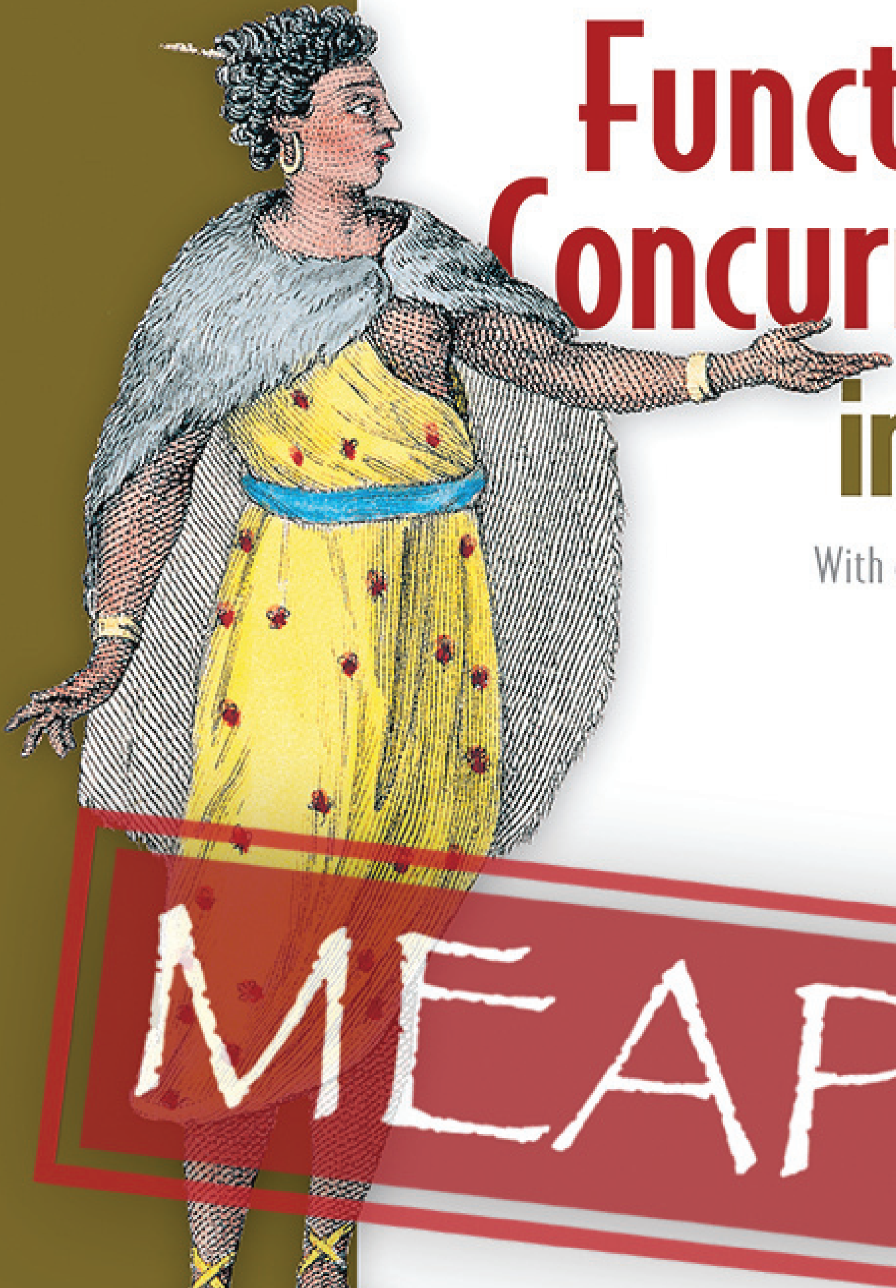


Functional Concurrency in .NET

With examples in C# and F#

Riccardo Terrell

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Functional Concurrency in .NET

With Examples in C# and F#
Version 4

welcome

Thank you for purchasing this MEAP edition of *Functional Concurrency in .NET*. This book aims to give you rapid access to a set of programming skills for writing multithreaded code using the functional programming paradigm. The book is in development – due for release in 2017 - and I am extremely excited to continue authoring it with your help for a successful final manuscript. Don't forget to visit https://www.manning.com/books/functional-concurrency-in-dotnet and give me your feedback!

This is an intermediate level book aimed at developers, architects and passionate computer programmers who are interested in writing code with improved speed and effectiveness by adopting a declarative and pain-free programming style. Additionally, by embracing the techniques explained in this book, your programs will continue to improve their speed and performance as the hardware evolves, making it the enduring solution for the future.

I have been involved in concurrent programming for more than a decade, and am passionate about sharing the benefits of functional programming with you. From the first time I started to apply the functional paradigm to programming languages C# and F#, I realized that this knowledge was a force multiplier and made writing concurrent programs fun. Still today, I am amazed by the possibilities and look forward to growing in learning with you…

This book is divided into three main sections, which will build on one another.

The first part of the book will cover the foundations of concurrency. There will be discussion about why conventional object-oriented programming is not effective in multithreaded programs and how functional programming is instead the right choice. This part of the book includes functional techniques and paradigms that will be used in the follow on sections of the book.

The second part is divided into concurrent and parallel programming designs, emphasizing the functional paradigm. This part covers both theory and practice with lots of code samples. The programs are written in both C# and F#, sometimes demonstrating the interpolation between the two languages to illustrate how to best choose the right tool for the job.

The third part of the book covers a real "cradle to grave" application implementation, covering the techniques and skills learned during the book. Different scenarios will be discussed, demonstrating alternate approaches and how to choose the right concurrent programming model. The application will have a dedicated chapter with the steps for developing a highly scalable server side program and reactive client side, which in this case is a mobile application.

Concurrency is present, and here to stay. Technology has brought us to the edge of our current capabilities and it will be up to "us", the developers to take our programs beyond. Through the exchange of ideas, we can continue to grow and sharpen our skills. I encourage you to share your insight and feedback on this version of the book to ensure it is a worthwhile and enjoyable product for all.

