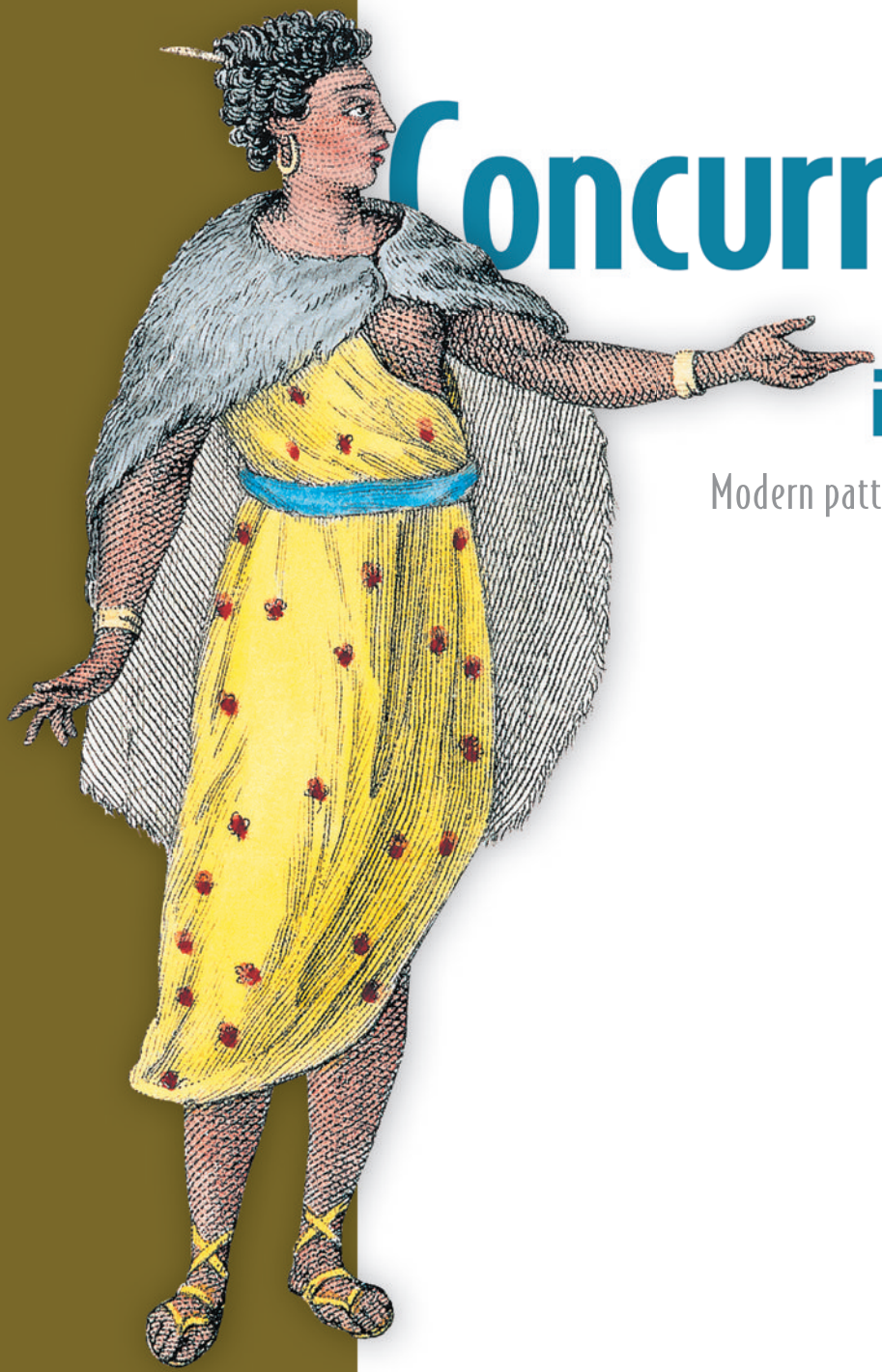
With examples in C# and F#



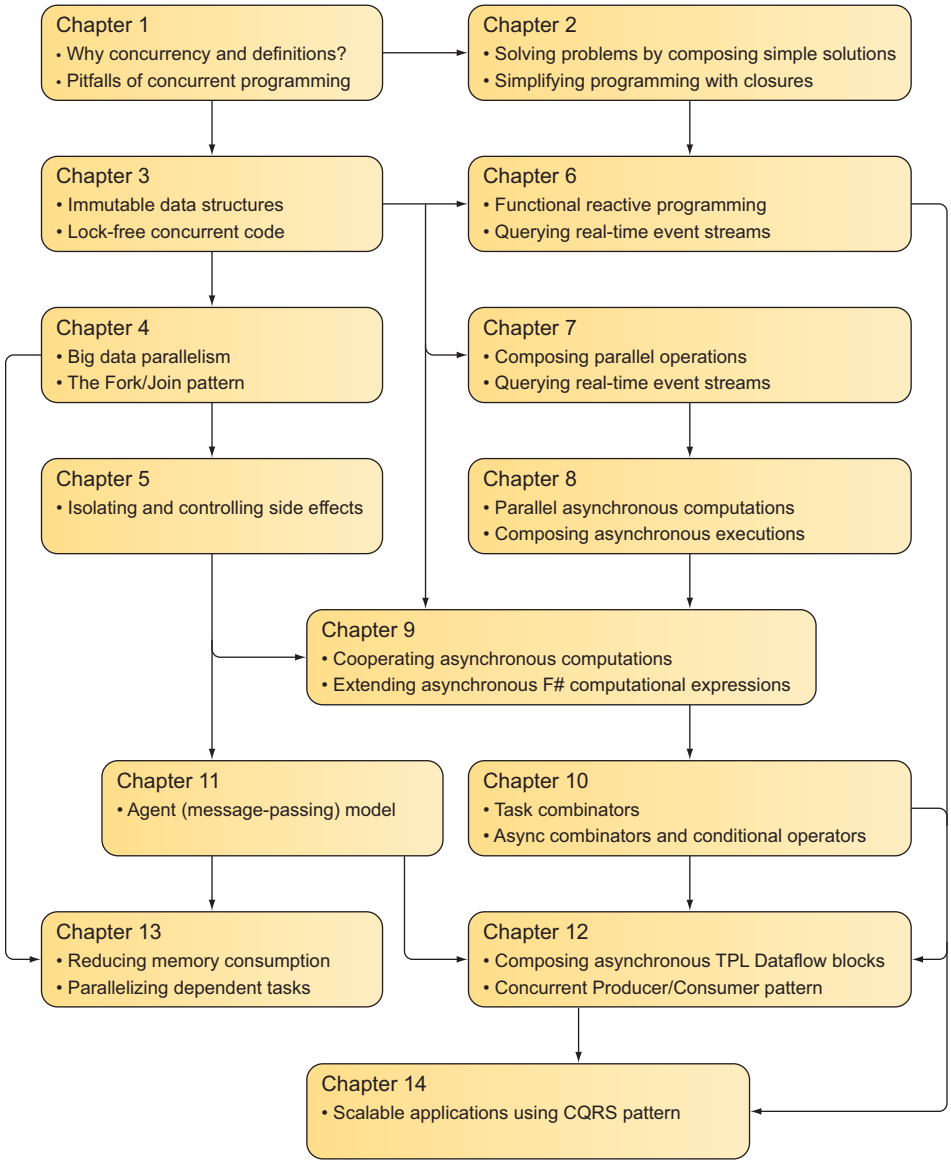
Concurrency in .NET

Modern patterns of concurrent and parallel programming

Riccardo Terrell

MANNING

Chapter dependency graph



Concurrency in .NET

Modern patterns of concurrent and parallel programming

RICCARDO TERRELL

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Marina Michaels
Technical development editor:	Michael Lund
Technical proofreader:	Viorel Moisei
Review editor:	Aleksandar Dragosavljević
Project manager:	Tiffany Taylor
Copy editor:	Katie Petito
Proofreader:	Elizabeth Martin
Typesetter:	Happenstance Type-O-Rama
Cover designer:	Marija Tudor

