C# Multithreaded and Parallel Programming

Develop powerful C# applications to take advantage of today's multicore hardware

Rodney Ringler
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 “Data Parallelism”
- A synopsis of the book’s content
- More information on C# Multithreaded and Parallel Programming

About the Author

Rodney Ringler has 25 years' experience developing multitasking and parallel applications, with the last 10 focused on C# and .NET. He graduated cum laude from Clemson University with a BS degree in Computer Engineering. He then worked for 12 years in the fiber optic manufacturing industry on C-based real-time multitasking process control systems, where he went from being a developer to a project manager to an IT architect. After this, he spent 8 years running his own application development and hosting company focused on both .NET and open source technologies. He then spent several years as a consultant, working with companies in the retail, software, and manufacturing industries.

Currently, Rodney works as a senior .NET developer at a manufacturing company based in Charlotte, NC, and takes .NET and object-oriented programming classes at Central Piedmont Community College.

In his spare time, Rodney enjoys life in Lake Wylie, SC, with his wife and four children. I would like to thank my lovely wife, Rachel, and our four wonderful children, Matt, Amanda, Caroline, and Dave, for all of their patience and support while I spent many nights and weekends working on this project. I would also like to thank my parents, Fred and Joyce Ringler, for all their help, encouragement, love, and support.

I would like to offer one last thank you to all of the people at Fellowship Hall, especially Mike and Mahala, who helped save my life and taught me that miracles do happen as long as you show up and do your part.
C# Multithreaded and Parallel Programming

Welcome to C# Multithreaded and Parallel Programming. This book will take you through all of the ways to perform multithreaded and concurrent programming using the C# programming language and the .NET Framework. We will start with a description of what concurrent and parallel programming is, why it is important, and when you should implement it. We will then go through the different classes provided by the .NET Framework and the different design patterns commonly used when developing multithreaded applications.

Most modern machines have dual-core processors. This means that the present-day computer has the ability to multitask. Using multiple cores means your applications can process data faster and be more responsive to users. However, to fully exploit this in your applications, you need to write multithreaded code.

This will take us on a journey from the BackgroundWorker component, the Thread class, the Task Parallel Library, to the async and await keywords. We will also explore common design patterns such as Pipelining, producer-consumer, and the IAsyncResult interface.

Using the concurrent and parallel classes provided by .NET allows you to easily write powerful multithreaded applications. In the latest version of .NET, Microsoft has added the Task Parallel Library and the async keyword to make concurrent programming functionality much easier than using threads.

We will cover all aspects of developing multithreaded applications using the latest version of .NET in this book.
What This Book Covers

Chapter 1, Understanding Multiprocessing and Multiple Cores, covers the evolution of computer hardware from single-processor systems to multiprocessor and multiple-core systems. It will also discuss how the Windows Scheduler allots time to threads. This chapter then discusses design considerations for concurrency, how to take advantage of multiprocessor/multiple core systems, and the expected performance improvements these designs can realize. During this chapter, we will walk through a simple single-threaded and then multithreaded example to show how performance is improved.

Chapter 2, Looking at Multithreaded Classes – BackgroundWorker, enables us to examine the basics of multithreaded programming with the BackgroundWorker class. We will go through a WPF example showing how we can update a UI while processing is going on in the background. This chapter will discuss the basics of coordinating work between multiple processes and the concept of concurrency.

Chapter 3, Thread Class – Heavyweight Concurrency in C#, allows us to examine and work with the Thread class and namespace. We will learn how to create threads, coordinate between threads, share data between threads, and stop threads. This chapter will explore the idea of heavyweight concurrency versus lightweight concurrency, which will be explained in detail later in the book. We will focus on manually using and coordinating multiple threads in an application.

Chapter 4, Advanced Thread Processing, explains in detail the concept of heavyweight concurrency and working with multiple threads using the Thread class. This chapter will further expand the image processing application to demonstrate how to coordinate between threads and wait on threads to complete by avoiding deadlocks, locking, and error handling. This chapter will leave the reader with a clear understanding of how to develop applications with multiple threads and have complete control over their execution and interaction.

Chapter 5, Lightweight Concurrency – Task Parallel Library (TPL), introduces the Task Parallel Library and the next evolution of multithreaded programming in C#/.NET. Now that we have a full understanding of how to develop, manage, and control applications with many threads, we will learn how to take advantage of the Parallel namespace in .NET to do a lot of the heavy lifting for us. We will introduce the idea of lightweight concurrency by leveraging the Parallel classes that .NET now provides. This will allow us to focus more on designing an efficient and powerful application and less on coordinating individual threads.
Chapter 6, Task-based Parallelism, enables us to examine the Task Parallel Library and task parallelism. A task is an asynchronous set of operations that can be run concurrently with other tasks. We will examine designing an application as a series of tasks that can be performed in parallel. With the help of examples, we will demonstrate how to create, manage, and coordinate tasks. We will further examine additional topics with the Task Parallel Library and task parallelism. We will learn how to perform exception handling when running multiple tasks, how to schedule tasks under certain conditions, and how to cancel running tasks before they complete when needed.

