥KU≥KFGPVKHKGF≥CPF≥VQ≥VYQ≥PWODGTU≥UVQTGF≥KPF[i]≥CPF
the_num[j]≥CTG≥RTKPVGF≥YKV≥C≥OGUUCIG≥CDQWV≥VQ≥KT≥TGNCVKQPU≥R≥6≥KU≥KU≥VQ≥TGFWEG≥RQUG

6Q≥EQFG≥KP≥ZCORNG≥≥WUGU≥1RGP/2≥RTCIOCU≥VQ≥KORNGOGPV≥EQPEWTTGPE[≥6Q≥EQFG≥KPENWFGU
C≥RCTCNNGN≥TGIKQP≥CTQWPF≥DQV≥VQ≥OCR≥CPF≥TGFWEG≥RQTVKQPU≥9KV≥KP≥V≥KU≥TGIKQP≥CTG≥VQ
FGENCTCVKQPU≥QH≥VQ≥NQECN≥XCTKCDNGU≥i≥j≥factor≥ii≥sum≥done≥CPF≥n

6Q≥for≥NQQR≥QH≥VQ≥OCR≥RQUG≥KU≥NQECVGF≥YKV≥KP≥CP≥1RGP/2≥NQQR≥YQTMUQTKPI≥EQPUVTWEV≥VQ
FKXKFG≥KVGTCVKQPU≥QH≥VQ≥NQQR≥COQPI≥VQ≥VJGCFU≥OQVKEG≥VQV≥+oXG≥CFFGFFGF≥schedule≥ENCWUG≥VQ≥VQ
RTCIOC≥+oXG≥URGEKHKGF≥C≥dynamic≥UEQFWNG≥UKPEG≥VQ≥COQWPV≥QH≥EQORWVCVKQP≥PGGFGF≥VQ≥HKPF
FKXKUQTU≥QH≥PWODGTU≥YKNN≥XCT[≥YKFGN[≥FGRGPFKPI≥QP≥VQ≥PWODGT≥KVUGNH≥9KV≥KP≥VQ≥KPPGT≥while
NQQR≥VQ≥PWODGT≥QH≥HCEVQTU≥VQV≥OWUV≥DG≥EQPUKFGTGF≥YKNN≥DG≥UOCNNGT≥HQT≥C≥EQORQUKVG≥PWODGT
VQP≥C≥RTKOG≥QH≥UKOKNCT≥OCIPKVWFG≥GI≥≥CPF≥≥#NUQ≥NCTIGT≥PWODGTU≥YKNN≥VCMG≥OQTG≥VKOG
VQP≥UOCNNGT≥PWODGTU≥UKPEG≥VQTG≥YKNN≥DG≥OQTG≥HCEVQTU≥VQ≥VGUV≥CPF≥NKMGN[≥VQTG≥YKNN≥DG≥OQTG
FKXKUQTU≥QH≥VQ≥NCTIGT≥PWODGT≥GI≥≥CPF≥≥*GPEG≥VQ≥DCNCPEG≥VQ≥NQCF≥CUUKIPGF≥VQ≥VJGCFU
+oXG≥GNGEVGF≥VQ≥WUG≥C≥F[PCOKE≥UEQFWNG≥YKV≥C≥EQPM≥UK\G≥6JGCFU≥VQV≥CTG≥CUUKIPGF≥UWDTCPIG
EQPMU≥VQV≥ECP≥DG≥EQORWVGF≥SWKEMN[≥YKNN≥DG≥CDNG≥VQ≥TGSWGUV≥C≥PGY≥EQPM≥VQ≥YQTM≥QP≥YKNG
VJGCFU≥PGGFKPI≥OQTG≥VKOG≥VQ≥EQPVKPWG≥YKV≥VQ≥CUUKIPGF≥UWDTCPIG≥EQPM≥YKNN≥EQPVKPWG
HCEVQTKPI

(QT≥VQ≥TGFWEG≥RQUG≥KORNGOGPVCVKQP≥CPQVQT≥NQQR≥YQTMUQTKPI≥EQPUVTWEV≥KU≥RNCEGF≥QP≥VQ
QWVGT≥for≥NQQR≥5KORNG≥4WNG≥≥#U≥YKV≥VQ≥YQTMUQTKPI≥EQPUVTWEV≥KP≥VQ≥OCRRKPI≥EQFG≥C
schedule≥ENCWUG≥QU≥DGGP≥CFFGF≥VQ≥[KGNF≥C≥OQTG≥NQCF≥DCNCPEGF≥GZGEWVKQP≥+P≥V≥KU≥ECUG≥+oXG≥WUGF
C≥static≥UEQFWNG≥;QW≥UQWNF≥TGCNK\G≥VQV≥VQ≥PWODGT≥QH≥KPPGT≥NQQR≥KVGTCVKQPU≥KU≥FKHHGTGPV≥HQT
GXGT[≥KVGTCVKQP≥QH≥VQ≥QWVGT≥NQQR≥*QYGXGT≥WPNKMG≥VQ≥KPPGT≥NQQR≥schedule≥YKV≥KP≥VQ≥OCR≥RQUG
VQ≥COQWPV≥QH≥YQTM≥RGT≥QWVGT≥NQQR≥KVGTCVKQP≥KU≥OQPQVQPKECNN[≥FGETGCUKPI≥CPF≥RTGFKEVCDNG≥6Q
V[RKECN≥FGHCWNV≥HQT≥CP≥1RGP/2≥YQTMUQTKPI≥EQPUVTWEV≥YKVQWV≥VQ≥schedule≥ENCWUG≥KU≥VQ≥FKXKFG
VQ≥KVGTCVKQPU≥KPVQ≥C≥PWODGT≥QH≥UKOKNCTUK\GF≥EQPM≥GSWCN≥VQ≥VQ≥PWODGT≥QH≥VJGCFU≥YKV≥KP≥VQ
1RGP/2≥VGCO≥+P≥V≥KU≥ECUG≥UWE≥C≥UEQFWNG≥YQWNF≥CUUKIP≥OWE≥OQTG≥YQTM≥VQ≥VQ≥HKTUV≥EQPM
VQP≥VQ≥UWDUGSWGPV≥EQPMU

+≥XKUWCNK\G≥UWE≥C≥FGHCWNV≥UVCVKE≥FKXKUKQP≥QH≥KVGTCVKQPU≥NKMG≥VQ≥VTKCPING≥UQYP≥KP≥(KIWTG≥≥YQTG
VQ≥XGTVKECN≥CZKU≥KU≥VQ≥QWVGT≥NQQR≥KVGTCVKQPU≥VQ≥QTK\QPVCN≥CZKU≥KU≥VQ≥KPPGT≥NQQR≥CPF≥VQ≥YKFVJ
QH≥VQ≥VTKCPING≥TGRTGUGPVU≥VQ≥PWODGT≥QH≥KPPGT≥NQQR≥KVGTCVKQPU≥GZGEWVGF≥6Q≥CTGC≥QH≥VQ≥VTKCPING
CUUQEKCVGF≥YKV≥C≥VJGCF≥KU≥KP≥FKTGEV≥RTQRQTVKQP≥VQ≥VQ≥COQWPV≥QH≥YQTM≥VQV≥VQ≥VJGCF≥KU
CUUKIPGF≥6Q≥HQWT≥FKHHGTGPV≥UQFGU≥QH≥ITC[≥TGRTGUGPV≥C≥FKHHGTGPV≥VJGCF≥VQ≥YKE≥VQ≥EQPM≥QH
YQTM≥QU≥DGGP≥CUUKIPGF

(a) Divided into 4 static chunks; assigned to 4 threads

(b) Divided into 16 static chunks; assigned to 4 threads

FIGURE 7-3. Two static distributions of monotonically decreasing amounts of work among four threads

The more equitable division of area (i.e., work) among threads is shown in Figure 7-3 (b), which includes smaller and more numerous chunks. Those chunks are assigned to each thread in a round-robin fashion. The sum of the areas is then much more equal between threads and, thus, the load will be better balanced. You can accomplish the division of work shown in Figure 7-3 (b) by using the `schedule (static, 8)`≥ENCWUG≥IKXGP≥KP≥ZCORNG≥≥9KNG≥C≥F[PCOKE
UEGFWNG≥OC[≥IKXG≥C≥VKIVGT≥QXGTCNN≥NQCF≥DCNCPEG≥VG≥QXGTGCF≥CUUQEKCVGF≥YKVZFKUVTKDWVKPI
EWPMU≥QH≥KVGTCVKQPU≥VQ≥VTGCFU≥OKIV≥DG≥OQTG≥CFXGTUG≥VGP≥UKORN[≥WUKPI≥VG≥pIQQF≥GPQWIq
UVCVKE≥UEGFWNG

6GTG≥KU≥PQ≥OCIKE≥TGCUQP≥HQT≥WUKPI≥VG≥EWPM≥UK\GU≥VGV≥YG≥WUGF≥GTG≥9GP≥WUKPI≥CP≥1RGP/2
UEGFWNG≥ENCWUG≥QT≥KORNGOGPVKPI≥UWE3DGCXKQT≥WUKPI≥CP≥GZRNKEKV≥VTGCFKPI≥NKDTCT[≥VGUV
UGXGTCN≥FKHHGTGPV≥XCNWGU≥VQ≥UGG≥YGVGT≥VGTG≥CTG≥UKIPKHKECPV≥RGTHQTOCPEG≥FKHHGTGPEGU≥%QQUG
VG≥QPG≥VGV≥IKXGU≥VG≥DGVVGT≥RGTHQTOCPEG≥KP≥VG≥OCLQTKV[≥QH≥RQVGPVKCN≥KPRWV≥FCVC≥UGV≥ECUGU≥-GGR
KP≥OKPF≥VG≥UK\G≥QH≥C≥ECEG≥NKPG≥CPF≥EQQUG≥C≥EWPM≥UK\G≥VGV≥YKNN≥CNNQY≥HWNN≥ECEG≥NKPGU≥VQ≥DG
WUGF≥D[≥C≥UKPING≥VTGCF≥YGPGXGT≥RQUUKDNG≥GURGEKCNN[≥YGP≥WRFCVGU≥CTG≥TGSWKTGF

Friendly Numbers Example Summary

In this section, I've shown how you can apply the MapReduce framework to a serial code in order to find a concurrent equivalent. While the reduce phase of the friendly numbers problem might seem atypical, you need to be prepared to see past the standard many-to-one reduction case in order to be better equipped to translate serial codes to a MapReduce solution.