ISBN 9781617292996
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 23 22 21 20 19 18

I dedicate this book to my wonderful and supportive wife, Bryony. Your support, love, care, and continued encouragement throughout my writing process are what allowed me to turn this project into a reality. I love you and admire you for your many efforts and patience while I was busy writing. Thank you for always believing in me.

I also dedicate this book to my loyal pugs, Bugghina and Stellina, who were always unconditionally and enthusiastically beside me while writing this book. You're man's best friends and this author's furriest fans.

brief contents

contents

preface

You're probably reading this book, *Concurrency in .NET,* because you want to build blazingly fast applications, or learn how to dramatically increase the performance of an existing one. You care about performance because you're dedicated to producing faster programs, and because you feel excited when a few changes in your code make your application faster and more responsive. Parallel programming provides endless possibilities for passionate developers who desire to exploit new technologies. When considering performance, the benefits of utilizing parallelism in your programming can't be overstated. But using imperative and object-oriented programming styles to write concurrent code can be convoluted and introduces complexities. For this reason, concurrent programming hasn't been embraced as common practice writ large, leading programmers to search for other options.

When I was in college, I took a class in functional programming. At the time, I was studying Haskell; and even though there was a steep learning curve, I enjoyed every lesson. I remember watching the first examples and being amazed by the elegance of the solutions as well as their simplicity. Fifteen years later, when I began searching for other options to enhance my programs utilizing concurrency, my thoughts returned to these lessons. This time, I was able to fully realize how powerful and useful functional programming would be in designing my daily programs. There are several benefits to using a functional approach in your programming style, and I discuss each of them in this book.

My academic adventures met my professional work when I was challenged to build a software system for the health care industry. This project involved making an application to analyze radiological medical images. The image processing required several steps such as image noise reduction, Gaussian algorithm, image interpolation, and image filtering to apply color to the gray image. The application was developed using Java and initially ran as anticipated. Eventually the department increased the demand, as often happens, and problems started to appear. The software didn't have any problems or bugs, but with the increasing number of images to analyze, it became slower.

Naturally, the first proposed solution to this problem was to buy a more powerful server. Although this was a valid solution at the time, today if you buy a new machine with the intention of gaining more CPU computation speed, you'll be disappointed. This is because the modern CPU has more than one core, but the speed of a single core isn't any faster than a single core purchased in 2007. The better and more enduring alternative to buying a new server/computer was to introduce parallelism to take advantage of multicore hardware and all of its resources, ultimately speeding up the image processing.

In theory, this was a simple task, but in practice it wasn't so trivial. I had to learn how to use threads and locking; and, unfortunately, I gained firsthand experience in what a deadlock is.

This deadlock spurred me to make massive changes to the code base of the application. There were so many changes that I introduced bugs not even related to the original purpose of my changes. I was frustrated, the code base was unmaintainable and fragile, and the overall process was prone to bugs. I had to step back from the original problem and look for a solution from a different perspective. There had to be a better way.

> *The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*
>
> *—Edsger Dijkstra*

After spending a few days looking for a possible solution to solve the multithreading madness, I realized the answer. Everything I researched and read was pointing toward the functional paradigm. The principles I had learned in that college class so many years ago became my mechanism for moving forward. I rewrote the core of the image processing application to run in parallel using a functional language. Initially, transitioning from an imperative to a functional style was a challenge. I had forgotten almost all that I learned in college, and I'm not proud to say that during this experience, I wrote code that looked very object-oriented in functional language; but it was a successful decision overall. The new program compiled and ran with a dramatic performance improvement, and the hardware resources were completely utilized and bug free. Above all, an unanticipated and fantastic surprise was that functional programming resulted in an impressive reduction in the number of lines of code: almost 50% fewer than the original implementation using object-oriented language.

This experience made me reconsider OOP as the answer for all my programming problems. I realized that this programming model and approach to problem solving had a limited perspective. My journey into functional programming began with a requirement for a good concurrent programming model.

Ever since, I've had a keen interest in functional programming applied to multithreading and concurrency. Where others saw a complex problem and a source of issues, I saw a solution in functional programming as a powerful tool that could use the available hardware to run faster. I came to appreciate how the discipline leads to a coherent, composable, beautiful way of writing concurrent programs.

I first had the idea for this book in July 2010, after Microsoft introduced F# as part of Visual Studio 2010. It was already clear at that time that an increasing number of mainstream programming languages supported the functional paradigm, including C#, C++, Java, and Python. In 2007, C# 3.0 introduced first-class functions to the language, along with new constructs such as lambda expressions and type inference to allow programmers to introduce functional programming concepts. Soon to follow was Language Integrate Query (LINQ), permitting a declarative programming style.

In particular, the .NET platform has embraced the functional world. With the introduction of F#, Microsoft has full-featured languages that support both object-oriented and functional paradigms. Additionally, object-oriented languages like C# are becoming more hybrid and bridging the gap between different programming paradigms, allowing for both programming styles.

Furthermore, we're facing the multicore era, where the power of CPUs is measured by the number of cores available, instead of clock cycles per second. With this trend, single-threaded applications won't achieve improved speed on multicore systems unless the applications integrate parallelism and employ algorithms to spread the work across multiple cores.

It has become clear to me that multithreading is in demand, and it has ignited my passion to bring this programming approach to you. This book combines the power of concurrent programming and functional paradigm to write readable, more modular, maintainable code in both the C# and F# languages. Your code will benefit from these techniques to function at peak performance with fewer lines of code, resulting in increased productivity and resilient programs.

It's an exciting time to start developing multithreaded code. More than ever, software companies are making tools and capabilities available to choose the right programming style without compromise. The initial challenges of learning parallel programming will diminish quickly, and the reward for your perseverance is infinite. No matter what your field of expertise is, whether you're a backend developer or a frontend web developer, or if you build cloud-based applications or video games, the use of parallelism to obtain better performance and to build scalable applications is here to stay.

This book draws on my experience with functional programming for writing concurrent programs in .NET using both C# and F#. I believe that functional programming is

becoming the de facto way to write concurrent code, to coordinate asynchronous and parallel programs in .NET, and that this book will give you everything you need to be ready and master this exciting world of multicore computers.

acknowledgments

Writing a book is a daunting feat for anyone. Doing so in your secondary language is infinitely more challenging and intimidating. For me, you don't dare entertain dreams such as this without being surrounded by a village of support. I would like to thank all of those who have supported me and participated in making this book a reality.

