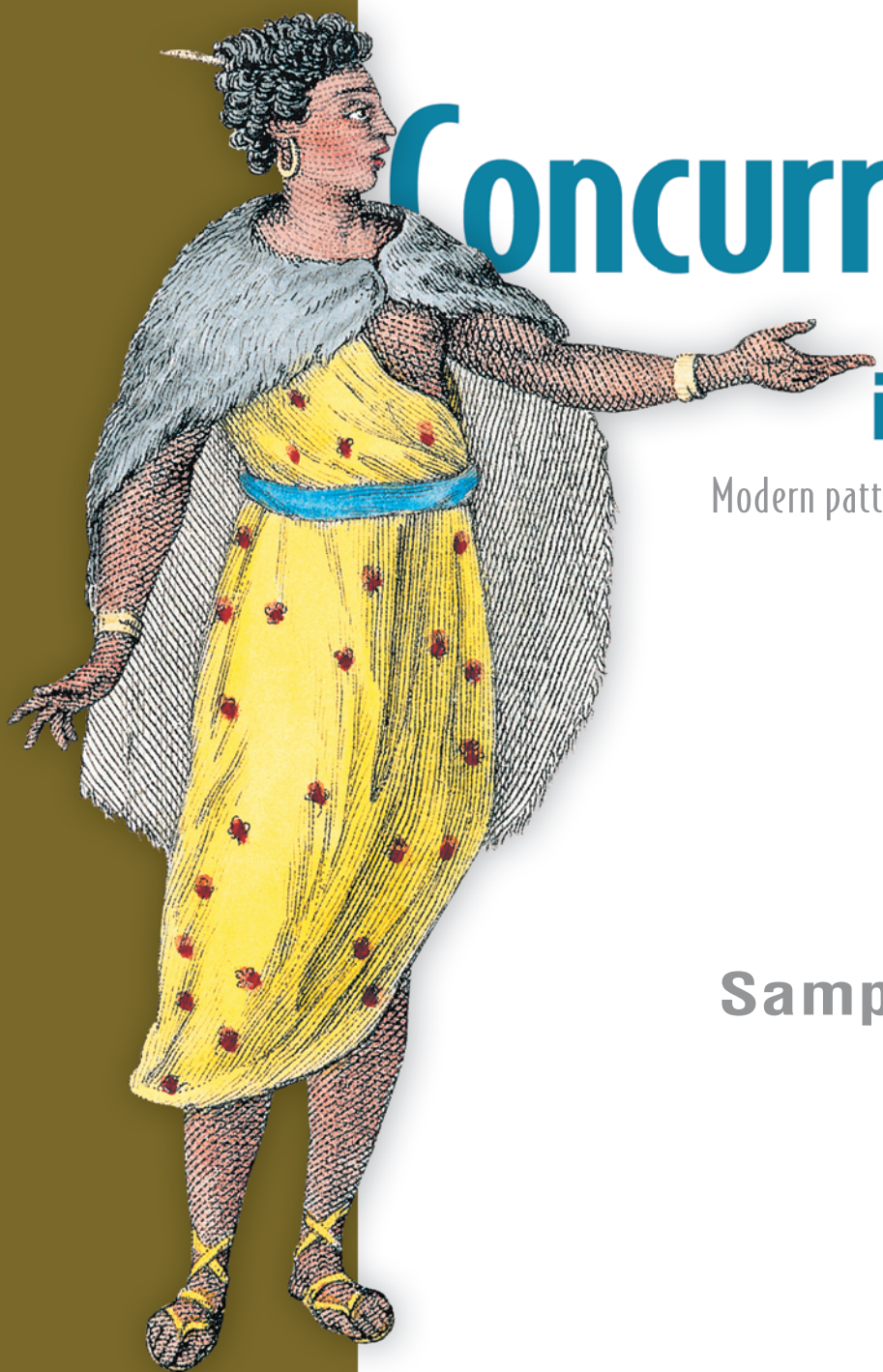
With examples in C# and F#



Concurrency in .NET

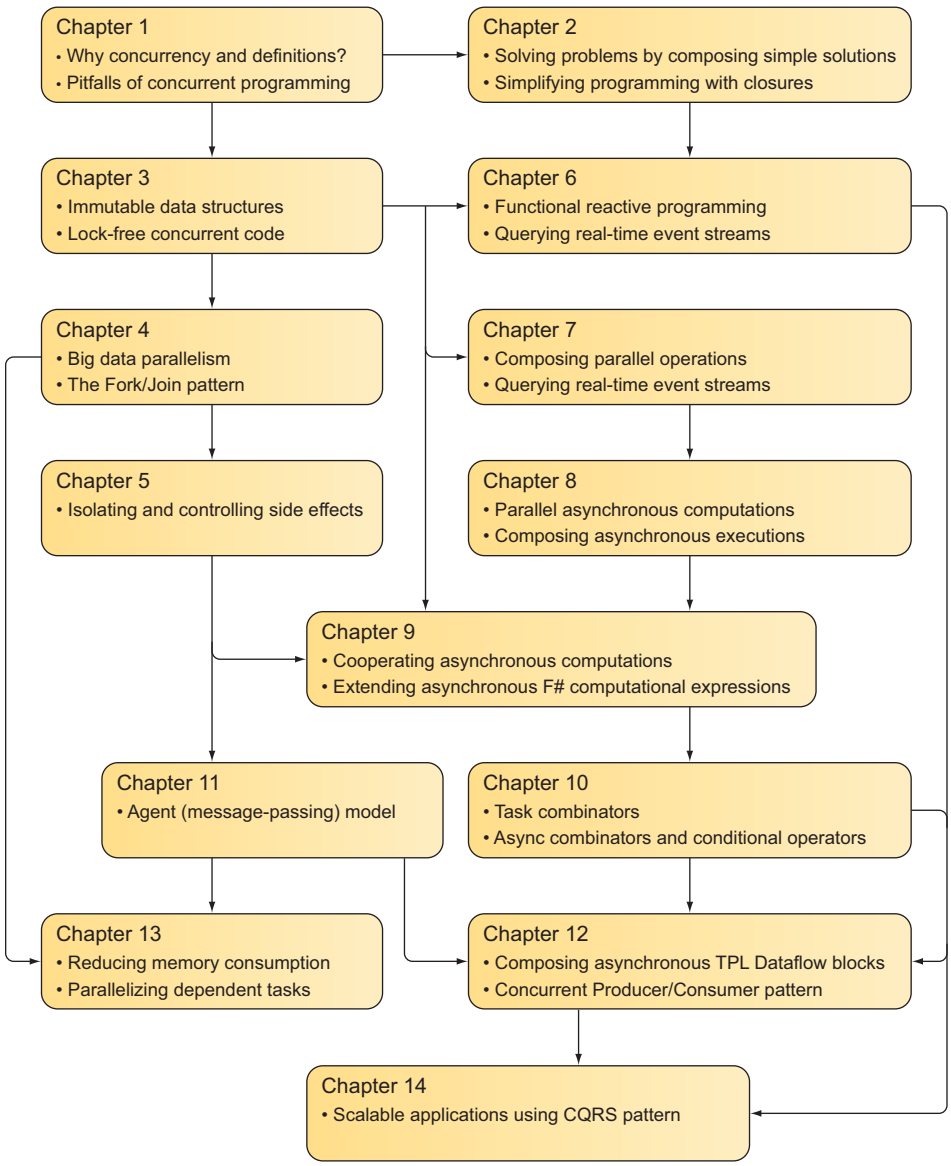
Modern patterns of concurrent and parallel programming

Riccardo Terrell

Sample Chapter

MANNING

Chapter dependency graph



Concurrency in .NET

Modern patterns of concurrent and parallel programming

by Riccardo Terrell

Chapter 7

brief contents

7 Task-based functional parallelism

This chapter covers

- Task parallelism and declarative programming semantics
- Composing parallel operations with functional combinators
- Maximizing resource utilization with the Task Parallel Library
- Implementing a parallel functional pipeline pattern

The task parallelism paradigm splits program execution and runs each part in parallel by reducing the total runtime. This paradigm targets the distribution of tasks across different processors to maximize processor utilization and improve performance. Traditionally, to run a program in parallel, code is separated into distinct areas of functionality and then computed by different threads. In these scenarios, primitive locks are used to synchronize the access to shared resources in the presence of multiple threads. The purpose of locks is to avoid race conditions and memory corruption by ensuring concurrent mutual exclusion. The main reason locks are

used is due to the design legacy of waiting for the current thread to complete before a resource is available to continue running the thread.

A newer and better mechanism is to pass the rest of the computation to a callback function (which runs after the thread completes execution) to continue the work. This technique in FP is called *continuation-passing style (CPS)*. In this chapter, you'll learn how to adopt this mechanism to run multiple tasks in parallel without blocking program execution. With this technique, you'll also learn how to implement task-based parallel programs by isolating side effects and mastering function composition, which simplifies the achievement of task parallelism in your code. Because compositionality is one of the most important features in FP, it eases the adoption of a declarative programming style. Code that's easy to understand is also simple to maintain. Using FP, you'll engage task parallelism in your programs without introducing complexity, as compared to conventional programming.

7.1 A short introduction to task parallelism

Task parallelism refers to the process of running a set of independent tasks in parallel across several processors. This paradigm partitions a computation into a set of smaller tasks and executes those smaller tasks on multiple threads. The execution time is reduced by simultaneously processing multiple functions.

In general, parallel jobs begin from the same point, with the same data, and can either terminate in a fire-and-forget fashion or complete altogether in a task-group continuation. Any time a computer program simultaneously evaluates different and autonomous expressions using the same starting data, you have task parallelism. The core of this concept is based on small units of computations called *futures*. Figure 7.1 shows the comparison between data parallelism and task parallelism.

