PARALLELISM IN .NET

Using Task Parallel Library

Abstract

This series introduces new features in .NET 4 for concurrent programming. It takes a pattern and design first approach to introducing new features within both the Task Parallel Library and PLINQ, explaining how to approach introducing concurrency within an application.

Reed Copsey, Jr.

http://reedcopsey.com/series/parallelism-in-net4/

Contents

Introduction

Parallel programming is something that every professional developer should understand, but is rarely discussed or taught in detail in a formal manner. Software users are no longer content with applications that lock up the user interface regularly, or take large amounts of time to process data unnecessarily. Modern development requires the use of parallelism. There is no longer any excuses for us as developers.

Learning to write parallel software is challenging. It requires more than reading that one chapter on parallelism in our programming language book of choice…

Today's systems are no longer getting faster with each generation; in many cases, newer computers are actually slower than previous generation systems. Modern hardware is shifting towards conservation of power, with processing scalability coming from having multiple computer cores, not faster and faster CPUs. Our CPU frequencies no longer double on a regular basis, but Moore's Law is still holding strong. Now, however, instead of scaling transistors in order to make processors faster, hardware manufacturers are scaling the transistors in order to add more discrete hardware processing threads to the system.

This changes how we should think about software. In order to take advantage of modern systems, we need to redesign and rewrite our algorithms to work in parallel. As with any design domain, it helps tremendously to have a common language, as well as a common set of patterns and tools.

For .NET developers, this is an exciting time for parallel programming. Version 4 of the .NET Framework is adding the Task Parallel Library. This has been back-ported to .NET 3.5sp1 as part of the Reactive Extensions for .NET and is available for use today in both .NET 3.5 and .NET 4.0 beta.

In order to fully utilize the Task Parallel Library and parallelism, both in .NET 4 and previous versions, we need to understand the proper terminology. For this series, I will provide an introduction to some of the basic concepts in parallelism and relate them to the tools available in .NET.

Part 1, Decomposition

The first step in designing any parallelized system is Decomposition. Decomposition is nothing more than taking a problem space and breaking it into discrete parts. When we want to work in parallel, we need to have at least two separate things that we are trying to run. We do this by taking our problem and decomposing it into parts.

There are two common abstractions that are useful when discussing parallel decomposition: Data Decomposition and Task Decomposition. These two abstractions allow us to think about our problem in a way that helps leads us to correct decision making in terms of the algorithms we'll use to parallelize our routine.

To start, I will make a couple of minor points.

- I'd like to stress that Decomposition has nothing to do with specific algorithms or techniques. It's about how you approach and think about the problem, not how you solve the problem using a specific tool, technique, or library. Decomposing the problem is about constructing the appropriate mental model: once this is done, you can choose the appropriate design and tools, which is a subject for future posts.
- Decomposition, being unrelated to tools or specific techniques, is not specific to .NET in any way. This should be the first step to parallelizing a problem and is valid using any framework, language, or toolset. However, this gives us a starting point – without a proper understanding of decomposition, it is difficult to understand the proper usage of specific classes and tools within the .NET framework.

Data Decomposition is often the simpler abstraction to use when trying to parallelize a routine. In order to decompose our problem domain by data, we take our entire set of data and break it into smaller, discrete portions, or chunks. We then work on each chunk in the data set in parallel.

This is particularly useful if we can process each element of data independently of the rest of the data. In a situation like this, there are some wonderfully simple techniques we can use to take advantage of our data. By decomposing our domain by data, we can very simply parallelize our routines. In general, we, as developers, should be always searching for data that can be decomposed.

Finding data to decompose if fairly simple, in many instances. Data decomposition is typically used with collections of data. Any time you have a collection of items and you're going to perform work on or with each of the items, you potentially have a situation where parallelism can be exploited. This is fairly easy to do in practice: look for iteration statements in your code, such as for and foreach.

Granted, every for loop is not a candidate to be parallelized. If the collection is being modified as it's iterated, or the processing of elements depends on other elements, the iteration block may need to be processed in serial. However, if this is not the case, data decomposition may be possible.