MapReduce As Generic Concurrency

I think the biggest reason that the MapReduce framework has gotten such a large amount of notoriety is that it can be handled in such a way that the programmer need not know much about concurrent programming. You can write a MapReduce ®engine¯ to execute concurrently when it is given the specifications on how the mapping operation is applied to individual elements, how the reduction operation is applied to individual elements, and how the reduction operation handles pairs of elements. For the programmer, these are simply serial functions (dealing with one or two objects). The engine would take care of dividing up the computations among concurrent threads. The TBB `parallel_reduce`≥CNIQTKVO≥KU≥CP≥GZCORNG≥QH VKU≥YOTG≥VO≥ `operator()`≥EQFG≥YQWNF≥EQPVCKP≥VO≥OCR≥RTUG≥EQORWVCVKQP≥QXGT≥C≥UWDTCPIG≥QH KVGOU≥CPF≥VO≥`join`≥OGVQF≥YQWNF≥KORNGOGPV≥VO≥TGFWEG≥RTUG

6KU≥KU≥VO≥TGCUQP≥VOV≥+≥TGEQOOGPF≥UVTWEVWTKPI≥[QWT≥OCR≥CPF≥TGFWEG≥RTUGU≥KP≥UWE3C≥YC[VOV≥[QW≥ECP≥CRRN[≥VO≥TGFWEVKQP≥EQORWVCVKQPU≥VQ≥KPFKXKFWCN≥KVGOU≥CPF≥RCTVKCN≥TGUWNVU≥QH RTGXKQWU≥TGFWEVKQPU≥(QT≥GZCORNG≥KP≥HKPFKPI≥VO≥OCZKOWO≥XCNWG≥HTQO≥C≥FCVC≥UGV≥VO≥FGHKPKVKQP QH≥VO≥TGFWEVKQP≥QRGTCVKQP≥KU≥VQ≥UKORN[≥EQORCTG≥VYQ≥KVGOU≥CPF≥TGVWTP≥VO≥XCNWG≥QH≥VO≥NCTIGUV 5WE3EQFG≥YQWNF≥YQTM≥YOVOT≥KV≥YCU≥DGKPI≥CRRNKGF≥VQ≥RCKTU≥QH≥GNGOGPVU≥HTQO≥VO≥QTKIKPCN EQNNGEVKQP≥QT≥HTQO≥RCTVKCN≥TGUWNVU≥VOV≥OF≥WUGF≥VKU≥EQFG≥VQ≥YKVVNG≥FQYP≥VO≥QTKIKPCN≥UGV≥KPVQ HGYGT≥RCTVKCN≥TGUWNVU≥+H≥VO≥/CR4GFWEG≥GPIKPG≥QPN[≥OU≥VQ≥FGCN≥YKV3VO≥FGVCKNU≥QH≥FKXKFKPI≥WR VO≥FCVC≥CPF≥TGEQODKPKPI≥RCTVKCN≥TGUWNVU≥VO≥RTQITCOOGT≥UKORN[≥UWRRNKGU≥VO≥EQORCTKUQP HWPEVKQP≥VQ≥VO≥GPIKPG≥6O≥NKOKVCVKQPU≥QH≥C≥IGPGTKE≥/CR4GFWEG≥GPIKPG≥OKIV≥RTGENWFG≥VO≥WUG QH≥UWE3C≥U[UVGO≥KH≥VO≥TGFWEVKQP≥EQORWVCVKQP≥YGTG≥OQTG≥EQORNGZ≥UWE3CU≥VO≥TGFWEVKQP EQORWVCVKQP≥WUGF≥KP≥VO≥HTKGPFN[≥PWODGTU≥RTQDNGO

+≥RTGFKEV≥VOV≥RTQXKFKPI≥IGPGTKE≥EQPEWTTGPE[≥GPIKPGU≥CPF≥CNIQTKVOU≥VOV≥CNNQY≥RTQITCOOGTU VQ≥YTKVG≥QPN[≥UGTKCN≥EQFG≥QT≥TGSWKTG≥C≥OKPKOWO≥QH≥EQPEWTTGPE[≥MPQYNGFIG≥YKNN≥DGEQOG≥RQRWNCT KP≥VO≥EQOKPI≥[GCTU≥6KU≥YKNN≥CNNQY≥RTQITCOOGTU≥YQ≥FQ≥PQV≥OXG≥VO≥VTCKPKPI≥QT≥UMKNNU≥KP EQPEWTTGPV≥RTQITCOOKPI≥VQ≥VCMG≥CFXCPVCIG≥QH≥OWNVKEQTG≥CPF≥OCP[EQTG≥RTQEGUUQTU≥PQY≥CPF KP≥VO≥HWVWTG≥1H≥EQWTUG≥WPVKN≥YG≥IGV≥VQ≥VO≥RQKPV≥YOTG≥YG≥ECP≥RTQITCO≥D[≥FGUETKDKPI≥QWT RTQDNGO≥QT≥CNIQTKVO≥KP≥?PINKU3NKMG≥KP≥EQWPVNGUU≥GRKUQFGU≥QKT≥6TGM2UQOGQPG≥OU≥VQ WPFGTUVCPF≥EQPEWTTGPV≥RTQITCOOKPI≥VQ≥DWKNF≥UWE3GPIKPGU≥YKE3EQWNF≥DG≥[QW

CHAPTER EIGHT

Sorting

THIS CHAPTER EXAMINES THE OPERATION OF CONCURRENT SORTING. At times, sorting has been estimated to account for over 80$ of all processing cycles. Presenting the results from database queries, compiling a list of business investments with associated risk-reward measures, and figuring the company payroll are all operations that require sorting. Even with the large amounts of processing time spent on graphical interfaces, visualization processing, and video games, sorting remains a vital part of computation. Every time you get a list of URLs from a search engine, the results have been sorted, typically by some measure of relevance to your original query.

In this chapter I will first consider compare-exchange sorts. These are sorting algorithms that use the results from comparing two keys to determine the relative order of the elements with those keys. Movement of data items will be based on those results and will be the exchange of the positions of the two items under consideration. The final algorithm considered is radix sort, which compares bits within keys to determine movement of data. Example codes presented in this chapter will use arrays of integers, but you can apply these techniques to structures with a key field to distinguish between elements.

Bubblesort

Bubblesort was the first sorting algorithm I ever learned. It is easy to code and simple to understand. A serial version of the algorithm appears in Example 8-1.

EXAMPLE 8-1. Serial Bubblesort

```
void BubbleSort(int *A, int N)
{
  int i, j;
  int temp;
  for (i = N-1; i > 0; i--) {
    for (j = 0; j < i; j++) {
      if (A[j] > A[j+1]) {
        temp = A[j]; A[j] = A[j+1]; A[j+1] = temp;
      }
    }
  }
}
```

```
6JTQWIJ GCEJ KVGTCVKQP QH VJG QWVGT NQQR  UQOG GNGOGPV
RQUKVKQP YKVJKP VJG CTTC[ YKNN OQXG CV NGCUV QPG UVGR E
VJG GNGOGPV YKVJ VJG NCTIGUV MG[ XCNWG VJCV KU PQV CV KV
HKPCN UQTVGF RQUKVKQP  6JWU  VJG KPPGT NQQR YKNN RCUU V
VJG CTTC[ WPVKN VJGTG CTG QPN[ VYQ GNGOGPVU NGHV VQ EQPU
RTQRGT QTFGT  VJG UQTV YKNN DG EQORNGVG  6JKU pDTWVG HQ
VGUV HQT GZEJCPIGU  +H PQ GZEJCPIGU CTG HQWPF KP CP[ KVGT
UQTVGF CPF VJGTG KU PQ PGGF HQT CFFKVKQPCN RCUUGU  (KIWT
```

BubbleSort()≥HWPEVKQP≥DCUGF≥QP≥VCUUG≥KFGCU≥WUKPI≥9KPFQYU≥6JGCFU
OQVKEG≥VCTV≥VCTU≥UKORNG≥EQFG≥UQYP≥KP≥ZCORNG≥≥CTU≥DNQCVGF≥VQ≥CV≥NGCUV≥HKXG≥VKOGU≥VCTU≥UK\G
CPF≥VCTV≥KU≥YKVQWV≥EQWPVKPI≥CNN≥VCTU≥UWRRQTV≥EQFG≥QWVUKFG≥QH≥VKMU≥HWPEVKQP≥PGGFGF≥VQ≥RTGRCTG
VCTU≥UGV≥QH≥NQEMU≥HQT≥VCTU≥ETKVKECN≥TGIKQPU