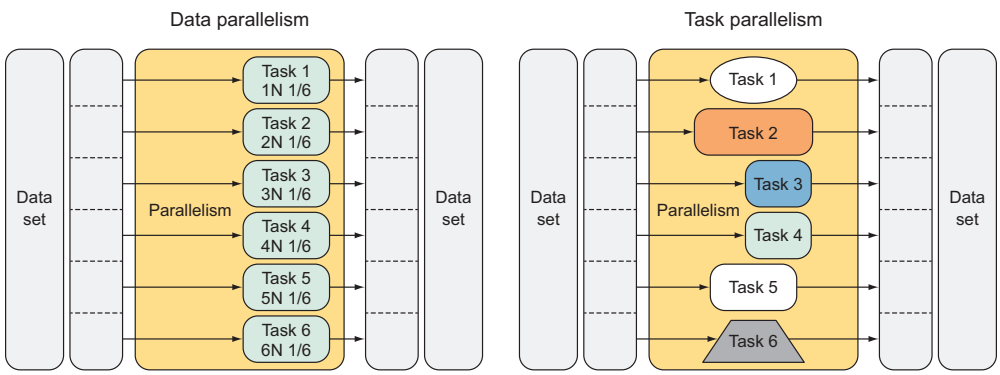


Figure 7.1 Data parallelism is the simultaneous execution of the same function across the elements of a data set. Task parallelism is the simultaneous execution of multiple and different functions across the same or different data sets.

> **Task parallelism isn't data parallelism**
>
> Chapter 4 explains the differences between task parallelism and data parallelism. To refresh your memory, these paradigms are at two ends of the spectrum. *Data parallelism* occurs when a single operation is applied to many inputs. *Task parallelism* occurs when multiple diverse operations perform against their own input. It is used to query and call multiple Web APIs at one time, or to store data against different database servers. In short, task parallelism parallelizes functions; data parallelism parallelizes data.

Task parallelism achieves its best performance by adjusting the number of running tasks, depending on the amount of parallelism available on your system, which corresponds to the number of available cores and, possibly, their current loads.

7.1.1 Why task parallelism and functional programming?

In the previous chapters, you've seen code examples that deal with data parallelism and task composition. Those data-parallel patterns, such as Divide and Conquer, Fork/Join, and MapReduce, aim to solve the computational problem of splitting and computing in parallel smaller, independent jobs. Ultimately, when the jobs are terminated, their outputs are combined into the final result.

In real-world parallel programming, however, you commonly deal with different and more complex structures that aren't so easily split and reduced. For example, the computations of a task that processes input data could rely on the result of other tasks. In this case, the design and approach to coordinating the work among multiple tasks is different than for the data parallelism model and can sometimes be challenging. This challenge is due to task dependencies, which can reach convoluted connections where execution times can vary, making the job distribution tough to manage.

The purpose of task parallelism is to tackle these scenarios, providing you, the developer, with a toolkit of practices, patterns, and, in the case of programming, the .NET Framework, a rich library that simplifies task-based parallel programming. In addition, FP eases the compositional aspect of tasks by controlling side effects and managing their dependencies in a declarative programming style.

Functional paradigm tenets play an essential role in writing effective and deterministic task-based parallel programs. These functional concepts were discussed in the early chapters of this book. To summarize, here's a list of recommendations for writing parallel code:

- Tasks should evaluate side-effect-free functions, which lead to referential transparency and deterministic code. Pure functions make the program more predictable because the functions always behave in the same way, regardless of the external state.
- Remember that pure functions can run in parallel because the order of execution is irrelevant.

- If side effects are required, control them locally by performing the computation in a function with run-in isolation.
- Avoid sharing data between tasks by applying a defensive copy approach.
- Use immutable structures when data sharing between tasks cannot be avoided.

NOTE A *defensive copy* is a mechanism that reduces (or eliminates) the negative side effects of modifying a shared mutable object. The idea is to create a copy of the original object that can be safely shared; its modification won't affect the original object.

7.1.2 Task parallelism support in .NET

Since its first release, the .NET Framework has supported the parallel execution of code through multithreading. Multithreaded programs are based on an independent execution unit called a *thread*, which is a lightweight process responsible for multitasking within a single application. (The `Thread` class can be found in the Base Class Library (BCL) `System.Threading` namespace.) Threads are handled by the CLR. The creation of new threads is expensive in terms of overhead and memory. For example, the memory stack size associated with the creation of a thread is about 1 MB in an x86 architecture-based processor because it involves the stack, thread local storage, and context switches.

Fortunately, the .NET Framework provides a class `ThreadPool` that helps to overcome these performance problems. In fact, it's capable of optimizing the costs associated with complex operations, such as creating, starting, and destroying threads. Furthermore, the .NET `ThreadPool` is designed to reuse existing threads as much as possible to minimize the costs associated with the instantiation of new ones. Figure 7.2 compares the two processes.

The ThreadPool class

The .NET Framework provides a `ThreadPool` static class that loads a set of threads during the initialization of a multithreaded application and then reuses those threads, instead of creating new threads, to run new tasks as required. In this way, the `ThreadPool` class limits the number of threads that are running at any given point, avoiding the overhead of creating and destroying application threads. In the case of parallel computation, `ThreadPool` optimizes the performance and improves the application's responsiveness by avoiding context switches.

The `ThreadPool` class exposes the static method `QueueUserWorkItem`, which accepts a function (delegate) that represents an asynchronous operation.

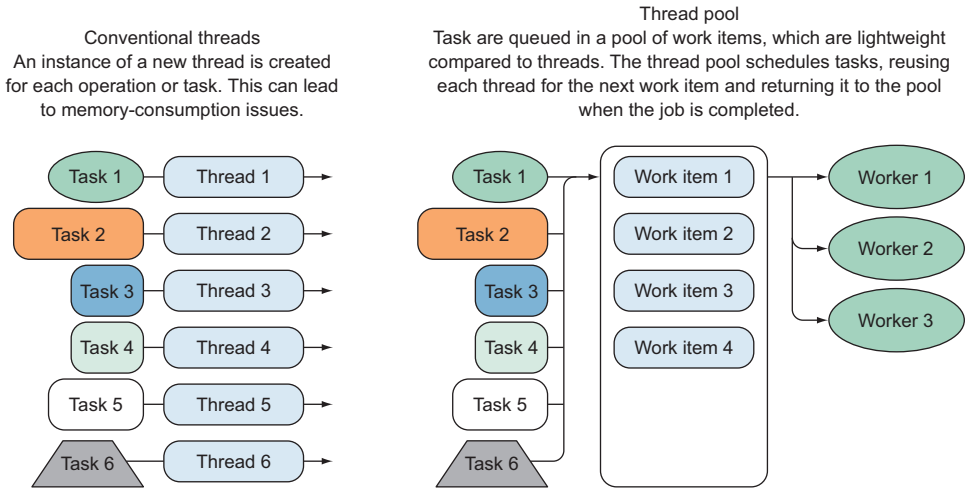


Figure 7.2 If using conventional threads, you must create an instance of a new thread for each operation or task. This can create memory consumption issues. By contrast, if using a thread pool, you queue a task in a pool of work items, which are lightweight compared to threads. The thread pool then schedules these tasks, reusing the thread for the next work item and returning it back to the pool when the job is completed.

The following listing compares starting a thread in a traditional way versus starting a thread using the `ThreadPool.QueueUserWorkItem` static method.

Listing 7.1 Spawning threads and `ThreadPool.QueueUserWorkItem`

```
Action<string> downloadSite = url => {
    var content = new WebClient().DownloadString(url);
    Console.WriteLine($"The size of the web site {url} is
➥ {content.Length}");
};

var threadA = new Thread(() => downloadSite("http://www.nasdaq.com"));
var threadB = new Thread(() => downloadSite("http://www.bbc.com"));

threadA.Start();
threadB.Start();
threadA.Join();
threadB.Join();

ThreadPool.QueueUserWorkItem(o => downloadSite("http://www.nasdaq.com"));
ThreadPool.QueueUserWorkItem(o => downloadSite("http://www.bbc.com"));
```

Uses a function to download a given website

Threads must start explicitly, providing the option to wait (join) for completion.

The ThreadPool.QueueUserWorkItem immediately starts an operation considered "fire and forget," which means the work item needs to produce side effects in order for the calculation to be visible.

A thread starts explicitly, but the `Thread` class provides an option using the instance method `Join` to wait for the thread. Each thread then creates an additional memory load, which is harmful to the runtime environment. Initiating an asynchronous

computation using `ThreadPool`'s `QueueUserWorkItem` is simple, but there are a few restraints when using this technique that introduce serious complications in developing a task-based parallel system:

- No built-in notification mechanism when an asynchronous operation completes
- No easy way to get back a result from a `ThreadPool` worker
- No built-in mechanism to propagate exceptions to the original thread
- No easy way to coordinate dependent asynchronous operations

To overcome these limitations, Microsoft introduced the notion of tasks with the TPL, accessible through the `System.Threading.Tasks` namespace. The tasks concept is the recommended approach for building task-based parallel systems in .NET.

7.2 The .NET Task Parallel Library

The .NET TPL implements a number of extra optimizations on top of `ThreadPool`, including a sophisticated `TaskScheduler` work-stealing algorithm (http://mng.bz/j4K1) to scale dynamically the degree of concurrency, as shown in figure 7.3. This algorithm guarantees an effective use of the available system processor resources to maximize the overall performance of the concurrent code.

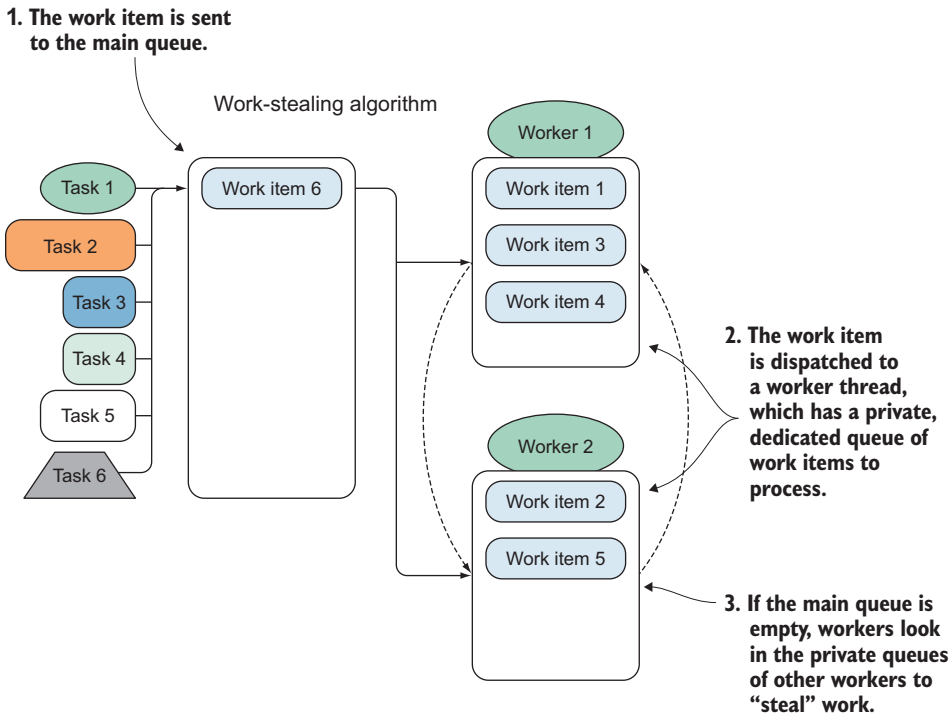


Figure 7.3 The TPL uses the work-stealing algorithm to optimize the scheduler. Initially, the TPL sends jobs to the main queue (step 1). Then it dispatches the work items to one of the worker threads, which has a private and dedicated queue of work items to process (step 2). If the main queue is empty, the workers look in the private queues of the other workers to "steal" work (step 3).