My adventure with F# started in 2013, when I attended a FastTrack to F# in NYC. I met Tomas Petricek, who inspired me to fall headfirst into the F# world. He welcomed me into the community and has been a mentor and confidant ever since.

I owe a huge debt of gratitude to the fantastic staff at Manning Publications. The heavy lifting for this book began 15 months ago with my development editor, Dan Maharry, and continued with Marina Michaels, both of whom have been patient and sage guides in this awesome task.

Thank you to the many technical reviewers, especially technical development editor Michael Lund and technical proofer Viorel Moisei. Your critical analysis was essential to ensuring that I communicated on paper all that is in my head, much of which was at risk of being "lost in translation." Thank you also to those who participated in Manning's MEAP program and provided support as peer reviewers: Andy Kirsch, Anton Herzog, Chris Bolyard, Craig Fullerton, Gareth van der Berg, Jeremy Lange, Jim Velch, Joel Kotarski, Kevin Orr, Luke Bearl, Pawel Klimczyk, Ricardo Peres, Rohit Sharma, Stefano Driussi, and Subhasis Ghosh.

I received endless support from the members of the F# community who have rallied behind me along the way, especially Sergey Tihon, who spent countless hours as my sounding board.

And thank you to my family and friends who have cheered me on and patiently waited for me to join the world again for social weekends, dinner outings, and the rest.

Above all, I would like to acknowledge my wife, who supports my every endeavor and has never allowed me to shy away from a challenge.

I must also recognize my dedicated and loyal pugs, Bugghina and Stellina, who were always at my side or on my lap while I was writing this book deep into the night. It was also during our long evening walks that I was able to clear my head and find the best ideas for this book.

about this book

Concurrency in .NET provides insights into the best practices necessary to build concurrent and scalable programs in .NET, illuminating the benefits of the functional paradigm to give you the right tools and principles for handling concurrency easily and correctly. Ultimately, armed with your newfound skills, you'll have the knowledge needed to become an expert at delivering successful high-performance solutions.

Who should read this book

If you're writing multithreaded code in .NET, this book can help. If you're interested in using the functional paradigm to ease the concurrent programming experience to maximize the performance of your applications, this book is an essential guide. This book will benefit any .NET developers who want to write concurrent, reactive, and asynchronous applications that scale and perform by self-adapting to the current hardware resources wherever the program runs.

This book is also suitable for developers who are curious about exploiting functional programming to implement concurrent techniques. Prior knowledge or experience with the functional paradigm isn't required, and the basic concepts are covered in appendix A.

The code examples use both the C# and F# languages. Readers familiar with C# will feel comfortable right away. Familiarity with the F# language isn't strictly required, and

a basic overview is covered in appendix B. Functional programming experience and knowledge isn't required; the necessary concepts are included in the book.

A good working knowledge of .NET is assumed. You should have moderate experience in working with .NET collections and knowledge of the .NET Framework, with a minimum of .NET version 3.5 required (LINQ, `Action<>`, and `Func<>` delegates). Finally, this book is suitable for any platform supported by .NET (including .NET Core).

How this book is organized: a roadmap

The book's 14 chapters are divided into 3 parts. Part 1 introduces functional concurrent programming concepts and the skills you need in order to understand the functional aspects of writing multithreaded programs:

- Chapter 1 highlights the main foundations and purposes behind concurrent programming and the reasons for using functional programming to write multithreaded applications.
- Chapter 2 explores several functional programming techniques to improve the performance of a multithreaded application. The purpose of this chapter is to provide concepts used during the rest of the book, and to make you familiar with powerful ideas that have originated from the functional paradigm.
- Chapter 3 provides an overview of the functional concept of immutability. It explains how immutability is used to write predictable and correct concurrent programs, and how it's applied to implement and use functional data structures, which are intrinsically thread safe.

Part 2 dives into the different concurrent programming models of the functional paradigm. We'll explore subjects such as the Task Parallel Library (TPL), and implementing parallel patterns such as Fork/Join, Divide and Conquer, and MapReduce. Also discussed are declarative composition, high-level abstraction in asynchronous operations, the agent programming model, and the message-passing semantic:

- Chapter 4 covers the basics of processing a large amount of data in parallel, including patterns such as Fork/Join.
- Chapter 5 introduces more advanced techniques for parallel processing massive data, such as aggregating and reducing data in parallel and implementing a parallel MapReduce pattern.
- Chapter 6 provides details of the functional techniques to process real-time streams of events (data), using functional higher-order operators with .NET Reactive Extensions to compose asynchronous event combinators. The techniques learned are used to implement a concurrent friendly and reactive publisher-subscriber pattern.
- Chapter 7 explains the task-based programming model applied to functional programming to implement concurrent operations using the Monadic pattern based on a continuation-passing style. This technique is then used to build a concurrent- and functional-based pipeline.

- Chapter 8 concentrates on the C# asynchronous programming model to implement unbounded parallel computations. This chapter also examines error handling and compositional techniques for asynchronous operations.
- Chapter 9 focuses on the F# asynchronous workflow, explaining how the deferred and explicit evaluation of this model permits a higher compositional semantic. Then, we explore how to implement custom computation expressions to raise the level of abstraction, resulting in a declarative programming style.
- Chapter 10 wraps up the previous chapters and culminates in implementing combinators and patterns such as Functor, Monad, and Applicative to compose and run multiple asynchronous operations and handle errors, while avoiding side effects.
- Chapter 11 delves into reactive programming using the message-passing programming model. It covers the concept of natural isolation as a complementary technique with immutability for building concurrent programs. This chapter focuses on the F# `MailboxProcessor` for distributing parallel work using the agent model and the share-nothing approach.
- Chapter 12 explains the agent programming model using the .NET TPL Dataflow, with examples in C#. You'll implement both a stateless and stateful agent using C# and run multiple computations in parallel that communicate with each other using (passing) messages in a pipeline style

Part 3 puts into practice all the functional concurrent programming techniques learned in the previous chapters:

- Chapter 13 contains a set of reusable and useful recipes to solve complex concurrent issues based on real-world experiences. The recipes use the functional patterns you've seen throughout the book.
- Chapter 14 presents a full application designed and implemented using the functional concurrent patterns and techniques learned in the book. You'll build a highly scalable, responsive server application, and a reactive client-side program. Two versions are presented: one using Xamarin Visual Studio for an iOS (iPad)-based program, and one using WPF. The server-side application uses a combination of different programming models, such as asynchronous, agent-based, and reactive, to ensure maximum scalability.

The book also has three appendices:

- Appendix A summarizes the concepts of functional programming. This appendix provides basic theory about functional techniques used in the book.
- Appendix B covers the basic concepts of F#. It's a basic overview of F# to make you feel comfortable and help you gain familiarity with this programming language.
- Appendix C illustrates few techniques to ease the interoperability between the F# asynchronous workflow and the .NET task in C#.

About the code

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **`in bold`** to highlight the topic under discussion.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In some cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

The source code for this book is available to download from the publisher's website (www.manning.com/books/concurrency-in-dotnet) and from GitHub (https://github.com/rikace/fConcBook). Most of the code is provided in both C# and F# versions. Instructions for using this code are provided in the README file included in the repository root.