EXAMPLE 8-2. Threaded version of Bubblesort

```
unsigned __stdcall BubbleSort(LPVOID pArg)
{
  int i, j, k, releasePoint, temp, rpInc;
  BOOL exch;

  rpInc = N/NUM_LOCKS;
  rpInc++;

  while (!Done) {
    k = 0;
    exch = FALSE;
    EnterCriticalSection(&BLock[k]);
      i = iCounter--;
      releasePoint = rpInc;
      if (i <= 0) {
        Done = TRUE;
        LeaveCriticalSection(&BLock[k]);
        break;
      }

      for (j = 0; j < i; j++) {
        if (A[j] > A[j+1]) {
          temp = A[j]; A[j] = A[j+1]; A[j+1] = temp;
          exch = TRUE;
        }
        if (j == releasePoint) {
          LeaveCriticalSection(&BLock[k++]);
          EnterCriticalSection(&BLock[k]);
          releasePoint += rpInc;
        }
      }
    LeaveCriticalSection(&BLock[k]);
    if (!exch) Done = TRUE;
  }
  return 0;
}
```

6MU≥GZCORNG≥WUGU≥C≥while≥NQQR≥VQ≥MGGR≥VJGCFU≥GZGEWVKPI≥RCUUGU≥CU≥NQPI≥CU≥VCTG≥KU≥UVKNN≥C≥ECTPEG
VCTV≥UQOG≥FCVC≥TGOCKPU≥VQ≥DG≥UQTVGF≥(QT≥GCE3RCUU≥VJQWI3VCTU≥FCVC≥VCTU≥OCZKOWO≥PWODGT

A[5]≥CPF≥A[6]≥GNGOGPVU≥OWUV≥DG
UYCRRGF≥CPF≥6≥YKNN≥PQV≥PGGF≥VQ≥UYCR≥A[4]≥CPF≥A[5]≥6WU≥VCTG≥KU≥PQ≥FCVC≥TCEG≥QP≥A[5]≥6
CHVGT≥HKPFKPI≥PQ≥PGGF≥VQ≥UYCR≥YKNN≥YCKV≥HQT≥6≥VQ≥TGNKPSWKU≥VC≥NQEM≥HQT≥VC≥UGEQPF≥\QPG≥DGHQTG
KV≥YKNN≥VQWE≥A[5]≥CICKP

+P≥%CUG≥≥UQYP≥KP≥(KIWTG≥≥D≥6≥YKNN≥PQV≥PGGF≥VQ≥UYCR≥ A[5]≥CPF≥A[6]≥YKNG≥6≥YKNN≥UYCR
A[4]≥CPF≥A[5]≥#ICKP≥VCTG≥KU≥PQ≥FCVC≥TCEG≥QP≥A[5]

+P≥%CUG≥≥UQYP≥KP≥(KIWTG≥≥E≥YCTG≥DQV≥6≥CPF≥6≥YKNN≥DG≥UYCRRKPI≥VCKT≥TGURGEVKXG
RCKTU≥QH≥GNGOGPVU≥VKPIU≥IGV≥C≥NQV≥OQTG≥KPVGTGUVKPI≥(KIWTG≥≥IKXGU≥CP≥GZCORNG≥QH≥QPG≥RQVGPVKCN
KPVGTNGCXKPI≥QH≥VC≥UYCR≥KPUVTWEVKQPU≥CHVGT≥GCE≥VJ≥TGCF≥CU≥GZGEWVGF≥VQ≥YGUV≥CPF≥HQWPF≥VCV
C≥UYCR≥KU≥TGSWKTGF≥(KIWTG≥≥CNUQ≥UQYU≥VC≥TGUWNVKPI≥UVCVG≥QH≥VC≥CTTC[≥RQTVKQP≥HQT≥GCE≥UVGR
QH≥VC≥KPVGTNGCXKPI

FIGURE 8-3. BubbleSort() Case 3 interleaving

By Grabthar's Hammer Anytime you can "lose" a data item from a concurrent algorithm, you've got a big problem. The interleaving shown in Figure 8-3 looks like a disastrous data race that could scuttle use of the code in Example 8-2. What if, just before the last comparison in a zone, we required the thread to acquire the lock on the next zone? For that last element where the overlap occurs, a thread would need to be holding two locks. This solves our problem, but the already complicated code would just get more convoluted. Plus, the additional overhead wouldn't help performance and the number of threads that can be executing within the array at the same time would be restricted even further (lowering the scalability of the code). It's looking grim.

Take heart—there is a silver lining to this gray cloud. If you do some algorithmic analysis, you will find that Case 3 is impossible. In fact, Case 2 can never occur in practice either. After a thread has gone through a zone, the first element just beyond the boundary item will be greater

A[5]
DGHQTG≥TGNKPSWKUKPI≥VQ≥NQEM≥QP≥VQ≥HKTUV≥\QPG≥VQ≥6≥+H≥≥YGTG≥PQV≥VQ≥NCTIGUV≥MG[≥XCNWG≥KP≥VQ
HKTUV≥\QPG≥A[5]≥YQWNF≥EQPVCKP≥VQV≥NCTIGUV≥XCNWG≥QT≥C≥XCNWG≥NCTIGT≥VQP≥VQ≥NCTIGUV≥KP≥VQ≥HKTUV
\QPG≥6WU≥VQ≥QPN[≥XCNKF≥ECUG≥YG≥PGGF≥VQ≥EQPUKFGT≥HQT≥KPVGTNGCXKPI≥CPCN[UKU≥KU≥%CUG≥≥CPF
VQV≥ECUG≥QU≥PQ≥FCVC≥TCEG

9QP≥VT[KPI≥VQ≥RTQXG≥QT≥FKURTQXG≥VQ≥EQTTGEVPGUU≥QH≥[QWT≥EQPEWTTGPV≥CNIQTKVOU≥FQPoV≥TGN[
UQNGN[≥QP≥CP≥KPVGTNGCXKPI≥CPCN[UKU≥;QW≥OWUV≥CNUQ≥EQPUKFGT≥VQ≥RTQRGTVKGU≥QH≥VQ≥UGTKCN≥CNIQTKVO
CPF≥QY≥VQUG≥OC[≥QT≥OC[≥PQV≥DG≥EQPIGF≥D[≥VQ≥VTCPUHQTOCVKQP≥QH≥VQ≥UGTKCN≥EQFG

Design Factor Scorecard

Bubblesort is a difficult algorithm to parallelize and may not be worth the effort. Even so, how efficient, simple, portable, and scalable is the concurrent Bubblesort code described earlier? Let's review the algorithm with respect to each of these categories.

Efficiency

The linear access of array elements provides an almost foolproof way to predict which cache line of data will be needed next. If you can define the data sizes of zones on cache line boundaries, you should be able to eliminate any chance of false sharing.

The linear access of array elements and the need to pass over all potentially unsorted items doesn't allow for the reuse of cache. Sure, this ®property¯ is inherent in the nature of the serial algorithm, but when multiple threads are repeatedly bringing array elements into cache, looking at them twice, and then not needing them until the next iteration of the `while`≥NQQR
[QW≥ECP≥SWKEMN[≥EQMG≥VQ≥OGOQT[≥DWU≥YKV3VQQ≥OWE3FCVC≥6KU≥KU≥IQKPI≥VQ≥FKTGEVN[≥CHHGEV≥VQ
UECNCDKNKV[≥QH≥VKU≥CNIQTKVO

Simplicity

It sucks. I mean, just look at the difference between the serial code (Example 8-1) and the threaded code (Example 8-2). There's nothing simple about the zone implementation to keep threads out of the way of each other.

Fortunately, not all sorting algorithms are this difficult to transform. Later sections cover concurrent versions of other compare-exchange based sorting algorithms that will prove to be easier to understand and be much simpler to parallelize.

Portability

Not so good. You could do a translation of Example 8-2 into Pthreads. However, since this is a task decomposition, I'm not too sure how easy it would be to use an implicit threading library. And while you could use the explicit task facilities under OpenMP or Intel TBB, the problem of coordinating threads to not overlap each other would still remain.

A distributed-memory implementation would, by necessity, need to be an algorithm based on data decomposition. The movement of data between assigned blocks would require a message to be sent and received. The amount of such data traffic, even in the average case, would be prohibitive.

Scalability

This sucks, too. There will be a tradeoff between the number of threads you can use, based on the number of zones, and the amount of synchronization overhead and time spent waiting at locks. The more zones within the data, the more threads you can use: the more zones, the more locks will have to be acquired and released, and the more opportunity there will be for threads to sit idle waiting for access into the next data zone.

I suppose you could set up twice as many zones as threads to alleviate the contention on locks. You could delay threads starting so that there is a ®free¯ zone between each thread as it passes through the array. There would be less chance for idle threads, but using twice as many locks would increase the overhead by a factor of two.

None of the above even takes into account the fact that after each pass, the number of potentially unsorted elements decreases. When the algorithm progresses to the point where there are fewer zones than threads, you can't help but have idle threads.