With the introduction of the task concept in place of the traditional and limited thread model, the Microsoft TPL eases the process of adding concurrency and parallelism to a program with a set of new types. Furthermore, the TPL provides support through the `Task` object to cancel and manage state, to handle and propagate exceptions, and to control the execution of working threads. The TPL abstracts away the implementation details from the developer, offering control over executing the code in parallel.

When using a task-based programming model, it becomes almost effortless to introduce parallelism in a program and concurrently execute parts of the code by converting those parts into tasks.

> **NOTE** The TPL provides the necessary infrastructure to achieve optimal utilization of CPU resources, regardless of whether you're running a parallel program on a single or multicore computer.

You have several ways to invoke parallel tasks. This chapter reviews the relevant techniques to implement task parallelism.

7.2.1 Running operations in parallel with TPL Parallel.Invoke

Using the .NET TPL, you can schedule a task in several ways, the `Parallel.Invoke` method being the simplest. This method accepts an arbitrary number of actions (delegates) as an argument in the form `ParamArray` and creates a task for each of the delegates passed. Unfortunately, the action-delegate signature has no input arguments, and it returns `void`, which is contrary to functional principles. In imperative programming languages, functions returning `void` are used for side effects.

When all the tasks terminate, the `Parallel.Invoke` method returns control to the main thread to continue the execution flow. One important distinction of the `Parallel.Invoke` method is that exception handling, synchronous invocation, and scheduling are handled transparently to the developer.

Let's imagine a scenario where you need to execute a set of independent, heterogeneous tasks in parallel as a whole, then continue the work after all tasks complete. Unfortunately, PLINQ and parallel loops (discussed in the previous chapters) cannot be used because they don't support heterogeneous operations. This is the typical case for using the `Parallel.Invoke` method.

> **NOTE** *Heterogeneous tasks* are a set of operations that compute as a whole regardless of having different result types or diverse outcomes.

Listing 7.2 runs functions in parallel against three given images and then saves the result in the filesystem. Each function creates a locally defensive copy of the original image to avoid unwanted mutation. The code example is in F#; the same concept applies to all .NET programming languages.

Listing 7.2 `Parallel.Invoke` executing multiple heterogeneous tasks

```
let convertImageTo3D (sourceImage:string) (destinationImage:string) =
    let bitmap = Bitmap.FromFile(sourceImage) :?> Bitmap
    let w,h = bitmap.Width, bitmap.Height
    for x in 20 .. (w-1) do
        for y in 0 .. (h-1) do
            let c1 = bitmap.GetPixel(x,y)
            let c2 = bitmap.GetPixel(x - 20,y)
            let color3D = Color.FromArgb(int c1.R, int c2.G, int c2.B)
            bitmap.SetPixel(x - 20 ,y,color3D)
    bitmap.Save(destinationImage, ImageFormat.Jpeg)

let setGrayscale (sourceImage:string) (destinationImage:string) =
    let bitmap = Bitmap.FromFile(sourceImage) :?> Bitmap
    let w,h = bitmap.Width, bitmap.Height
    for x = 0 to (w-1) do
        for y = 0 to  (h-1) do
            let c = bitmap.GetPixel(x,y)
            let gray = int(0.299 * float c.R + 0.587 * float c.G + 0.114 *
➥ float c.B)
            bitmap.SetPixel(x,y, Color.FromArgb(gray, gray, gray))
    bitmap.Save(destinationImage, ImageFormat.Jpeg)

let setRedscale (sourceImage:string) (destinationImage:string) =
    let bitmap = Bitmap.FromFile(sourceImage) :?> Bitmap
    let w,h = bitmap.Width, bitmap.Height
    for x = 0 to (w-1) do
        for y = 0 to  (h-1) do
            let c = bitmap.GetPixel(x,y)
            bitmap.SetPixel(x,y, Color.FromArgb(int c.R,
➥ abs(int c.G - 255), abs(int c.B - 255)))
    bitmap.Save(destinationImage, ImageFormat.Jpeg)

System.Threading.Tasks.Parallel.Invoke(
    Action(fun ()-> convertImageTo3D "MonaLisa.jpg" "MonaLisa3D.jpg"),
    Action(fun ()-> setGrayscale "LadyErmine.jpg" "LadyErmineRed.jpg"),
    Action(fun ()-> setRedscale "GinevraBenci.jpg" "GinevraBenciGray.jpg"))
```

Creates an image with a 3D effect from a given image

Creates a copy of an image from a given file path

A nested for loop accesses the pixels of the image.

Saves the newly created image in the filesystem

Creates an image, transforming the colors to shades of gray

A nested for loop accesses the pixels of the image.

Creates an image by applying the Red color filter

A nested for loop accesses the pixels of the image.

Saves the newly created image in the filesystem

In the code, `Parallel.Invoke` creates and starts the three tasks independently, one for each function, and blocks the execution flow of the main thread until all tasks complete. Due to the parallelism achieved, the total execution time coincides with the time to compute the slower method.

NOTE The source code intentionally uses the methods `GetPixel` and `SetPixel` to modify a `Bitmap`. These methods (especially `GetPixel`) are notoriously slow; but for the sake of the example we want to test the parallelism creating little

extra overhead to induce extra CPU stress. In production code, if you need to iterate throughout an entire image, you're better off marshaling the entire image into a byte array and iterating through that.

It's interesting to notice that the `Parallel.Invoke` method could be used to implement a Fork/Join pattern, where multiple operations run in parallel and then join when they're all completed. Figure 7.4 shows the images before and after the image processing.

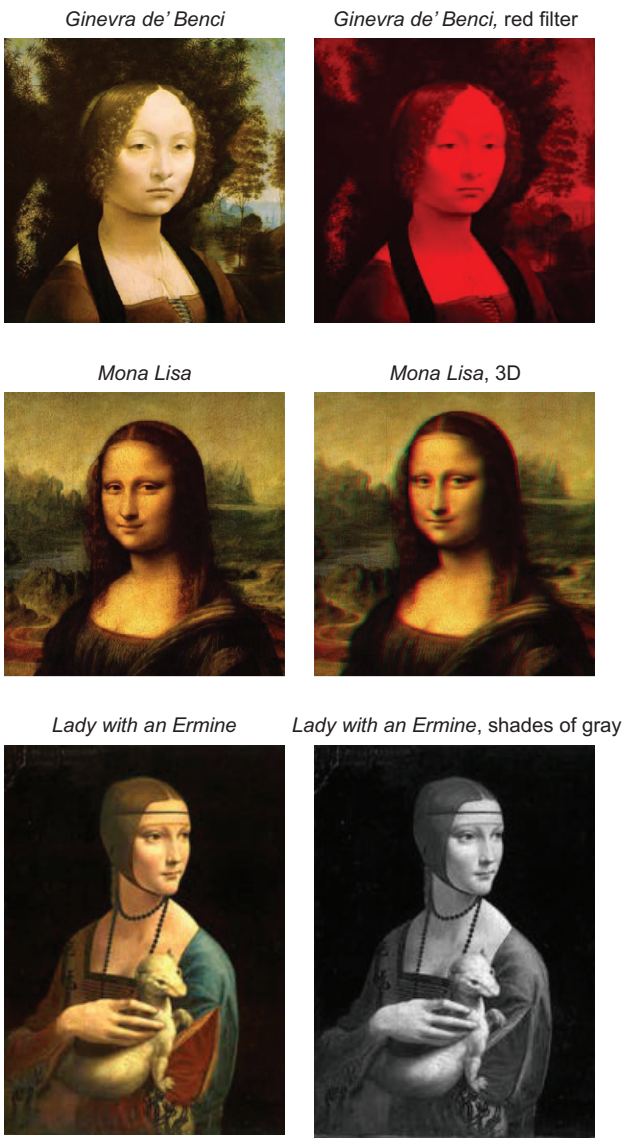


Figure 7.4 The resulting images from running the code in listing 7.2. You can find the full implementation in the downloadable source code.

Despite the convenience of executing multiple tasks in parallel, `Parallel.Invoke` limits the control of the parallel operation because of the `void` signature type. This method doesn't expose any resources to provide details regarding the status and outcome, either succeed or fail, of each individual task. `Parallel.Invoke` can either complete successfully or throw an exception in the form of an `AggregateException` instance. In the latter case, any exception that occurs during the execution is postponed and rethrown when all tasks have completed. In FP, exceptions are side effects that should be avoided. Therefore, FP provides a better mechanism to handle errors, a subject which will be covered in chapter 11.

Ultimately, there are two important limitations to consider when using the `Parallel.Invoke` method:

- The signature of the method returns `void`, which prevents compositionality.
- The order of task execution isn't guaranteed, which constrains the design of computations that have dependencies.

7.3 The problem of void in C#

It's common, in imperative programming languages such as C#, to define methods and delegates that don't return values (`void`), such as the `Parallel.Invoke` method. This method's signature prevents compositionality. Two functions can compose when the output of a function matches the input of the other function.

In function-first programming languages such as F#, every function has a return value, including the case of the `unit` type, which is comparable to a `void` but is treated as a value, conceptually not much different from a Boolean or integer.

`unit` is the type of any expression that lacks any other specific value. Think of functions used for printing to the screen. There's nothing specific that needs to be returned, and therefore functions may return `unit` so that the code is still valid. This is the F# equivalent of C#'s `void`. The reason F# doesn't use `void` is that every valid piece of code has a return type, whereas `void` is the absence of a return. Rather than the concept of `void`, a functional programmer thinks of `unit`. In F#, the `unit` type is written as `()`. This design enables function composition.

