With examples in C# and F#

Concurrency in .NET

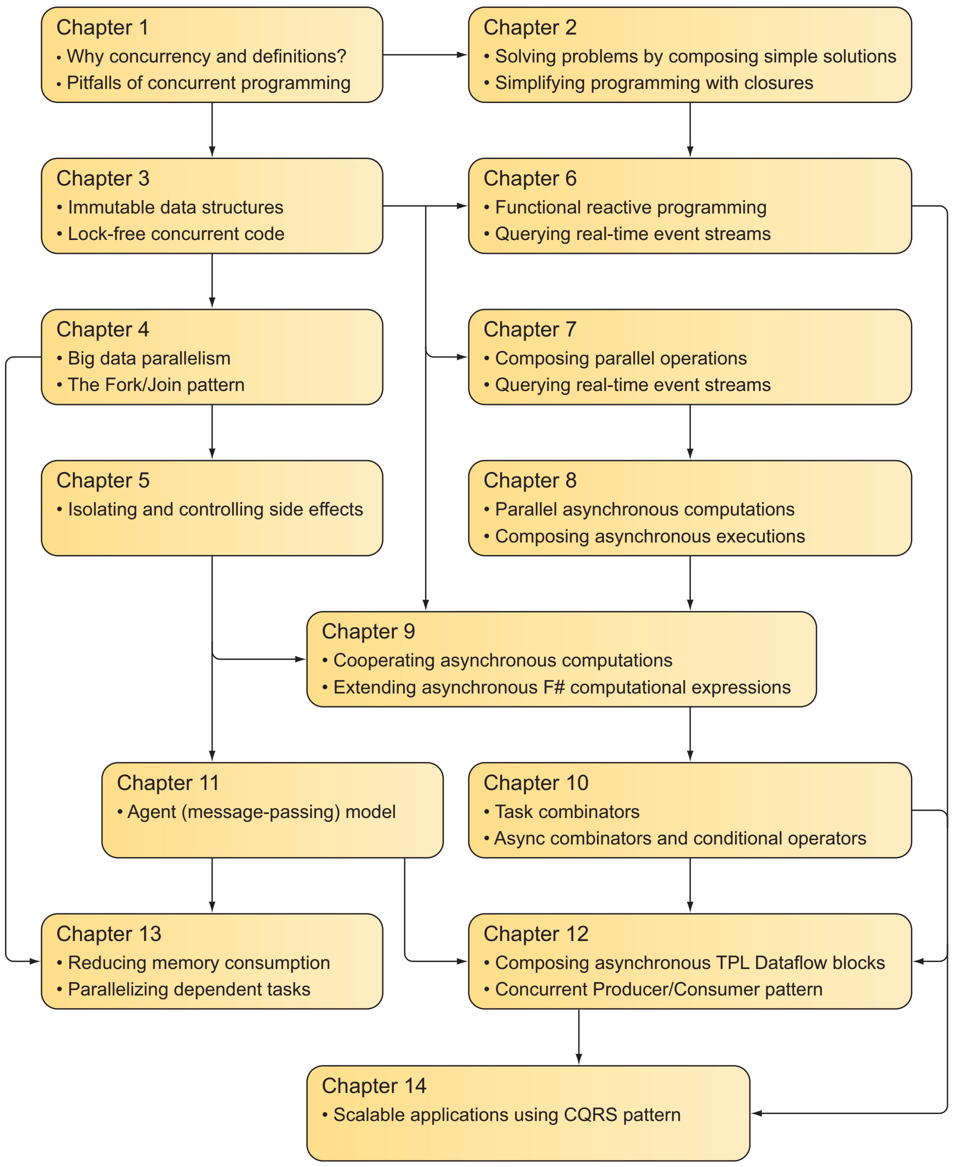
Modern patterns of concurrent and parallel programming

Riccardo Terrell

Sample Chapter

MANNING

Chapter dependency graph


Chapter 1
• Why concurrency and definitions?
• Pitfalls of concurrent programming
Chapter 2
• Solving problems by composing simple solutions
• Simplifying programming with closures
Chapter 3
• Immutable data structures
• Lock-free concurrent code
Chapter 6
• Functional reactive programming
• Querying real-time event streams
Chapter 4
• Big data parallelism
• The Fork/Join pattern
Chapter 7
• Composing parallel operations
• Querying real-time event streams
Chapter 5
• Isolating and controlling side effects
Chapter 8
• Parallel asynchronous computations
• Composing asynchronous executions
Chapter 9
• Cooperating asynchronous computations
• Extending asynchronous F# computational expressions
Chapter 11
• Agent (message-passing) model
Chapter 10
• Task combinators
• Async combinators and conditional operators
Chapter 13
• Reducing memory consumption
• Parallelizing dependent tasks
Chapter 12
• Composing asynchronous TPL Dataflow blocks
• Concurrent Producer/Consumer pattern
Chapter 14
• Scalable applications using CQRS pattern

Concurrency in .NET

Modern patterns of concurrent and parallel programming

by Riccardo Terrell

Chapter 1

brief contents

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.

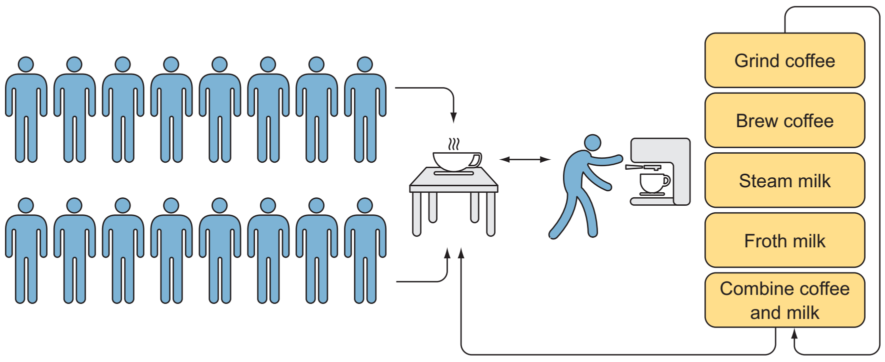


Figure 1.1 For each person in line, the barista is sequentially repeating the same set of instructions (grind coffee, brew coffee, steam milk, froth milk, and combine the coffee and the milk to make a cappuccino).