Book forum

Purchase of *Concurrency in .NET* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://forums.manning.com/forums/concurrency-in-dotnet. You can also learn more about Manning's forums and the rules of conduct at https://forums.manning.com/forums/about.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author

Riccardo Terrell is a seasoned software engineer and Microsoft MVP who is passionate about functional programming. He has over 20 years' experience delivering cost-effective technology solutions in a competitive business environment.

In 1998, Riccardo started his own software business in Italy, where he specialized in providing customized medical software to his clients. In 2007, Riccardo moved to the United States and ever since has been working as a .NET senior software developer and senior software architect to deliver cost-effective technology solutions in the business environment. Riccardo is dedicated to integrating advanced technology tools to increase internal efficiency, enhance work productivity, and reduce operating costs.

He is well known and actively involved in the functional programming community, including .NET meetups and international conferences. Riccardo believes in multi-paradigm programming as a mechanism to maximize the power of code. You can keep up with Riccardo and his coding adventures on his blog, www.rickyterrell.com.

about the cover illustration

The figure on the cover of *Concurrency in .NET* is a man from a village in Abyssinia, today called Ethiopia. The illustration is taken from a Spanish compendium of regional dress customs first published in Madrid in 1799, engraved by Manuel Albuerne (1764-1815). The book's title page states

> *Coleccion general de los Trages que usan actualmente todas las Nacionas del Mundo desubierto, dibujados y grabados con la mayor exactitud por R.M.V.A.R. Obra muy util y en special para los que tienen la del viajero universal*

which we translate, as literally as possible, thus:

> *General collection of costumes currently used in the nations of the known world, designed and printed with great exactitude by R.M.V.A.R. This work is very useful especially for those who hold themselves to be universal travelers*

Although little is known of the engraver, designers, and workers who colored this illustration by hand, the exactitude of their execution is evident in this drawing. The Abyssinian is just one of many figures in this colorful collection. Their diversity speaks vividly of the uniqueness and individuality of the world's towns and regions just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The collection brings to life a sense of isolation and distance of that period—and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then, and the diversity by region, so rich at the time, has faded away. It's now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

Part 1

Benefits of functional programming applicable to concurrent programs

Functional programming is a programming paradigm that focuses on abstraction and composition. In these first three chapters you'll learn how to treat computations as the evaluation of expressions to avoid the mutation of data. To enhance concurrent programming, the functional paradigm provides tools and techniques to write deterministic programs. Output only depends upon input and not on the state of the program at execution time. The functional paradigm also facilitates writing code with fewer bugs by emphasizing separation of concerns between purely functional aspects, isolating side effects, and controlling unwanted behaviors.

This part of the book introduces the main concepts and benefits of functional programming applicable to concurrent programs. Concepts discussed include programming with pure functions, immutability, laziness, and composition.

1 Functional concurrency foundations

This chapter covers

- Why you need concurrency
- Differences between concurrency, parallelism, and multithreading
- Avoiding common pitfalls when writing concurrent applications
- Sharing variables between threads
- Using the functional paradigm to develop concurrent programs

In the past, software developers were confident that, over time, their programs would run faster than ever. This proved true over the years due to improved hardware that enabled programs to increase speed with each new generation.

For the past 50 years, the hardware industry has experienced uninterrupted improvements. Prior to 2005, the processor evolution continuously delivered faster single-core CPUs, until finally reaching the limit of CPU speed predicted by Gordon Moore. Moore, a computer scientist, predicted in 1965 that the density and speed of transistors would double every 18 months before reaching a maximum speed beyond which technology couldn't advance. The original prediction for the increase of CPU

speed presumed a speed-doubling trend for 10 years. Moore's prediction, known as Moore's Law, was correct—except that progress continued for almost 50 years (decades past his estimate).

Today, the single-processor CPU has nearly reached the speed of light, all the while generating an enormous amount of heat due to energy dissipation; this heat is the limiting factor to further improvements.

CPU has nearly reached the speed of light

The speed of light is the absolute physical limit for electric transmission, which is also the limit for electric signals in the CPU. No data propagation can be transmitted faster than the light medium. Consequentially, signals cannot propagate across the surface of the chip fast enough to allow higher speeds. Modern chips have a base cycle frequency of roughly 3.5 GHz, meaning 1 cycle every 1/3,500,000,000 seconds, or 2.85 nanoseconds. The speed of light is about 3e8 meters per second, which means that data can be propagated around 0.30 cm (about a foot) in a nanosecond. But the bigger the chip, the longer it takes for data to travel through it.

A fundamental relationship exists between circuit length (CPU physical size) and processing speed: the time required to perform an operation is a cycle of circuit length and the speed of light. Because the speed of light is constant, the only variable is the size of the CPU; that is, you need a small CPU to increase the speed, because shorter circuits require smaller and fewer switches. The smaller the CPU, the faster the transmission. In fact, creating a smaller chip was the primary approach to building faster CPUs with higher clock rates. This was done so effectively that we've nearly reached the physical limit for improving CPU speed.

For example, if the clock speed is increased to 100 GHz, a cycle will be 0.01 nanoseconds, and the signals will only propagate 3 mm in this time. Therefore, a CPU core ideally needs to be about 0.3 mm in size. This route leads to a physical size limitation. In addition, this high frequency rate in such a small CPU size introduces a thermal problem in the equation. Power in a switching transistor is roughly the frequency ^2, so in moving from 4 GHz to 6 GHz there is a 225% increase of energy (which translates to heat). The problem besides the size of the chip becomes its vulnerability to suffer thermal damage such as changes in crystal structure.

Moore's prediction about transistor speed has come to fruition (transistors cannot run any faster) but it isn't dead (modern transistors are increasing in density, providing opportunities for parallelism within the confines of that top speed). The combination of multicore architecture and parallel programming models is keeping Moore's Law alive! As CPU single-core performance improvement stagnates, developers adapt by segueing into multicore architecture and developing software that supports and integrates concurrency.

The processor revolution has begun. The new trend in multicore processor design has brought parallel programming into the mainstream. Multicore processor architecture offers the possibility of more efficient computing, but all this power requires additional work for developers. If programmers want more performance in their code, they

must adapt to new design patterns to maximize hardware utilization, exploiting multiple cores through parallelism and concurrency.

In this chapter, we'll cover general information about concurrency by examining several of its benefits and the challenges of writing traditional concurrent programs. Next, we'll introduce functional paradigm concepts that make it possible to overcome traditional limitations by using simple and maintainable code. By the end of this chapter, you'll understand why concurrency is a valued programming model, and why the functional paradigm is the right tool for writing correct concurrent programs.

1.1 What you'll learn from this book

In this book I'll look at considerations and challenges for writing concurrent multithreaded applications in a traditional programming paradigm. I'll explore how to successfully address these challenges and avoid concurrency pitfalls using the functional paradigm. Next, I'll introduce the benefits of using abstractions in functional programming to create declarative, simple-to-implement, and highly performant concurrent programs. Over the course of this book, we'll examine complex concurrent issues providing an insight into the best practices necessary to build concurrent and scalable programs in .NET using the functional paradigm. You'll become familiar with how functional programming helps developers support concurrency by encouraging immutable data structures that can be passed between threads without having to worry about a shared state, all while avoiding side effects. By the end of the book you'll master how to write more modular, readable, and maintainable code in both C# and F# languages. You'll be more productive and proficient while writing programs that function at peak performance with fewer lines of code. Ultimately, armed with your newfound skills, you'll have the knowledge needed to become an expert at delivering successful high-performance solutions.