In principle, it isn't required for a programming language to support methods with return values. But a method without a defined output (`void`) suggests that the function performs some side effect, which makes it difficult to run tasks in parallel.

7.3.1 The solution for void in C#: the unit type

In functional programming, a function defines a relationship between its input and output values. This is similar to the way mathematical theorems are written. For example, in the case of a pure function, the return value is only determined by its input values.

In mathematics, every function returns a value. In FP, a function is a mapping, and a mapping has to have a value to map. This concept is missing in mainstream imperative programming languages such as C#, C++, and Java, which treat `void`s as methods that don't return anything, instead of as functions that can return something meaningful.

In C#, you can implement a `Unit` type as a `struct` with a single value that can be used as a return type in place of a `void`-returning method. Alternatively, the Rx, discussed in chapter 6, provides a `unit` type as part of its library. This listing shows the implementation of the `Unit` type in C#, which was borrowed from the Microsoft Rx (http://bit.ly/2vEzMeM).

Listing 7.3 `Unit` type implementation in C#

```
public struct Unit : IEquatable<Unit>
{
    public static readonly Unit Default = new Unit();
    public override int GetHashCode() => 0;
    public override bool Equals(object obj) => obj is Unit;
    public override string ToString() => "()";

    public bool Equals(Unit other) => true;
    public static bool operator ==(Unit lhs, Unit rhs) => true;
    public static bool operator !=(Unit lhs, Unit rhs) => false;
}
```

The unit struct that implements the IEquatable interface to force the definition of a type-specific method for determining equality

Uses a helper static method to retrieve the instance of the Unit type

Overrides the base methods to force equality between Unit types

Equality between unit types is always true.

The `Unit` struct implements the `IEquatable` interface in such a way that forces all values of the `Unit` type to be equal. But what's the real benefit of having the `Unit` type as a value in a language type system? What is its practical use?

Here are two main answers:

- The type `Unit` can be used to publish an acknowledgment that a function is completed.
- Having a `Unit` type is useful for writing generic code, including where a generic first-class function is required, which reduces code duplication.

Using the `Unit` type, for example, you could avoid repeating code to implement `Action<T>` or `Func<T, R>`, or functions that return a `Task` or a `Task<T>`. Let's consider a function that runs a `Task<TInput>` and transforms the result of the computation into a `TResult` type:

```
TResult Compute<TInput, TResult>(Task<TInput> task,
            Func<TInput, TResult> projection) => projection(task.Result);

Task<int> task = Task.Run<int>(() => 42);
bool isTheAnswerOfLife = Compute(task, n => n == 42);
```

This function has two arguments. The first is a `Task<TInput>` that evaluates to an expression. The result is passed into the second argument, a `Func<TInput, TResult>` delegate, to apply a transformation and then return the final value.

NOTE This code implementation is for demo purposes only. It isn't recommended to block the evaluation of the task to retrieve the result as the `Compute` function does in the previous code snippet. Section 7.4 covers the right approach.

How would you convert the `Compute` function into a function that prints the result? You're forced to write a new function to replace the `Func<T>` delegate projection into an `Action` delegate type. The new method has this signature:

```
void Compute<TInput>(Task<TInput> task, Action<TInput> action) =>
                     action(task.Result);

Task<int> task = Task.Run<int>(() => 42);
Compute(task, n => Console.WriteLine($"Is {n} the answer of life?
➥ {n == 42}"));
```

It's also important to point out that the `Action` delegate type is performing a side effect: in this case, printing the result on the console, which is a function conceptually similar to the previous one.

It would be ideal to reuse the same function instead of having to duplicate code for the function with the `Action` delegate type as an argument. To do so, you'll need to pass a `void` into the `Func` delegate, which isn't possible in C#. This is the case where the `Unit` type removes code repetition. By using the `struct Unit` type definition, you can use the same function that takes a `Func` delegate to also produce the same behavior as the function with the `Action` delegate type:

```
Task<int> task = Task.Run<int>(() => 42);

Unit unit = Compute(task, n => {
     Console.WriteLine($"Is {n} the answer of life? {n == 42}");
     return Unit.Default;});
```

In that way, introducing the `Unit` type in the C# language, you can write one `Compute` function to handle both cases of returning a value or computing a side effect. Ultimately, a function returning a `Unit` type indicates the presence of side effects, which is meaningful information for writing concurrent code. Moreover, there are FP languages, such as Haskell, where the `Unit` type notifies the compiler, which then distinguishes between pure and impure functions to apply more granular optimization.

7.4 Continuation-passing style: a functional control flow

Task continuation is based on the functional idea of the CPS paradigm, discussed in chapter 3. This approach gives you execution control, in the form of continuation, by passing the result of the current function to the next one. Essentially, function continuation is a delegate that represents "what happens next." CPS is an alternative for the conventional control flow in imperative programming style, where each command is executed one after another. Instead, using CPS, a function is passed as an argument into a method, explicitly defining the next operation to execute after its own computation is completed. This lets you design your own flow-of-control commands.

7.4.1 Why exploit CPS?

The main benefit of applying CPS in a concurrent environment is avoiding inconvenient thread blocking that negatively impacts the performance of the program. For example, it's inefficient for a method to wait for one or more tasks to complete, blocking the main execution thread until its child tasks complete. Often the parent task, which in this case is the main thread, can continue, but cannot proceed immediately because its thread is still executing one of the other tasks. The solution: CPS, which allows the thread to return to the caller immediately, without waiting on its children. This ensures that the continuation will be invoked when it completes.

One downside of using explicit CPS is that code complexity can escalate quickly because CPS makes programs longer and less readable. You'll see later in this chapter how to combat this issue by combining TPL and functional paradigms to abstract the complexity behind the code, making it flexible and simple to use. CPS enables several helpful task advantages:

- Function continuations can be composed as a chain of operations.
- A continuation can specify the conditions under which the function is called.
- A continuation function can invoke a set of other continuations.
- A continuation function can be canceled easily at any time during computation or even before it starts.

In the .NET Framework, a task is an abstraction of the classic (traditional) .NET thread (http://mng.bz/DK6K), representing an independent asynchronous unit of work. The `Task` object is part of the `System.Threading.Tasks` namespace. The higher level of abstraction provided by the `Task` type aims to simplify the implementation of concurrent code and facilitate the control of the life cycle for each task operation. It's possible, for example, to verify the status of the computation and confirm whether the operation is terminated, failed, or canceled. Moreover, tasks are composable in a chain of operations by using continuations, which permit a declarative and fluent programming style.

The following listing shows how to create and run operations using the `Task` type. The code uses the functions from listing 7.2.

Listing 7.4 Creating and starting tasks

```
Task monaLisaTask = Task.Factory.StartNew(() =>
    convertImageTo3D("MonaLisa.jpg", "MonaLisa3D.jpg"));

Task ladyErmineTask = new Task(() =>
    setGrayscale("LadyErmine.jpg", "LadyErmine3D.jpg"));
ladyErmineTask.Start();

Task ginevraBenciTask = Task.Run(() =>
    setRedscale("GinevraBenci.jpg", "GinevraBenci3D.jpg"));
```

Runs the method convertImageTo3D using the StartNew Task static helper method

Runs the method setGrayscale by creating a new Task instance, and then calls the Start() Task instance method

Runs the method setRedscale using the simplified static method Run(), which runs a task with the default common properties

The code shows three different ways to create and execute a task:

- The first technique creates and immediately starts a new task using the built-in `Task.Factory.StartNew` method constructor.
- The second technique creates a new instance of a task, which needs a function as a constructor parameter to serve as the body of the task. Then, calling the `Start` instance method, the `Task` begins the computation. This technique provides the flexibility to delay task execution until the `Start` function is called; in this way, the `Task` object can be passed into another method that decides when to schedule the task for execution.
- The third approach creates the `Task` object and then immediately calls the `Run` method to schedule the task. This is a convenient way to create and work with tasks using the default constructor that applies the standard option values.

The first two options are a better choice if you need a particular option to instantiate a task, such as setting the `LongRunning` option. In general, tasks promote a natural way to isolate data that depends on functions to communicate with their related input and output values, as shown in the conceptual example in figure 7.5.

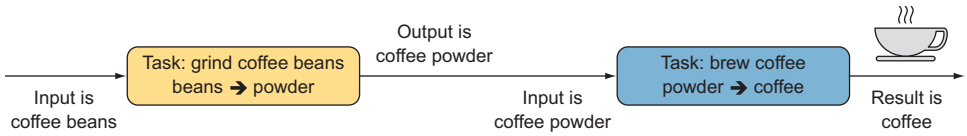


Figure 7.5 When two tasks are composed together, the output of the first task becomes the input for the second. This is the same as function composition.

> **NOTE** The `Task` object can be instantiated with different options to control and customize its behavior. The `TaskCreationOptions.LongRunning` option notifies the underlying scheduler that the task will be a long-running one, for example. In this case, the task scheduler might be bypassed to create an additional and dedicated thread whose work won't be impacted by thread-pool scheduling. For more information regarding `TaskCreationOptions`, see the Microsoft MSDN documentation online (http://bit.ly/2uxg1R6).

7.4.2 Waiting for a task to complete: the continuation model

You've seen how to use tasks to parallelize independent units of work. But in common cases the structure of the code is more complex than launching operations in a fire-and-forget manner. The majority of task-based parallel computations require a more sophisticated level of coordination between concurrent operations, where order of execution can be influenced by the underlying algorithms and control flow of the program. Fortunately, the .NET TPL library provides mechanisms for coordinating tasks.

Let's start with an example of multiple operations running sequentially, and incrementally redesign and refactor the program to improve the code compositionality and

performance. You'll start with the sequential implementation, and then you'll apply different techniques incrementally to improve and maximize the overall computational performance.

Listing 7.5 implements a face-detection program that can detect specific faces in a given image. For this example, you'll use the images of the presidents of the United States on $20, $50, and $100 bills, using the side on which the president's image is printed. The program will detect the face of the president in each image and return a new image with a square box surrounding the detected face. In this example, focus on the important code without being distracted by the details of the UI implementation. The full source code is downloadable from the book's website.