Odd-Even Transposition Sort

The *odd-even transposition sort* compares adjacent pairs of items in an array to be sorted and exchanges them if they are found to be out of order relative to each other. Unlike Bubblesort, which also compares adjacent elements, the odd-even transposition sort compares disjointed pairs by using alternating odd and even index values during different phases of the sort. That is, on one pass through the data, it compares an odd index [i]≥CPF≥VQ≥CFLCEGPV≥GXGP≥KPFGZ
[i+1]≥GNGOGPV≥KP≥VQ≥UWEEGGFKPI≥RQUG≥KV≥EQORCTGU≥CP≥GXGP≥KPFGZ≥[i]≥CPF≥VQ≥CFLCEGPV≥QFF
KPFGZ≥[i+1]≥GNGOGPV≥6Q≥QFF≥CPF≥GXGP≥RQUGU≥CTG≥TGRGCVGF≥WPVKN≥PQ≥GZEQPIGU≥QH≥FCVC≥CTG
TGSWKTGF≥6YQ≥RQUGU≥QH≥VQ≥QFFGXGP≥VTCPURQUKVKQP≥UQTV≥CTG≥UQQYP≥KP≥(KIWTG≥

[4]≥CPF≥[5]≥GNGOGPVU≥KP
(KIWTG≥≥YG≥ECP≥UKORN[≥NGV≥VQ≥VJGCF≥CUUKIPGF≥VQ≥EQ0PM≥EQPVCKPKPI≥VQ≥NQYGT≥KPFGZGF
GNGOGPV≥FQ≥VQ≥EQORCTKUQP≥6QTG≥YKNN≥DG≥PQ≥RQUUKDKNKV[≥QH≥EQPVGPVKQP≥HTQO≥VQ≥VJGCF≥CUUKIPGF
VQ≥CFLCEGPV≥EQ0PM≥UKPEG≥CP[≥KVGO≥WUGF≥YKVKP≥C≥EQORCTKUQP≥KP≥GKVQT≥VQ≥QFF≥QT≥VQ≥GXGP
RQUUGU≥YKNN≥QPN[≥DG≥VQWEQF≥QPEG

6Q≥KVGTCVKQPU≥QH≥VQ≥KPPGTCT≥NQQR≥ECP≥DG≥FKXKFGF≥VQ≥HQTO≥KPFGRGPFGPV≥FCVC≥EQ0PMU≥HQT≥VJGCFU
1RGP/2≥ECP≥GCUKN[≥RCTCNNGNK\G≥VQ≥NQQR≥CU≥UQQYP≥KP≥?CORNG≥≥9G≥ECP≥WUG≥VQ≥KORNKEKV
DCTTKGT≥CV≥VQ≥GPF≥QH≥VQ≥RCTCNNGN≥TGIKQP≥VQ≥GPUWTG≥VQV≥CNN≥VQ≥EQORCTKUQPU≥CPF≥FCVC≥OQXGOGPV
YKVKP≥C≥RQUUGUXG≥EQORNGVGF≥DGHQTG≥VQ≥PGZV≥RQUUG≥KU≥NCWPEQF

EXAMPLE 8-4. Concurrent version of odd-even transposition sort with OpenMP

```
void OddEvenSort(int *A, int N)
{
  int exch = 1, start = 0, i;
  int temp;

  while (exch || start) {
    exch = 0;

#pragma omp parallel for private(temp) shared(start, exch)
    for (i = start; i < N-1; i += 2) {
      if (A[i] > A[i+1]) {
        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
        exch = 1;
      }
    }

    if (start == 0) start = 1;
    else start = 0;
  }
}
```

6Q≥ temp≥XCTKCDNG≥KU≥C≥YQTM≥XCTKCDNG≥CPF≥GCE3VJGCF≥TGSWKTGU≥C≥UGRCTCVG≥EQR[≥VQ0U≥+oXG≥CFFGF
C≥private≥ENCWUG≥VQ≥VQ≥RTCIOC≥+oXG≥CFFGF≥VQ≥shared≥ENCWUG≥OQTG≥HQT≥FQEWOGPVCVKQP≥VQP
PGEGUUKV[≥6Q≥EQFG≥WRFCVGU≥VQ≥exch≥XCTKCDNG≥YKVKP≥VQ≥NQQR≥CPF≥VQP≥TGCFU≥VQ≥XCNWG≥QWVUKFG
QH≥VQ≥RCTCNNGN≥TGIKQP≥5KPEG≥KV≥KU≥QPN[≥TGCF≥YKVKP≥VQ≥RCTCNNGN≥TGIKQP≥CPF≥WRFCVGF≥QWVUKFG≥QH
VQ≥TGIKQP≥start≥FQGU≥PQV≥PGGF≥CP[≥RTQVGEVKQP

+H≥[QW≥HWTVQT≥EQPUKFGT≥VQ≥WRFCVG≥exch≥[QW≥OC[≥YQPFGT≥YQVQT≥CP[≥RTQVGEVKQP≥KU≥TGSWKTGF
UKPEG≥GCE3VJGCF≥KU≥WRFCVKPI≥exch≥YKV3VQ≥UCOG≥XCNWG≥+P≥HCEV≥YQPGXGT≥VQ≥UCOG≥VJGCF
FGVGEVU≥OWNVKRNG≥UYCRU≥YKVKP≥VQ≥FCVC≥EQ0PM≥CUUKIPGF≥D[≥VQ≥1RGP/2≥NQQR≥YQTMUQTKPI
EQPUVTWEV≥KV≥YKNN≥TGCUUKIP≥VQ≥UCOG≥XCNWG≥6KU≥KU≥MPQYP≥*QUPKIP≥FCVC≥TCEG*≥ECWUG≥OWNVKRNG
VJGCFU≥OC[≥DG≥WRFCVKPI≥VQ≥XCNWG≥EQPEWTTGPVN[≥KV≥KU≥EQPUKFGTGF≥C≥FCVC≥TCEG≥DWV≥KV≥FQGU≥PQ
QTO≥VQ≥NGCXG≥QHH≥VQ≥RTQVGEVKQP≥QH≥VKU≥XCTKCDNG≥UKPEG≥KV≥KU≥VQ≥UCOG≥XCNWG≥DGKPI≥CUUKIPGF≥GCEJ
VKOG≥$UKFGU≥VQ≥ atomic≥RTCIOC≥FQGUPoV≥YQTM≥QP≥C≥UKORNG≥CUUKIPOGPV≥QH≥C≥XCNWG≥1P≥VQ≥QVQT
QPF≥KH≥VQ≥CNIQTKVO≥PGGFGF≥VQ≥MGGR≥EQWPV≥QH≥VQ≥PWODGT≥QH≥GZEQPIGUtTCVQT≥VQP≥LWUV

start≥QNFKPI≥C≥≥CPF≥YQWNF≥EQPIG≥KV≥VQ≥≥+H≥VQTG≥YGTG≥CP≥QFF≥PWODGT≥QH≥VJGCFU≥VQ
HKPCN≥XCNWG≥CUUKIPGF≥KP≥VQ≥ETKVKECN≥TGIKQP≥YQWNF≥DG≥EQTTGEV≥+oO≥PQV≥CYCTG≥QH≥CP[≥RQYGTU≥QT
OWNVKRNGU≥QH≥≥YKE3UGGOU≥VQ≥DG≥VQ≥DCUKU≥HQT≥VQ≥PWODGT≥QH≥EQTGU≥CXCKNCDNG≥QP≥OWNVKEQTG
RTQEGUUQTU≥VQV≥CTG≥QFF

+PUVGCF≥QH≥CNNQYKPI≥QPG≥VJGCF≥CV≥C≥VKOG≥VQ≥QXG≥CEEGUU≥VQ≥WRFCVG≥start≥YG≥PGGF≥QPG≥VJGCF
CNQPG≥VQ≥WRFCVG≥VKU≥XCTKCDNG≥(QT≥VKU≥YG≥WUG≥C≥single≥EQPUVTWEV≥6QTG≥KU≥VQ≥CFFGF≥DQPWU≥QH
CP≥KORNKEKV≥DCTTKGT≥CV≥VQ≥GPF≥QH≥VQ≥single≥EQPUVTWEV≥VQ≥GPUWTG≥VQV≥PQ≥VJGCFU≥YKNN≥IQ≥QPVQ≥VQ
PGZV≥UGV≥QH≥for≥NQQR≥KVGTCVKQPU≥YKV3VQ≥QNF≥XCNWG≥QH≥start

6QTG≥KU≥PQ≥GCU[≥YC[≥VQ≥RTQVGEV≥VQ≥TGCFKPI≥QH≥VQ≥exch≥XCNWG≥CU≥VQ≥EQPFKVKQPCN≥GZRTGUUKQP≥QH
VQ≥ while≥NQQR≥&QGU≥VQ≥TGCFKPI≥QH≥exch≥PGGF≥VQ≥DG≥RTQVGEVGF!≥+UPoV≥VKU≥CNUQ≥C≥DGPKIP≥FCVC≥TCEG
DGECWUG≥GCE3VJGCF≥YKNN≥DG≥TGCFKPI≥VQ≥UCOG≥XCNWG≥HTQO≥VQ≥XCTKCDNG≥KP≥QTFGT≥VQ≥VGUV≥YGVQT
QT≥PQV≥VQ≥RTQEGGF≥KPVQ≥VQ≥while≥NQQR!≥7PHQTVWPCVGN[≥VKU≥YQNG≥UQNWVKQP≥KUPoV≥CU≥DGPKIP≥CU≥YG
OKIV≥QXG≥QRGF