Let's look at one example of how we might use data decomposition. Suppose we were working with an image and we were applying a simple contrast stretching filter. When we go to apply the filter, once we know the minimum and maximum values, we can apply this to each pixel independently of the other pixels. This means that we can easily decompose this problem based off data – we will do the same operation, in parallel, on individual chunks of data (each pixel).

Task Decomposition, on the other hand, is focused on the individual tasks that need to be performed instead of focusing on the data. In order to decompose our problem domain by tasks, we need to think about our algorithm in terms of discrete operations, or tasks, which can then later be parallelized.

Task decomposition, in practice, can be a bit more tricky than data decomposition. Here, we need to look at what our algorithm actually does and how it performs its actions. Once we have all of the basic steps taken into account, we can try to analyze them and determine whether there are any constraints in terms of shared data or ordering. There are no simple things to look for in terms of finding tasks we can decompose for parallelism; every algorithm is unique in terms of its tasks, so every algorithm will have unique opportunities for task decomposition.

For example, say we want our software to perform some customized actions on startup, prior to showing our main screen. Perhaps we want to check for proper licensing, notify the user if the license is not valid and also check for updates to the program. Once we verify the license and that there are no updates, we'll start normally. In this case, we can decompose this problem into tasks – we have a few tasks, but there are at least two discrete, independent tasks (check licensing, check for updates) which we can perform in parallel. Once those are completed, we will continue on with our other tasks.

One final note – Data Decomposition and Task Decomposition are not mutually exclusive. Often, you'll mix the two approaches while trying to parallelize a single routine. It's possible to decompose your problem based off data, then further decompose the processing of each element of data based on tasks.

This just provides a framework for thinking about our algorithms and for discussing the problem.

Part 2, Simple Imperative Data Parallelism

In my discussion of Decomposition of the problem space, I mentioned that Data Decomposition is often the simplest abstraction to use when trying to parallelize a routine. If a problem can be decomposed based off the data, we will often want to use what MSDN refers to as Data Parallelism as our strategy for implementing our routine. The Task Parallel Library in .NET 4 makes implementing Data Parallelism, for most cases, very simple.

Data Parallelism is the main technique we use to parallelize a routine which can be decomposed based off data. Data Parallelism refers to taking a single collection of data and having a single operation be performed concurrently on elements in the collection.

One side note here: Data Parallelism is also sometimes referred to as the Loop Parallelism Pattern or Loop-level Parallelism. In general, for this series, I will try to use the terminology used in the MSDN Documentation for the Task Parallel Library. This should make it easier to investigate these topics in more detail.

Once we've determined we have a problem that, potentially, can be decomposed based on data, implementation using Data Parallelism in the TPL is quite simple. Let's take our example from the Data Decomposition discussion – a simple contrast stretching filter. Here, we have a collection of data (pixels) and we need to run a simple operation on each element of the pixel. Once we know the minimum and maximum values, we most likely would have some simple code like the following:

```
for (int row=0; row < pixelData.GetUpperBound(0); ++row)
{
    for (int col=0; col < pixelData.GetUpperBound(1); ++col)
    {
        pixelData[row, col] = AdjustContrast(pixelData[row, col], minPixel, maxPixel);
    }
}
```

This simple routine loops through a two dimensional array of pixelData and calls the AdjustContrast routine on each pixel.

As I mentioned, when you're decomposing a problem space, most iteration statements are potentially candidates for data decomposition. Here, we're using two for loops – one looping through rows in the image and a second nested loop iterating through the columns. We then perform one, independent operation on each element based on those loop positions.

This is a prime candidate – we have no shared data, no dependencies on anything but the pixel which we want to change. Since we're using a for loop, we can easily parallelize this using the Parallel.For method in the TPL:

```
Parallel.For(0, pixelData.GetUpperBound(0), row =>
{
    for (int col=0; col < pixelData.GetUpperBound(1); ++col)
    {
        pixelData[row, col] = AdjustContrast(pixelData[row, col], minPixel, maxPixel);
    }
});
```

Here, by simply changing our first for loop to a call to Parallel.For, we can parallelize this portion of our routine. Parallel.For works, as do many methods in the TPL, by creating a delegate and using it as an argument to a method. In this case, our for loop iteration block becomes a delegate creating via a lambda expression. This lets you write code that, superficially, looks similar to the familiar for loop, but functions quite differently at runtime.