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.

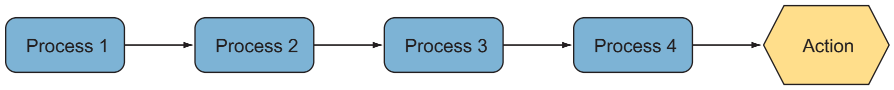


Figure 1.2 Typical sequential coding involving a consecutive, progressively ordered execution of processes

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.

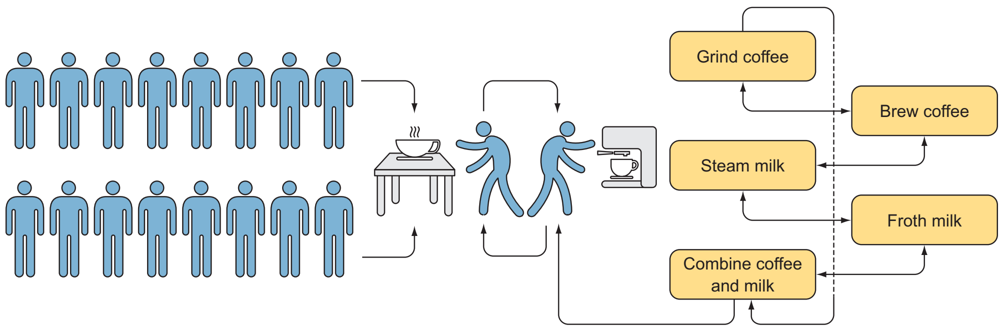


Figure 1.3 The barista switches between the operations (multitasking) of preparing the coffee (grind and brew) and preparing the milk (steam and froth). As a result, the barista executes segments of multiple tasks in an interleaved manner, giving the illusion of multitasking. But only one operation is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

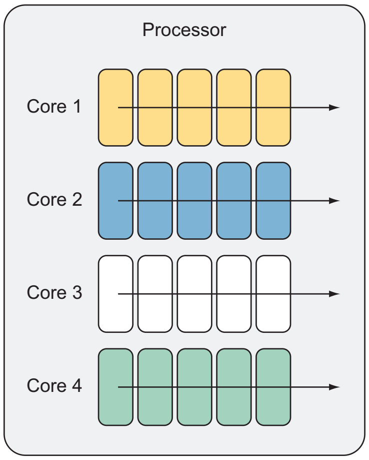


Figure 1.4 Only multicore machines allow parallelism for simultaneously executing different tasks. In this figure, each core is performing an independent task.

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.

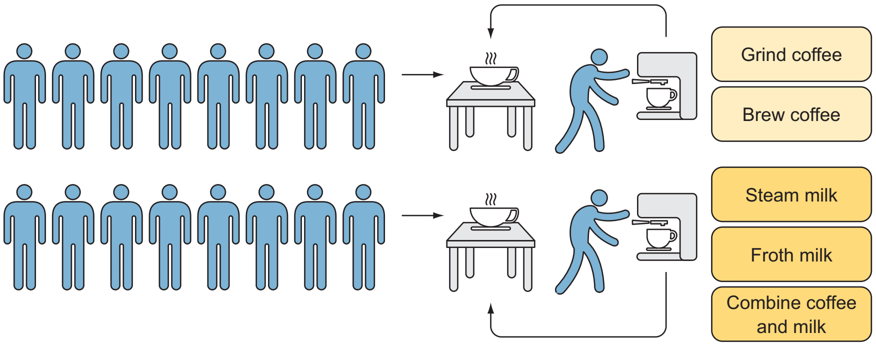


Figure 1.5 The production of cappuccinos is faster because two baristas can work in parallel with two coffee stations.

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).

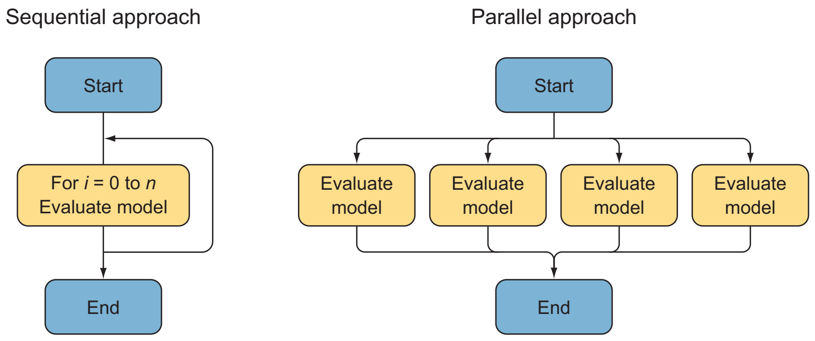


Figure 1.6 Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time.

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.

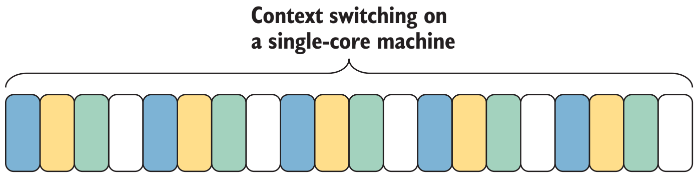


Figure 1.7 Each task has a different shade, indicating that the context switch in a single-core machine gives the illusion that multiple tasks run in parallel, but only one task is processed at a time.

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.

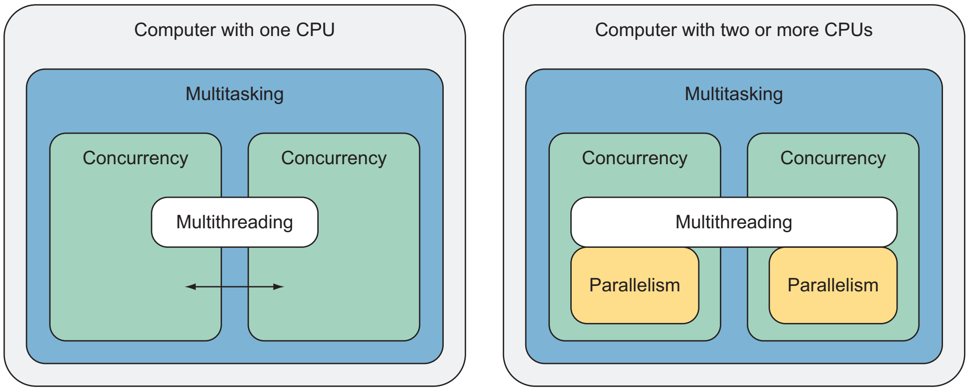


Figure 1.8 Relationship between concurrency, parallelism, multithreading, and multitasking in a single and a multicore device

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).