Thanks for your interest and for purchasing this book.

—Riccardo A. Terrell

brief contents

1 Functional Concurrency Foundations

This chapter covers

- **Why we need concurrency**
- **The difference between concurrency, parallelism, and multithreading The most common pitfalls when writing concurrent applications**
- **The problem of sharing variables between threads**
- **The benefits of 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, which proved true over the years due to improved hardware that enabled programs to increase speed with each new generation.

For the past fifty 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 Moore's Law. Moore's Law is named after Gordon Moore, a computer scientist, who in 1965 predicted that the density and speed of transistors would double every eighteen months before reaching a maximum speed beyond which technology could not advance. The original prediction for the increase of CPU speed presumed a speed-doubling trend for ten years. Moore's prediction was correct—except for the fact that progress continued for almost fifty years (decades past his estimate).

Today, the single-processor CPU has nearly reached the speed of light, in the process generating an enormous amount of heat due to energy dissipation; it is this heat that is the limiting factor to further improvements. Moore's Law prediction about transistor speed has come to fruition; transistors simply cannot run any faster. However, Moore's Law is not dead; modern transistors are increasing in density, providing opportunities for parallelism within the confines of that top speed. Consequently, the combination of multicore architecture and parallel programming models is keeping Moore’s Law alive. As CPU single-core performance improvement have stagnated, developers have adapted 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 some 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 Let’s start with terminology

In computer programming, there are some terms (such as *concurrency*, *parallelism,* and *multithreading*) that are used in the same context, but which have different meanings. In some ways, due to their similarities, the tendency to treat these terms as the same thing is correct. But, when it becomes important to reason about the behavior of a program, it is crucial to make a distinction between computer programming terms. For example, concurrency is by definition multithreading, but multithreading is not necessarily concurrent. In other words, we can easily make a multicore CPU function like a single-core CPU, but not the other way around. Let’s discuss some terminology.

1.1.1 Sequential programming

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, he or she is only able to make one drink at a time. Thus, you must wait patiently in line before you order. Then, because there are a few steps to making a cappuccino (grinding the coffee, brewing the coffee, steaming the milk, frothing the milk, and combining the coffee and milk), it takes a while longer to get your cappuccino. Figure 1.1 shows this process.

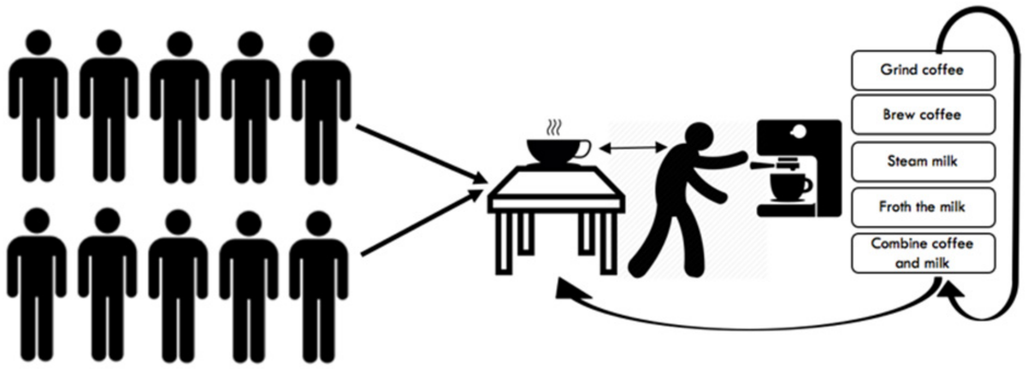


Figure 1.1. For each person in line, the barista is sequentially repeating the same set of instructions (grind coffee, brew coffee, steam the milk, froth the milk, and combine the coffee and the milk to make a cappuccino).

This is an example of sequential work, where one task must be completed before the next. It's a convenient approach, with a clear set of step-by-step instructions of what to do and when to do it. Moreover, in our example, the barista will likely not get confused and make any mistakes while preparing the cappuccino because the steps are clear and ordered (there are no context switches). 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. For example, in imperative and object-oriented programming we tend to write code that behaves sequentially, with all attention and resources focused on the task currently running. We express the program by performing an ordered set of statements, one after another. However, while the unit of code follows this sequential pattern, the application as a whole can perform differently.

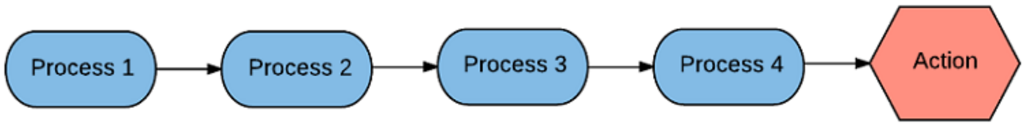


Figure 1.2. Typical sequential coding involving a consecutive, progressively ordered execution of processes

Concurrent programming Suppose the barista would prefer to initiate multiple steps and execute them concurrently, thus moving the customer line along much faster (and, consequently, increasing garnered tips). For example, once the coffee is ground, the barista can start brewing the espresso; in the meantime, taking a new order or starting 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. In fact, because there is only one espresso machine, the barista must stop one task to start or continue another, which means the barista, in reality, executes only one task at a time as shown in figure 1.3. In modern multicore computers, this is a waste of valuable resources.