We could easily do this to our second for loop as well, but that may not be a good idea. There is a balance to be struck when writing parallel code. We want to have enough work items to keep all of our processors busy, but the more we partition our data, the more overhead we introduce. In this case, we have an image of data – most likely hundreds of pixels in both dimensions. By just parallelizing our first loop, each row of pixels can be run as a single task. With hundreds of rows of data, we are providing fine enough granularity to keep all of our processors busy.

If we parallelize both loops, we're potentially creating millions of independent tasks. This introduces extra overhead with no extra gain and will actually reduce our overall performance. This leads to my first guideline when writing parallel code:

Partition your problem into enough tasks to keep each processor busy throughout the operation, but not more than necessary to keep each processor busy.

Also note that I parallelized the outer loop. I could have just as easily partitioned the inner loop. However, partitioning the inner loop would have led to many more discrete work items, each with a smaller amount of work (operate on one pixel instead of one row of pixels). My second guideline when writing parallel code reflects this:

Partition your problem in a way to place the most work possible into each task.

This typically means, in practice, that you will want to parallelize the routine at the "highest" point possible in the routine, typically the outermost loop. If you're looking at parallelizing methods which call other methods, you'll want to try to partition your work high up in the stack – as you get into lower level methods, the performance impact of parallelizing your routines may not overcome the overhead introduced.

Parallel.For works great for situations where we know the number of elements we're going to process in advance. If we're iterating through an IList<T> or an array, this is a typical approach. However, there are other iteration statements common in C#. In many situations, we'll use foreach instead of a for loop. This can be more understandable and easier to read, but also has the advantage of working with collections which only implement IEnumerable<T>, where we do not know the number of elements involved in advance.

As an example, lets take the following situation. Say we have a collection of Customers and we want to iterate through each customer, check some information about the customer and if a certain case is met, send an email to the customer and update our instance to reflect this change. Normally, this might look something like:

```
foreach(var customer in customers)
{
    // Run some process that takes some time...
    DateTime lastContact = theStore.GetLastContact(customer);
    TimeSpan timeSinceContact = DateTime.Now - lastContact;

    // If it's been more than two weeks, send an email and update...
    if (timeSinceContact.Days > 14)
    {
       theStore.EmailCustomer(customer);
       customer.LastEmailContact = DateTime.Now;
    }
}
```

Here, we are doing a fair amount of work for each customer in our collection, but we don't know how many customers exist. If we assume that theStore.GetLastContact(customer) and theStore.EmailCustomer(customer) are both side-effect free, thread safe operations, we could parallelize this using Parallel.ForEach:

```
Parallel.ForEach(customers, customer =>
{
    // Run some process that takes some time...
    DateTime lastContact = theStore.GetLastContact(customer);
    TimeSpan timeSinceContact = DateTime.Now - lastContact;

    // If it's been more than two weeks, send an email and update...
    if (timeSinceContact.Days > 14)
    {
       theStore.EmailCustomer(customer);
```

```
        customer.LastEmailContact = DateTime.Now;
    }
});
```

Just like Parallel.For, we rework our loop into a method call accepting a delegate created via a lambda expression. This keeps our new code very similar to our original iteration statement, however, this will now execute in parallel. The same guidelines apply with Parallel.ForEach as with Parallel.For.

The other iteration statements, do and while, do not have direct equivalents in the Task Parallel Library. These, however, are very easy to implement using Parallel.ForEach and the yield keyword.

Most applications can benefit from implementing some form of Data Parallelism. Iterating through collections and performing "work" is a very common pattern in nearly every application. When the problem can be decomposed by data, we often can parallelize the workload by merely changing foreach statements to Parallel.ForEach method calls and for loops to Parallel.For method calls. Any time your program operates on a collection and does a set of work on each item in the collection where that work is not dependent on other information, you very likely have an opportunity to parallelize your routine.

Part 3, Imperative Data Parallelism: Early Termination

Although simple data parallelism allows us to easily parallelize many of our iteration statements, there are cases that it does not handle well. In my previous discussion, I focused on data parallelism with no shared state and where every element is being processed exactly the same.