Listing 7.5 Face-detection function in C#

```
Bitmap DetectFaces(string fileName) {
    var imageFrame = new Image<Bgr, byte>(fileName);
    var cascadeClassifier = new CascadeClassifier();
    var grayframe = imageFrame.Convert<Gray, byte>();
    var faces = cascadeClassifier.DetectMultiScale(
        grayframe, 1.1, 3, System.Drawing.Size.Empty);
    foreach (var face in faces)
        imageFrame.Draw(face,
                  new Bgr(System.Drawing.Color.BurlyWood), 3);
    return imageFrame.ToBitmap();
}

void StartFaceDetection(string imagesFolder) {
    var filePaths = Directory.GetFiles(imagesFolder);
        foreach (string filePath in filePaths) {
            var bitmap = DetectFaces(filePath);
            var bitmapImage = bitmap.ToBitmapImage();
            Images.Add(bitmapImage);
        }
}
```

Uses a classifier to detect face features in an image

Instance of an Emgu.CV image to interop with the OpenCV library

Face-detection process

The detected face(s) is highlighted here, using a box that's drawn around it.

The processed image is added to the Images observable collection to update the UI.

The function `DetectFaces` loads an image from the filesystem using the given filename path and then detects the presence of any faces. The library `Emgu.CV` is responsible for performing the face detection. The `Emgu.CV` library is a .NET wrapper that permits interoperability with programming languages such as C# and F#, both of which can interact and call the functions of the underlying Intel OpenCV image-processing library.[1] The function `StartFaceDetection` initiates the execution, getting the filesystem path of the images to evaluate, and then sequentially processes the face detection in a `for-each` loop by calling the function `DetectFaces`. The result is a new `BitmapImage` type, which is added to the observable collection `Images` to update the UI. Figure 7.6 shows the expected result—the detected faces are highlighted in a box.

[1] OpenCV (Open Source Computer Vision Library) is a high-performance image processing library by Intel (https://opencv.org).

Figure 7.6 Result of the face-detection process. The right side has the images with the detected face surrounded by a box frame.

The first step in improving the performance of the program is to run the face-detection function in parallel, creating a new task for each image to evaluate.

Listing 7.6 Parallel-task implementation of the face-detection program

```
void StartFaceDetection(string imagesFolder)
{
    var filePaths = Directory.GetFiles(imagesFolder);

    var bitmaps = from filePath in filePaths
                  select Task.Run<Bitmap>(() => DetectFaces(filePath));

    foreach (var bitmap in bitmaps) {
         var bitmapImage = bitmap.Result;
             Images.Add(bitmapImage.ToBitmapImage());
    }
}
```

Starts a task sequentially from the TPL for each image to process

In this code, a LINQ expression creates an `IEnumerable` of `Task<Bitmap>`, which is constructed with the convenient `Task.Run` method. With a collection of tasks in place, the code starts an independent computation in the `for-each` loop; but the performance of the program isn't improved. The problem is that the tasks still run sequentially, one by one. The loop processes one task at a time, awaiting its completion before continuing to the next task. The code isn't running in parallel.

You could argue that choosing a different approach, such as using `Parallel.ForEach` or `Parallel.Invoke` to compute the `DetectFaces` function, could avoid the problem and guarantee parallelism. But you'll see why this isn't a good idea.

Let's adjust the design to fix the problem by analyzing what the foundational issue is. The `IEnumerable` of `Task<Bitmap>` generated by the LINQ expression is materialized during the execution of the `for-each` loop. During each iteration, a `Task<Bitmap>` is retrieved, but at this point, the task isn't competed; in fact, it's not even started. The reason lies in the fact that the `IEnumerable` collection is lazily evaluated, so the underlying task starts the computation at the last possible moment during its materialization. Consequently, when the result of the task bitmap inside the loop is accessed through the `Task<Bitmap>.Result` property, the task will block the joining thread until the task is

done. The execution will resume after the task terminates the computation and returns the result.

To write scalable software, you can't have any blocked threads. In the previous code, when the task's `Result` property is accessed because the task hasn't yet finished running, the thread pool will most likely create a new thread. This increases resource consumption and hurts performance.

After this analysis, it appears that there are two issues to be corrected to ensure parallelism (figure 7.7):

- Ensure that the tasks run in parallel.
- Avoid blocking the main working thread and waiting for each task to complete.

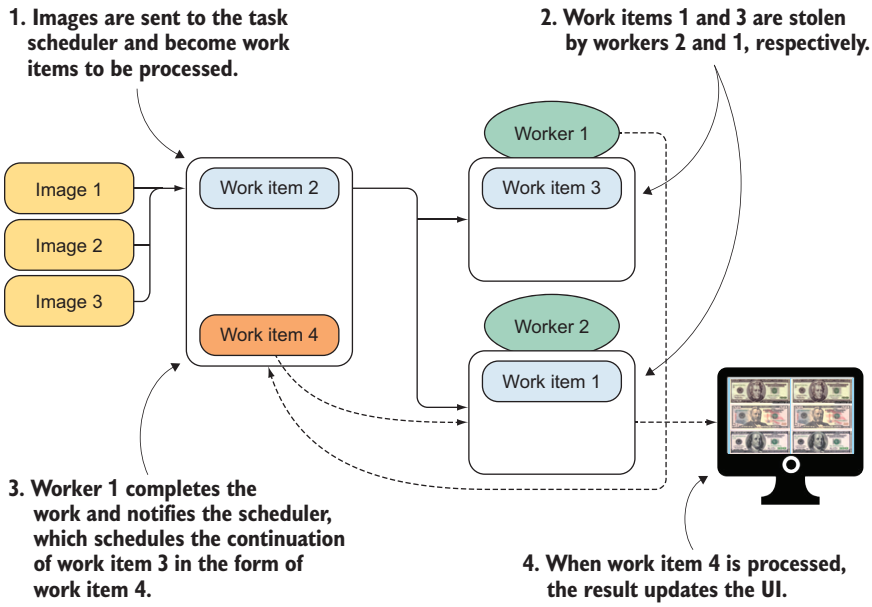


Figure 7.7 The images are sent to the task scheduler, becoming work items to be processed (step 1). Work item 3 and work item 1 are then "stolen" by worker 1 and worker 2, respectively (step 2). Worker 1 completes the work and notifies the task scheduler, which schedules the rest of the work for continuation in the form of the new work item 4, which is the continuation of work item 3 (step 3). When work item 4 is processed, the result updates the UI (step 4).

Here is how to fix issues to ensure the code runs in parallel and reduces memory consumption.

Listing 7.7 Correct parallel-task implementation of the `DetectFaces` function

```
ThreadLocal<CascadeClassifier> CascadeClassifierThreadLocal =
   new ThreadLocal<CascadeClassifier>(() => new CascadeClassifier());

Bitmap DetectFaces(string fileName) {
```

Uses a ThreadLocal instance to ensure a defensive copy of CascadeClassifier for each working task

```
      var imageFrame = new Image<Bgr, byte>(fileName);
      var cascadeClassifier = CascadeClassifierThreadLocal.Value;
      var grayframe = imageFrame.Convert<Gray, byte>();
      var faces = cascadeClassifier.DetectMultiScale(grayframe, 1.1, 3,
➥ System.Drawing.Size.Empty);

      foreach (var face in faces)
         imageFrame.Draw(face, new Bgr(System.Drawing.Color.BurlyWood), 3);
      return imageFrame.ToBitmap();
}

void StartFaceDetection(string imagesFolder) {
      var filePaths = Directory.GetFiles(imagesFolder);
      var bitmapTasks =
       (from filePath in filePaths
       select Task.Run<Bitmap>(() => DetectFaces(filePath))).ToList();

      foreach (var bitmapTask in bitmapTasks)
            bitmapTask.ContinueWith(bitmap => {
                  var bitmapImage = bitmap.Result;
                  Images.Add(bitmapImage.ToBitmapImage());
            }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

Uses a LINQ expression on the file paths that starts image processing in parallel

Task continuation ensures no blocking; the operation passes the continuation of the work when it completes.

TaskScheduler FromCurrentSynchronizationContext chooses the appropriate context to schedule work on the relevant UI.

In the example, to keep the code structure simple, there's the assumption that each computation completes successfully. A few code changes exist, but the good news is that true parallel computation is achieved without blocking any threads (by continuing the task operation when it completes). The main function `StartFaceDetection` guarantees executing the tasks in parallel by materializing the LINQ expression immediately with a call to `ToList()` on the `IEnumerable` of `Task<Bitmap>`.

> **NOTE** When you write a computation that creates a load of tasks, fire a LINQ query and make sure to materialize the query first. Otherwise, there's no benefit, because parallelism will be lost and the task will be computed sequentially in the `for-each` loop.

Next, a `ThreadLocal` object is used to create a defensive copy of `CascadeClassifier` for each thread accessing the function `DetectFaces`. `CascadeClassifier` loads into memory a local resource, which isn't thread safe. To solve this problem of thread unsafety, a local variable `CascadeClassifier` is instantiated for each thread that runs the function. This is the purpose of the `ThreadLocal` object (discussed in detail in chapter 4).

Then, in the function `StartFaceDetection`, the `for-each` loop iterates through the list of `Task<Bitmap>`, creating a continuation for each task instead of blocking the execution if the task is not completed. Because `bitmapTask` is an asynchronous operation, there's no guarantee that the task has completed executing before the `Result` property is accessed. It's good practice to use task continuation with the function `ContinueWith` to access the result as part of a continuation. Defining a task continuation is similar to

creating a regular task, but the function passed with the `ContinueWith` method takes as an argument a type of `Task<Bitmap>`. This argument represents the antecedent task, which can be used to inspect the status of the computation and branch accordingly.

When the antecedent task completes, the function `ContinueWith` starts execution as a new task. Task continuation runs in the captured current synchronization context, `TaskScheduler.FromCurrentSynchronizationContext`, which automatically chooses the appropriate context to schedule work on the relevant UI thread.

> **NOTE** When the `ContinueWith` function is called, it's possible to initiate starting the new task only if the first task terminates with certain conditions, such as if the task is canceled, by specifying the `TaskContinuationOptions.OnlyOnCanceled` flag, or if an exception is thrown, by using the `TaskContinuationOptions.OnlyOnFaulted` flag.

As previously mentioned, you could have used `Parallel.ForEach`, but the problem is that this approach waits until all the operations have finished before continuing, blocking the main thread. Moreover, it makes it more complex to update the UI directly because the operations run in different threads.

7.5 Strategies for composing task operations

Continuations are the real power of the TPL. It's possible, for example, to execute multiple continuations for a single task and to create a chain of task continuations that maintains dependencies with each other. Moreover, using task continuation, the underlying scheduler can take full advantage of the work-stealing mechanism and optimize the scheduling mechanisms based on the available resources at runtime.

Let's use task continuation in the face-detection example. The final code runs in parallel, providing a boost in performance. But the program can be further optimized in terms of scalability. The function `DetectFaces` sequentially performs the series of operations as a chain of computations. To improve resource use and overall performance, a better design is to split the tasks and subsequent task continuations for each `DetectFaces` operation run in a different thread.

Using task continuation, this change is simple. The following listing shows a new `DetectFaces` function, with each step of the face-detection algorithm running in a dedicated and independent task.

Listing 7.8 `DetectFaces` function using task continuation