Chapter 7, Data Parallelism, explores the concept of data parallelism. We will see how to perform the same operations on elements of a collection concurrently using the Task Parallel Library. The Parallel class has the For and ForEach loops, and we will show examples of each to demonstrate how they handle concurrent data processing. We will convert our image processing application from heavyweight to lightweight concurrency using the Task Parallel Library instead of the Thread class.

Chapter 8, Debugging Multithreaded Applications with Visual Studio, teaches us how to take full advantage of Visual Studio 2012 to debug our multithreaded applications. We will demonstrate using the Threads view, and the Tasks, Parallel Stacks, and Parallel Watch windows. We will finish with debugging our image processing application.

Chapter 9, Pipeline and Producer-consumer Design Patterns, helps us explore two of the most popular parallel patterns for development—Pipelining and producer-consumer. In Pipelining, we will see how to accomplish a parallel task where a simple parallel loop will not work due to data dependencies. The producer-consumer pattern allows a producer, which is generating results, to run along with the consumer so that the consumer can consume the results concurrently. We will expand our image processing application to implement these two patterns in combination.

Chapter 10, Parallel LINQ – PLINQ, details the benefits and functionality provided by Parallel LINQ (PLINQ). We will see how PLINQ speeds up traditional LINQs by separating the data source into sections and executing the query on each section. We will also discuss what kind of queries to use PLINQ for because not all queries will run faster using PLINQ.

Chapter 11, The Asynchronous Programming Model, explains the Asynchronous Programming Model (APM), which is a design pattern that is based on classes implementing the IAsyncResult interface. We will see how to begin and end asynchronous operations and use delegates to call methods asynchronously. This chapter will also cover the new async and await keywords and how to use them to implement an asynchronous design in your custom classes.
Concurrently performing a task or a set of operations on a collection of data is referred to as data parallelism. For example, if we have a list of files in a folder and we want to rename them all, we can create a For loop that goes through the collection and, during each iteration, the loop performs a rename command. We can also iterate through a collection datatype such as a List or DataView using a foreach statement. These are specialized For and ForEach statements that are part of the Task Parallel Library (TPL) in the System.Threading.Tasks.Parallel namespace.

The TPL provides the Parallel library to make it easy to perform concurrent operations on a dataset or data collection using the different overloads of the Parallel.For and Parallel.ForEach methods.

In this chapter, we will learn how to process items of a data source in parallel using the Parallel.For and Parallel.ForEach methods. We will also examine the ParallelLoopState class, which allows us to examine the results of a concurrent loop and perform actions with the results. Finally, we will learn how to cancel a concurrent loop before it has completed. In this chapter, we will cover:

- Parallel data processing with Parallel.For
- Parallel data processing of IEnumerable collections
- Using the results of concurrent data loops
- Canceling a parallel loop operation
Data Parallelism

Parallel loop processing
In this first section, we will examine the `Parallel.For` method and several variations of it. The most basic overload to this method takes a starting index, an ending index, and an `Action` delegate. In the `Parallel` method, the `Action` delegate can be implemented with a named method, anonymous method, or a lambda expression. The following is the basic syntax for each method:

```csharp
// Named method.
Parallel.For(0, i, DoWork);

// Anonymous method
Parallel.For(0, i, delegate(int j)
{
    // Do Work.
});

// Lambda expression.
Parallel.For(0, i, j =>
{
    // Do Work
});
```

In each example, the method or lambda expression takes a single parameter that is the iteration value. If you need more control over the execution of the concurrent loop, there are overload methods that take a `ParallelLoopState` parameter that is internally generated by .NET. We will talk about this later in the chapter, but it allows us to do things such as canceling a parallel loop or performing an action for each iteration of the loop once it is completed.

Here is a list of all of the overloads of the `Parallel.For` method:
Also, as well as a ParallelLoopState parameter, some overloads allow the method to return a thread-safe local variable.

Now, let's take a look at an example. We will build a simple Windows Presentation Foundation (WPF) application that takes an array of integer numbers, performs a calculation on them, and then updates the item in the array. This will be done concurrently instead of sequentially.

### How to do it

For this example, we will create a WPF application that allows the user to enter numbers in 10 boxes and click on a button. Once the button is clicked, it will concurrently take each number and multiply it by the numbers 1 through 10 and sum the results. The result of each calculation will be placed back in each box. Perform the following steps to do so:

1. Open Visual Studio and create a WPF application named ParallelMath1.
2. In the MainWindow.xaml design view, change the page title to ParallelMath: Title="ParallelMath" Height="350" Width="525"
3. Now, create 10 textbox controls and place them in the `MainWindow.xaml` file with the names, `tb1`, `tb2`, `tb3`, `tb4`, `tb5`, `tb6`, `tb7`, `tb8`, `tb9`, and `tb10` respectively. Also, set their `Text` properties to 0.