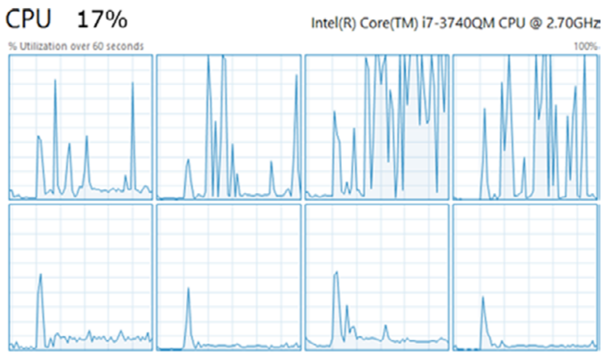

CPU 17%
Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
% Utilization over 60 seconds
100%

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.

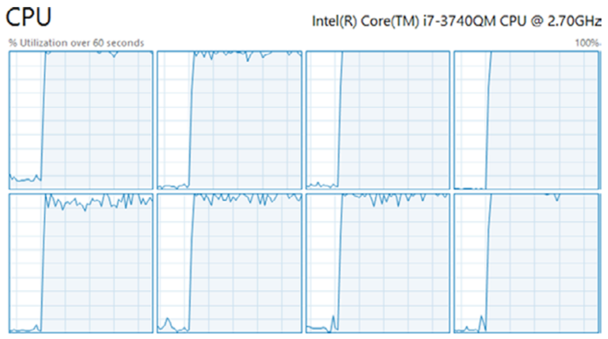

CPU
Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
% Utilization over 60 seconds
100%

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.*

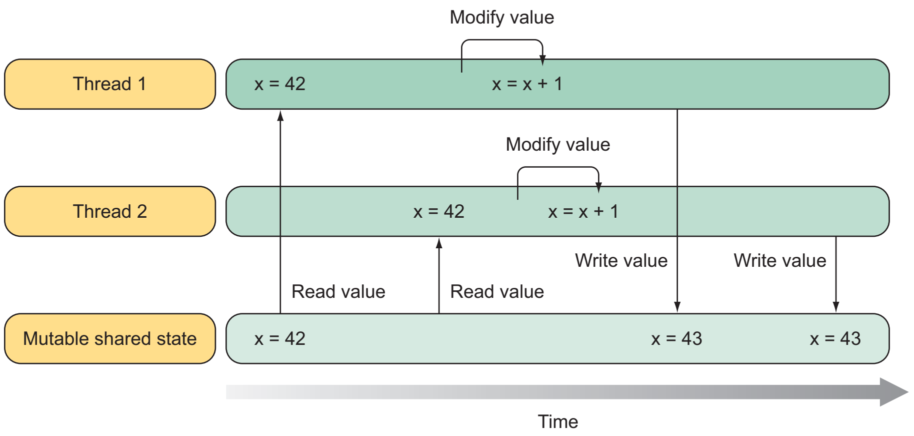


Figure 1.11 Two threads (Thread 1 and Thread 2) run in parallel, both trying to access and modify the shared value *x*. If Thread 2 tries to read from the same memory location while Thread 1 writes back an updated value, the value of *x* changes. This result is data corruption or *race condition*.

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

The combination of a mutable state and parallelism in a program is synonymous with problems. The solution from the imperative paradigm perspective is to protect the mutable state by locking access to more than one thread at a time. This technique is called *mutual exclusion* because the access of one thread to a given memory location prevents access of other threads at that time. The concept of timing is central as multiple threads must access the same data at the same time to benefit from this technique. The introduction of locks to synchronize access by multiple threads to shared resources solves the problem of data corruption, but introduces more complications that can lead to *deadlock*.

Consider the case in figure 1.12 where Thread 1 and Thread 2 are waiting for each other to complete work and are blocked indefinitely in that waiting. Thread 1 acquires Lock A, and, right after, Thread 2 acquires Lock B. At this point, both threads are waiting on a lock that will never be released. This is a case of deadlock.

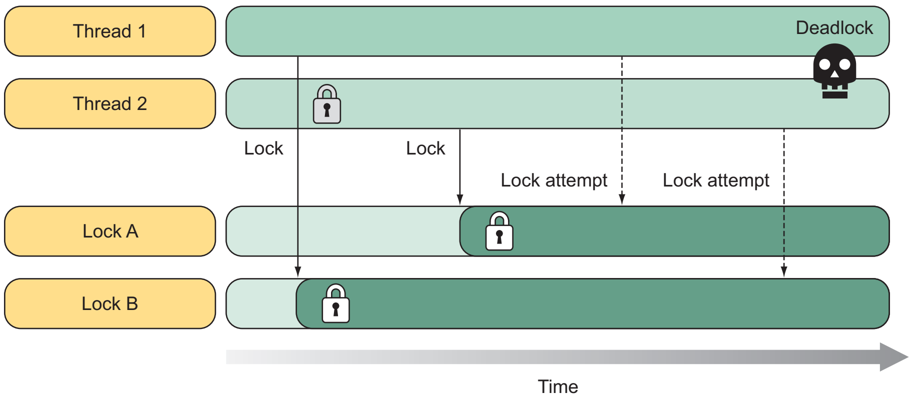


Figure 1.12. In this scenario, Thread 1 acquires Lock A, and Thread 2 acquires Lock B. Then, Thread 2 tries to acquire Lock A while Thread 1 tries to acquire Lock B that is already acquired by Thread 2, which is waiting to acquire Lock A before releasing Lock B. At this point, both threads are waiting at the lock that'll never be released. This is a case of deadlock.