```
Task<Bitmap> DetectFaces(string fileName)
{
    var imageTask = Task.Run<Image<Bgr, byte>>(
        () => new Image<Bgr, byte>(fileName)
   );
    var imageFrameTask = imageTask.ContinueWith(
        image => image.Result.Convert<Gray, byte>()
   );
```

Uses task continuation to pass the result of the work into the attached function without blocking

```
    var grayframeTask = imageFrameTask.ContinueWith(
        imageFrame => imageFrame.Result.Convert<Gray, byte>()
    );

    var facesTask = grayframeTask.ContinueWith(grayFrame =>
       {
          var cascadeClassifier = CascadeClassifierThreadLocal.Value;
          return cascadeClassifier.DetectMultiScale(
             grayFrame.Result, 1.1, 3, System.Drawing.Size.Empty);
       }
    );

    var bitmapTask = facesTask.ContinueWith(faces =>
        {
           foreach (var face in faces.Result)
               imageTask.Result.Draw(
               face, new Bgr(System.Drawing.Color.BurlyWood), 3);
           return imageTask.Result.ToBitmap();
        }
    );
    return bitmapTask;
}
```

Uses task continuation to pass the result of the work into the attached function without blocking

The code works as expected; the execution time isn't enhanced, although the program can potentially handle a larger number of images to process while still maintaining lower resource consumption. This is due to the smart `TaskScheduler` optimization. Because of this, the code has become cumbersome and hard to change. For example, if you add error handling or cancellation support, the code becomes a pile of spaghetti code—hard to understand and to maintain. It can be better. Composition is the key to controlling complexity in software.

The objective is to be able to apply a LINQ-style semantic to compose the functions that run the face-detection program, as shown here (the command and module names to note are in bold):

```
from image in Task.Run<Emgu.CV.Image<Bgr, byte>()
from imageFrame in Task.Run<Emgu.CV.Image<Gray, byte>>()
from faces in Task.Run<System.Drawing.Rectangle[]>()
select faces;
```

This is an example of how mathematical patterns can help to exploit declarative compositional semantics.

7.5.1 Using mathematical patterns for better composition

Task continuation provides support to enable task composition. How do you combine tasks? In general, function composition takes two functions and injects the result from the first function into the input of the second function, thereby forming one function. In chapter 2, you implemented this `Compose` function in C# (in bold):

```
Func<A, C> Compose<A, B, C>(this Func<A, B> f, Func<B, C> g) =>
                                                (n) => g(f(n));
```

Can you use this function to combine two tasks? Not directly, no. First, the return type of the compositional function should be exposing the task's elevated type as follows (noted in bold):

```
Func<A, Task<C>> Compose<A, B, C>(this Func<A, Task<B>> f,
                                   Func<B, Task<C>> g) => (n) => g(f(n));
```

But there's a problem: the code doesn't compile. The return type from the function `f` doesn't match the input of the function `g`: the function `f(n)` returns a type `Task<B>`, which isn't compatible with the type `B` in function `g`.

The solution is to implement a function that accesses the underlying value of the elevated type (in this case, the task) and then passes the value into the next function. This is a common pattern, called Monad, in FP; the Monad pattern is another design pattern, like the Decorator and Adapter patterns. This concept was introduced in section 6.4.1, but let's analyze this idea further so you can apply the concept to improve the face-detection code.

Monads are mathematical patterns that control the execution of side effects by encapsulating program logic, maintaining functional purity, and providing a powerful compositional tool to combine computations that work with elevated types. According to the monad definition, to define a monadic constructor, there are two functions, `Bind` and `Return`, to implement.

THE MONADIC OPERATORS, BIND AND RETURN

`Bind` takes an instance of an elevated type, unwraps the underlying value, and then invokes the function over the extracted value, returning a new elevated type. This function is performed in the future when it's needed. Here the `Bind` signature uses the `Task` object as an elevated type:

```
Task<R> Bind<T, R>(this Task<T> m, Func<T, Task<R>> k)
```

The `Return` value is an operator that wraps any type `T` into an instance of the elevated type. Following the example of the `Task` type, here's the signature:

```
Task<T> Return(T value)
```

> **NOTE** The same applies to other elevated types: for example, replacing the `Task` type with another elevated type such as the `Lazy` and `Observable` types.

THE MONAD LAWS

Ultimately, to define a correct monad, the `Bind` and `Return` operations need to satisfy the monad laws:

1. *Left identity*—Applying the `Bind` operation to a value wrapped by the `Return` operation and then passed into a function is the same as passing the value straight into the function:

   ```
   Bind(Return value, function) = function(value)
   ```

2 *Right identity*—Returning a bind-wrapped value is equal to the wrapped value directly:

```
Bind(elevated-value, Return) = elevated-value
```

3 *Associative*—Passing a value into a function `f` whose result is passed into a second function `g` is the same as composing the two functions `f` and `g` and then passing the initial value:

```
Bind(elevated-value, f(Bind(g(elevated-value)) =
            Bind(elevated-value, Bind(f.Compose(g), elevated-value))
```

Now, using these monadic operations, you can fix the error in the previous `Compose` function to combine the `Task` elevated types as shown here:

```
Func<A, Task<C>> Compose<A, B, C>(this Func<A, Task<B>> f,
                                  Func<B, Task<C>> g) => (n) => Bind(f(n), g);
```

Monads are powerful because they can represent any arbitrary operations against elevated types. In the case of the `Task` elevated type, monads let you implement function combinators to compose asynchronous operations in many ways, as shown in figure 7.8.

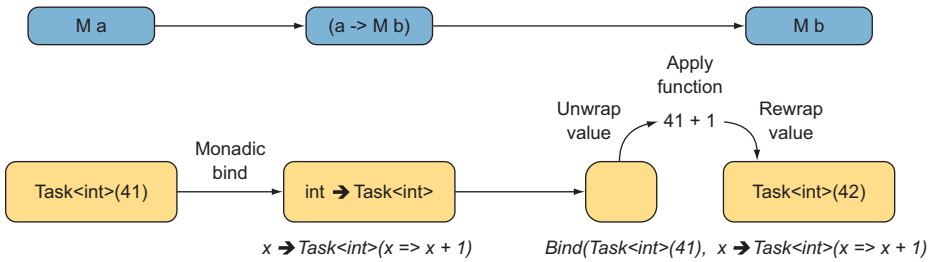


Figure 7.8 The monadic `Bind` operator takes the elevated value `Task` that acts as a container (wrapper) for the value 42, and then it applies the function `x → Task<int>(x => x + 1)`, where `x` is the number 41 unwrapped. Basically, the `Bind` operator unwraps an elevated value (`Task<int>(41)`) and then applies a function (`x + 1`) to return a new elevated value (`Task<int>(42)`).

Surprisingly, these monadic operators are already built into the .NET Framework in the form of LINQ operators. The LINQ `SelectMany` definition corresponds directly to the monadic `Bind` function. Listing 7.9 shows both the `Bind` and `Return` operators applied to the `Task` type. The functions are then used to implement a LINQ-style semantic to compose asynchronous operations in a monadic fashion. The code is in F# and then consumed in C# to keep proving the easy interoperability between these programming languages (the code to note is in bold).

Listing 7.9 Task extension in F# to enable LINQ-style operators for tasks

```
[<Sealed; Extension; CompiledName("Task")>]
type TaskExtensions =
  // 'T -> M<'T>
  static member Return value : Task<'T> = Task.FromResult<'T> (value)
```

The Return monadic operator takes any type T and returns a Task<T>.

```
// M<'T> * ('T -> M<'U>) -> M<'U>

static member Bind (input : Task<'T>, binder : 'T -> Task<'U>) =
    let tcs = new TaskCompletionSource<'U>()
    input.ContinueWith(fun (task:Task<'T>) ->
       if (task.IsFaulted) then
            tcs.SetException(task.Exception.InnerExceptions)
       elif (task.IsCanceled) then
            tcs.SetCanceled()
       else
            try

                (binder(task.Result)).ContinueWith(fun
➥ (nextTask:Task<'U>) -> tcs.SetResult(nextTask.Result)) |> ignore
            with
            | ex -> tcs.SetException(ex)) |> ignore
    tcs.Task

static member Select (task : Task<'T>, selector : 'T -> 'U) : Task<'U> =
      task.ContinueWith(fun (t:Task<'T>) -> selector(t.Result))

static member SelectMany(input:Task<'T>, binder:'T -> Task<'I>,
   projection:'T -> 'I -> 'R): Task<'R> =
      TaskExtensions.Bind(input,
          fun outer -> TaskExtensions.Bind(binder(outer), fun inner ->
              TaskExtensions.Return(projection outer inner)))

static member SelectMany(input:Task<'T>, binder:'T -> Task<'R>) : Task<'R>
   =
      TaskExtensions.Bind(input,
          fun outer -> TaskExtensions.Bind(binder(outer), fun inner ->
              TaskExtensions.Return(inner)))
```

The Bind operator takes a Task object as an elevated type, applies a function to the underlying type, and returns a new elevated type Task<U>

The Bind operator unwraps the result from the Task elevated type and passes the result into the continuation that executes the monadic function binder.

TaskCompletionSource initializes a behavior in the form of Task, so it can be treated like one.

The LINQ SelectMany operator acts as the Bind monadic operator.

The implementation of the `Return` operation is straightforward, but the `Bind` operation is a little more complex. The `Bind` definition can be reused to create other LINQ-style combinators for tasks, such as the `Select` and two variants of the `SelectMany` operators. In the body of the function `Bind`, the function `ContinueWith`, from the underlying task instance, is used to extract the result from the computation of the input task. Then to continue the work, it applies the binder function to the result of the input `task`. Ultimately, the output of the `nextTask` continuation is set as the result of the `tcs` `TaskCompletionSource`. The returning task is an instance of the underlying `TaskCompletionSource`, which is introduced to initialize a task from any operation that starts and finishes in the future. The idea of the `TaskCompletionSource` is to create a task that can be governed and updated manually to indicate when and how a given operation completes. The power of the `TaskCompletionSource` type is in the capability of creating tasks that don't tie up threads.

TaskCompletionSource

The purpose of the `TaskCompletionSource<T>` object is to provide control and refer to an arbitrary asynchronous operation as a `Task<T>`. When a `TaskCompletionSource` (http://bit.ly/2vDOmSN) is created, the underlying task properties are accessible through a set of methods to manage the lifetime and completion of the task. This includes `SetResult`, `SetException`, and `SetCanceled`.

APPLYING THE MONAD PATTERN TO TASK OPERATIONS

With the LINQ operations `SelectMany` on tasks in place, you can rewrite the `DetectFaces` function using an expressive and comprehension query (the code to note is in bold).

Listing 7.10 `DetectFaces` using task continuation based on a LINQ expression