6Q≥KORNKEKV≥DCTTKGT≥QH≥VQ≥single≥EQPUVTWEV≥CUUWTGU≥WU≥VQV≥VQTG≥YKNN≥DG≥PQ≥FCVC≥TCEGU≥QP≥VQ
EQORCTKPI≥CPF≥UYCRRKPI≥QH≥FCVC≥DGVYGGP≥QFF≥CPF≥GXGP≥RCUUGU≥*QYGXGT≥YG≥ECP≥UVKNN≥TWP≥KPVQ
C≥ECVCUVTQRKE≥FCVC≥TCEG≥CV≥VQ≥RQKPV≥YQTG≥VJGCFU≥GPVGT≥WQ≥while≥NQQR≥GXGP≥VQ≥KPKVKCN≥GPVT[
%QPUKFGT≥VQ≥KPKVKCN≥GPVT[≥KPVQ≥VQ≥RCTCNNGN≥TGIKQP≥6Q≥UQTGF≥XCNWG≥QH≥KU≥≥CPF≥CNN≥VJGCFU
UQWNF≥GPVGT≥VQ≥while≥NQQR≥9QV≥KU≥VQ≥HKTUV≥VKPI≥VQV≥C≥VJGCF≥FQGU≥YQP≥KV≥GPVGTU≥VQ
while≥NQQR!≥6QV≥VJGCF≥TGUGVU≥VQ≥XCNWG≥QH≥exch≥VQ≥≥+H≥CP[≥VJGCFU≥YKVKP≥VQ≥1RGP/2≥VGCO
FQPoV≥CEEGUU≥VQ≥XCNWG≥QH≥exch≥KP≥VQ≥EQPFKVKQPCN≥GZRTGUUKQP≥DGHQTG≥UQOG≥VJGCF≥QU≥EQPIGF≥KV
VQUG≥VJGCFU≥YKNN≥DG≥EQPXKPEGF≥VQV≥VQ≥UQTVKPI≥QU≥EQORNGVGF≥CPF≥YKNN≥RTQEGGF≥VQ≥VQ≥GPF≥QH
VQ≥RCTCNNGN≥TGIKQP≥VQ≥YCKV≥HQT≥VQ≥QVQT≥VJGCFU≥VQ≥TGCNK\G≥VQV≥VQ≥UQTVKPI≥QU≥HKPKUGF≥+P≥VQ
YQTUV≥ECUG≥CNN≥DWV≥QPG≥VJGCF≥VQ≥VJGCF≥VQV≥YCU≥NWEM[≥GPQWI3VQ≥RGTHQTO≥VQ≥QTKIKPCN≥TGUGV
QH≥exch≥YKNN≥DG≥UKFGNKPGF≥9GoNN≥QXG≥C≥FGCFNQEM≥UKVWCVKQP≥DGECWUG≥CP[≥VJGCF≥VQV≥IGVU≥KPVQ≥VQ
UQTVKPI≥EQFG≥YKNN≥DG≥YCKVKPI≥CV≥VQ≥DCTTKGT≥QH≥VQ≥NQQR≥YQTMUQTKPI≥EQPUVTWEV≥YKNG≥CNN≥QVQTU
CTG≥YCKVKPI≥CV≥VQ≥KORNKEKV≥DCTTKGT≥QH≥VQ≥RCTCNNGN≥TGIKQP

%CP≥YG≥HKZ≥VKU!≥6Q≥DKPCT[≥PCVWTG≥QH≥exch≥KU≥VTKRRKPI≥WU≥WR≥5KORN[≥RTQVGEVKPI≥VQ≥TGUGV≥QH
exch≥YKV3UQOG≥HQTO≥QH≥U[PEJQPK\CVKQP≥YQPoV≥IGV≥CTQWPF≥VKU≥9G≥PGGF≥VQ≥UGV≥WR≥exch≥VQ≥CNNQY
O≥ VJGCFU≥VQ≥RCUU≥KPVQ≥VQ≥while≥NQQR≥KH≥CP[≥QH≥VQ≥VJGCFU≥QXG≥RGTHQTOGF≥CP≥GZECPIG≥KP≥VQ
RTGXKQWU≥RCUUG≥9QV≥KH≥TCVQT≥VQP≥UGVVKPI≥exch≥VQ≥≥YQP≥CP≥GZECPIG≥KU≥GZGEWVGF≥YG
CUUKIPGF≥VQ≥PWODGT≥QH≥VJGCFU≥KP≥VQ≥VGCO≥VQ≥exch!≥9KV3VQV≥ECPIG≥VQ≥TGUGV≥QH≥ exch≥YKVKP
VQ≥while≥NQQR≥YQWNF≥DG≥C≥RTQVGEVGF≥FGETGOGPV≥CPF≥exch≥YQWNF≥PQV≥DG≥≥WPVKN≥GXGT[≥VJGCF
QF≥GPVGTGF≥VQ≥while≥NQQR≥6Q≥ECPIGU≥PGGFGF≥VQ≥KORNGOGPV≥VKU≥OGVQF≥YKVKP≥VQ≥while
NQQR≥EQPFKVKQPCN≥GZRTGUUKQP≥CPF≥DQF[≥CTG≥UQYP≥KP≥ZCORNG≥

EXAMPLE 8-6. While-loop changes to odd-even transposition sort code

```
#pargma omp parallel
 {int temp;
  while (1) {
    if (exch == 0 && start == 0) break;
```

```
#pragma omp critical
    exch--;
#pragma omp for
    for (i = start; i < N; i += 2) {
      if (A[i] > A[i+1]) {
        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
#pragma omp critical
        exch = omp_num_threads();  // Assign with the number of threads
      }
    }
#pragma omp single
    if (start == 0) start = 1;
    else start = 0;
  }
 }
```

```
6C≥HKTUV≥VX[PI≥VQ≥PQVG≥KU≥VCV≥YGoXG≥VWTPGF≥VX[U≥KPVQ≥CP≥KPHKPKVG≥NQQR≥VGV≥YKVCP≥VCTG
CXG≥DGGP≥PQ≥GZECPIGU≥KP≥VC≥RTGXKQWU≥RCUG≥CPF≥VC≥OKPKOWO≥PWODGT≥QH≥RCUGU≥CXG≥DGGP
RTQEGUUGF≥6X[U≥ECPIG≥KU≥PGEGUUKVCVGF≥D[≥VC≥HCEV≥VCV≥YG≥ECPPQV≥RTQVGEV≥VC≥TGCF≥CEEGUU≥KP≥VC
while≥NQQR≥EQPFKVKQPCN≥2WNNKPI≥VCV≥TGHGTGPEG≥QWV≥QH≥VC≥GZRTGUUKQP≥CPF≥TGUVTWEVWTKPI≥VC≥NQQR
CU≥UQYP≥KU≥PGEGUUCT[≥VQ≥OCMG≥UWTG≥VCV≥CNN≥TGSWKTGF≥CEEGUUGU≥VQ≥exch≥CTG≥RTQVGEVGF
```

```
6C≥KPKVKCN≥CEEGUU≥QH≥exch≥YKVX[P≥VC≥ if≥EQPFKVKQP≥KU≥PQV≥KP≥C≥ETKVKECN≥TGIKQP≥+H≥YG≥CF≥RNCEGF≥VCV
NKPG≥QH≥EQFG≥YKVX[P≥VC≥ETKVKECN≥TGIKQP≥RTQVGEVKPI≥VC≥FGETGOGPV≥QH≥exch≥YG≥YQWNF≥CXG≥VYQ
GZKVU≥HTQO≥C≥DNQEM≥QH≥EQFG≥YKVX[P≥C≥ETKVKECN≥TGIKQP≥VC≥PQTOCN≥GZKV≥RQKPV≥HQT≥VC≥ECUG≥VCV
exch≥KU≥PQV≥≥CPF≥VC≥GZGEWVKQP≥QH≥VC≥break≥XGP≥KH≥UWE3DCF≥EQFKPI≥RTCEVKEG≥YGTG≥CNNQYGF
EQPUKFGT≥VC≥EQPUGSWGPEGU≥QH≥VC≥ECUG≥YCTG≥VC≥KPKVKCN≥if≥UVCVGOGPV≥CU≥DGGP≥RWV≥KP≥CP
1RGP/2≥ETKVKECN≥TGIKQP≥9CPGXGT≥VCTG≥YGTG≥PQ≥GZECPIGU≥QH≥KVGOU≥HTQO≥VC≥RTGXKQWU≥RCUG
VC≥HKTUV≥VTGCF≥KPVQ≥VC≥TGIKQP≥YQWNF≥CESWKTG≥VC≥U[PETQPK\CVKQP≥QDLGEV≥VCV≥1RGP/2≥WUGU
HQT≥C≥critical≥GZGEWVG≥VC≥DTGCM≥HCNN≥VQ≥VC≥KORNKEKV≥DCTTKGT≥CV≥VC≥GPF≥QH≥VC≥RCTCNNGN≥TGIKQP
CPF≥TGOCKP≥QNFKPI≥VC≥ETKVKECN≥U[PE≥QDLGEV≥6X[U≥YQWNF≥FGCFNQEM≥VC≥TGUV≥QH≥VC≥VTGCFU≥UKPEG
VC≥PGZV≥VTGCF≥CPF≥CNN≥QVCT≥VTGCFU≥YQWNF≥UKV≥YCKVKPI≥CV≥VC≥GPVT[≥VQ≥VC≥critical≥TGIKQP≥OQV
IQQF
```

```
9CV≥CDQWV≥VC≥WPUCHG≥UVCVG≥QH≥PQV≥RTQVGEVKPI≥VC≥CEEGUU≥QH≥exch≥KP≥VC≥KPKVKCN≥if≥VGUV≥CEEGUU!
%QWNF≥VC≥RTQVGEVGF≥FGETGOGPV≥QH≥VC≥exch≥XCTKCDNG≥KPVGTHGTG≥YKV3VX[U≥VQ≥ECWUG≥RTQDNGOU!≥+H
VC≥XCNWG≥QH≥exch≥KU≥≥VCTG≥KU≥PQ≥RTQDNGO≥UKPEG≥CNN≥VTGCFU≥YKNN≥GZGEWVCP≥VC≥break≥CPF≥GZKV≥VC
RCTCNNGN≥TGIKQP≥+H≥VC≥XCNWG≥QH≥exch≥KU≥VC≥PWODGT≥QH≥VTGCFU≥GZGEWVKPI≥VC≥UQTV≥GKVCT≥UGV≥KP
VC≥FGENCTCVKQP≥QWVUKFG≥VC≥RCTCNNGN≥TGIKQP≥QT≥UGV≥HTQO≥CP≥GZECPIG≥QH≥KVGOU≥KP≥VC≥RTGXKQWU
RCUG≥CU≥GCE3VTGCF≥GPVGTU≥VCV≥HKTUV≥ETKVKECN≥TGIKQP≥VQ≥FGETGOGPV≥QH≥exch≥YG≥MPQY≥VCV≥QPN[≥VC
NCUV≥VTGCF≥VQ≥RGTHQTO≥VC≥FGETGOGPV≥YKNN≥DTKPI≥VC≥XCNWG≥FQYP≥VQ≥≥6X[U≥PQ≥VTGCF≥YKNN≥HCKN
VQ≥GPVGT≥VC≥NQQR≥YQTMUCTKPI≥TGIKQP≥HTQO≥CP≥KPVGTCEVKQP≥QH≥VTGCFU≥ECEMKPI≥HQT≥VC≥GPF≥QH
UQTVKPI≥CPF≥GZGEWVKPI≥VC≥FGETGOGPV≥QH≥exch
```

```
9CV≥CDQWV≥VC≥QVCT≥WRFCVG≥QH≥ exch≥YKVX[P≥VC≥NQQR≥YQTMUCTKPI≥TGIKQP!≥6C≥UCOG≥1RGP/2
ETKVKECN≥TGIKQP≥RTQVGEVU≥KV≥UQ≥PQ≥VYQ≥VTGCFU≥ECP≥DG≥FGETGOGPVKPI≥exch≥CPF≥TGUGVVKPI≥KV≥VQ≥VC
PWODGT≥QH≥VTGCFU≥CV≥VC≥UCOG≥VKOG≥5VKNN≥VCTG≥KU≥C≥UWDVNG≥DWI≥VCV≥GZKUVU≥YKVX[P≥VC≥EQFG≥QH
```