Here's what you'll learn:

- How to combine asynchronous operations with the Task Parallel Library
- How to avoid common problems and troubleshoot your multithreaded and asynchronous applications
- Knowledge of concurrent programming models that adopt the functional paradigm (functional, asynchronous, event-driven, and message passing with agents and actors)
- How to build high-performance, concurrent systems using the functional paradigm
- How to express and compose asynchronous computations in a declarative style
- How to seamlessly accelerate sequential programs in a pure fashion by using data-parallel programming
- How to implement reactive and event-based programs declaratively with Rx-style event streams
- How to use functional concurrent collections for building lock-free multithreaded programs

- How to write scalable, performant, and robust server-side applications
- How to solve problems using concurrent programming patterns such as the Fork/Join, parallel aggregation, and the Divide and Conquer technique
- How to process massive data sets with parallel streams and parallel Map/Reduce implementations

This book assumes you have knowledge of general programming, but not functional programming. To apply functional concurrency in your coding, you only need a subset of the concepts from functional programming, and I'll explain what you need to know along the way. In this fashion, you'll gain the many benefits of functional concurrency in a shorter learning curve, focused on what you can use right away in your day-to-day coding experiences.

1.2 Let's start with terminology

This section defines terms related to the topic of this book, so we start on common ground. In computer programming, some terms (such as *concurrency*, *parallelism,* and *multithreading*) are used in the same context, but have different meanings. Due to their similarities, the tendency to treat these terms as the same thing is common, but it is not correct. When it becomes important to reason about the behavior of a program, it's crucial to make a distinction between computer programming terms. For example, concurrency is, by definition, multithreading, but multithreading isn't necessarily concurrent. You can easily make a multicore CPU function like a single-core CPU, but not the other way around.

This section aims to establish a common ground about the definitions and terminologies related to the topic of this book. By the end of this section, you'll learn the meaning of these terms:

- Sequential programming
- Concurrent programming
- Parallel programming
- Multitasking
- Multithreading

1.2.1 Sequential programming performs one task at a time

Sequential programming is the act of accomplishing things in steps. Let's consider a simple example, such as getting a cup of cappuccino at the local coffee shop. You first stand in line to place your order with the lone barista. The barista is responsible for taking the order and delivering the drink; moreover, they are able to make only one drink at a time so you must wait patiently—or not—in line before you order. Making a cappuccino involves grinding the coffee, brewing the coffee, steaming the milk, frothing the milk, and combining the coffee and milk, so more time passes before you get your cappuccino. Figure 1.1 shows this process.

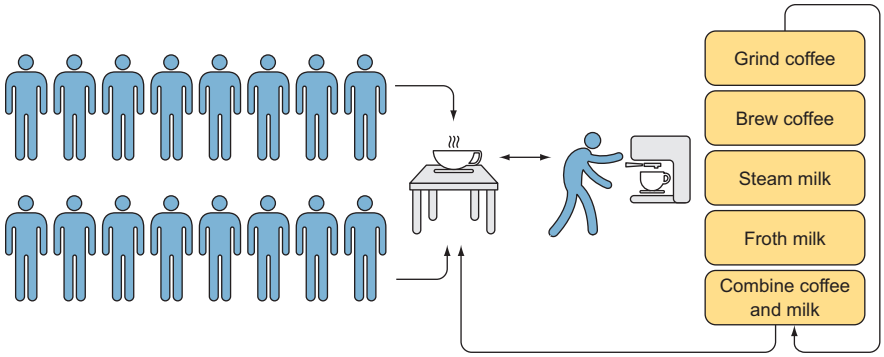


Figure 1.1 For each person in line, the barista is sequentially repeating the same set of instructions (grind coffee, brew coffee, steam milk, froth milk, and combine the coffee and the milk to make a cappuccino).

Figure 1.1 is an example of sequential work, where one task must be completed before the next. It is a convenient approach, with a clear set of systematic (step-by-step) instructions of what to do and when to do it. In this example, the barista will likely not get confused and make any mistakes while preparing the cappuccino because the steps are clear and ordered. The disadvantage of preparing a cappuccino step-by-step is that the barista must wait during parts of the process. While waiting for the coffee to be ground or the milk to be frothed, the barista is effectively inactive (blocked). The same concept applies to sequential and concurrent programming models. As shown in figure 1.2, sequential programming involves a consecutive, progressively ordered execution of processes, one instruction at a time in a linear fashion.

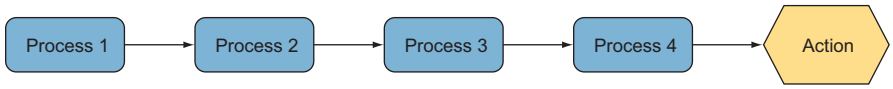


Figure 1.2 Typical sequential coding involving a consecutive, progressively ordered execution of processes

In imperative and object-oriented programming (OOP) we tend to write code that behaves sequentially, with all attention and resources focused on the task currently running. We model and execute the program by performing an ordered set of statements, one after another.

1.2.2 Concurrent programming runs multiple tasks at the same time

Suppose the barista prefers to initiate multiple steps and execute them concurrently? This moves the customer line along much faster (and, consequently, increases garnered tips). For example, once the coffee is ground, the barista can start brewing the espresso. During the brewing, the barista can take a new order or start the process of steaming and frothing the milk. In this instance, the barista gives the perception of

doing multiple operations at the same time (multitasking), but this is only an illusion. More details on multitasking are covered in section 1.2.4. In fact, because the barista has only one espresso machine, they must stop one task to start or continue another, which means the barista executes only one task at a time, as shown in figure 1.3. In modern multicore computers, this is a waste of valuable resources.

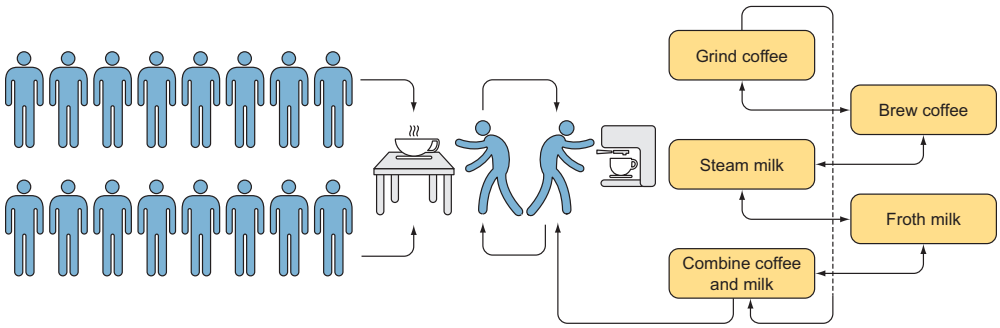


Figure 1.3 The barista switches between the operations (multitasking) of preparing the coffee (grind and brew) and preparing the milk (steam and froth). As a result, the barista executes segments of multiple tasks in an interleaved manner, giving the illusion of multitasking. But only one operation is executed at a time due to the sharing of common resources.