```
Task<Bitmap> DetectFaces(string fileName)  {
     Func<System.Drawing.Rectangle[],Image<Bgr, byte>, Bitmap>
➥ drawBoundries =
        (faces, image) => {
              faces.ForAll(face => image.Draw(face, new
➥ Bgr(System.Drawing.Color.BurlyWood), 3));
            return image.ToBitmap();
      };

         return from image in Task.Run(() => new Image<Bgr, byte>(fileName))
                from imageFrame in Task.Run(() => image.Convert<Gray,
    byte>())
                from bitmap in Task.Run(() =>
           CascadeClassifierThreadLocal.Value.DetectMultiScale(imageFrame,
➥ 1.1, 3, System.Drawing.Size.Empty)).Select(faces =>
                                                drawBoundries(faces, image))
                select bitmap;
}
```

The detected face(s) are highlighted using a box that's drawn around them.

Task composition using the LINQ-like Task operators defined with the Task monadic operators

This code shows the power of the monadic pattern, providing composition semantics over elevated types such as tasks. Moreover, the code of the monadic operations is concentrated into the two operators `Bind` and `Return`, making the code maintainable and easy to debug. To add logging functionality or special error handling, for example, you only need to change one place in code, which is convenient.

In listing 7.10, the `Return` and `Bind` operators were exposed in F# and consumed in C#, as a demonstration of the simple interoperability between the two programming languages. The source code for this book contains the implementation in C#. A beautiful composition of elevated types requires monads; the *continuation monad* shows how monads can readily express complex computations.

USING THE HIDDEN FMAP FUNCTOR PATTERN TO APPLY TRANSFORMATION

One important function in FP is `Map`, which transforms one input type into a different one. The signature of the `Map` function is

```
Map :  (T -> R) -> [T] -> [R]
```

An example in C# is the LINQ `Select` operator, which is a map function for `IEnumerable` types:

```
IEnumerable<R>  Select<T,R>(IEnumerable<T> en, Func<T, R> projection)
```

In FP, this similar concept is called a *functor*, and the map function is defined as `fmap`. Functors are basically types that can be mapped over. In F#, there are many:

```
Seq.map : ('a -> 'b) -> 'a seq -> 'b seq
List.map : ('a -> 'b) -> 'a list -> 'b list
Array.map : ('a -> 'b) -> 'a [] -> 'b []
Option.map : ('a -> 'b) -> 'a Option -> 'b Option
```

This mapping idea seems simple, but the complexity starts when you have to map elevated types. This is when the functor pattern becomes useful.

Think about a functor as a container that wraps an elevated type and offers a way to transform a normal function into one that operates on the contained values. In the case of the `Task` type, this is the signature:

```
fmap : ('T -> 'R) -> Task<'T> -> Task<'R>
```

This function has been previously implemented for the `Task` type in the form of the `Select` operator as part of the LINQ-style operators set for tasks built in F#. In the last LINQ expression computation of the function `DetectFaces`, the `Select` operator projects (`map`) the input `Task<Rectangle[]>` into a `Task<Bitmap>`:

```
from image in Task.Run(() => new Image<Bgr, byte>(fileName))
from imageFrame in Task.Run(() => image.Convert<Gray, byte>())
from bitmap in Task.Run(() =>
            CascadeClassifierThreadLocal.Value.DetectMultiScale
                        (imageFrame, 1.1, 3, System.Drawing.Size.Empty))
     .select(faces => drawBoundries(faces, image))
select bitmap;
```

The concept of functors becomes useful when working with another functional pattern, applicative functors, which will be covered in chapter 10.

> **NOTE** The concepts of functors and monads come from the branch of mathematics called *category theory*,[2] but it isn't necessary to have any category theory background to follow and use these patterns.

THE ABILITIES BEHIND MONADS

Monads provide an elegant solution to composing elevated types. Monads aim to control functions with side effects, such as those that perform I/O operations, providing a mechanism to perform operations directly on the result of the I/O without having a value from impure functions floating around the rest of your pure program. For this reason, monads are useful in designing and implementing concurrent applications.

[2] For more information, see https://wiki.haskell.org/Category_theory.

7.5.2 Guidelines for using tasks

Here are several guidelines for using tasks:

- It's good practice to use immutable types for return values. This makes it easier to ensure that your code is correct.
- It's good practice to avoid tasks that produce side effects; instead, tasks should communicate with the rest of the program only with their returned values.
- It's recommended that you use the task continuation model to continue with the computation, which avoids unnecessary blocking.

7.6 The parallel functional Pipeline pattern

In this section, you're going to implement one of the most common coordination techniques—the Pipeline pattern. In general, a pipeline is composed of a series of computational steps, composed as a chain of stages, where each stage depends on the output of its predecessor and usually performs a transformation on the input data. You can think of the Pipeline pattern as an assembly line in a factory, where each item is constructed in stages. The evolution of an entire chain is expressed as a function, and it uses a message queue to execute the function each time new input is received. The message queue is non-blocking because it runs in a separate thread, so even if the stages of the pipeline take a while to execute, it won't block the sender of the input from pushing more data to the chain.

This pattern is similar to the Producer/Consumer pattern, where a producer manages one or more worker threads to generate data. There can be one or more consumers that consume the data being created by the producer. Pipelines allow these series to run in parallel. The implementation of the pipeline in this section follows a slightly different design as compared to the traditional one seen in figure 7.9.

The traditional Pipeline pattern with serial stages has a speedup, measured in throughput, which is limited to the throughput of the slowest stage. Every item pushed into the pipeline must pass through that stage. The traditional Pipeline pattern cannot scale automatically with the number of cores, but is limited to the number of stages. Only a linear pipeline, where the number of stages matches the number of available logical cores, can take full advantage of the computer power. In a computer with eight cores, a pipeline composed of four stages can use only half of the resources, leaving 50% of the cores idle.

FP promotes composition, which is the concept the Pipeline pattern is based on. In listing 7.11, the pipeline embraces this tenet by composing each step into a single function and then distributing the work in parallel, fully using the available resources. In an abstract way, each function acts as the continuation of the previous one, behaving as a continuation-passing style. The code listing implementing the pipeline is in F#, then consumed in C#. But in the downloadable source code, you can find the full implementation in both programming languages. Here the `IPipeline` interface defines the functionality of the pipeline.

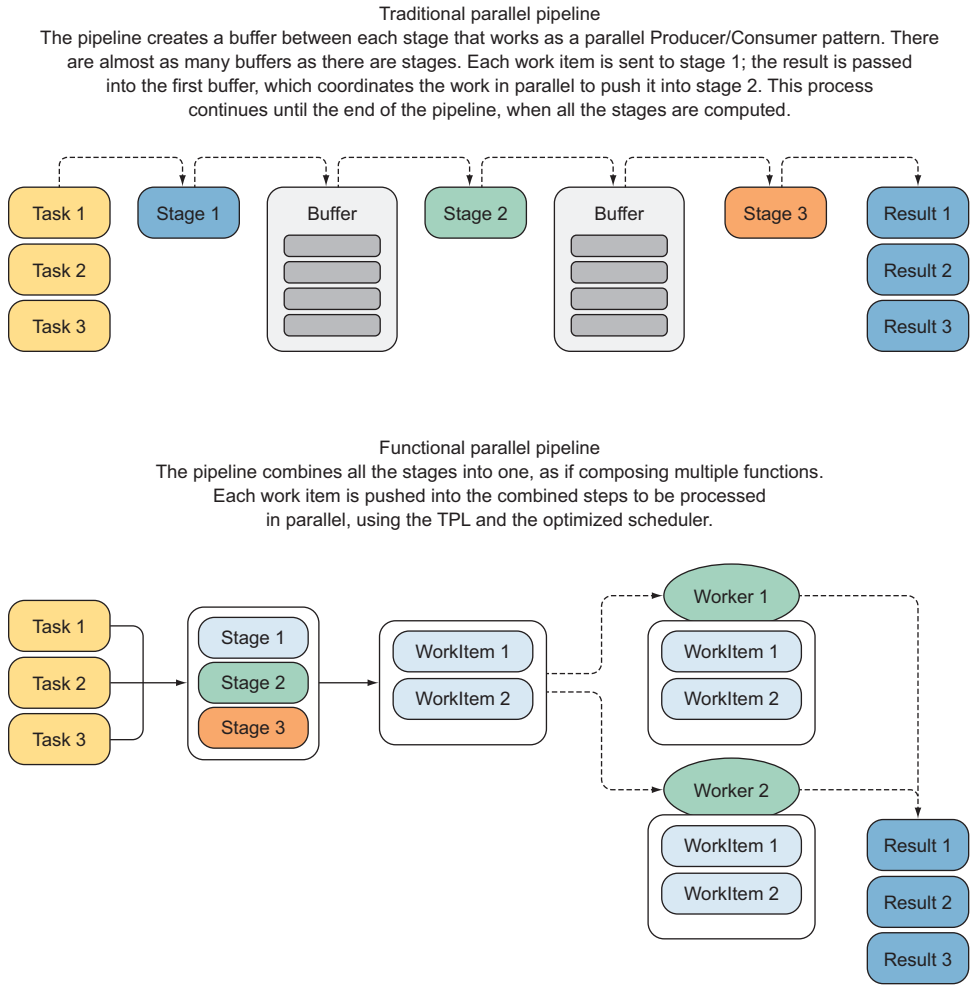


Figure 7.9 The traditional pipeline creates a buffer between each stage that works as a parallel Producer/Consumer pattern. There are almost as many buffers as there are number of stages. With this design, each work item to process is sent to the initial stage, then the result is passed into the first buffer, which coordinates the work in parallel to push it into the second stage. This process continues until the end of the pipeline when all the stages are computed. By contrast, the functional parallel pipeline combines all the stages into one, as if composing multiple functions. Then, using a `Task` object, each work item is pushed into the combined steps to be processed in parallel and uses the TPL and the optimized scheduler.

Listing 7.11 `IPipeline` interface