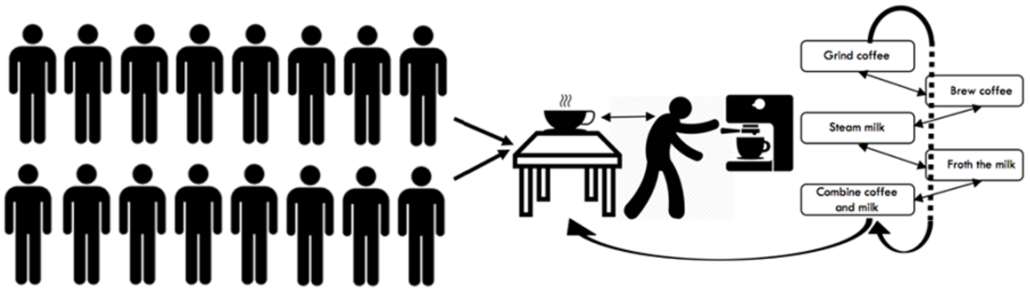


Figure 1.3. The barista is switching between the operations (multitasking) of preparing the coffee (grind and brew) and preparing the milk (steam and froth). As a result, the barista executes segments of multiple tasks in an interleaved manner, giving the illusion of multitasking. However, only one operation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

In computer programming, utilization of 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. *Concurrency* describes the ability to run several programs or multiple parts of a program at the same time. This can happen either in a single CPU core or in parallel, if multiple CPU cores are available. The throughput and responsiveness of the program can be improved through the asynchronous or parallel execution of a task. For example, as application that streams video content is concurrent 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. However, in a single-core environment, the execution of one thread is actually being 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*.

1.1.2 Parallelism

From the developer's prospective, we think 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 about executing multiple steps at once, literally at the same time, to improve the speed of the application. *Parallelism* is the concept of multiple threads concurrently executing a series of tasks on different cores. Thus, all parallel programs are concurrent, though, as we have seen, not all concurrency is parallel. That's because parallelism requires hardware support (multiple cores). Thus, parallelism is achievable only in multicore devices (figure 1.4) and is the means to increasing performance and throughput (the rate at which it processes a computation) of a program.

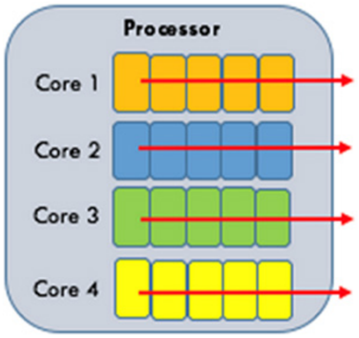


Figure 1.4. Only multicore machines allow parallelism for simultaneously executing different tasks. In this figure, each core is performing an independent task, each marked with a different color.

To return to the coffee shop, imagine for a moment you are the manager of the coffee shop and want to reduce the waiting time for customers by speeding up drink production. A simple solution is to hire a second barista and set up a second coffee station. In fact, with two baristas working simultaneously, the queues of customers can be processed independently and in parallel, thus speeding up the preparation of cappuccinos (figure 1.5).

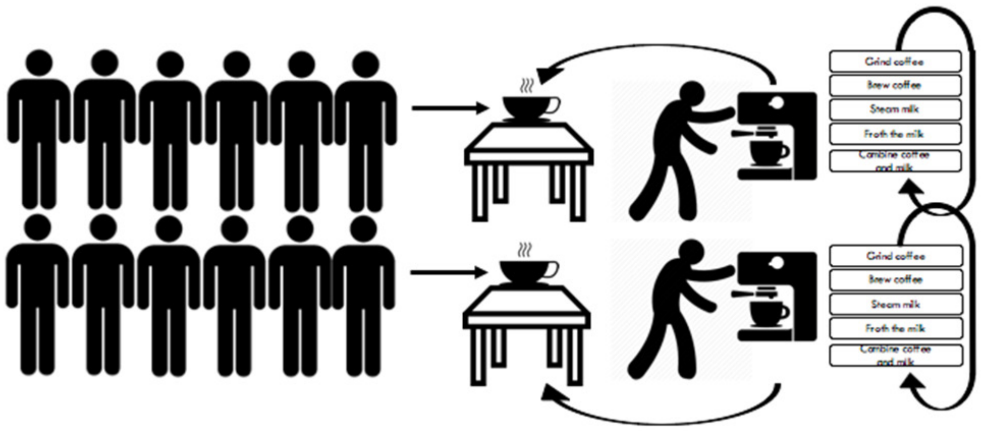


Figure 1.5. The production of cappuccinos is faster because two baristas can work in parallel with two coffee stations.

No break in production results in a benefit in performance. The goal of parallelism is to maximize the utilization 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.

Parallelism and concurrency are related programming models. A parallel program is also concurrent, but a concurrent program is not always parallel, with parallel programming being a subset of concurrent programming. In other words, while *concurrency* refers to the design of the system, *parallelism* relates to the execution. In fact, concurrent and parallel programming models are directly linked to the local hardware environment where they are performed.

1.1.3 Multitasking

Multitasking is the concept of performing multiple tasks over a period of time by executing them concurrently. We are familiar with this idea because we multitask all the time in our daily lives. For example, while we are waiting for the barista to prepare our cappuccino, we use our smartphone to check our emails or scan a new story. We are 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, however, 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 1.6); however, in a single-core computer, it's quite possible that multitasking can slow down the performance of a program by introducing extra overhead for context switching between threads.

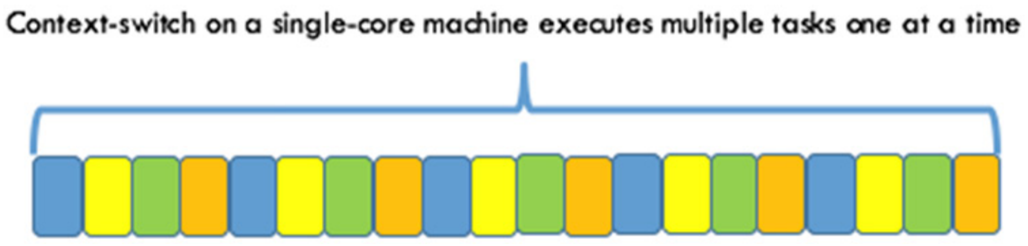

Context-switch on a single-core machine executes multiple tasks one at a time

Figure 1.6. Each task has a different color, 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 generally 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 user interface (UI) responsiveness to help avoid freezing the user interface during long operations.