while≥NQQR≥YGP≥QVGTU≥CTG≥UVKNN≥YCKVKPI≥VQ≥IGV≥VJQWI3VG≥ETKVKECN
TGIKQP≥HQT≥FGETGOGPVKPI≥exch≥9GP≥C≥UYCR≥QH≥KVGOU≥KU≥FGVGEVGF≥VG≥XCNWG≥QH≥exch≥KU≥UGV≥VQ≥VG
PWODGT≥QH≥VJGCFU≥VQUG≥VJGCFU≥VGV≥GXGPoV≥RCUUGF≥VJQWI3VG≥ETKVKECN≥TGIKQP≥YKNN≥VGP
FGETGOGPV≥VKU≥XCNWG≥VQ≥C≥PWODGT≥VGV≥KU≥PQV≥≥CPF≥KU≥NGUU≥VGP≥VG≥PWODGT≥QH≥VJGCFU≥#P[
UYCR≥CHVGT≥CNN≥VJGCFU≥GXG≥RCUUGF≥VG≥KPKVKCN≥ETKVKECN≥TGIKQP≥YKNN≥UGV≥VG≥XCNWG≥QH≥exch≥EQTTGEVN[
CPF≥VGTG≥YKNN≥DG≥PQ≥RTQDNGO≥HQT≥VG≥PGZV≥RCUUG≥WV≥ECP≥[QW≥IWCTCPVGG≥VGV≥VGTG≥YKNN≥DG
CPQVGT≥UYCR!

&QKPI≥KPVGTNGCXKPI≥CPCN[UKU≥QP≥[QWT≥EQPEWTTGPV≥CNIQTKVOU≥[QW≥OC[≥CNUQ≥GXG≥VQ≥URGEKH[≥VG
FCVC≥UGV≥VGV≥ECP≥VTKIIGT≥DCF≥DGGXKQT≥COQPI≥VJGCFU≥6Q≥UQY≥QHH≥VKU≥UWDVNG≥DWI≥KP≥VG
KORNGOGPVCVKQP≥KOCIKPG≥VG≥YQTUVECUG≥FCVC≥UGV≥HQT≥QFFGXGP≥VTCPURQUKVKQP≥UQTV≥C≥UQTVGF≥NKUV
DWV≥YKV3VG≥HKPCN≥GNGOGPV≥CV≥VG≥GCF≥QH≥VG≥NKUV≥&WTKPI≥CP[≥RCUUG≥VGTG≥YKNN≥DG≥C≥UKPING≥UYCR
WPVKN≥VG≥HKPCN≥GNGOGPV≥TGCEGU≥KVU≥UQTVGF≥RQUKVKQP≥9KV3QPN[≥C≥UKPING≥UYCR≥KP≥C≥UKPING≥VJGCF
KH≥VGTG≥CTG≥VJGCFU≥VGV≥GXGPoV≥RCUUGF≥VJQWI3VG≥FGETGOGPV≥ETKVKECN≥TGIKQP≥VGTG≥KU≥PQ
EGPEG≥VQ≥EQTTGEV≥VG≥RTQDNGO≥QH≥GXKPI≥VG≥YTQPI≥XCNWG≥KP≥exch≥HQT≥VG≥PGZV≥VKOG≥VJQWI3VG
while≥NQQR

.GV≥OG≥KNNWUVTCVG≥YJ≥YKV3C≥EQPETGVG≥KPVGTNGCXKPI≥GXKPI≥VG≥YTQPI≥XCNWG≥KP≥exch≥YKNN≥NGCF
VQ≥ECVCUVTQRG≥#UUWOG≥YG≥GXG≥GKIV≥VJGCFU≥TWPPKPI≥VG≥EQFG≥UQYP≥KP≥ZCORNG≥≥#HVGT
VJGG≥QH≥VGUG≥VJGCFU≥RCUU≥VJQWI3VG≥ETKVKECN≥TGIKQP≥CPF≥FGETGOGPV≥exch≥QPG≥QH≥VG≥VJGG
KFGPVKHKGU≥VYQ≥GNGOGPVU≥VGV≥OWUV≥DG≥UYCRRGF≥CPF≥GPVGTU≥VG≥ETKVKECN≥TGIKQP≥YKVKP≥VG≥NQQR
YQTMUGTKPI≥TGIKQP≥XGP≥KH≥VGTG≥KU≥C≥SWGWG≥VQ≥QTFGT≥CEEGUU≥VQ≥VG≥critical≥U[PEJQPK\CVKQP
QDLGEV≥VKU≥KPVGTNGCXKPI≥GZCORNG≥YKNN≥GXG≥VG≥QVGT≥HKXG≥VJGCFU≥UVKNN≥RTGRCTKPI≥VQ≥EGEM
YGVGT≥exch≥KU≥GSWCN≥VQ≥≥6G≥VJGCF≥VGV≥HQWPF≥VG≥QPG≥GZECPIG≥VQ≥DG≥FQPG≥FWTKPI≥VKU
RCUUG≥QH≥VG≥UQTV≥YKNN≥UGV≥exch≥VQ≥≥#HVGT≥VG≥HKPCN≥HKXG≥VJGCFU≥RCUU≥VJQWI3VG≥FGETGOGPVKPI
ETKVKECN≥TGIKQP≥VG≥XCNWG≥QH≥exch≥YKNN≥DG≥≥1PEG≥VG≥VJGCFU≥OGGV≥CV≥VG≥KORNKEKV≥DCTTKGT≥CV≥VG
GPF≥QH≥VG≥single≥TGIKQP≥VG[≥YKNN≥CNN≥UVCTV≥DCEM≥KPVQ≥VG≥DQF[≥QH≥VG≥NQQR≥*QYGXGT≥CHVGT
VG≥HKTUV≥VJGG≥VJGCFU≥GXG≥GZGEWVGF≥VG≥FGETGOGPV≥CP[≥QH≥VG≥QVGT≥HKXG≥VJGCFU≥VGV≥GXGPoV
CNTGCF[≥DGGP≥CDNG≥VQ≥RGTHQTO≥VG≥HKTUV≥EGEM≥QH≥(exch == 0)≥YKNN≥break≥QWV≥QH≥VG≥while≥NQQR≥CPF
YCKV≥CV≥VG≥GPF≥QH≥VG≥RCTCNNGN≥TGIKQP≥9G≥YKNN≥GXG≥UQOG≥VJGCFU≥CV≥VG≥DCTTKGT≥CV≥VG≥GPF≥QH
VG≥RCTCNNGN≥TGIKQP≥CPF≥VG≥TGUV≥CV≥VG≥DCTTKGT≥CV≥VG≥GPF≥QH≥VG≥NQQR≥YQTMUGTKPI≥EQPUVTWEV
OQPG≥QH≥VGUG≥VJGCFU≥ECP≥DG≥TGNGCUGF≥WPVKN≥CNN≥VJGCFU≥GXG≥OGV≥CV≥VG≥UCOG≥DCTTKGT≥5KPEG≥VGV
YKNN≥PGXGT≥GRRGP≥VG≥EQFG≥KU≥KP≥FGCFNQEM

N O T E

Even using the interlocked intrinsics of Windows Threads will not eliminate this problem.