Here is a list of concurrency hazards with a brief explanation. Later, you'll get more details on each, with a specific focus on how to avoid them:

- Race condition is a state that occurs when a shared mutable resource (a file, image, variable, or collection, for example) is accessed at the same time by multiple threads, leaving an inconsistent state. The consequent data corruption makes a program unreliable and unusable.
- Performance decline is a common problem when multiple threads share state contention that requires synchronization techniques. Mutual exclusion locks (or mutexes), as the name suggests, prevent the code from running in parallel by forcing multiple threads to stop work to communicate and synchronize memory access. The acquisition and release of locks comes with a performance penalty, slowing down all processes. As the number of cores gets larger, the cost of lock

contention can potentially increase. As more tasks are introduced to share the same data, the overhead associated with locks can negatively impact the computation. Section 1.4.3 demonstrates the consequences and overhead costs due to introducing lock synchronization.

- Deadlock is a concurrency problem that originates from using locks. It occurs when a cycle of tasks exists in which each task is blocked while waiting for another to proceed. Because all tasks are waiting for another task to do something, they're blocked indefinitely. The more that resources are shared among threads, the more locks are needed to avoid race condition, and the higher the risk of deadlocks.
- Lack of composition is a design problem originating from the introduction of locks in the code. Locks don't compose. Composition encourages problem dismantling by breaking up a complex problem into smaller pieces that are easier to solve, then gluing them back together. Composition is a fundamental tenet in FP.

1.4.2 The sharing of state evolution

Real-world programs require interaction between tasks, such as exchanging information to coordinate work. This cannot be implemented without sharing data that's accessible to all the tasks. Dealing with this shared state is the root of most problems related to parallel programming, unless the shared data is immutable or each task has its own copy. The solution is to safeguard all the code from those concurrency problems. No compiler or tool can help you position these primitive synchronization locks in the correct location in your code. It all depends on your skill as a programmer.

Because of these potential problems, the programming community has cried out, and in response, libraries and frameworks have been written and introduced into mainstream object-oriented languages (such as C# and Java) to provide concurrency safeguards, which were not part of the original language design. This support is a design correction, illustrated with the presence of shared memory in imperative and object-oriented, general-purpose programming environments. Meanwhile, functional languages don't need safeguards because the concept of FP maps well onto concurrent programming models.

1.4.3 A simple real-world example: parallel quicksort

Sorting algorithms are used generally in technical computing and can be a bottleneck. Let's consider a Quicksort algorithm,[3] a CPU-bound computation amenable to parallelization that orders the elements of an array. This example aims to demonstrate the pitfalls of converting a sequential algorithm into a parallel version and points out that introducing

[3] Tony Hoare invented the Quicksort algorithm in 1960, and it remains one of the most acclaimed algorithms with great practical value.

parallelism in your code requires extra thinking before making any decisions. Otherwise, performance could potentially have an opposite outcome to that expected.

Quicksort is a Divide and Conquer algorithm; it first divides a large array into two smaller sub-arrays of low elements and high elements. Quicksort can then recursively sort the sub-arrays, and is amenable to parallelization. It can operate in place on an array, requiring small additional amounts of memory to perform the sorting. The algorithm consists of three simple steps, as shown in figure 1.13:

1. Select a pivot element.
2. Partition the sequence into subsequences according to their order relative to the pivot.
3. Quicksort the subsequences.

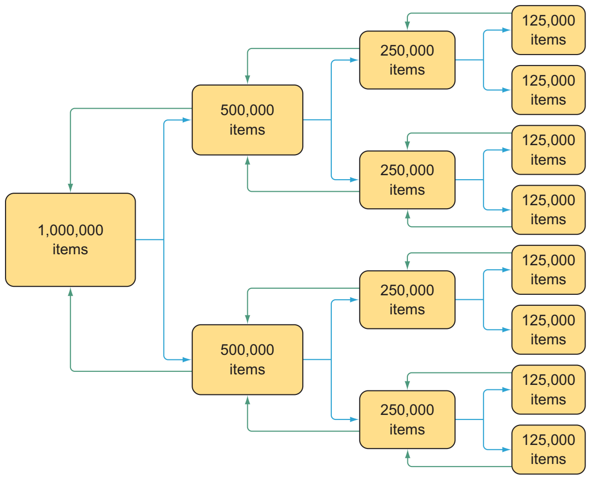


Figure 1.13. The recursive function divides and conquers. Each block is divided into equal halves, where the pivot element must be the median of the sequence, until each portion of code can be executed independently. When all the single blocks are completed, they send the result back to the previous caller to be aggregated. Quicksort is based on the idea of picking a pivot point and partitioning the sequence into sub-sequence elements smaller than the pivot and bigger than the pivot elements before recursively sorting the two smaller sequences.

Recursive algorithms, especially ones based on a form of Divide and Conquer, are a great candidate for parallelization and CPU-bound computations.

The Microsoft Task Parallel Library (TPL), introduced after the release of .NET 4.0, makes it easier to implement and exploit parallelism for this type of algorithm. Using the TPL, you can divide each step of the algorithm and perform each task in parallel, recursively. It's a straight and easy implementation, but you must be careful of the level of depth to which the threads are created to avoid adding more tasks than necessary.

To implement the Quicksort algorithm, you'll use the FP language F#. Due to its intrinsic recursive nature, however, the idea behind this implementation can also be applied to C#, which requires an imperative `for` loop approach with a mutable state. C# doesn't support optimized tail-recursive functions such as F#, so a hazard exists of raising a stack overflow exception when the call-stack pointer exceeds the stack constraint. In chapter 3, we'll go into detail on how to overcome this C# limitation.

Listing 1.1 shows a Quicksort function in F# that adopts the Divide and Conquer strategy. For each recursive iteration, you select a pivot point and use that to partition the total array. You partition the elements around the pivot point using the `List.partition` API, then recursively sort the lists on each side of the pivot. F# has great built-in support for data structure manipulation. In this case, you're using the `List.partition` API, which returns a tuple containing two lists: one that satisfies the predicate and another that doesn't.