1.1.4 Multithreading

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 does not necessarily imply multithreading. Multithreading enables an application to subdivide specific tasks into individual threads that run in parallel within the same process.[1]

A *thread* is a unit of computation (an independent set of programming instructions designed to achieve a particular result), which the Operating System (OS) scheduler independently executes and manages. Multithreading differs from multitasking: unlike multitasking, with multithreading the threads share resources. However, this "sharing resources" design presents more programming challenges than does multitasking. We discuss the problem of sharing variables between threads in the subsection "Concurrency hazards."

[1] 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.

The concepts of parallel and multithreading programming are closely related. However, 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, extending well beyond multithreading. For example, we could use multithreading to parallelize a program by sharing resources in the same process, but we could also parallelize a program by executing the computation in multiple processes or even in different computers. In Chapter #, "The Agent Programming Model," we discuss the multi-process approach in more detail.

Figure 1.7 shows the relationship between these terms.

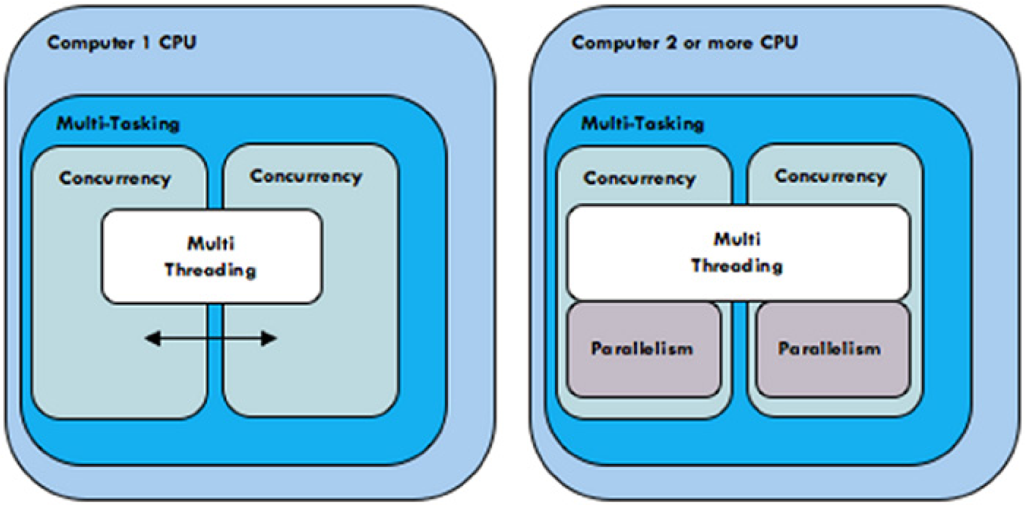


Figure 1.7. Relationship between concurrency, parallelism, multithreading, and multitasking in a single- and a multicore device.

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 does not 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 for that is that parallelism is achievable only on multi-cores devices.
- *Multitasking* concurrently performs multiple threads from different processes. Multitasking does not necessarily mean parallel execution, which is achieved only if there are multiple CPUs.

- *Multithreading* extends the idea of multitasking; it is 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.

1.2 Why the need for concurrency?

Concurrency is a natural part of life—we humans are accustomed to multitasking. We can read an email while drinking a cup of coffee, or type on the keyboard 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 is 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. So it is with applications. The problem is, however, that most applications are not written to evenly split the tasks required among the CPUs available.

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. In 2012, Google received more than 2 million search queries per minute; in 2014, that number has more than doubled. I found this Pixar story interesting and inspiring. In 1995, Pixar produced the first completely computer-generated movie, *Toy Story*. In computer animation, a multitude of details and information must be rendered for each image, such as shading and lighting. All of 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* utilized 100 dual-processor machines connected together to create their movie, and the utilization of parallel computation was indispensable. A few years later, Pixar evolved for *Toy Story 2*, using an incredible 1,400 computer processors for digital movie editing, thereby vastly improving digital quality and editing time. In the beginning of 2000, this incredible computer power increased even more, to 3,500 processors. Sixteen years later, the computer power utilized to process a fully animated movie reached an absurd 24,000 cores. Thus, the need for parallel computing is increasing exponentially.

Let's consider a processor with *N* (as any number) running cores. In a single-threaded application, there will be only one core running. 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 was not designed with concurrency in mind, then you are 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 very high, possibly 100%, while all the other cores are under-utilized or idle. In a machine with eight cores, running non-concurrent programs means the overall utilization of the resources could be as low as 15% (figure 1.8).