Unfortunately, there are many common cases where this does not happen. If we are dealing with a loop that requires early termination, extra care is required when parallelizing.

Often, while processing in a loop, once a certain condition is met, it is no longer necessary to continue processing. This may be a matter of finding a specific element within the collection, or reaching some error case. The important distinction here is that, it is often impossible to know until runtime, what set of elements needs to be processed.

In my initial discussion of data parallelism, I mentioned that this technique is a candidate when you can decompose the problem based on the data involved and you wish to apply a single operation concurrently on all of the elements of a collection. This covers many of the potential cases, but sometimes, after processing some of the elements, we need to stop processing.

As an example, lets go back to our previous Parallel.ForEach example with contacting a customer. However, this time, we'll change the requirements slightly. In this case, we'll add an extra condition – if the store is unable to email the customer, we will exit gracefully. The thinking here, of course, is that if the store is currently unable to email, the next time this operation runs, it will handle the same situation, so we can just skip our processing entirely. The original, serial case, with this extra condition, might look something like the following:

```
foreach(var customer in customers)
{
    // Run some process that takes some time...
    DateTime lastContact = theStore.GetLastContact(customer);
    TimeSpan timeSinceContact = DateTime.Now - lastContact;

    // If it's been more than two weeks, send an email and update...
    if (timeSinceContact.Days > 14)
    {
         // Exit gracefully if we fail to email, since this
         // entire process can be repeated later without issue.
         if (theStore.EmailCustomer(customer) == false)
             break;
```

```
            customer.LastEmailContact = DateTime.Now;
        }
    }
```

Here, we're processing our loop, but at any point, if we fail to send our email successfully, we just abandon this process and assume that it will get handled correctly the next time our routine is run. If we try to parallelize this using Parallel.ForEach, as we did previously, we'll run into an error almost immediately: the break statement we're using is only valid when enclosed within an iteration statement, such as foreach. When we switch to Parallel.ForEach, we're no longer within an iteration statement – we're a delegate running in a method.

This needs to be handled slightly differently when parallelized. Instead of using the break statement, we need to utilize a new class in the Task Parallel Library: ParallelLoopState. The ParallelLoopState class is intended to allow concurrently running loop bodies a way to interact with each other and provides us with a way to break out of a loop. In order to use this, we will use a different overload of Parallel.ForEach which takes an IEnumerable<T> and an Action<T, ParallelLoopState> instead of an Action<T>. Using this, we can parallelize the above operation by doing:

```
Parallel.ForEach(customers, (customer, parallelLoopState) =>
{
    // Run some process that takes some time...
    DateTime lastContact = theStore.GetLastContact(customer);
    TimeSpan timeSinceContact = DateTime.Now - lastContact;

    // If it's been more than two weeks, send an email and update...
    if (timeSinceContact.Days > 14)
    {
        // Exit gracefully if we fail to email, since this
        // entire process can be repeated later without issue.
        if (theStore.EmailCustomer(customer) == false)
            parallelLoopState.Break();
        else
            customer.LastEmailContact = DateTime.Now;
    }
});
```

There are a couple of important points here. First, we didn't actually instantiate the ParallelLoopState instance. It was provided directly to us via the Parallel class. All we needed to do was change our lambda expression to reflect that we want to use the loop state and the Parallel class creates an instance for our use. We also needed to change our logic slightly when we call Break(). Since Break() doesn't stop the program flow within our block, we needed to add an else case to only set the property in customer when we succeeded. This same technique can be used to break out of a Parallel.For loop.

That being said, there is a huge difference between using ParallelLoopState to cause early termination and to use break in a standard iteration statement. When dealing with a loop serially, break will immediately terminate the processing within the closest enclosing loop statement. Calling ParallelLoopState.Break(), however, has a very different behavior.

The issue is that, now, we're no longer processing one element at a time. If we break in one of our threads, there are other threads that will likely still be executing. This leads to an important observation about termination of parallel code:

Early termination in parallel routines is not immediate. Code will continue to run after you request a termination.