Concurrency describes the ability to run several programs or multiple parts of a program at the same time. In computer programming, using concurrency within an application provides actual multitasking, dividing the application into multiple and independent processes that run at the same time (concurrently) in different threads. This can happen either in a single CPU core or in parallel, if multiple CPU cores are available. The throughput (the rate at which the CPU processes a computation) and responsiveness of the program can be improved through the asynchronous or parallel execution of a task. An application that streams video content is concurrent, for example, because it simultaneously reads the digital data from the network, decompresses it, and updates its presentation onscreen.

Concurrency gives the impression that these threads are running in parallel and that different parts of the program can run simultaneously. But in a single-core environment, the execution of one thread is temporarily paused and switched to another thread, as is the case with the barista in figure 1.3. If the barista wishes to speed up production by simultaneously performing several tasks, then the available resources must be increased. In computer programming, this process is called parallelism.

1.2.3 Parallel programming executes multiples tasks simultaneously

From the developer's prospective, we think of parallelism when we consider the questions, "How can my program execute many things at once?" or "How can my program solve one problem faster?" *Parallelism* is the concept of executing multiple tasks at once concurrently, literally at the same time on different cores, to improve the speed of

the application. Although all parallel programs are concurrent, we have seen that not all concurrency is parallel. That's because parallelism depends on the actual runtime environment, and it requires hardware support (multiple cores). Parallelism is achievable only in multicore devices (figure 1.4) and is the means to increasing performance and throughput of a program.

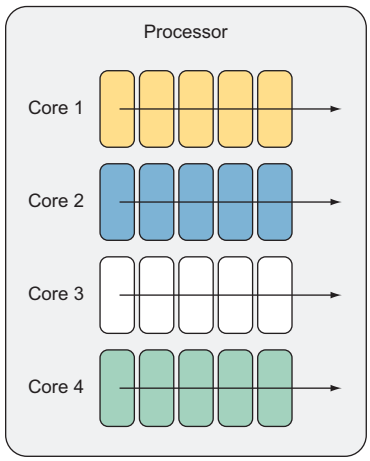


Figure 1.4 Only multicore machines allow parallelism for simultaneously executing different tasks. In this figure, each core is performing an independent task.

To return to the coffee shop example, imagine that you're the manager and wish to reduce the waiting time for customers by speeding up drink production. An intuitive solution is to hire a second barista and set up a second coffee station. With two baristas working simultaneously, the queues of customers can be processed independently and in parallel, and the preparation of cappuccinos (figure 1.5) speeds up.

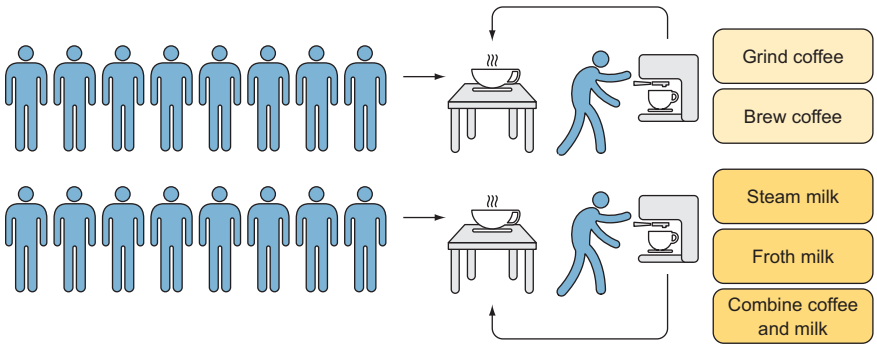


Figure 1.5 The production of cappuccinos is faster because two baristas can work in parallel with two coffee stations.

No break in production results in a benefit in performance. The goal of parallelism is to maximize the use of all available computational resources; in this case, the two baristas are working in parallel at separate stations (multicore processing).

Parallelism can be achieved when a single task is split into multiple independent subtasks, which are then run using all the available cores. In figure 1.5, a multicore machine (two coffee stations) allows parallelism for simultaneously executing different tasks (two busy baristas) without interruption.

The concept of timing is fundamental for simultaneously executing operations in parallel. In such a program, operations are *concurrent* if they can be executed in parallel, and these operations are *parallel* if the executions overlap in time (see figure 1.6).

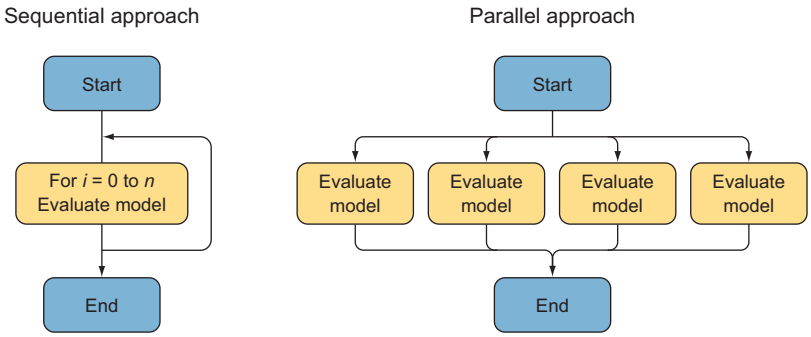


Figure 1.6 Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time.

Parallelism and concurrency are related programming models. A parallel program is also concurrent, but a concurrent program isn't always parallel, with parallel programming being a subset of concurrent programming. While concurrency refers to the design of the system, parallelism relates to the execution. Concurrent and parallel programming models are directly linked to the local hardware environment where they're performed.

1.2.4 Multitasking performs multiple tasks concurrently over time

Multitasking is the concept of performing multiple tasks over a period of time by executing them concurrently. We're familiar with this idea because we multitask all the time in our daily lives. For example, while waiting for the barista to prepare our cappuccino, we use our smartphone to check our emails or scan a news story. We're doing two things at one time: waiting and using a smartphone.

Computer multitasking was designed in the days when computers had a single CPU to concurrently perform many tasks while sharing the same computing resources. Initially, only one task could be executed at a time through time slicing of the CPU. (*Time slice* refers to a sophisticated scheduling logic that coordinates execution between multiple threads.) The amount of time the schedule allows a thread to run before scheduling a different thread is called *thread quantum*. The CPU is time sliced so that each thread gets to perform one operation before the execution context is switched to another thread. Context switching is a procedure handled by the operating system to

multitask for optimized performance (figure 1.7). But in a single-core computer, it's possible that multitasking can slow down the performance of a program by introducing extra overhead for context switching between threads.

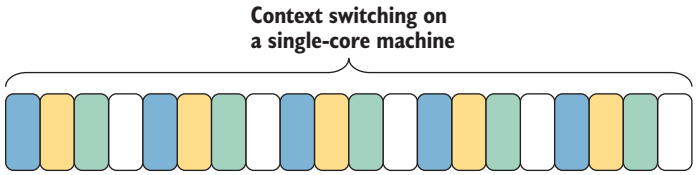


Figure 1.7 Each task has a different shade, indicating that the context switch in a single-core machine gives the illusion that multiple tasks run in parallel, but only one task is processed at a time.