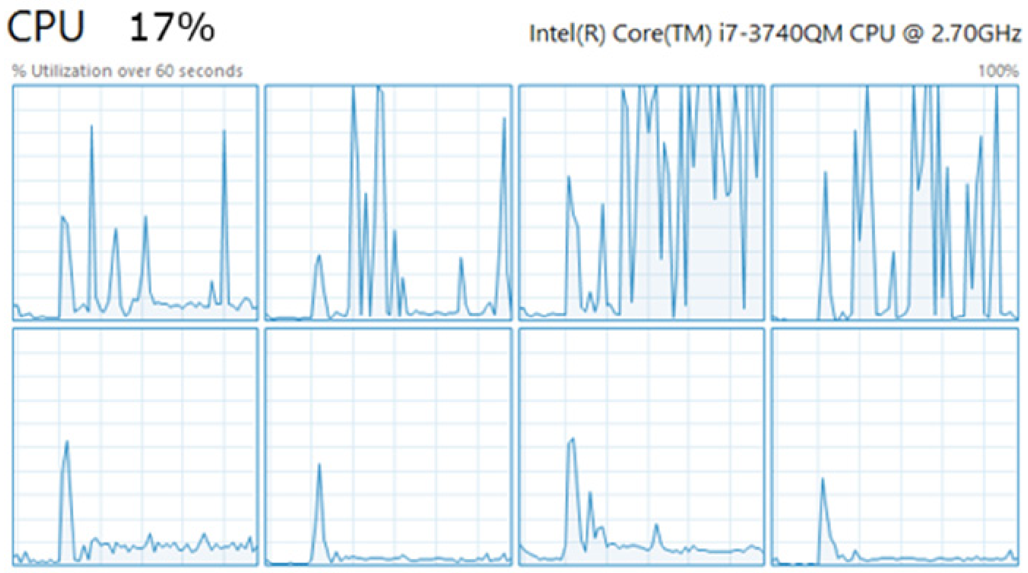
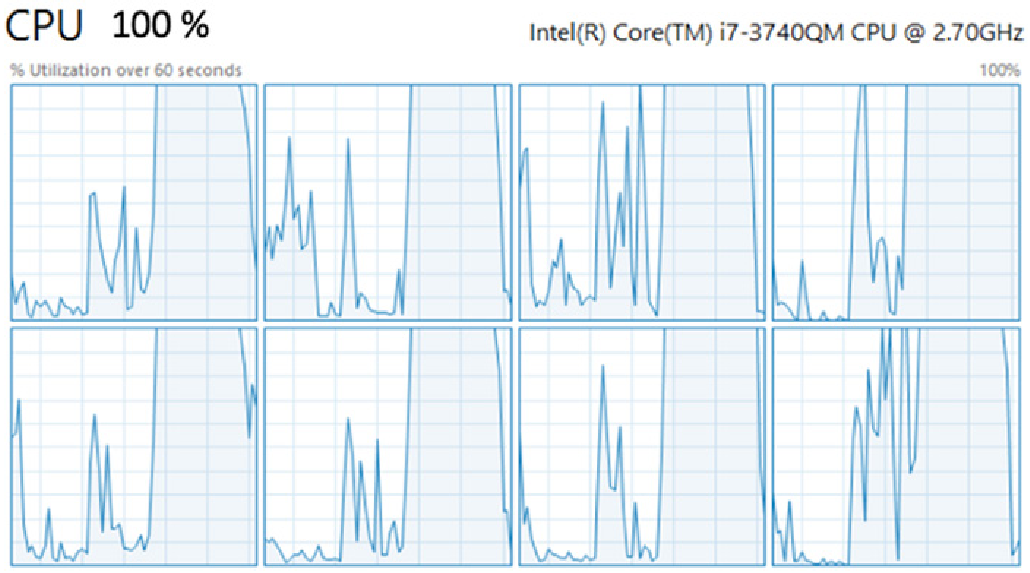

CPU 17%
Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
% Utilization over 60 seconds
100%

Figure 1.8. Windows Task Manager shows a program poorly utilizing the CPU resources.

Such waste of computing power unequivocally illustrates that sequential code is not the correct programming model for multicore processers. To maximize the utilization of the computational resources available, the Microsoft .NET platform provides parallel execution of code through multithreading. By leveraging parallelism, a program can take full advantage of the resources available, as illustrated by the CPU performance counter in figure 1.9, where you'll notice that all the processor cores are running very high, possibly at 100%.


CPU 100 %
Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
% Utilization over 60 seconds
100%
Figure 1.9. A program that is written with concurrency in mind can maximize CPU resources, possibly up to 100 percent.

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.

1.2.1 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 is cheaper to build scalable programs that leverage the computational resources available with fewer servers, then to keep buying and adding expensive hardware that are under-utilized 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 is rewarding to be able to improve the performance of your program with this functional programming approach. While it is 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. There is something magical and spectacular about opening the Windows Task Manager and proudly noticing that the CPU usage spikes to 100% after your code changes. Performance can be measured, which is why you should care about improving it. 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 user interfaces will continue to spur the need for concurrency. In lockstep with the direction of hardware, it is evident that concurrency and parallelism will be the future of programming.

1.3 The Pitfalls of Concurrent Programming

Concurrent and parallel programming are without doubt beneficial for rapid responsiveness and speedy execution of a given computation. However, this gain of performance and reactive experience comes with a price. Utilizing 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 are 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. However, this property becomes hard to resolve in a parallel execution. External circumstances, such as the Operating System (OS) scheduler or cache coherence (which will be covered in chapter x, “Parallel Streams in Action”) 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 just 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 possible to leverage concurrency and parallelism to reach incredible computational performance and a highly responsive application?
- How can this program take full advantage of the multicore computer power?
- How can communication with and access to the same memory location between threads be coordinated while ensuring thread safety?[2]
- How can the deterministic execution of a program be ensured?
- How can the execution of a program be parallelized without jeopardizing the quality of the final result?

[2] A method is called thread-safe when the data and state do not get corrupted if two or more threads attempt to access and modify the data or state at the same time.

These are not easy questions to answer. However, there are patterns and techniques that can help. For example, in the presence of side effects,3 the determinism of the computation is lost because the order in which concurrent tasks execute becomes variable. The obvious solution is to avoid side effects in favor of pure functions. We will study and learn these techniques and practices during the course of the book.

1.3.1 Concurrency hazards

Writing concurrent programs is not easy, and there are many sophisticated elements to consider during program design. Creating new threads or queueing multiple jobs on the thread pools is relatively simple, but how do you ensure correctness in the program? When there are many threads continually accessing shared data, you must consider how to safeguard the data structure to guarantee its integrity. A thread should be able to write and modify a memory location atomically,[4] 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 (figure 1.10).

Consider the case of two threads (Thread 1 and Thread 2) running in parallel, both trying to access and modify the same shared value (*x*) as shown in figure 1.10. For Thread 1 to modify a variable requires more than just 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* will have changed. More precisely, it is 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*. This result is data corruption. This phenomenon is called *race condition*.