This may seem problematic at first, but it is something you just need to keep in mind while designing your routine. ParallelLoopState.Break() should be thought of as a request. We are telling the runtime that no elements that were in the collection past the element we're currently processing need to be processed and leaving it up to the runtime to decide how to handle this as gracefully as possible. Although this may seem problematic at first, it is a good thing. If the runtime tried to immediately stop processing, many of our elements would be partially processed. It would be like putting a return statement in a random location throughout our loop body – which could have horrific consequences to our code's maintainability.

In order to understand and effectively write parallel routines, we, as developers, need a subtle, but profound shift in our thinking. We can no longer think in terms of sequential processes, but rather need to think in terms of requests to the system that may be handled differently than we'd first expect. This is more natural to developers who have dealt with asynchronous models previously, but is an important distinction when moving to concurrent programming models.

As an example, I'll discuss the Break() method. ParallelLoopState.Break() functions in a way that may be unexpected at first. When you call Break() from a loop body, the runtime will continue to process all elements of the collection that were found prior to the element that was being processed when the Break() method was called. This is done to keep the behavior of the Break() method as close to the behavior of the break statement as possible. We can see the behavior in this simple code:

```
var collection = Enumerable.Range(0, 20);
var pResult = Parallel.ForEach(collection, (element, state) =>
{
    if (element > 10)
    {
        Console.WriteLine("Breaking on {0}", element);
        state.Break();
    }
    Console.WriteLine(element);
});
```

If we run this, we get a result that may seem unexpected at first:

```
0
2
1
5
6
3
4
10
Breaking on 11
11
Breaking on 12
12
9
Breaking on 13
13
7
8
Breaking on 15
15
```

What is occurring here is that we loop until we find the first element where the element is greater than 10. In this case, this was found, the first time, when one of our threads reached element 11. It requested that the loop stop by calling Break() at this point. However, the loop continued processing until all of the elements less than 11 were completed, then terminated. This means that it will guarantee that elements 9, 7 and 8 are completed before it stops processing. You can see our other threads that were running each tried to break as well, but since Break() was called on the element with a value of 11, it decides which elements (0-10) must be processed.

If this behavior is not desirable, there is another option. Instead of calling ParallelLoopState.Break(), you can call ParallelLoopState.Stop(). The Stop() method requests that the runtime terminate as soon as possible , without guaranteeing that any other elements are processed. Stop() will not stop the processing within an element, so elements already being processed will continue to be processed. It will prevent new elements, even ones found earlier in the collection, from being processed. Also, when Stop() is called, the ParallelLoopState's IsStopped property will return true. This lets longer running processes poll for this value and return after performing any necessary cleanup.

The basic rule of thumb for choosing between Break() and Stop() is the following.

- Use ParallelLoopState.Stop() when possible, since it terminates more quickly. This is particularly useful in situations where you are searching for an element or a condition in the collection. Once you've found it, you do not need to do any other processing, so Stop() is more appropriate.
- Use ParallelLoopState.Break() if you need to more closely match the behavior of the C# break statement.

Both methods behave differently than our C# break statement. Unfortunately, when parallelizing a routine, more thought and care needs to be put into every aspect of your routine than you may otherwise expect. This is due to my second observation:

Parallelizing a routine will almost always change its behavior.

This sounds crazy at first, but it's a concept that's so simple its easy to forget. We're purposely telling the system to process more than one thing at the same time, which means that the sequence in which things get processed is no longer deterministic. It is easy to change the behavior of your routine in very subtle ways by introducing parallelism. Often, the changes are not avoidable, even if they don't have any adverse side effects. This leads to my final observation for this post:

Parallelization is something that should be handled with care and forethought, added by design and not just introduced casually.

Part 4, Imperative Data Parallelism: Aggregation

In the article on simple data parallelism, I described how to perform an operation on an entire collection of elements in parallel. Often, this is not adequate, as the parallel operation is going to be performing some form of aggregation.

Simple examples of this might include taking the sum of the results of processing a function on each element in the collection, or finding the minimum of the collection given some criteria. This can be done using the techniques described in simple data parallelism, however, special care needs to be taken into account to synchronize the shared data appropriately. The Task Parallel Library has tools to assist in this synchronization.