Listing 1.1 Simple Quicksort algorithm

```
let rec quicksortSequential aList =
    match aList with
    | [] -> []
    | firstElement :: restOfList ->
        let smaller, larger =
            List.partition (fun number -> number < firstElement) restOfList
        quicksortSequential smaller @ (firstElement ::
➥ quicksortSequential larger)
```

Running this Quicksort algorithm against an array of 1 million random, unsorted integers on my system (eight logical cores; 2.2 GHz clock speed) takes an average of 6.5 seconds. But when you analyze this algorithm design, the opportunity to parallelize is evident. At the end of `quicksortSequential`, you recursively call into `quicksortSequential` with each partition of the array identified by the `(fun number -> number < firstElement) restOfList`. By spawning new tasks using the TPL, you can rewrite in parallel this portion of the code.

Listing 1.2 Parallel Quicksort algorithm using the TPL

```
let rec quicksortParallel aList =
    match aList with
    | [] -> []
    | firstElement :: restOfList ->
        let smaller, larger =
            List.partition (fun number -> number < firstElement) restOfList
        let left  = Task.Run(fun () -> quicksortParallel smaller)
        let right = Task.Run(fun () -> quicksortParallel larger)
        left.Result @ (firstElement :: right.Result)
```

Appends the result for each task into a sorted array

Task.Run executes the recursive calls in tasks that can run in parallel; for each recursive call, tasks are dynamically created.

The algorithm in listing 1.2 is running in parallel, which now is using more CPU resources by spreading the work across all available cores. But even with improved resource utilization, the overall performance result isn't meeting expectations.

Execution time dramatically increases instead of decreases. The parallelized Quicksort algorithm is passed from an average of 6.5 seconds per run to approximately 12 seconds. The overall processing time has slowed down. In this case, the problem is that the algorithm is *over-parallelized*. Each time the internal array is partitioned, two new tasks are spawned to parallelize this algorithm. This design is spawning too many tasks in relation to the cores available, which is inducing parallelization overhead. This is especially true in a Divide and Conquer algorithm that involves parallelizing a recursive function. It's important that you don't add more tasks than necessary. The disappointing result demonstrates an important characteristic of parallelism: inherent limitations exist on how much extra threading or extra processing will help a specific algorithmic implementation.

To achieve better optimization, you can refactor the previous `quicksortParallel` function by stopping the recursive parallelization after a certain point. In this way, the algorithm's first recursions will still be executed in parallel until the deepest recursion, which will revert to the serial approach. This design guarantees taking full advantage of cores. Plus, the overhead added by parallelizing is dramatically reduced.

Listing 1.3 shows this new design approach. It takes into account the level where the recursive function is running; if the level is below a predefined threshold, it stops parallelizing. The function `quicksortParallelWithDepth` has an extra argument, `depth`, whose purpose is to reduce and control the number of times a recursive function is parallelized. The `depth` argument is decremented on each recursive call, and new tasks are created until this argument value reaches zero. In this case, you're passing the value resulting from `Math.Log(float System.Enviroment.ProcessorCount, 2.) + 4.` for the `max depth`. This ensures that every level of the recursion will spawn two child tasks until all the available cores are enlisted.

Listing 1.3 A better parallel Quicksort algorithm using the TPL

```
let rec quicksortParallelWithDepth depth aList =
    match aList with
    | [] -> []
    | firstElement :: restOfList ->
      let smaller, larger =
          List.partition (fun number -> number < firstElement) restOfList
      if depth < 0 then
          let left  = quicksortParallelWithDepth depth smaller
          let right = quicksortParallelWithDepth depth larger
          left @ (firstElement :: right)
      else
          let left  = Task.Run(fun () ->
➥ quicksortParallelWithDepth (depth - 1) smaller)
          let right = Task.Run(fun () ->
➥ quicksortParallelWithDepth (depth - 1) larger)
          left.Result @ (firstElement :: right.Result)
```

Tracks the function recursion level with the depth parameter

If the value of depth is negative, skips the parallelization

If the value of depth is positive, allows the function to be called recursively, spawning two new tasks

Sequentially executes the Quicksort using the current thread

One relevant factor in selecting the number of tasks is how similar the predicted run time of the tasks will be. In the case of `quicksortParallelWithDepth`, the duration of the tasks can vary substantially, because the pivot points depend on the unsorted data. They don't necessarily result in segments of equal size. To compensate for the uneven sizes of the tasks, the formula in this example calculates the `depth` argument to produce more tasks than cores. The formula limits the number of tasks to approximately 16 times the number of cores because the number of tasks can be no larger than `2 ^ depth`. Our objective is to have a Quicksort workload that is balanced, and that doesn't start more tasks than required. Starting a `Task` during each iteration (recursion), when the depth level is reached, saturates the processors.

In most cases, the Quicksort generates an unbalanced workload because the fragments produced are not of equal size. The conceptual formula `log2(ProcessorCount) + 4` calculates the `depth` argument to limit and adapt the number of running tasks regardless of the cases.[4] If you substitute `depth = log2(ProcessorCount) + 4` and simplify the expression, you see that the number of tasks is 16 times `ProcessorCount`. Limiting the number of subtasks by measuring the recursion depth is an extremely important technique.[5]

For example, in the case of four-core machines, the depth is calculated as follows:

```
depth = log2(ProcessorCount) + 4
depth = log2(2) + 4
depth = 2 + 4
```

The result is a range between approximately 36 to 64 concurrent tasks, because during each iteration two tasks are started for each branch, which in turn double in each iteration. In this way, the overall work of partitioning among threads has a fair and suitable distribution for each core.

1.4.4 Benchmarking in F#

You executed the Quicksort sample using the F# REPL (Read-Evaluate-Print-Loop), which is a handy tool to run a targeted portion of code because it skips the compilation step of the program. The REPL fits quite well in prototyping and data-analysis development because it facilitates the programming process. Another benefit is the built-in `#time` functionality, which toggles the display of performance information. When it's enabled, F# `Interactive` measures real time, CPU time, and garbage collection information for each section of code that's interpreted and executed.

Table 1.1 sorts a 3 GB array, enabling the 64-bit environment flag to avoid size restriction. It's run on a computer with eight logical cores (four physical cores with hyper-threading). On an average of 10 runs, table 1.1 shows the execution times in seconds.

[4] The function log2 is an abbreviation for *Log in base 2*. For example, log2(x) represents the logarithm of x to the base 2.

[5] Recall that for any value a, $2 ^\wedge (a + 4)$ is the same as $16 \times 2^\wedge a$; and that if $a = \log2(b)$, $2^\wedge a = b$.