```
[<Interface>]
type IPipeline<'a,'b> =
    abstract member Then : Func<'b, 'c> -> IPipeline<'a,'c>

    abstract member Enqueue : 'a * Func<('a * 'b), unit)> -> unit
```

Interface that defines the pipeline contract

Uses a function to expose a fluent API approach

Uses a function to push new input to process into the pipeline

```
abstract member Execute : (int * CancellationToken) -> IDisposable
abstract member Stop : unit -> unit
```

Starts the pipeline execution

The pipeline can be stopped at any time; this function triggers the underlying cancellation token.

The function `Then` is the core of the pipeline, where the input function is composed of the previous one, applying a transformation. This function returns a new instance of the pipeline, providing a convenient and fluent API to build the process.

The `Enqueue` function is responsible for pushing work items into the pipeline for processing. It takes a `Callback` as an argument, which is applied at the end of the pipeline to further process the final result. This design gives flexibility to apply any arbitrary function for each item pushed.

The `Execute` function starts the computation. Its input arguments set the size of the internal buffer and a cancellation token to stop the pipeline on demand. This function returns an `IDisposable` type, which can be used to trigger the cancellation token to stop the pipeline. Here is the full implementation of the pipeline (the code to note is in bold).

Listing 7.12 Parallel functional pipeline pattern

```
[<Struct>]
type Continuation<'a, 'b>(input:'a, callback:Func<('a * 'b), unit) =
    member this.Input with get() = input
    member this.Callback with get() = callback

type Pipeline<'a, 'b> private (func:Func<'a, 'b>) as this =
    let continuations = Array.init 3 (fun _ -> new
                    BlockingCollection<Continuation<'a,'b>>(100))

    let then' (nextFunction:Func<'b,'c>) =
        Pipeline(func.Compose(nextFunction)) :> IPipeline<_,_>

    let enqueue (input:'a) (callback:Func<('a * 'b), unit>) =
        BlockingCollection<Continuation<_,_>>.AddToAny(continuations,
➥ Continuation(input, callback))

    let stop() = for continuation in continuations do continuation.
     CompleteAdding()
```

The Continuation struct encapsulates the input value for each task with the callback to run when the computation completes.

Initializes the BlockingCollection that buffers the work

Uses function composition to combine the current function of the pipeline with the new one passed and returns a new pipeline. The compose function was introduced in chapter 2.

The Enqueue function pushes the work into the buffer.

The BlockingCollection is notified to complete, which stops the pipeline.

```
let execute blockingCollectionPoolSize
 (cancellationToken:CancellationToken) =

    cancellationToken.Register(Action(stop)) |> ignore

    for i = 0 to blockingCollectionPoolSize - 1 do
        Task.Factory.StartNew(fun ( )->
            while (not <| continuations.All(fun bc -> bc.IsCompleted))
              && (not <| cancellationToken.IsCancellationRequested) do
                let continuation = ref
➥ Unchecked.defaultof<Continuation<_,_>>
                BlockingCollection.TakeFromAny(continuations,
➥ continuation)
                let continuation = continuation.Value
                continuation.Callback.Invoke(continuation.Input,
➥ func.Invoke(continuation.Input)),
        cancellationToken, TaskCreationOptions.LongRunning,
➥ TaskScheduler.Default) |> ignore

static member Create(func:Func<'a, 'b>) =
    Pipeline(func) :> IPipeline<_,_>

interface IPipeline<'a, 'b> with
   member this.Then(nextFunction) = then' nextFunction
   member this.Enqueue(input, callback) = enqueue input callback
   member this.Stop() = stop()
   member this.Execute (blockingCollectionPoolSize,cancellationToken) =
        execute blockingCollectionPoolSize cancellationToken
        { new IDisposable with member self.Dispose() = stop() }
```

Registers the cancellation token to run the stop function when it's triggered

Starts the tasks to compute in parallel

The static method creates a new instance of the pipeline.

The `Continuation` structure is used internally to pass through the pipeline functions to compute the items. The implementation of the pipeline uses an internal buffer composed by an array of the concurrent collection `BlockingCollection<Collection>`, which ensures thread safety during parallel computation of the items. The argument to this collection constructor specifies the maximum number of items to buffer at any given time. In this case, the value is `100` for each buffer.

Each item pushed into the pipeline is added to the collection, which in the future will be processed in parallel. The `Then` function is composing the function argument `nextFunction` with the function `func`, which is passed into the pipeline constructor. Note that you use the `Compose` function defined in chapter 2 in listing 2.3 to combine the functions `func` and `nextFunction`:

```
Func<A, C> Compose<A, B, C>(this Func<A, B> f, Func<B, C> g) =>
(n) => g(f(n));
```

When the pipeline starts the process, it applies the final composed function to each input value. The parallelism in the pipeline is achieved in the `Execute` function, which spawns one task for each `BlockingCollection` instantiated. This guarantees a buffer for running the thread. The tasks are created with the `LongRunning` option to schedule a dedicated thread. The `BlockingCollection` concurrent collection allows thread-safe access to the items stored using the static methods `TakeFromAny` and `AddToAny`, which

internally distribute the items and balance the workload among the running threads. This collation is used to manage the connection between the input and output of the pipeline, which behave as producer/consumer threads.

> **NOTE** Using `BlockingCollection`, remember to call `GetConsumingEnumerable` because the `BlockingCollection` class implements `IEnumerable<T>`. Enumerating over the blocking collection instance won't consume values.

The pipeline constructor is set as `private` to avoid direct instantiation. Instead, the static method `Create` initializes a new instance of the pipeline. This facilitates a fluent API approach to manipulate the pipeline.

This pipeline design ultimately resembles a parallel Produce/Consumer pattern capable of managing the concurrent communication between many-producers to many-consumers.

The following listing uses the implemented pipeline to refactor the `DetectFaces` program from the previous section. In C#, a fluent API approach is a convenient way to express and compose the steps of the pipeline.

Listing 7.13 Refactored `DetectFaces` code using the parallel pipeline

```
var files = Directory.GetFiles(ImagesFolder);

var imagePipe = Pipeline<string, Image<Bgr, byte>>
    .Create(filePath => new Image<Bgr, byte>(filePath))
    .Then(image => Tuple.Create(image, image.Convert<Gray, byte>()))
    .Then(frames => Tuple.Create(frames.Item1,
     CascadeClassifierThreadLocal.Value.DetectMultiScale(frames.Item2, 1.1,
         3, System.Drawing.Size.Empty)))
     .Then(faces =>{
           foreach (var face in faces.Item2)
               faces.Item1.Draw(face,
➥ new Bgr(System.Drawing.Color.BurlyWood), 3);
           return faces.Item1.ToBitmap();
         });

imagePipe.Execute(cancellationToken);

foreach (string fileName in files)
   imagePipe.Enqueue(file, (_, bitmapImage)
                                    => Images.Add(bitmapImage));
```

Constructs the pipeline using fluent API.

Starts the execution of the pipeline. The cancellation token stops the pipeline at any given time.

The iteration pushes (enqueues) the file paths into the pipeline queue, whose operation is non-blocking.

By exploiting the pipeline you developed, the code structure is changed considerably.

> **NOTE** The F# pipeline implementation, in the previous section, uses the `Func` delegate to be consumed effortlessly by C# code. In the source code of the book you can find the implementation of the same pipeline that uses F# functions in place of the .NET `Func` delegate, which makes it a better fit for projects completely built in F#. In the case of consuming native F# functions from C#, the helper extension method `ToFunc` provides support for interoperability. The `ToFunc` extension method can be found in the source code.

The pipeline definition is elegant, and it can be used to construct the process to detect the faces in the images using a nice, fluent API. Each function is composed step by step, and then the `Execute` function is called to start the pipeline. Because the underlying pipeline processing is already running in parallel, the loop to push the file path of the images is sequential. The `Enqueue` function of the pipeline is non-blocking, so there are no performance penalties involved. Later, when an image is returned from the computation, the `Callback` passed into the `Enqueue` function will update the result to update the UI. Table 7.1 shows the benchmark to compare the different approaches implemented.

Table 7.1 Benchmark processing of 100 images using four logical core computers with 16 GB RAM. The results, expressed in seconds, represent the average from running each design three times.

Serial loop	Parallel	Parallel continuation	Parallel LINQ combination	Parallel pipeline
68.57	22.89	19.73	20.43	17.59

The benchmark shows that, over the average of downloading 100 images for three times, the pipeline parallel design is the fastest. It's also the most expressive and concise pattern.

Summary

- Task-based parallel programs are designed with the functional paradigm in mind to guarantee more reliable and less vulnerable (or corrupt) code from functional properties such as immutability, isolation of side effects, and defensive copy. This makes it easier to ensure that your code is correct.
- The Microsoft TPL embraces functional paradigms in the form of using a continuation-passing style. This allows for a convenient way to chain a series of non-blocking operations.
- A method that returns `void` in C# code is a string signal that can produce side effects. A method with `void` as output doesn't permit composition in tasks using continuation.
- FP unmasks mathematical patterns to ease parallel task composition in a declarative and fluent programming style. (The monad and functor patterns are hidden in LINQ.) The same patterns can be used to reveal monadic operations with tasks, exposing a LINQ-semantic style.
- A functional parallel pipeline is a pattern designed to compose a series of operations into one function, which is then applied concurrently to a sequence of input values queued to be processed. Pipelines are often useful when the data elements are received from a real-time event stream.
- Task dependency is the Achilles heel of parallelism. Parallelism is restricted when two or more operations cannot run until other operations have completed. It's essential to use tools and patterns to maximize parallelism as much as possible. A functional pipeline, CPS, and mathematical patterns like monad are the keys.

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.

"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

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

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]