4. Now, place a button control on the `MainWindow.xaml` file and set the `Content` property to Calculate and the `Name` property to `btnCalculate`.

5. Next, create an event handler for the button’s click event named `btnCalculate_Click`.

6. Inside the class definition, place a line of code to create an array of 10 integers:
   ```csharp
   int[] numbers = new int[10];
   ```

7. Now, put the following code inside the `btnCalculate_Click` event handler:
   ```csharp
   int[] numbers = new int[10];
   numbers[0] = Convert.ToInt32(tb1.Text);
   numbers[1] = Convert.ToInt32(tb2.Text);
   numbers[2] = Convert.ToInt32(tb3.Text);
   numbers[3] = Convert.ToInt32(tb4.Text);
   numbers[4] = Convert.ToInt32(tb5.Text);
   numbers[6] = Convert.ToInt32(tb7.Text);
   numbers[7] = Convert.ToInt32(tb8.Text);
   numbers[8] = Convert.ToInt32(tb9.Text);
   numbers[9] = Convert.ToInt32(tb10.Text);
   
   Parallel.For(0, 9, CalculateNumbers);
   
   tb1.Text = numbers[0].ToString();
   tb2.Text = numbers[1].ToString();
   tb3.Text = numbers[2].ToString();
   tb4.Text = numbers[3].ToString();
   tb5.Text = numbers[4].ToString();
   tb6.Text = numbers[5].ToString();
   tb7.Text = numbers[6].ToString();
   tb8.Text = numbers[7].ToString();
   tb9.Text = numbers[8].ToString();
   tb10.Text = numbers[9].ToString();
   ```

8. Finally, create a method named `CalculateNumbers` and put the following code into it:
   ```csharp
   private void CalculateNumbers(int i)
   {
       int j = numbers[i];
   ```
for (int k = 1; k <= 10; k++)
{
    j *= k;
}

numbers[i] = j;

That should be all. Now, let's run our application and see what happens. Remember we have not put in any error handling. The application expects a number and only a number in each textbox when the Calculate button is clicked. If it is not there, the application will throw an argument out of range exception.

You should see results similar to this before you click on the button:
Data Parallelism

Now, let's enter 10 numbers into our textboxes so that the application looks something like the following:

Now, click on the Calculate button and you should see the following results very quickly since we are doing these calculations concurrently:
How it works

In the preceding exercise, we entered 10 numbers into 10 textboxes and then clicked on Calculate. The program then took each number, multiplied it by the numbers 1 through 10, and summed them together. It then placed the result back in the textbox it came from.

This was all done concurrently. Each textbox was processed in parallel. This may have been on 10 separate threads or fewer, depending on the hardware we run the program on. Unlike using threads directly by using the Parallel class and TPL, .NET manages the threadpool and maximizes how many threads to perform the concurrent operation on, using the processing cores available on the machine.

Let's look at how the concurrent loop is executed. It is the single command Parallel.For(0, 9, CalculateNumbers);. This command queues 10 tasks to the threadpool and each task will execute the CalculateNumbers method with an integer parameter.

Now, let's look at the Parallel.ForEach command.

Data parallelism on collections using Parallel.ForEach

The form of data parallelism that I find most helpful is performing concurrent operations on collections of data. This allows us to take data collections such as lists, dataviews, dictionaries, and so on and perform a task on each item in the collection in parallel with a single line of code! This makes using data parallelism simpler; you have to do nothing more than normal data processing. This is one of the reasons why the TPL is such a wonderful enhancement to .NET.

Much like the Parallel.For method, the structure of Parallel.ForEach looks like the following:

   Parallel.ForEach(dataCollection, item => DoWork(item));

There are two parameters in the most basic version of this method. There is a data collection and an Action delegate to perform a task on an item of the dataCollection. The Action delegate takes a single parameter that is an item in the collection.
The following are all of the different overloads of the `ForEach` method:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;, long=&quot;step&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;, long=&quot;step&quot;, long=&quot;initial&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;, long=&quot;step&quot;, long=&quot;initial&quot;, long=&quot;increment&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;, long=&quot;step&quot;, long=&quot;initial&quot;, long=&quot;increment&quot;, long=&quot;min&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;, long=&quot;step&quot;, long=&quot;initial&quot;, long=&quot;increment&quot;, long=&quot;min&quot;, long=&quot;max&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
<tr>
<td><code>ForEach&lt;TSource&gt;(IEnumerable&lt;TSource&gt;, Action&lt;TSource&gt;, ParallelLoopState, int=&quot;start&quot;, int=&quot;end&quot;, long=&quot;step&quot;, long=&quot;initial&quot;, long=&quot;increment&quot;, long=&quot;min&quot;, long=&quot;max&quot;, long=&quot;initialIndex&quot;)</code></td>
<td>Executes a <code>foreach</code> loop on <code>IEnumerable&lt;TSource&gt;</code> in which iterations may run in parallel, and the state of the loop can be monitored and manipulated.</td>
</tr>
</tbody>
</table>


As you can see, there are many different overloads to this method. They allow us to use a `ParallelLoopState` object or a thread-safe local variable.
We will focus on the simple form of just performing concurrent processing on a data collection. To further reiterate this point, let's revisit a project we worked on earlier in the book. In Chapter 4, Advanced Thread Processing, we wrote an application that took a JPG image, divided it into separate bitmaps, and then performed parallel functions on each bitmap to find old stars. It then reassembled the individual bitmaps back into a single image.