We can use the intrinsics for every access of exchDWCEUDWGFUECP

UWNUGexchQWGQHVGFUDHQTGNNVGFUGXGTVWQTKGUG

%CWW#NEWCTTGCHVGTKCNTQQFGTGexch≥UQWNF≥FQ≥VG
VTKEM≥*QYGXGT≥QWT≥QTKIKPCN≥IQCN≥YCU≥VQ≥TGOQXG≥VG≥QXGTGCF≥QH≥EQPVKPWCNN[≥UVCTVKPI≥CPF

critical≥TGIKQPU≥CP≥CFFKVKQPCN≥KORNKEKV≥DCTTKGT≥HTQO≥VG≥single≥EQPUVTWEV≥CPF≥KH≥YG≥YCPV≥VQ
HKZ≥VKU≥NCVGUV≥RTQDNGO≥CP≥GZRNKEKV≥DCTTKGT≥+≥VKPM≥YG≥ECP≥DG≥TRR[≥VTV≥VG≥EQFG≥KP
ZCORNG≥≥YQTMU≥6TG≥KU≥OKPKOCN≥U[PEJQPK\CVKQP≥QXGTCF≥TGSWKTGF≥CPF≥VG≥QXGTCF
QH≥TGRGCVGFN[≥GPVGTKPI≥CPF≥GZKVKPI≥C≥RCTCNNGN≥TGIKQP≥NQQMU≥VQ≥DG≥OKVKICVGF≥D[≥PQV≥PGGFKPI≥CNN≥QH
VG≥CFFKVKQPCN≥U[PEJQPK\CVKQP≥VTV≥YQWNF≥DG≥PGEGUUCT[≥VQ≥FTKXG≥VG≥EQPEWTTGPE[≥KIGT≥KP≥VG
EQFG

Keeping threads awake longer without caffeine

If we can't efficiently minimize thread sleeping and waking overhead without adding a bunch of other overheads, we can cut down on the number of times we need to start and stop threads within the odd-even transposition sort implementation. Inside each parallel region, rather than doing either an odd phase or an even phase, we can do one of each. Code for this variation of the sort algorithm is given in Example 8-7.

EXAMPLE 8-7. Concurrent double-phase implementation

```
void OddEvenSort(int *A, int N)
{
  int exch0, exch1 = 1, trips = 0, i;

  while (exch1) {
    exch0 = 0;
    exch1 = 0;

#pragma omp parallel
   {int temp;

#pragma omp  for
    for (i = 0; i < N-1; i += 2) {
      if (A[i] > A[i+1]) {
        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
        exch0 = 1;
      }
    }
    if (exch0 || !trips) {
#pragma omp  for
      for (i = 1; i < N-1; i += 2) {
        if (A[i] > A[i+1]) {
          temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
          exch1 = 1;
        }
      }
    }  // if exch0
   }   // end parallel
    trips = 1;
  }
}
```

```
                    exch0≥CPF≥exch1≥ZGEWVKQP≥QH≥VG≥while
NQQR≥DQF[≥KU≥RTGFKECVGF≥QP≥VG≥exch1≥XCTKCDNG≥6G≥VYQ≥HNCIU≥CTG≥TGUGV≥CPF≥VG≥VJGCF≥GPVGTU≥VG
RCTCNNGN≥TGIKQP≥6G≥HKTUV≥NQQR≥YQTMUCTKPI≥EQPUVTWEV≥FKXKFGU≥WR≥VG≥FCVC≥CTTC[≥CPF≥KH≥CP[≥VJGCF
RGTHQTOU≥CP≥GZECIPIG≥KV≥UGVU≥exch0≥#HVGT≥DGKPI≥TGNGCUGF≥HTQO≥VG≥KORNKEKV≥DCTTKGT≥QH≥VG≥HKTUV
YQTMUCTKPI≥EQPUVTWEV≥GCE3VJGCF≥VGUVU≥exch0≥+H≥VG≥XCTKCDNG≥CU≥DGGP≥UGV≥QT≥VG≥zips≥EQWPVGT
CU≥PQV≥DGGP≥UGV≥CHVGT≥VG≥HKTUV≥RCKT≥QH≥RCUGU≥VG≥VJGCF≥GPVGTU≥VG≥UGEQPF≥NQQR≥YQTMUCTKPI
EQPUVTWEV≥+H≥CP[≥VJGCF≥RGTHQTOU≥CP≥GZECIPIG≥KP≥VG≥UGEQPF≥NQQR≥exch1≥KU≥UGV≥#HVGT≥VJGCFU
CTG≥TGNGCUGF≥HTQO≥VG≥GPF≥QH≥VG≥RCTCNNGN≥TGIKQP≥DCTTKGT≥VG≥EQPFKVKQPCN≥GZRTGUUKQP≥QH≥VG
while≥NQQR≥KU≥GXCNWCVGF≥+H≥VGTG≥YCU≥CP≥GZECIPIG≥exch1≥VG≥VJGCFU≥TGUGV≥VG≥HNCIU≥CPF≥GPVGT
VG≥RCTCNNGN≥TGIKQP

6G≥FQWDNGRCUG≥XCTKCVKQP≥EWVU≥VG≥PWODGT≥QH≥GPVTKGU≥CPF≥GZKVU≥HTQO≥VG≥RCTCNNGN≥TGIKQP≥KP
CNH≥6G≥UVCTV≥KPFGZ≥QP≥GCE3QH≥VG≥for≥NQQRU≥KU≥CTFEQFGF≥KPVQ≥VG≥CNIQTKVO≥YKE3GNKOKPCVGU
VG≥PGGF≥HQT≥VG≥single≥TGIKQP≥VQ≥HNKR≥VG≥part≥XCTKCDNG≥6G≥TGFWEVKQP≥QH≥VJGCF≥U[PEJQPK\CVKQP
HTQO≥WPTQNNKPI≥KVGTCVKQPU≥QH≥VG≥QTKIKPCN≥while≥NQQR≥UQWNF≥RTQXG≥VQ≥DG≥YGNN≥YQTV3VG≥GZVTC
EQFKPI
```

Design Factor Scorecard

How efficient, simple, portable, and scalable are the first concurrent version and the double-phase version of the odd-even transposition sort code? Let's review these algorithms with respect to each of these categories.

Efficiency

Cache use for the odd-even transposition sort (Example 8-4) is very good. Items from the array are accessed in linear order. Once a thread has a starting point within the array, it will be easy to predict the next cache line needed to be brought in from memory. Thus, we should assign large blocks of the array to each thread. With OpenMP, the `schedule (static)`≥ENCWUG≥GPUWTGU
```
VCV≥VG≥FCVC≥KU≥FKXKFGF≥KPVQ≥C≥PWODGT≥QH≥EQPMU≥GSWCN≥VQ≥VG≥PWODGT≥QH≥VJGCFU≥KH≥VKU≥KU≥PQV
CNTGCF[≥VG≥FGHCWNV≥HQT≥NQQR≥YQTMUCTKPI≥ZRNKEKV≥VJGCFKPI≥OQFGNU≥ECP≥FKXKFG≥VG≥KPPGT≥for
NQQR≥VQ≥CUUKIP≥FCVC≥KP≥VG≥UCOG≥YC[

+H≥VG≥FCVC≥UGVU≥CTG≥NCTIG≥GPQWI3VCV≥[QW≥OWUV≥RGTHQTO≥ECEG≥GXKEVKQPU≥DGHQTG≥C≥RCUU≥VJQWIJ
VG≥FCVC≥KU≥EQORNGVG≥[QW≥ECP≥VYGCM≥VG≥FQWDNGRCUG≥KORNGOGPVCVKQP≥ZCORNG≥≥VQ
CNVGTPCVG≥VG≥EQKEG≥QH≥KPETGOGPVKPI≥QT≥FGETGOGPVKPI≥VG≥NQQR≥KVGTCVQT≥6CV≥KU≥HQT≥VG≥HKTUV
for≥NQQR≥RCUG≥KPETGOGPV≥VG≥KVGTCVKQPU≥EQPVTQNNKPI≥VG≥KPFGZ≥QH≥CTTC[≥KVGOU≥CPF≥FWTKPI≥VG
UGEQPF≥for≥NQQR≥RCUG≥FGETGOGPV≥VG≥KVGTCVKQPU≥HTQO≥VG≥WRRGT≥GPF≥QH≥VG≥CTTC[≥#HVGT≥VG≥HKTUV
RCUG≥VG≥ECEG≥YKNN≥EQPVCKP≥VG≥KIGUV≥KPFGZGF≥GNGOGPVU≥HTQO≥VG≥CTTC[≥EQPM≥CPF≥VG
UGEQPF≥RCUG≥YQWNF≥UVCTV≥VG≥EQORCTGGZECIPIG≥QRGTCVKQPU≥CPF≥CEEGUU≥KVGOU≥KP≥TGXGTUG≥QTFGT
HTQO≥VG≥HKTUV≥RCUG≥6G≥UWEEGGFKPI≥HKTUV≥RCUG≥YKNN≥VGP≥CXG≥VG≥NQYGUV≥KPFGZGF≥CTTC[
GNGOGPVU≥UVQTGF≥KP≥ECEG≥QP≥YKE3VQ≥YQTM
```

Simplicity

From the previous discussions, you should be able to see that the simplest implementation of a concurrent odd-even transposition sort is given in Example 8-4. The original serial code is well preserved and recognizable. In fact, even the addition of a second `for`≥NQQR≥CPF≥NQQR
YQTMUCTKPI≥EQPUVTWEV≥ZCORNG≥≥[KGNFU≥EQFG≥VCV≥KU≥GCU[≥VQ≥WPFGTUVCPF≥6C≥EQFG≥HTQO
ZCORNG≥≥YKVZ3CP≥CFFGF≥GZRNKEKV≥DCTTKGT≥YQWNF≥PQV≥DG≥UQ≥UKORNG