[3] **A side effect arises when a method changes some state from outside its scope, or it communicates with the "outside world," like calling a database or writing to the file system.**
[4] **An atomic operation accesses a shared memory and completes in a single step relative to other threads.**

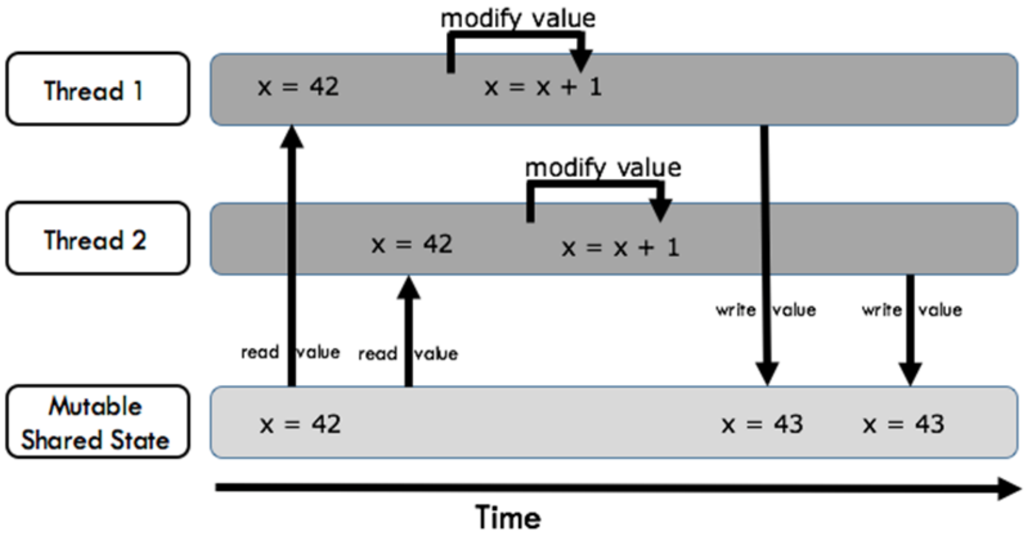


Figure 1.10. Two threads (Thread 1 and Thread 2) running in parallel, both trying to access and modify the same shared value (*x*). If Thread 2 tries to read from the same memory location while Thread 1 is writing back an updated value, the value of *x* will have changed. This result is data corruption or *race condition*.

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.

However, 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* (figure 1.11). Consider the case where Thread 1 and Thread 2 are waiting on each other to complete work and get blocked indefinitely in that waiting. In the figure, 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.

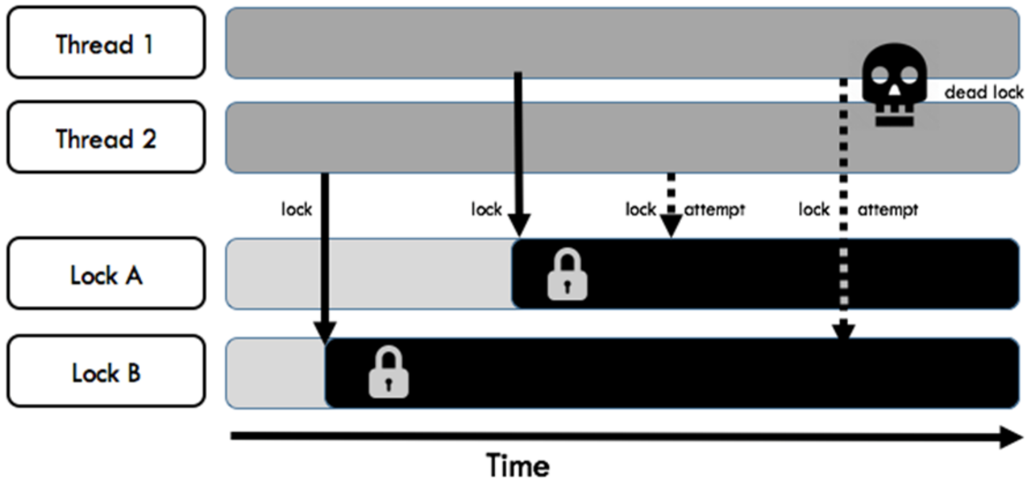


Figure 1.11. In this scenario, Thread 1 acquires Lock A, and, right after, Thread 2 acquires Lock B. At this point, both threads are waiting at the lock that will never be released. This is a case of deadlock.

Following is a list of some concurrency hazards with a brief explanation. Chapters 5 ("Data Parallel in Action") and 7 ("Light Task based concurrency") will provide 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[5] 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 treads 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 in order to communicate and synchronize memory access. The acquisition and release of locks comes with some performance penalty, slowing down all processes. Furthermore, as the number of cores gets larger, the cost of lock contention can potentially increase. Thus, as more tasks are introduced to share the same data, the overhead associated with locks can negatively impact the computation. The section "A simple real-world example (parallel quicksort)" demonstrates the consequence and overhead costs due to introducing lock synchronization.

Deadlock is a concurrency problem that originates from using *locks*. In particular, it occurs when there is a cycle of tasks in which each task is blocked while waiting for

5 The definition of a resource is pretty broad, and can be a file, an image, a variable or a collection.

another to proceed. Because all are waiting for another task to do something, they are blocked indefinitely. The more that resources are shared among threads, the more locks are needed to avoid race condition, and the higher is the risk of deadlocks.

Lack of composition is a design problem originating from the introduction of locks in the code. Simply put, locks do not 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 functional programming.

1.3.2 The sharing-of-state evolution

Real-world programs require some interaction between tasks, such as exchanging information to coordinate work. This cannot be implemented without sharing some data that is 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 mutable or each task has its own copy. The solution is to safeguard all the code from those concurrency problems. There is no compiler or tool that 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, there has been a rise of libraries and frameworks 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 do not need safeguards because the concept of functional programming maps well onto concurrent programming models.