There are two kinds of multitasking operating systems:

- *Cooperative multitasking systems*, where the scheduler lets each task run until it finishes or explicitly yields execution control back to the scheduler
- *Preemptive multitasking systems* (such as Microsoft Windows), where the scheduler prioritizes the execution of tasks, and the underlying system, considering the priority of the tasks, switches the execution sequence once the time allocation is completed by yielding control to other tasks

Most operating systems designed in the last decade have provided preemptive multitasking. Multitasking is useful for UI responsiveness to help avoid freezing the UI during long operations.

1.2.5 Multithreading for performance improvement

Multithreading is an extension of the concept of multitasking, aiming to improve the performance of a program by maximizing and optimizing computer resources. Multithreading is a form of concurrency that uses multiple threads of execution. Multithreading implies concurrency, but concurrency doesn't necessarily imply multithreading. Multithreading enables an application to explicitly subdivide specific tasks into individual threads that run in parallel within the same process.

> **NOTE** A *process* is an instance of a program running within a computer system. Each process has one or more threads of execution, and no thread can exist outside a process.

A *thread* is a unit of computation (an independent set of programming instructions designed to achieve a particular result), which the operating system scheduler independently executes and manages. Multithreading differs from multitasking: unlike multitasking, with multithreading the threads share resources. But this "sharing resources" design presents more programming challenges than multitasking does. We discuss the problem of sharing variables between threads later in this chapter in section 1.4.1.

The concepts of parallel and multithreading programming are closely related. But in contrast to parallelism, multithreading is hardware-agnostic, which means that it can be performed regardless of the number of cores. Parallel programming is a superset of multithreading. You could use multithreading to parallelize a program by sharing resources in the same process, for example, but you could also parallelize a program by executing the computation in multiple processes or even in different computers. Figure 1.8 shows the relationship between these terms.

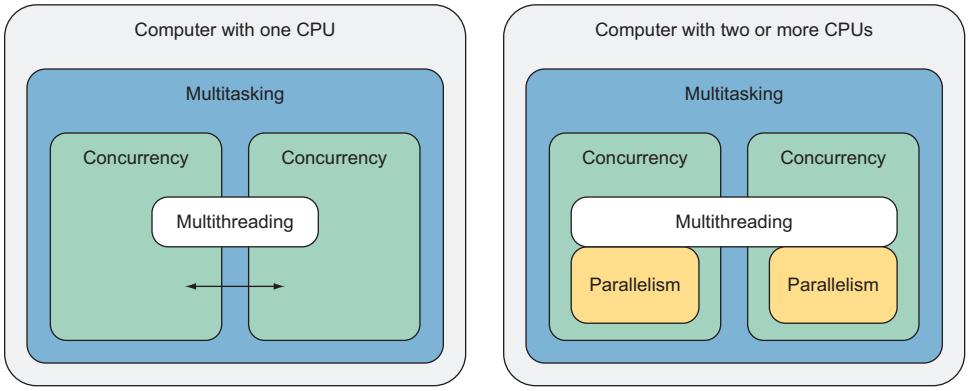


Figure 1.8 Relationship between concurrency, parallelism, multithreading, and multitasking in a single and a multicore device

To summarize:

- *Sequential programming* refers to a set of ordered instructions executed one at a time on one CPU.
- *Concurrent programming* handles several operations at one time and doesn't require hardware support (using either one or multiple cores).
- *Parallel programming* executes multiple operations at the same time on multiple CPUs. All parallel programs are concurrent, running simultaneously, but not all concurrency is parallel. The reason is that parallelism is achievable only on multicore devices.
- *Multitasking* concurrently performs multiple threads from different processes. Multitasking doesn't necessarily mean parallel execution, which is achieved only when using multiple CPUs.
- *Multithreading* extends the idea of multitasking; it's a form of concurrency that uses multiple, independent threads of execution from the same process. Each thread can run concurrently or in parallel, depending on the hardware support.

1.3 Why the need for concurrency?

Concurrency is a natural part of life—as humans we're accustomed to multitasking. We can read an email while drinking a cup of coffee, or type while listening to our favorite song. The main reason to use concurrency in an application is to increase

performance and responsiveness, and to achieve low latency. It's common sense that if one person does two tasks one after another it would take longer than if two people did those same two tasks simultaneously.

It's the same with applications. The problem is that most applications aren't written to evenly split the tasks required among the available CPUs. Computers are used in many different fields, such as analytics, finance, science, and health care. The amount of data analyzed is increasing year by year. Two good illustrations are Google and Pixar.

In 2012, Google received more than 2 million search queries per minute; in 2014, that number more than doubled. In 1995, Pixar produced the first completely computer-generated movie, *Toy Story*. In computer animation, myriad details and information must be rendered for each image, such as shading and lighting. All this information changes at the rate of 24 frames per second. In a 3D movie, an exponential increase in changing information is required.

The creators of *Toy Story* used 100 connected dual-processor machines to create their movie, and the use of parallel computation was indispensable. Pixar's tools evolved for *Toy Story 2*; the company used 1,400 computer processors for digital movie editing, thereby vastly improving digital quality and editing time. In the beginning of 2000, Pixar's computer power increased even more, to 3,500 processors. Sixteen years later, the computer power used to process a fully animated movie reached an absurd 24,000 cores. The need for parallel computing continues to increase exponentially.

Let's consider a processor with N (as any number) running cores. In a single-threaded application, only one core runs. The same application executing multiple threads will be faster, and as the demand for performance grows, so too will the demand for N to grow, making parallel programs the standard programming model choice for the future.

If you run an application in a multicore machine that wasn't designed with concurrency in mind, you're wasting computer productivity because the application as it sequences through the processes will only use a portion of the available computer power. In this case, if you open Task Manager, or any CPU performance counter, you'll notice only one core running high, possibly at 100%, while all the other cores are underused or idle. In a machine with eight cores, running non-concurrent programs means the overall use of the resources could be as low as 15% (figure 1.9).

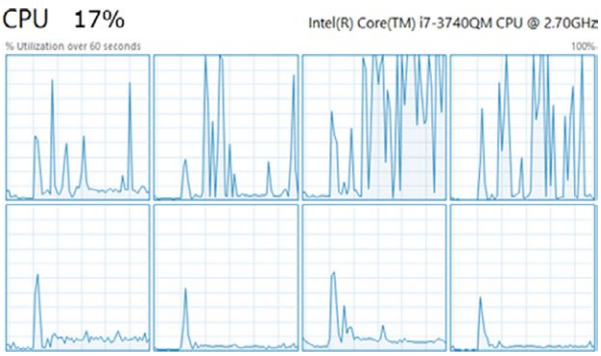


Figure 1.9 Windows Task Manager shows a program poorly utilizing CPU resources.

Such waste of computing power unequivocally illustrates that sequential code isn't the correct programming model for multicore processers. To maximize the use of the available computational resources, Microsoft's .NET platform provides parallel execution of code through multithreading. By using parallelism, a program can take full advantage of the resources available, as illustrated by the CPU performance counter in figure 1.10, where you'll notice that all the processor cores are running high, possibly at 100%. Current hardware trends predict more cores instead of faster clock speeds; therefore, developers have no choice but to embrace this evolution and become parallel programmers.