We will rewrite this application using data parallelism and the TPL instead of threads directly. This will demonstrate how TPL can simplify multithreaded code development.

No longer do we have to manage threads (start them, wait on them to complete, or track them). We no longer have to manage the number of processing cores our machine has to maximize performance without starting too many individual threads. All we have to do is separate our large image into a collection of smaller bitmaps and use a Parallel.ForEach concurrent loop to process each bitmap. That's it. Let's get started.

How to do it

We will take our original OldStarsFinder Windows Form application and change it. To do this let's perform the following steps:

1. First, let's open our OldStarsFinder application in Visual Studio.
2. Let's add a new using statement so we can access the Parallel library:
   ```csharp
   using System.Threading.Tasks;
   ```
3. At the beginning of the class definition, remove all of the old variable declarations and replace them with just these:
   ```csharp
   private int Count = 0;
   // Number of bitmaps to break the original into and add to
   // the list of bitmaps.
   private List<Bitmap> BitmapList;
   // List of bitmaps to use the ParallelForEach on.
   Bitmap OriginalBitmap;
   private String prsOldStarsCount = "0";
   // Old stars count using a lock to protect thread safety.
   ```
4. Replace the `CropBitmap` method with this method definition:

   ```csharp
   private Bitmap CropBitmap(Bitmap proBitmap, Rectangle proRectangle)
   {
       // Create a new bitmap copying the portion of the
       // original defined by proRectangle
       // and keeping its PixelFormat
       Bitmap loCroppedBitmap = proBitmap.Clone(proRectangle, proBitmap.PixelFormat);

       return loCroppedBitmap;
   }
   ```

5. You can leave the `IsOldStar` method as is.

6. Change the `ThreadOldStarsFinder` method to look like this:

   ```csharp
   private void ThreadOldStarsFinder(Bitmap loBitmap)
   {
       int liRow;                  // The pixel matrix (bitmap) row number (Y)
       int liCol;                  // The pixel matrix (bitmap) col number (X)
       Color loPixelColor;         // The pixel color

       // Iterate through each pixel matrix (bitmap) rows
       for (liRow = 0; liRow < loBitmap.Height; liRow++)
       {
           // Iterate through each pixel matrix (bitmap) cols
           for (liCol = 0; liCol < loBitmap.Width; liCol++)
           {
               // Get the pixel Color for liCol and liRow
               loPixelColor = loBitmap.GetPixel(liCol, liRow);
               if (IsOldStar(loPixelColor))
               {
                   // The color range corresponds to an old star
                   // Change its color to a pure blue
                   loBitmap.SetPixel(liCol, liRow, Color.Blue);
               }
           }

           lock (prsOldStarsCount)
           {
               int i = Convert.ToInt32(prsOldStarsCount);
               i = Convert.ToInt32(prsOldStarsCount);
           }
       }
   }
   ```
i = i + 1;
prsOldStarsCount = i.ToString();
}
}
}

7. Then change the ShowBitmapWithOldStars method to look like this:

```csharp
private void ShowBitmapWithOldStars()
{
    int liThreadNumber;
    // Each bitmap portion
    Bitmap loBitmap;
    // The starting row in each iteration
    int liStartRow = 0;

    // Calculate each bitmap's height
    int liEachBitmapHeight = ((int)(OriginalBitmap.Height / Count)) + 1;

    // Create a new bitmap with the whole width and height
    loBitmap = new Bitmap(OriginalBitmap.Width,
                          OriginalBitmap.Height);
    Graphics g = Graphics.FromImage((Image)loBitmap);
    g.InterpolationMode = System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;

    for (liThreadNumber = 0; liThreadNumber < Count;
         liThreadNumber++)
    {
        // Draw each portion in its corresponding absolute starting row
        g.DrawImage(BitmapList[liThreadNumber], 0, liStartRow);
        // Increase the starting row
        liStartRow += liEachBitmapHeight;
    }

    // Show the bitmap in the PictureBox picStarsBitmap
    picStarsBitmap.Image = loBitmap;
    //picStarsBitmap.Image.Save("c:\packt\resulting_image.png", ImageFormat.Png);

    tbCount.Text = prsOldStarsCount;
}
8. Now, change the butFindOldStars_Click event handler to have the following code:

```csharp
private void butFindOldStars_Click(object sender, EventArgs e)
{
    Count = Convert.ToInt32(tbTasks.Text);
    OriginalBitmap = new Bitmap(picStarsBitmap.Image);
    BitmapList = new List<Bitmap>(Count);
    int StartRow = 0;
    int EachBitmapHeight = ((int)(OriginalBitmap.Height / Count)) + 1;
    int HeightToAdd = OriginalBitmap.Height;
    Bitmap loBitmap;

    // Breakup the bitmap into a list of bitmaps.
    for (int i = 0; i < Count; i++)
    {
        if (EachBitmapHeight > HeightToAdd)
            // The last bitmap height perhaps is less than the other bitmaps height
            EachBitmapHeight = HeightToAdd;

        loBitmap = CropBitmap(OriginalBitmap, new Rectangle(0, StartRow, OriginalBitmap.Width, EachBitmapHeight));
        HeightToAdd -= EachBitmapHeight;
        StartRow += EachBitmapHeight;
        BitmapList.Add(loBitmap);
    }

    // Iterate through the list of bitmaps with the Parallel.ForEach command.
    Parallel.ForEach(BitmapList, item =>
    ThreadOldStarsFinder(item));

    ShowBitmapWithOldStars();
}
```
9. Then add a label control with the text, *Number of bitmaps to divide into for processing:*.  
10. Also, add a textbox control and set its `Name` property to `tbTasks`. This will be used to allow you to designate the number of sections you want the bitmap divided into.  
11. Finally, we remove the `butFindOldStarsBatch` button because we do not need it in this application.

That should be all you need to do to run this application using data parallelism with the Task Parallel Library.

Let’s compile and run our application. You should get something like this:
Now, enter the number of bitmaps to divide the image into and click on the **Old Star Finder** button. The application will now look like this:

What just happened? We entered 8 for the number of bitmaps to divide into. The application splits the JPG image into 8 equal-sized bitmaps and then into a list collection of bitmaps. Then it concurrently processes each bitmap looking for old stars. Finally, it reassembles the bitmaps into one image and redisplayes it.

Let's take a closer look at what just happened.

**How it works**

If you compare the two versions of the program, you will see that the second version is much simpler with less code. If we examine the `butOldStarsFinder_Click` event handler method, we will see most of the work. First, we divide our image up into a `List` collection of smaller bitmaps based on the number we entered. Here is the code that does this:

```csharp
// Breakup the bitmap into a list of bitmaps.
for (int i = 0; i < Count; i++)
{
```
if (EachBitmapHeight > HeightToAdd)
{
    // The last bitmap height perhaps is less than
    // the other bitmaps height
    EachBitmapHeight = HeightToAdd;
}

loBitmap = CropBitmap(OriginalBitmap, new Rectangle(0, StartRow, OriginalBitmap.Width, EachBitmapHeight));
HeightToAdd -= EachBitmapHeight;
StartRow += EachBitmapHeight;
BitmapList.Add(loBitmap);
}

Next, we take our list collection, BitmapList, and use it in a parallel ForEach command in this line of code:

    Parallel.ForEach(BitmapList, item => ThreadOldStarsFinder(item));

Finally, when this loop has completed, we display the image with the old stars with this method:

    ShowBitmapWithOldStars();

That is it. We no longer have to find out how many cores the processor has and create that many threads. No matter how many items there are in our collection, .NET maximizes the threads in the threadpool to achieve optimal performance. It will create threads if needed or reuse existing threads if possible. This saves on the overhead of starting more threads than can be effectively used by the number of cores in the machine.

You can now see why writing multithreaded code using TPL is called lightweight concurrency. This version of the Old Stars Finder is definitely "lighter" on the code and logic than the previous version written directly with threads or heavyweight concurrency.

**Canceling a parallel loop**

Now that we have learned how to execute a parallel loop using a Parallel class library, let's take a look at how we can stop or break a loop if needed. With a normal For or ForEach loop we can use a Continue command to break from a loop.
Data Parallelism

If we break from a parallel loop, then we complete all iterations on the threads that are currently executing and then stop. If we stop a parallel loop, then we stop all currently running iterations of the loop as soon as possible, but we do not run them to completion. In either case, we will not schedule tasks on the threadpool for the rest of the iterations of the parallel loop that we are yet to get started with.

To perform a break or a stop of a parallel loop, we need to use the `ParallelLoopState` object. This means that we have to use one of the overloads or the `Parallel.For` or `Parallel.ForEach` method that takes a `ParallelLoopState` parameter.

What is the `ParallelLoopState` class? This class cannot be instantiated in your user code. It is provided by the TPL and .NET and hence is a special class. This object provides your parallel loop with a mechanism to interact with other iterations in the loop.

The `break` and `stop` methods are the methods you will use most often, as well as the `IsStopped` and `IsExceptional` properties. These properties allow you to check whether any iteration of the loop has called `Stop` or thrown an exception.

Now, we will take our `ParallelMath1` example and change it to stop the loop after seven iterations. This is arbitrary, for example purposes. But in a real example, there are many conditions where you will want to break or stop from a parallel loop.

How to do it

We just need to make a few adjustments to our previous program. Let's start by opening the `ParallelMath1` WPF application in Visual Studio and making the following changes:

1. Create a new method called `CalculateNumbers2` and place the following code into it:

   ```csharp
   private void CalculateNumbers2(int i, ParallelLoopState pls)
   {
       int j = numbers[i];

       if (i < 7)
       {
           for (int k = 1; k <= 10; k++)
           {
               j *= k;
           }
        
           numbers[i] = j;
       }
   }
   ```
else
{
    pls.Stop();
    return;
}

2. Then change our `Parallel.For` command to call this new method:

```
Parallel.For(0, 9, CalculateNumbers2);
```

That's it. Now, let's run our application and put numbers in each of the boxes so that it looks like the following screenshot:
Now, click on the **Calculate** button and your results should look like the following screenshot:

What do you see? Yes, after seven iterations of the parallel loop, the loop is stopped and the last three iterations are not finished. Let's examine why.

**How it works**

By adding the `ParallelLoopState` parameter to the method called by the parallel `For` method, we actually change the overload of the method that is called. We are now calling this overload:

| S | For(int i, int j, Action<int, ParallelLoopState>) | Executes a `for` in Visual Basic; loop in which iterations may run in parallel and the state of the loop can be monitored and manipulated. |
You will notice that we do not create this `ParallelLoopState` variable and pass it into the `CalculateNumbers2` method. It is done by .NET and we can just use it. Pretty handy!

Now, in our `Action` delegate, `CalculateNumbers2`, we call the `Stop` method of this object using the following command:

```csharp
pls.Stop();
```

Once this method is called, the rest of the iterations of the loop are not performed and the loop completes with the iterations it has already completed.

This is not a very practical example—why execute the loop for 10 iterations and just stop after seven? Why not execute the parallel for seven iterations in the first place? This is just an example for demonstration purposes. In your applications, you will find many conditions by which you will want to exit a parallel loop before completing all iterations, just like with a normal `For` loop.

## Handling exceptions in parallel loops

Well, no matter how hard we try to write error-free code, the real world intervenes and inevitably there are exceptions that our code will encounter, for example, file not found, argument out of range, and so on. When we are processing a parallel loop command, since all of the iterations are running on potentially different threads, we need a way to gather all of the exceptions that any of the iterations of the loop may produce.

.NET provides the `AggregateException` class for just this purpose. This allows us to collect all of the exceptions into the `AggregateException` object and then "catch" it once the loop has completed.

Think of this like you would a normal error-handling situation. Typically, you put a `try` block around a section of the code and then a `catch` block after it to process any exceptions that occurred in the `try` block of the code. This will behave the same way. We will put a `try` block around our parallel loop command and then catch the exceptions that occur in all of the iterations of the loop. Only in this case, our `catch` block will catch the `AggregateException`, which is just a collection of exceptions.

Also, in our `Action` delegate, we will catch any exceptions that occur and add them to the `AggregateException` object.

That is all there is to it! Let's try this for ourselves by modifying our `ParallelMath1` project to generate an exception if any of the iterations produce a sum of over 5 million.
**Data Parallelism**

**How to do it**

To start, let's open the `ParallelMath1` project in Visual Studio and make the following changes:

1. We will be using a Concurrent queue to collect the exceptions, so add this using statement:
   ```csharp
   using System.Collections.Concurrent;
   ```

2. Then, we need to declare our `ConcurrentQueue` instance where it is visible to the entire class. Add the following line right after the declaration on the number's integer array:
   ```csharp
   ConcurrentQueue<Exception> exceptions = new ConcurrentQueue<Exception>();
   ```

3. Next, we will add a new method to be called by the `Action` delegate of our parallel loop command. We will call this method `CalculateNumbers3`. Add the following code to this method:
   ```csharp
   private void CalculateNumbers3(int i, ParallelLoopState pls)
   {
       int j = numbers[i];
       try {
           for (int k = 1; k <= 10; k++)
           {
               j *= k;
               if (j > 5000000) throw new ArgumentException(String.Format("The value of text box {0} is {1}. ", i, j));
           }
       }
       catch (Exception e)
       {
           exceptions.Enqueue(e);
       }
       numbers[i] = j;
   }
   ```
4. Then, let’s alter our btnCalculate_Click event handler. Change the code between the population of the numbers array and the population of the textboxes to include the following lines of code:

```csharp
try
{
    Parallel.For(0, 10, CalculateNumbers3);

    if (exceptions.Count > 0) throw new AggregateException(exceptions);
}

catch (AggregateException ae)
{
    // This is where you can choose which exceptions to handle.
    foreach (var ex in ae.InnerExceptions)
    {
        if (ex is ArgumentException)
        {
            tbMessages.Text += ex.Message;
            tbMessages.Text += "\n";
        }
        else
        {
            throw ex;
        }
    }
}
```

5. Finally, in the MainWindow.xaml designer view, add a textblock control; set its Name property to tbMessages and its Text property to an empty string.

Now our changes are complete. When any of the iterations of the parallel loop reach above 5 million, we will throw an exception. All of the exceptions will be collected into a ConcurrentQueue and added to the AggregateException object. Once the parallel loop execution has completed, we will process the AggregateException, if there are any, and write their exception messages to the Messages textblock.
Build and run your application. Now, enter numbers in each of the boxes. You should have a screen that looks like the following screenshot:

Now, click on the **Calculate** button and you should see results that look like the following screenshot:
As you can see from the output, every box that has a total that goes over 5 million has a line printed in our Messages textblock. In this example, any number that goes over 5 million throws an exception. Once all of the iterations of the parallel loop have completed, we process these exceptions and print their messages to the tbMessages textblock.

**How it works**

The first thing we changed was adding an object that is in a concurrent queue, to hold all of our exceptions. This is done with this command:

```csharp
ConcurrentQueue<Exception> exceptions = new
ConcurrentQueue<Exception>();
```

Next, in our Action delegate of the parallel for command, which in this version executes CalculateNumbers3, we check for numbers greater than 5 million through an exception using the following command:

```csharp
if (j > 5000000) throw new ArgumentException(String.Format("The value of text box {0} is {1}. ", i, j));
```

We then catch this exception within the delegate and add it to our concurrent queue of exception objects using these statements:

```csharp
catch(Exception e)
{
    exceptions.Enqueue(e);
}
```

We do this because we do not want to interrupt other iterations of the parallel loop running on different threads. We want the loop to finish processing and then handle the exceptions. Since each iteration of the loop is running concurrently and is designed not to affect other iterations, we should not interrupt all of the iterations because one has an issue.

Once the loop has completed, we want to check for any exceptions and process them. Here is the code that handles this functionality:

```csharp
try
{
    Parallel.For(0, 10, CalculateNumbers3);

    if (exceptions.Count > 0) throw new
    AggregateException(exceptions);
}
catch (AggregateException ae)
{  
  // This is where you can choose which exceptions to handle.
  foreach (var ex in ae.InnerExceptions)
  {
    if (ex is ArgumentException)
    {
      tbMessages.Text += ex.Message;
      tbMessages.Text += "\r\n";
    }
    else
    {
      throw ex;
    }
  }
}

If we see any exceptions in our `ConcurrentQueue` object, we throw an `AggregateException` and give it the whole queue of exceptions. Then we catch this `AggregateException` and process all of the exceptions it contains.

We could have also performed different actions based on the exception type of each exception. You can play with your code and try this.

### Using thread-local variables in parallel loops

Once you are comfortable using parallel loops, stopping them, and performing exception handling with them, let's talk about how we can use thread-local variables to better coordinate results. If we want to sum the results of all the iterations of a parallel loop, how would we do that?

From what we have learned so far, we would create a class variable before the loop and access it by each iteration of the loop using a `lock` statement so that it remains thread-safe. This takes overhead and coordination time. To improve performance, we can implement our parallel loop using a thread-local variable.

The `Parallel.For` and `Parallel.ForEach` loops both have overloads that implement a thread-local variable. What do we mean by a thread-local variable? This is a variable whose scope lasts the duration of the parallel loop, from just before the first iteration starts to the completion of the last iteration. Each iteration of the loop gets its own copy of the thread-local variable.

In these overloads of the parallel loop methods, there are three functions that get passed to the loop as well as the iteration parameters. For a `For` loop, the iteration parameters are the starting and ending values of the loop index, and for the `ForEach` loop, it is the source collection.
The first function will initialize the thread-local variable. The second function is the `Action` delegate that the loop performs. The third function is the `Action` delegate that gets executed when all iterations of the loop have completed, and it receives the thread-local variable for each loop iteration. It can then process the results, which usually means combining the results.

Let's examine one of the `ForEach` overloads:

```csharp
ForEach<TSource, TLocal>(IEnumerable<TSource>, Func<TLocal>, Func<TSource, ParallelLoopState, TLocal, TLocal>, Action<TLocal>)
```

Let's dissect this for a minute. We will take each piece of the method definition and explain its role:

- **ForEach<TSource, TLocal>:** `TSource` is the datatype of the source collection, and `TLocal` is the datatype of the thread-local variable.
- **IEnumerable<TSource>:** This is the source collection. Since we are using a `ForEach` example, the source collection has to be `IEnumerable`.
- **Func<TLocal>:** This is the first function; it initializes the thread-local variable.
- **Func<TSource, ParallelLoopState, TLocal, TLocal>:** This is the second function; it is the `Action` delegate that is performed by each iteration of the loop.
- **Action<TLocal>:** This is the third function; it is the `Action` delegate that is performed on the local state of each iteration.

Even though this is a fairly straightforward concept, it leads itself to a method overload that looks very complicated. To make sure we understand, let's go to our `ParallelMath1` project and see how it works in the sample application we have been building.