1.3.3 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,[6] 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 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. `Quicksort` can operate in-place on an array,

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

requiring small additional amounts of memory to perform the sorting. The algorithm consists of three simple steps as shown in figure 1.12:

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

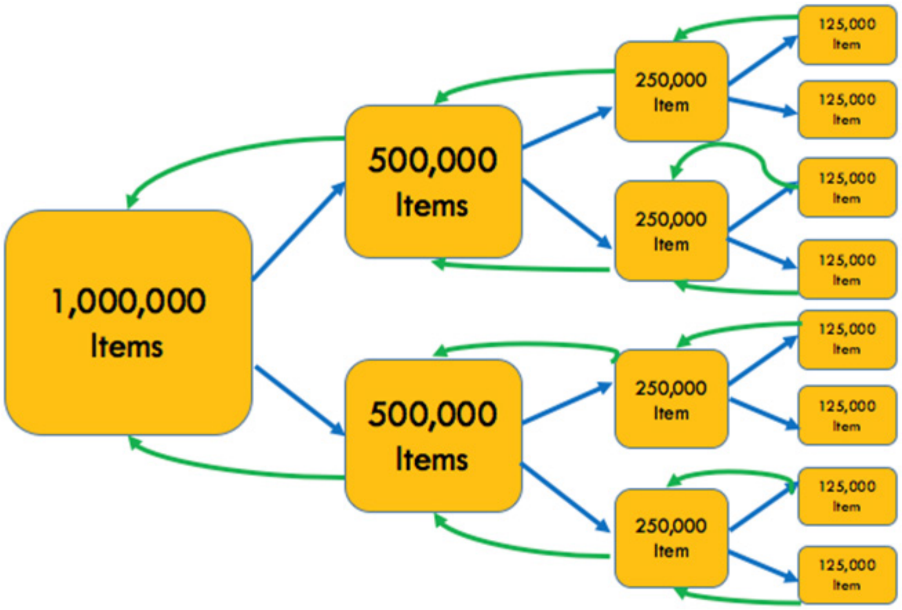


Figure 1.12. The recursive function `divide-and-conquer`. Each block is divided into equal halves, where the pivot element must be the median of the sequence, until each portion of code can be executed independently. When all of the single blocks are completed, they send the result back to the previous caller to be aggregated. Quicksorting is based on the idea of picking a pivot point and partitioning the sequence into smaller than the pivot and elements bigger than the pivot 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 .NET 4.0, makes it fairly simple to implement and exploit parallelism for this type of algorithm. Using the TPL, we can divide each step of the algorithm and perform each task in parallel, recursively. It is a straight and easy implementation, but we 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, we'll use the functional programming language F#. Due to its intrinsic recursive nature, however, the idea behind this implementation can also be applied to C#, which would require an imperative `for-loop` approach with a mutable state. C# does not support optimized tail-recursive functions like F#, thus, there is the hazard of raising a Stack Overflow exception when the call-stack pointer exceeds the stack constraint. In chapter 3, "Functional and Parallel Data Structures," we'll go into detail on how to overcome this C# limitation. Listing 1.1 shows a sequential `Quicksort` function in F# that adopts the divide-and-conquer strategy. For each recursive iteration, we select a pivot point and use that to partition the total array. We 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, we are using the `List.partition` API, which returns a tuple containing two lists: one list that satisfies the predicate and the other that does not.

Listing 1.1 A 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 ::
[CA] quicksortSequential larger)
```

Running this `Quicksort` algorithm against a 1,000,000 random and unsorted integer array on my system (an eight-logical core, 2.2 GHz clock speed) takes an average of 6.5 seconds. However, when we analyze this algorithm design, the opportunity to parallelize is evident. At the end of `quicksortSequential`, we recursively call into `quicksortSequential` with each partition of the array identified by the `(fun number -> number> firstElement) restofArray` as shown in the following listing. By spawning new tasks using the TPL library, we can rewrite in parallel this portion of the code.

Listing 1.2 A parallel Quicksort algorithm using the TPL library