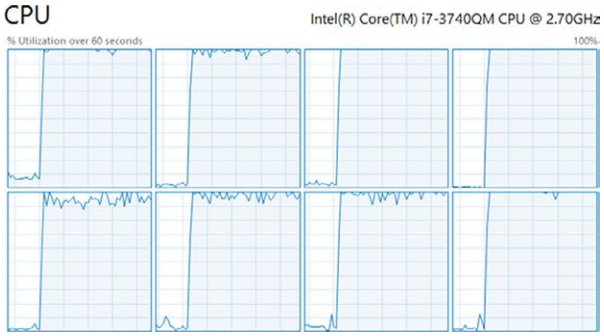


Figure 1.10 A program written with concurrency in mind can maximize CPU resources, possibly up to 100%.

1.3.1 Present and future of concurrent programming

Mastering concurrency to deliver scalable programs has become a required skill. Companies are interested in hiring and investing in engineers who have a deep knowledge of writing concurrent code. In fact, writing correct parallel computation can save time and money. It's cheaper to build scalable programs that use the computational resources available with fewer servers, than to keep buying and adding expensive hardware that is underused to reach the same level of performance. In addition, more hardware requires more maintenance and electric power to operate.

This is an exciting time to learn to write multithreaded code, and it's rewarding to improve the performance of your program with the functional programming (FP) approach. Functional programming is a programming style that treats computation as the evaluation of expressions and avoids changing-state and mutable data. Because immutability is the default, and with the addition of a fantastic composition and declarative programming style, FP makes it effortless to write concurrent programs. More details follow in section1.5.

While it's a bit unnerving to think in a new paradigm, the initial challenge of learning parallel programming diminishes quickly, and the reward for perseverance is infinite. You'll find something magical and spectacular about opening the Windows Task Manager and proudly noticing that the CPU usage spikes to 100% after your code changes. Once you become familiar and comfortable with writing highly scalable systems using the functional paradigm, it will be difficult to go back to the slow style of sequential code.

Concurrency is the next innovation that will dominate the computer industry, and it will transform how developers write software. The evolution of software requirements

in the industry and the demand for high-performance software that delivers great user experience through non-blocking UIs will continue to spur the need for concurrency. In lockstep with the direction of hardware, it's evident that concurrency and parallelism are the future of programming.

1.4 The pitfalls of concurrent programming

Concurrent and parallel programming are without doubt beneficial for rapid responsiveness and speedy execution of a given computation. But this gain of performance and reactive experience comes with a price. Using sequential programs, the execution of the code takes the happy path of predictability and determinism. Conversely, multithreaded programming requires commitment and effort to achieve correctness. Furthermore, reasoning about multiple executions running simultaneously is difficult because we're used to thinking sequentially.

Determinism

Determinism is a fundamental requirement in building software as computer programs are often expected to return identical results from one run to the next. But this property becomes hard to resolve in a parallel execution. External circumstances, such as the operating system scheduler or cache coherence (covered in chapter 4), could influence the execution timing and, therefore, the order of access for two or more threads and modify the same memory location. This time variant could affect the outcome of the program.

The process of developing parallel programs involves more than creating and spawning multiple threads. Writing programs that execute in parallel is demanding and requires thoughtful design. You should design with the following questions in mind:

- How is it possible to use concurrency and parallelism to reach incredible computational performance and a highly responsive application?
- How can such programs take full advantage of the power provided by a multicore computer?
- How can communication with and access to the same memory location between threads be coordinated while ensuring thread safety? (A method is called *thread-safe* when the data and state don't get corrupted if two or more threads attempt to access and modify the data or state at the same time.)
- How can a program ensure deterministic execution?
- How can the execution of a program be parallelized without jeopardizing the quality of the final result?

These aren't easy questions to answer. But certain patterns and techniques can help. For example, in the presence of side effects,[1] the determinism of the computation is lost because the order in which concurrent tasks execute becomes variable. The

[1] A side effect arises when a method changes some state from outside its scope, or it communicates with the "outside world," such as calling a database or writing to the file system.

obvious solution is to avoid side effects in favor of pure functions. You'll learn these techniques and practices during the course of the book.

1.4.1 Concurrency hazards

Writing concurrent programs isn't easy, and many sophisticated elements must be considered during program design. Creating new threads or queuing multiple jobs on the thread pool is relatively simple, but how do you ensure correctness in the program? When many threads continually access shared data, you must consider how to safeguard the data structure to guarantee its integrity. A thread should write and modify a memory location atomically,[2] without interference by other threads. The reality is that programs written in imperative programming languages or in languages with variables whose values can change (mutable variables) will always be vulnerable to data races, regardless of the level of memory synchronization or concurrent libraries used.

> **NOTE** A data race occurs when two or more threads in a single process access the same memory location concurrently, and at least one of the accesses updates the memory slot while other threads read the same value without using any exclusive locks to control their accesses to that memory.

Consider the case of two threads (Thread 1 and Thread 2) running in parallel, both trying to access and modify the shared value *x* as shown in figure 1.11. For Thread 1 to modify a variable requires more than one CPU instruction: the value must be read from memory, then modified and ultimately written back to memory. If Thread 2 tries to read from the same memory location while Thread 1 is writing back an updated value, the value of *x* changed. More precisely, it's possible that Thread 1 and Thread 2 read the value *x* simultaneously, then Thread 1 modifies the value *x* and writes it back to memory, while Thread 2 also modifies the value *x*. The result is data corruption. This phenomenon is called *race condition*.

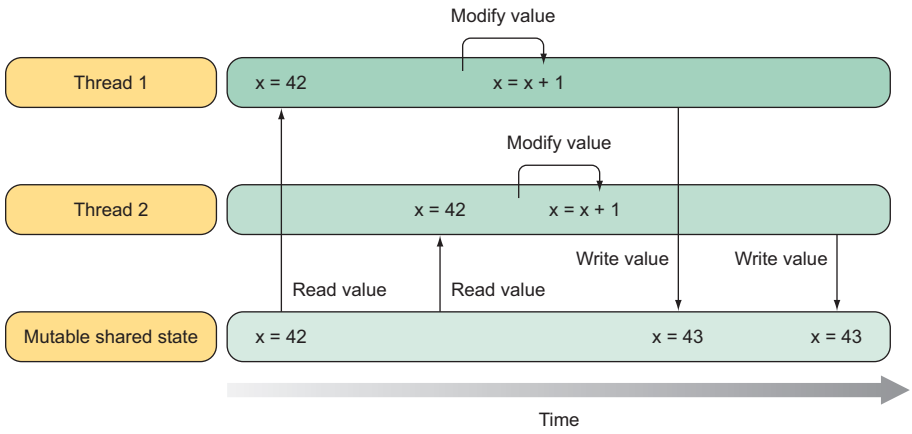


Figure 1.11 Two threads (Thread 1 and Thread 2) run in parallel, both trying to access and modify the shared value *x*. If Thread 2 tries to read from the same memory location while Thread 1 writes back an updated value, the value of *x* changes. This result is data corruption or *race condition*.

[2] An atomic operation accesses a shared memory and completes in a single step relative to other threads.