Portability

Using an explicit threading model, a concurrent odd-even transposition sort will require dividing the `for`≥NQQR≥KVGTCVKQPU≥COQPI≥VJGCFU≥6KU≥KU≥GCU[≥DWV≥YG≥OWUV≥RTGUGTXG≥VC≥DCTTKGT
CV≥VC≥GPF≥QH≥VC≥`for`≥NQQR≥VQ≥GPUWTG≥VCV≥CNN≥QH≥VC≥EQORCTGGZEC PIG≥QRGTCVKQPU≥CXG≥EQORNGVGF
KP≥QPG≥RCUUG≥DGHQTG≥VJGCFU≥OQXG≥QP≥VQ≥VC≥PGZV≥RCUUG≥+H≥PQV≥VCTG≥KU≥C≥RQVGPVKCN≥FCVC≥TCEG≥QP
VC≥NCUV≥KPFGZGU≥EQORCTGF≥D[≥QPG≥VJGCF≥FWTKPI≥RCUUGM≥CPF≥VC≥HKTUV≥KPFGZGU≥VQWEC F≥D[≥VC
VJGCF≥CUUKIPGF≥VQ≥VC≥CFLCEGPV≥CTTC[≥EQ PM≥KP≥RCUUGM ≥(QT≥GZCORNG≥VC≥NCUV≥EQORCTKUQP
FQPG≥D[≥VJGCF≥6≥KP≥RCUUG≥ M≥EQWNF≥DG≥GNGOGPVU≥YKVZ3KPFGZGU≥=?≥CPF≥=?≥+H≥VJGCF≥6≥CU
UVCTVGF≥QPVQ≥RCUUGM ≥CV≥VC≥UCOG≥VKOG≥KV≥YKNN≥EQORCTG≥GNGOGPVU≥YKVZ3KPFGZGU≥=?≥CPF≥=?
6CV≥UCTGF≥GXGP≥KPFGZGF≥GNGOGPV≥=?≥KU≥VC≥RQVGPVKCN≥FCVC≥TCEG≥6CU≥UQOG≥HQTO≥QH≥DCTTKGT
U[PEJTQPK\CVKQP≥DGVYGGP≥RCUUGU≥KU≥PGGFGF

#≥FKUVTKDWVGFOGOQT[≥XGTUKQP≥QH≥QFFGXGP≥VTCPURQUKVKQP≥UQTV≥VCMGU≥VC≥NKPGCT≥QTFGT≥QH≥VC≥TCPMU
QH≥RTQEGUUGU≥CPF≥GZGEWVGU≥C≥EQORCTGURNKV≥QRGTCVKQP≥DGVYGGP≥CNVGTPCVKPI≥PGKIDQT≥RTQEGUUGU
#HVGT≥UQTVKPI≥VC≥NQECN≥FCVC≥KP≥GCEZ3RTQEGUU≥VC≥UQTVKPI≥VCMGU≥RNCEG≥KP≥CNVGTPCVKPI≥RCUUGU≥+P≥VC
QFF≥RCUUG≥QFFTCPMGF≥RTQEGUUGU≥GZECPIG≥C≥EQR[≥QH≥VCKT≥FCVC≥YKVZ3C≥EQR[≥QH≥VC≥FCVC≥CNF
D[≥VC≥GXGPTCPMGF≥RTQEGUUGU≥YQUG≥TCPM≥KU≥QPG≥NGUU≥XGPTCPMGF≥RTQEGUUGU≥GZECPIG≥FCVC
YKVZ3VC≥QFFTCPMGF≥RTQEGUUGU≥YQUG≥TCPM≥KU≥QPG≥ITGCVGT≥#NN≥RTQEGUUGU≥RGTHQTOCG≥mergesort≥QH
VC≥FCVC≥UGVU≥YKVZ3VC≥QFFTCPMGF≥RTQEGUUGU≥MGGRKPI≥VC≥WRRGT≥CNH≥QH≥VC≥FCVC≥CPF≥VC≥GXGP
TCPMGF≥RTQEGUUGU≥MGGRKPI≥VC≥NQYGT≥CNH≥+P≥VC≥GXGP≥RCUUG≥GXGPTCPMGF≥RTQEGUUGU≥GZECPIG
YKVZ3QFFTCPMGF≥RTQEGUUGU≥YQUG≥TCPM≥KU≥QPG≥ITGCVGT≥CPF≥MGGR≥VC≥WRRGT≥CNH≥QH≥VC≥OGTIGF
FCVC≥QFFTCPMGF≥RTQEGUUGU≥GZECPIG≥YKVZ3GXGPTCPMGF≥RTQEGUUGU≥CPF≥MGGR≥VC≥NQYGT≥CNH≥QH
VC≥OGTIGF≥FCVC≥#HVGT≥C≥PWODGT≥QH≥RCUUGU≥GSWCN≥VQ≥VC≥PWODGT≥QH≥RTQEGUUGUtCNH≥QFF≥RCUUGU
CPF≥CNH≥GXGP≥RCUUGUtVC≥FCVC≥KU≥UQTVGF≥CNQPI≥VC≥NKPGCT≥QTFGT≥QH≥RTQEGUUGU

Scalability

The odd-even transposition sort is very scalable. The amount of data to be compared and exchanged per thread remains constant throughout the execution of the algorithm.

Shellsort

Before we look at Shellsort, let's review insertion sort, since the two are related. I won't give a concurrent algorithm or implementation of insertion sort, though I will describe an

Quick Review of Insertion Sort

Insertion sort is such a simple algorithm to remember and code that I've always used it as my ®go to¯ sort whenever I needed a quick and dirty algorithm. For me, the definitive example of the algorithm is to imagine being dealt and holding a hand of bridge. Every bridge player I know keeps cards of the same suit together and ranks the cards by face value within each suit. In the player's mind, at least, there is a total order to the 52 cards of a deck of standard playing cards. For each new card that is dealt, a player picks up the new card and scans the current cards in her hand from one side to the other until she finds the place that the new card fits into the sequence.

Implementation of insertion sort starts with an unsorted list of items. This is typically an array, but you can also use it with a linked list of items. At any time during the sort, the list of items is separated into two parts
new item, the next unsorted item is chosen, and the sorted list is scanned from the biggest index down to the smallest. Items larger than the item to be inserted are moved down one place in the array. When the first item from the sorted portion smaller than the item to be inserted is located, the position of the inserted item (within the current set of sorted items) has been found and the item is stored in the array. Example 8-8 has the code for sorting an array of integers with the insertion sort algorithm.

EXAMPLE 8-8. Insertion sort algorithm

```
void InsertionSort(int *A, int N)
{
  int i, j, v;
  for (i = 1; i < N; i++) {
    v = A[i];
    j = i;
    while (A[j-1] > v) {
      A[j] = A[j-1];
      j--;
      if (j <= 0) break;
    }
    A[j] = v;
  }
}
```

(KIWTG≥≥UQQYU≥VQ≥UVCVG≥QH≥CP≥CTTC[≥QH≥≥KPVGIGT≥MG[U≥CV≥VQ≥GPF≥QH≥GCE≥QH≥VJGG≥KVGTCVKQPU
QH≥VQ≥QWVGT≥for≥NQQR≥QP≥≥KPVGIGT≥MG[U≥6Q≥UQFGF≥GNGOGPVU≥QH≥VQ≥CTTC[≥UQQY≥VQ≥UQTVGF
RQTVKQP

FIGURE 8-5. Insertion sort

Insertion sort is an inherently serial algorithm. Items are inserted one at a time. For a concurrent version of insertion sort, you might imagine starting in the middle and working out, giving two fronts that could each be handled by a different thread. One unsorted area contains the lower indexed slots and the other contains the higher indexed slots. Looking at the original data list in Example 8-8, imagine the sorted portion to be the [4]≥CPF≥[5]≥KPFGZGF
KVGOU≥9GV≥GRRGPU≥YGP≥QPG≥VJGCF≥CVVGORVU≥VQ≥KPUGTV≥≥CPF≥VG≥QVGT≥VJGCF≥VTKGU≥VQ
RNCEG≥VG≥≥KVGO!≥CE≥VJGCFoU≥UGCTE≥HQT≥KPUGTVKQP≥RQKPVU≥ETQUUGU≥VG≥UGCTE≥QH≥VG≥QVGT
VJGCF≥9GV≥RTQVGEVKQPU≥QH≥VG≥UGTGF≥CTTC[≥YQWNF≥YG≥PGGF≥VQ≥FQ≥VKU≥EQTTGEVN[!≥*QY≥EQORNGZ
FQGU≥VG≥EQFG≥DGEQOG≥YGP≥FGCNKPI≥YKV≥C≥UKHVKPI≥GPF≥DQWPFCT[≥VQ≥VGTOKPCVG≥VG≥KPUGTVKQP
QH≥VG≥pNCTIGUVq≥QT≥pUOCNNGUVq≥UGGP≥KVGO!≥XGP≥KH≥CNN≥QH≥VGUG≥SWGUVKQPU≥ECP≥DG≥CPUYGTGF
UCVKUHCEVQTKN[≥VG≥CNIQTKVO≥KU≥TGUVTKEVGF≥VQ≥WUKPI≥VYQ≥VJGCFU≥CPF≥KU≥PQV≥XGT[≥UECNCDNG

Serial Shellsort

Shellsort is a version of insertion sort that does an *h-sort* across the whole array (see Example 8-9). That is, the algorithm is working an insertion sort in *h* separate, interleaved partitions at the same time. After each pass with a given *h*, the value of *h* is reduced and another pass over the list is done to *h*-sort the list. When the value of *h* reaches 1, a simple insertion sort of the whole list is performed.

EXAMPLE 8-9. Serial Shellsort algorithm

```
void Shellsort(int *A, int N)
{
  int i, j, h, v;
  h = 1; while (h < N) h = 3*h + 1;
  h /= 3;
  while (h != 1) {
    for (i = h; i < N; i++) {
      v = A[i]; j = i;
      while (A[j-h] > v) {
        A[j] = A[j-h];
        j -= h;
        if (j <= h) break;
```