Table 1.1 Benchmark of sorting with Quicksort

Serial	Parallel	Parallel 4 threads	Parallel 8 threads
6.52	12.41	4.76	3.50

It's important to mention that for a small array, fewer than 100 items, the parallel sort algorithms are slower than the serial version due to the overhead of creating and/or spawning new threads. Even if you correctly write a parallel program, the overhead introduced with concurrency constructors could overwhelm the program runtime, delivering the opposite expectation by decreasing performance. For this reason, it's important to benchmark the original sequential code as a baseline and then continue to measure each change to validate whether parallelism is beneficial. A complete strategy should consider this factor and approach parallelism only if the array size is greater than a threshold (recursive depth), which usually matches the number of cores, after which it defaults back to the serial behavior.

1.5 Why choose functional programming for concurrency?

The trouble is that essentially all the interesting applications of concurrency involve the deliberate and controlled mutation of shared state, such as screen real estate, the file system, or the internal data structures of the program. The right solution, therefore, is to provide mechanisms which allow the safe mutation of shared state section.

—Peyton Jones, Andrew Gordon, and Sigbjorn Finne ("Concurrent Haskell," Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, *St. Petersburg Beach, FL, January 1996)*

FP is about minimizing and controlling side effects, commonly referred to as *pure functional programming*. FP uses the concept of transformation, where a function creates a copy of a value *x* and then modifies the copy, leaving the original value *x* unchanged and free to be used by other parts of the program. It encourages considering whether mutability and side effects are necessary when designing the program. FP allows mutability and side effects, but in a strategic and explicit manner, isolating this area from the rest of the code by utilizing methods to encapsulate them.

The main reason for adopting functional paradigms is to solve the problems that exist in the multicore era. Highly concurrent applications, such as web servers and data-analysis databases, suffer from several architectural issues. These systems must be scalable to respond to a large number of concurrent requests, which leads to design challenges for handling maximum resource contention and high-scheduling frequency. Moreover, race conditions and deadlocks are common, which makes troubleshooting and debugging code difficult.

In this chapter, we discussed a number of common issues specific to developing concurrent applications in either imperative or OOP. In these programming paradigms, we're dealing with objects as a base construct. Conversely, in terms of concurrency, dealing with objects has caveats to consider when passing from a single-thread program to a massively parallelizing work, which is a challenging and entirely different scenario.

NOTE A *thread* is an operating system construct that functions like a virtual CPU. At any given moment, a thread is allowed to run on the physical CPU for a slice of time. When the time for a thread to run expires, it's swapped off the CPU for another thread. Therefore, if a single thread enters an infinite loop, it cannot monopolize all the CPU time on the system. At the end of its time slice, it will be switched out for another thread.

The traditional solution for these problems is to synchronize access to resources, avoiding contention between threads. But this same solution is a double-edged sword because using primitives for synchronization, such as `lock` for mutual exclusion, leads to possible deadlock or race conditions. In fact, the state of a variable (as the name *variable* implies) can mutate. In OOP, a variable usually represents an object that's liable to change over time. Because of this, you can never rely on its state and, consequentially, you must check its current value to avoid unwanted behaviors (figure 1.14).

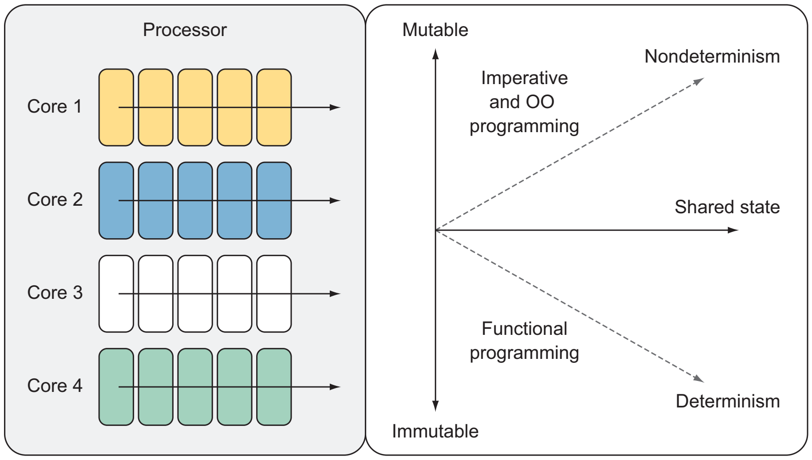


Figure 1.14 In the functional paradigm, due to immutability as a default construct, concurrent programming guarantees deterministic execution, even in the case of a shared state. Conversely, imperative and OOP use mutable states, which are hard to manage in a multithread environment, and this leads to nondeterministic programs.

It's important to consider that components of systems that embrace the FP concept can no longer interfere with each other, and they can be used in a multithreaded environment without using any locking strategies.

Development of safe parallel programs using a share of mutable variables and side-effect functions takes substantial effort from the programmer, who must make critical decisions, often leading to synchronization in the form of locking. By removing those fundamental problems through functional programming, you can also remove those concurrency-specific issues. This is why FP makes an excellent concurrent programming model. It is an exceptional fit for concurrent programmers to achieve correct high performance in highly multithreaded environments using simple code. At the heart of FP, neither variables nor state are mutable and cannot be shared, and functions may not have side effects.

FP is the most practical way to write concurrent programs. Trying to write them in imperative languages isn't only difficult, it also leads to bugs that are difficult to discover, reproduce, and fix.

How are you going to take advantage of every computer core available to you? The answer is simple: embrace the functional paradigm!

1.5.1 Benefits of functional programming

There are real advantages to learning FP, even if you have no plans to adopt this style in the immediate future. Still, it's hard to convince someone to spend their time on something new without showing immediate benefits. The benefits come in the form of idiomatic language features that can initially seem overwhelming. FP, however, is a paradigm that will give you great coding power and positive impact in your programs after a short learning curve. Within a few weeks of using FP techniques, you'll improve the readability and correctness of your applications.