```
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)      ❶
        let right = Task.Run(fun () -> quicksortParallel larger)       ❶
        left.Result @ (firstElement :: right.Result)                   ❷
```

❶ **Task.Run executes the recursive calls in tasks that can run in parallel; for each recursive call, tasks are dynamically created**
❷ **Appends the result for each task into a sorted array**

The algorithm in listing 1.2 is now running in parallel, which is using more CPU resources by spreading the work across all available cores. However, even with improved resource utilization, the overall performance result is not meeting expectations.Execution time dramatically increases instead of decreasing. 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 creating 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 is important that you do not add more tasks than necessary. The disappointing result demonstrates an important characteristic of parallelism: there are inherent limitations to how much extra threading or extra processing will actually help a specific algorithmic implementation.

To achieve better optimization, we 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. Furthermore, 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 currently 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. In this case, we are passing the value resulting from '*Math.Log(float System.Enviroment.ProcessorCount, 2.) + 1*' for the *max Depth*. *This ensures that every level of the recursion will spawn two child tasks until all the available cores are enlisted.* In this way, the overall work of partitioning among threads has a fair and suitable distribution for each core.

Listing 1.3 A better parallel Quicksort Algorithm using the TPL library

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

❶ **Tracks the function recursion level with the depth parameter**
❷ **If the value of depth is negative , skips the parallelization**
❸ **Sequentially executes the quicksort using the current thread**
❹ **If the value of depth is positive, allows the function to be called recursively, spawning two new tasks**

1.3.4 Benchmarking in F#

We executed the quicksort sample using the F# `REPL` (Read-Evaluate-Print-Loop), which is a very 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 as it facilitates the programming process. One more benefit of using the `REPL` 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 is interpreted and executed.

The following benchmark (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 hyperthreading) . On an average of ten runs, these are the execution times in seconds.

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 is important to mention that for a small array, less than 100 items, the parallel `sort` algorithms are slower than the serial version, due to the unnecessary overhead. 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 is 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.

1.4 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 in *Concurrent Haskell*

Functional programming (FP) is about minimizing and controlling side effects; this is more commonly called *pure functional programming*. For example, utilizing the concept of transformation, 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. FP encourages considering if mutability and side effects are really 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 have discussed a number of common issues specific to developing concurrent applications in either imperative or object-oriented programming (OOP). In these programming paradigms, we are dealing with objects as a base construct. Conversely, in terms of concurrency, dealing with objects has some caveats to consider when passing from a single-thread program to a massively parallelizing work, which is a challenging and entirely different scenario.

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 is 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 1.13).

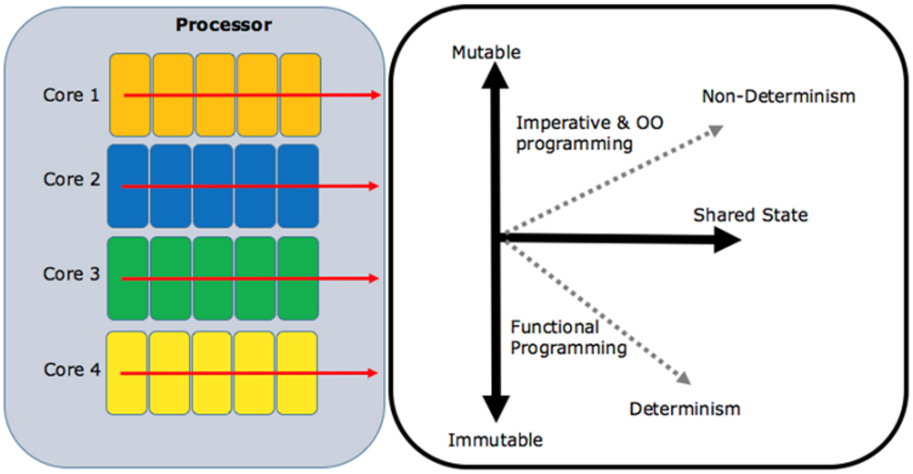


Figure 1.13. In the functional paradigm, due to immutability as a default construct, concurrent programming guarantees deterministic execution, even in the case of a shared state. On the other hand, imperative and OOP use mutable states, which are hard to manage in a multithread environment, and this leads to non-deterministic programs.

It is important to consider that components of systems that embrace the Functional programming 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, we are also able to remove those concurrency-specific issues. This is the reason why FP makes an excellent concurrent programming model. Functional programming is an exceptional fit for concurrent programmers to achieve correct high performance in highly multithreaded environments using simple code. At the very heart of Functional programming, neither variables nor state are mutable and cannot be shared, and functions may not have side effects.

Functional programming is the most practical way to write concurrent programs. Trying to write concurrent programs in imperative languages is not 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!

1.4.1 Benefits of functional programming

There are real advantages to learning functional programming, even if you have no actual plans to adopt this style in the immediate future. Still, it is 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 seem overwhelming at first. Functional programming, 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 functional programming techniques, you will improve the readability and correctness of your applications.

The benefits of functional programming (with focus on concurrency) include:

Immutability: A property that prevents modification of an object state after creation. In particular, in FP there is no concept of variable assignment. Once a value has been associated with an identifier (which replaces the name of variable in FP), it cannot change. Thus, 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: There are no side effects, which means that functions do not change any variables or data of any type outside the function. Functions are said to be pure if they are 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 a 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 exact same meaning. Thus, 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 functional programming 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 abstraction 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 will let you write more modular, expression-oriented, and conceptually simple code. The combinations of these functional programming 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.

1.5 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# is not difficult; in terms of concepts, they are basically the same. Going from an imperative paradigm to a functional paradigm, however, is a far more difficult challenge. Core concepts are completely replaced. There is no more state. There are no more variables. There are no more side effects.

However, 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 likens that to a patient whose doctor prescribes 30 minutes of exercise per day to be healthy. The patient knows that there are real benefits in exercise, but is also aware that daily exercise implies commitment and sacrifice.

Similarly, learning a new paradigm is not hard, but does require dedication, engagement, and time. I encourage everyone who wants to be a better programmer to consider learning the functional programming 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 and how to leverage them, 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 functional programming 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 leverage 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 functional programming.

1.6 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 does not mean you must learn a new language; you can apply the functional paradigm by using tools that you are already familiar with, such as multipurpose languages like C# and F#. There are several functional features that have been added to the C# and F# languages over the course of the last few years that will make it easier for you to shift to incorporating this new paradigm.

The intrinsically different approaches to solving problems are the reasons these languages have been 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, we *should* think of programming languages as tools.

Ideally, a solution should be a combination of C# and F# projects that work cohesively together. 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 encompasses functional, imperative, and OOP techniques.
- Both languages are part of the .NET ecosystem and derive a rich set of libraries that can be leveraged equally by both languages.
- F# is a functional-first programming language, which 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 data structures instead of lock-free code, due to default immutable constructors. An example of that is the discriminated union and the record types. These types have structural equality and do not allow nulls, which 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

The null reference was introduced by Tony Hoare 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:

"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"
https://en.wikipedia.org/wiki/Tony_Hoare#Apologies_and_retractions

- F# is naturally parallelizable because it uses immutably as 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.[7] These programming languages are great tools for writing concurrent software, and the power and options for workable solutions aggregate when used in coexistence.
- Furthermore, F# and C# can interoperate. In fact, a F# function can call a method in a

[7] Symmetric Multiprocessing (SMP) is the processing of programs by multiple processors that share a common operating system and memory.

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. In addition, we'll examine tools and libraries like the Task Parallel Library (TPL) and Reactive Extensions (RX) that have been successfully designed, inspired, and implemented by adopting the functional paradigm to obtain composable abstraction.

It is 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 take the level of abstraction away from the traditional and complex memory-synchronization models. Examples of these higher-level libraries are Intel's TBB (Threading Building Blocks) and Microsoft's TPL (Task Parallel Library). There are also some interesting open source projects, such as OpenMP (which provides pragmas[8] that you can insert into a program to make parts of it parallel) and OpenCL (which is a low-level language to communicate with Graphic Processing Units (GPUs)). GPU programming has a lot of traction and has been sanctioned by Microsoft with C++AMP extensions and Accelerator.NET.

1.7 Summary

The main takeaway from this chapter is that no silver bullet exists for the challenges and complexities of concurrent and parallel programming; as professional engineers, we need different types of ammunition, and we need to know how and when to use them to hit the target. Keep in mind the following guidelines:

- 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 is not 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.

[8] **Pragmas are compiler-specific definitions that can be used to create new preprocessor functionality or to send implementation-defined information to the compiler.**

- Functional programming excels in parallel computation because immutability is the default; thus, making it simpler to reason about the share of data.