**How to do it**

To use a thread-local variable to sum up our textboxes once we have performed our parallel loop on them, let's open our `ParallelMath1` project and make a few changes:

1. In the `MainWindow.xaml` file in the designer view, let's add a label control and set the `Content` property to `Sum:`.
2. Now, let's add a textbox control beside it and set the `Name` property to `tbSum` and make the `Text` property empty.
3. In the MainWindow.xaml.cs file, add the following using statement so that we can use the Interlocked.Add method:
   ```csharp
   using System.Threading;
   ```

4. Also, add a class variable below our ConcurrentQueue declaration for a sum variable we will call total:
   ```csharp
   long total = 0;
   ```

5. Then, comment out our current Parallel.For command because we are going to use the new overloaded version needed for thread-local variables:
   ```csharp
   //Parallel.For(0, 10, CalculateNumbers3);
   ```

6. It is easier for this version of a Parallel.For to use a lambda expression instead of the named methods for the Action delegates. So, use the following Parallel.For command:
   ```csharp
   Parallel.For<long>(0, 10,
       () => 0,
       (i, loop, subtotal) =>
       {
           int j = numbers[i];
           for (int k = 1; k <= 10; k++)
           {
               j *= k;
           }
           numbers[i] = j;
           subtotal += j;
           return subtotal;
       },
       (finalResult) => Interlocked.Add(ref total, finalResult))
   ```

7. Finally, right after this statement, add the following statement so that we can see the total on the user interface:
   ```csharp
   tbSum.Text = total.ToString();
   ```

That is all the changes we need to make so that we can use our thread-local variable with the Parallel.For loop to calculate the sum of our textbox.

Once these changes have been made, build and run the application. Enter numbers in the textboxes and you should have a screen that looks like the following screenshot:
Now, click on the Calculate button and see what happens. The results should look like the following screenshot:
Data Parallelism

As you can see from the example, we now have a sum of all of the boxes once the parallel loop has processed them. We are able to do this without having to continually lock the class variable in each iteration when it wants to update the loop. We can do the summing once at the end of the parallel loop using the thread-local value from each iteration of the loop.

Now, let's examine what just happened

How it works

Just like in the previous versions of this project, we take the numbers in 10 different textboxes and multiply them by the numbers 1 through 10 and sum them. The result is then put back in the textbox. But this time, we take the new results in the 10 textboxes and sum them, and the final total is displayed in the tbSum textbox.

The only real difference in this version is the Parallel.For command. Let's take a deeper look at it:

```csharp
Parallel.For<long>(0, 10,
    () => 0,
    (i, loop, subtotal) =>
    {
        int j = numbers[i];
        for (int k = 1; k <= 10; k++)
        {
            j *= k;
        }

        numbers[i] = j;
        subtotal += j;
        return subtotal;
    },
    (finalResult) => Interlocked.Add(ref total, finalResult)
);```

First, we now have a TResult parameter that is a long Parallel.For<long>. This tells us that the Action delegate that each iteration of the loop executes will return a value with the datatype long. This Action delegate is implemented as a lambda expression this time and is the fourth parameter of our Parallel.For method:

```csharp
(i, loop, subtotal) =>
{
    int j = numbers[i];
    for (int k = 1; k <= 10; k++)
    ```
{  
    j *= k;
}

numbers[i] = j;
subtotal += j;
return subtotal;

Let's create a back up; the first two parameters are the starting and ending indices of our iteration, 0 and 10. The third parameter is our Action delegate that initializes the thread-local variable. It is implemented with a lambda expression:

() => 0

Then, our final parameter to the Parallel.For method is the Action delegate that is executed on each iteration's thread-local variable:

(finalResult) => Interlocked.Add(ref total, finalResult)

We chose to use lambda expression for the three Action delegates in this example instead of named or anonymous methods because it is easier for us to see what is going on and what is being passed to what. However, we can use named methods to achieve the same results.

**Summary**

In this chapter, we covered all aspects of imperative data parallelism. In Chapter 10, Parallel LINQ – PLINQ, we will cover declarative data parallelism with PLINQ discussion. Data parallelism using TPL in .NET really comes down to performing parallel loops using the Parallel.For and Parallel.ForEach methods. These parallel loops allow us to iterate through a set or collection of data and perform the same function on each member of the set concurrently.

We learned how to perform a parallel loop on a set of data using Parallel.For and a collection of data using Parallel.ForEach. We then saw how to stop or break from a loop when a particular condition was reached; for this we used the ParallelLoopState object that .NET can generate.

Next, we explored error handling with parallel loops and the AggregateException object. We learned how to process all of the exceptions that might occur during the different iterations of the loop without affecting the other iterations.
Data Parallelism

In the last section, we saw how to use thread-local variables in our loops to have a thread-safe local copy of a variable and then use the results from all of these local copies at the end of the loop processing.

In the next chapter, we will take some time and explore the Visual Studio Debugger and the features it provides for debugging a parallel application that has multiple threads running at once.
Where to buy this book

You can buy C# Multithreaded and Parallel Programming from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.