The benefits of FP (with focus on concurrency) include the following:

- *Immutability*—A property that prevents modification of an object state after creation. In FP, variable assignment is not a concept. Once a value has been associated with an identifier, it cannot change. Functional code is immutable by definition. Immutable objects can be safely transferred between threads, leading to great optimization opportunities. Immutability removes the problems of memory corruption (race condition) and deadlocks because of the absence of mutual exclusion.
- *Pure function*—This has no side effects, which means that functions don't change any input or data of any type outside the function body. Functions are said to be pure if they're transparent to the user, and their return value depends only on the input arguments. By passing the same arguments into a pure function, the result won't change, and each process will return the same value, producing consistent and expected behavior.
- *Referential transparency*—The idea of a function whose output depends on and maps only to its input. In other words, each time a function receives the same arguments, the result is the same. This concept is valuable in concurrent programming because the definition of the expression can be replaced with its value and will have the same meaning. Referential transparency guarantees that a set of functions can be evaluated in any order and in parallel, without changing the application's behavior.
- *Lazy evaluation*—Used in FP to retrieve the result of a function on demand or to defer the analysis of a big data stream until needed.
- *Composability*—Used to compose functions and create higher-level abstractions out of simple functions. Composability is the most powerful tool to defeat complexity, letting you define and build solutions for complex problems.

Learning to program functionally allows you to write more modular, expression-oriented, and conceptually simple code. The combinations of these FP assets will let you understand what your code is doing, regardless of how many threads the code is executing.

Later in this book, you'll learn techniques to apply parallelism and bypass issues associated with mutable states and side effects. The functional paradigm approach to these concepts aims to simplify and maximize efficiency in coding with a declarative programming style.

1.6 Embracing the functional paradigm

Sometimes, change is difficult. Often, developers who are comfortable in their domain knowledge lack the motivation to look at programming problems from a different perspective. Learning any new program paradigm is hard and requires time to transition to developing in a different style. Changing your programming perspective requires a switch in your thinking and approach, not solely learning new code syntax for a new programming language.

Going from a language such as Java to C# isn't difficult; in terms of concepts, they're the same. Going from an imperative paradigm to a functional paradigm is a far more difficult challenge. Core concepts are replaced. You have no more state. You have no more variables. You have no more side effects.

But the effort you make to change paradigms will pay large dividends. Most developers will agree that learning a new language makes you a better developer, and liken that to a patient whose doctor prescribes 30 minutes of exercise per day to be healthy. The patient knows the real benefits in exercise, but is also aware that daily exercise implies commitment and sacrifice.

Similarly, learning a new paradigm isn't hard, but does require dedication, engagement, and time. I encourage everyone who wants to be a better programmer to consider learning the FP paradigm. Learning FP is like riding a roller coaster: during the process there will be times when you feel excited and levitated, followed by times when you believe that you understand a principle only to descend steeply—screaming—but the ride is worth it. Think of learning FP as a journey, an investment in your personal and professional career with guaranteed return. Keep in mind that part of the learning is to make mistakes and develop skills to avoid those in the future.

Throughout this process, you should identify the concepts that are difficult to understand and try to overcome those difficulties. Think about how to use these abstractions in practice, solving simple problems to begin with. My experience shows that you can break through a mental roadblock by finding out what the intent of a concept is by using a real example. This book will walk you through the benefits of FP applied to concurrency and a distributed system. It's a narrow path, but on the other side, you'll emerge with several great foundational concepts to use in your everyday programming. I am confident you'll gain new insights into how to solve complex problems and become a superior software engineer using the immense power of FP.

1.7 Why use F# and C# for functional concurrent programming?

The focus of this book is to develop and design highly scalable and performant systems, adopting the functional paradigm to write correct concurrent code. This doesn't mean you must learn a new language; you can apply the functional paradigm by using tools that you're already familiar with, such as the multipurpose languages C# and F#. Over the years several functional features have been added to those languages, making it easier for you to shift to incorporating this new paradigm.

The intrinsically different approach to solving problems is the reason these languages were chosen. Both programming languages can be used to solve the same problem in very different ways, which makes a case for choosing the best tool for the job. With a well-rounded toolset, you can design a better and easier solution. In fact, as software engineers, you *should* think of programming languages as tools.

Ideally, a solution should be a combination of C# and F# projects that work together cohesively. Both languages cover a different programming model, but the option to choose which tool to use for the job provides an enormous benefit in terms of productivity and efficiency. Another aspect to selecting these languages is their different concurrent programming model support, which can be mixed. For instance:

- F# offers a much simpler model than C# for asynchronous computation, called *asynchronous workflows*.
- Both C# and F# are strongly typed, multipurpose programming languages with support for multiple paradigms that encompass functional, imperative, and OOP techniques.
- Both languages are part of the .NET ecosystem and derive a rich set of libraries that can be used equally by both languages.
- F# is a functional-first programming language that provides an enormous productivity boost. In fact, programs written in F# tend to be more succinct and lead to less code to maintain.
- F# combines the benefits of a functional declarative programming style with support from the imperative object-oriented style. This lets you develop applications using your existing object-oriented and imperative programming skills.
- F# has a set of built-in lock-free data structures, due to default immutable constructors. An example is the discriminated union and the record types. These types have structural equality and don't allow `nulls` that lead to "trusting" the integrity of the data and easier comparisons.
- F#, different from C#, strongly discourages the use of `null` values, also known as the billion-dollar mistake, and, instead, encourages the use of immutable data structures. This lack of null reference helps to minimize the number of bugs in programming.

The null reference origin

Tony Hoare introduced the null reference in 1965, while he was designing the ALGOL object-oriented language. Some 44 years later, he apologized for inventing it by calling it the billion-dollar mistake. He also said this:

". . . I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes . . .." [6]

- F# is naturally parallelizable because it uses immutably as a default type constructor, and because of its .NET foundation, it integrates with the C# language with state-of-the-art capability at the implementation level.
- C# design tends toward an imperative language, first with full support for OOP. (I like to define this as imperative OO.) The functional paradigm, during the past years and since the release of .NET 3.5, has influenced the C# language with the addition of features like lambda expressions and LINQ for list comprehension.
- C# also has great concurrency tools that let you easily write parallel programs and readily solve tough real-world problems. Indeed, exceptional multicore development support within the C# language is versatile, and capable of rapid development and prototyping of highly parallel symmetric multiprocessing (SMP) applications. These programming languages are great tools for writing concurrent software, and the power and options for workable solutions aggregate when used in coexistence. SMP is the processing of programs by multiple processors that share a common operating system and memory.
- F# and C# can interoperate. In fact, an F# function can call a method in a C# library, and vice versa.

In the coming chapters, we'll discuss alternative concurrent approaches, such as data parallelism, asynchronous, and the message-passing programming model. We'll build libraries using the best tools that each of these programming languages can offer and compare those with other languages. We'll also examine tools and libraries like the TPL and Reactive Extensions (Rx) that have been successfully designed, inspired, and implemented by adopting the functional paradigm to obtain composable abstraction.

It's obvious that the industry is looking for a reliable and simple concurrent programming model, shown by the fact that software companies are investing in libraries that remove the level of abstraction from the traditional and complex memory-synchronization models. Examples of these higher-level libraries are Intel's Threading Building Blocks (TBB) and Microsoft's TPL.

There are also interesting open source projects, such as OpenMP (which provides pragmas [compiler-specific definitions that you can use to create new preprocessor

[6] From a speech at QCon London in 2009: http://mng.bz/u74T.

functionality or to send implementation-defined information to the compiler] that you can insert into a program to make parts of it parallel) and OpenCL (a low-level language to communicate with Graphic Processing Units [GPUs]). GPU programming has traction and has been sanctioned by Microsoft with C++ AMP extensions and Accelerator .NET.

Summary

- No silver bullet exists for the challenges and complexities of concurrent and parallel programming. As a professional engineer, you need different types of ammunition, and you need to know how and when to use them to hit the target.
- Programs must be designed with concurrency in mind; programmers cannot continue writing sequential code, turning a blind eye to the benefits of parallel programming.
- Moore's Law isn't incorrect. Instead, it has changed direction toward an increased number of cores per processor rather than increased speed for a single CPU.
- While writing concurrent code, you must keep in mind the distinction between concurrency, multithreading, multitasking, and parallelism.
- The share of mutable states and side effects are the primary concerns to avoid in a concurrent environment because they lead to unwanted program behaviors and bugs.
- To avoid the pitfalls of writing concurrent applications, you should use programming models and tools that raise the level of abstraction.
- The functional paradigm gives you the right tools and principles to handle concurrency easily and correctly in your code.
- Functional programming excels in parallel computation because immutability is the default, making it simpler to reason about the share of data.

Glossary

Asynchronicity—When a program performs requests that don't complete immediately but that are fulfilled later, and where the program issuing the request must do meaningful work in the meantime.

Concurrency—The notion of multiple things happening at the same time. Usually, concurrent programs have multiple threads of execution, each typically executing different code.

Parallelism—The state of a program when more than one thread runs simultaneously to speed up the program's execution.

Process—A standard operating system process. Each instance of the .NET CLR runs in its own process. Processes are typically independent.

Thread—The smallest sequence of programmed instructions that the OS can manage independently. Each .NET process has many threads running within the one process and sharing the same heap.

Selecting the right concurrent pattern

Application characteristic	Concurrent pattern
You have a sequential loop where each iteration runs an independent operation.	Use the Parallel Loop pattern to run autonomous operations simultaneously (chapter 3).
You write an algorithm that divides the problem domain dynamically at runtime.	Use dynamic task parallelism, which uses a Divide and Conquer technique to spawn new tasks on demand (chapter 4).
You have to parallelize the execution of a distinct set of operations without dependencies and aggregate the result.	Use the Fork/Join pattern to run in parallel a set of tasks that permit you to reduce the results of all the operations when completed (chapter 4).
You need to parallelize the execution of a distinct set of operations where order of execution depends on dataflow constraints.	Use the Task Graph pattern to make the dataflow dependencies between tasks clear (chapter 13).
You have to analyze and accumulate a result for a large data set by performing operations such as filtering, grouping, and aggregating.	Use the MapReduce pattern to parallelize the processing in a different and independent step of a massive volume of data in a timely manner (chapter 5).
You need to aggregate a large data set by applying a common operation.	Use the Parallel Aggregation, or Reducer, pattern to merge partial results (chapter 5).
You implement a program that repetitively performs a series of independent operations connected as a chain.	Use the Pipeline pattern to run in parallel a set of operations that are connected by queues, preserving the order of inputs (chapters 7 and 12).
You have multiple processes running independently for which work must be synchronized.	Use the Producer/Consumer pattern to safely share a common buffer. This buffer is used by the producer to queue the generated data in a thread-safe manner; the data is then picked up by the consumer to perform some operation (chapters 8 and 13).

.NET PROGRAMMING

Concurrency in .NET

Riccardo Terrell

Unlock the incredible performance built into your multiprocessor machines. Concurrent applications run faster because they spread work across processor cores, performing several tasks at the same time. Modern tools and techniques on the .NET platform, including parallel LINQ, functional programming, asynchronous programming, and the Task Parallel Library, offer powerful alternatives to traditional thread-based concurrency.

Concurrency in .NET teaches you to write code that delivers the speed you need for performance-sensitive applications. Featuring examples in both C# and F#, this book guides you through concurrent and parallel designs that emphasize functional programming in theory and practice. You'll start with the foundations of concurrency and master essential techniques and design practices to optimize code running on modern multiprocessor systems.

What's Inside

- The most important concurrency abstractions
- Employing the agent programming model
- Implementing real-time event-stream processing
- Executing unbounded asynchronous operations
- Best concurrent practices and patterns that apply to all platforms

For readers skilled with C# or F#.

Riccardo Terrell is a seasoned .NET software engineer, senior software architect, and Microsoft MVP who is passionate about functional programming.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/concurrency-in-dot-net

Free eBook
See first page

"A complementary source of knowledge about modern concurrent functional programming on the .NET platform—an absolute must-read."
—Pawel Klimczyk, Microsoft MVP

"Not just for those cutting code on Windows. You can use the gold dust in this book on any platform!"
—Kevin Orr, Sumus Solutions

"Presents real-world problems and offers different kinds of concurrency to solve them."
—Andy Kirsch, Rally Health

"Easiest entry into concurrency I've come across so far!"
—Anton Herzog
AFMG Technologies

ISBN-13: 978-1-61729-299-6
ISBN-10: 1-61729-299-0
55999
9 781617 292996

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]