The main issue with aggregation when parallelizing a routine is that you need to handle synchronization of data. Since multiple threads will need to write to a shared portion of data. Suppose, for example, that we wanted to parallelize a simple loop that looked for the minimum value within a dataset:

```
double min = double.MaxValue;
foreach(var item in collection)
{
    double value = item.PerformComputation();
    min = System.Math.Min(min, value);
}
```

This seems like a good candidate for parallelization, but there is a problem here. If we just wrap this into a call to Parallel.ForEach, we'll introduce a critical race condition and get the wrong answer. Let's look at what happens here:

```
// Buggy code!  Do not use!
```

```
double min = double.MaxValue;
Parallel.ForEach(collection, item =>
{
    double value = item.PerformComputation();
    min = System.Math.Min(min, value);
});
```

This code has a fatal flaw: **min** will be checked, then set, by multiple threads simultaneously. Two threads may perform the check at the same time and set the wrong value for min. Say we get a value of 1 in thread 1 and a value of 2 in thread 2 and these two elements are the first two to run. If both hit the min check line at the same time, both will determine that min should change, to 1 and 2 respectively. If element 1 happens to set the variable first, then element 2 sets the min variable, we'll detect a min value of 2 instead of 1. This can lead to wrong answers.

Unfortunately, fixing this, with the Parallel.ForEach call we're using, would require adding locking. We would need to rewrite this like:

```
// Safe, but slow
double min = double.MaxValue;
// Make a "lock" object
object syncObject = new object();
Parallel.ForEach(collection, item =>
{
    double value = item.PerformComputation();
    lock(syncObject)
        min = System.Math.Min(min, value);
});
```

This will potentially add a huge amount of overhead to our calculation. Since we can potentially block while waiting on the lock for every single iteration, we will most likely slow this down to where it is actually quite a bit slower than our serial implementation. The problem is the lock statement – any time you use lock(object), you're almost assuring reduced performance in a parallel situation. This leads to two observations I'll make:

When parallelizing a routine, try to avoid locks.

That being said:

Always add any and all required synchronization to avoid race conditions.

These two observations tend to be opposing forces – we often need to synchronize our algorithms, but we also want to avoid the synchronization when possible. Looking at our routine, there is no way to directly avoid this lock, since each element is potentially being run on a separate thread and this lock is necessary in order for our routine to function correctly every time.

However, this isn't the only way to design this routine to implement this algorithm. Realize that, although our collection may have thousands or even millions of elements, we have a limited number of Processing Elements (PE). Processing Element is the standard term for a hardware element which can process and execute instructions. This typically is a core in your processor, but many modern systems have multiple hardware execution threads per core. The Task Parallel Library will not execute the work for each item in the collection as a separate work item. Instead, when Parallel.ForEach executes, it will partition the collection into larger "chunks" which get processed on different threads via the ThreadPool. This helps reduce the threading overhead and help the overall speed. In general, the Parallel class will only use one thread per PE in the system.

Given the fact that there are typically fewer threads than work items, we can rethink our algorithm design. We can parallelize our algorithm more effectively by approaching it differently. Because the basic aggregation we are doing here (Min) is communitive, we do not need to perform this in a given order. We knew this to be true already – otherwise, we wouldn't have been able to parallelize this routine in the first place. With this in mind, we can treat each thread’s work independently, allowing each thread to serially process many elements with no locking, then, after all the threads are complete, “merge” together the results.

This can be accomplished via a different set of overloads in the Parallel class: Parallel.ForEach<TSource,TLocal>. The idea behind these overloads is to allow each thread to begin by initializing some local state (TLocal). The thread will then process an entire set of items in the source collection, providing that state to the delegate which processes an individual item. Finally, at the end, a separate delegate is run which allows you to handle merging that local state into your final results.

To rewriting our routine using Parallel.ForEach<TSource,TLocal>, we need to provide three delegates instead of one. The most basic version of this function is declared as:

```
public static ParallelLoopResult ForEach<TSource, TLocal>(
    IEnumerable<TSource> source,
    Func<TLocal> localInit,
    Func<TSource, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally
)
```

The first delegate (the localInit argument) is defined as Func<TLocal>. This delegate initializes our local state. It should return some object we can use to track the results of **a single thread’s operations**.

The second delegate (the body argument) is where our main processing occurs, although now, instead of being an Action<T>, we actually provide a Func<TSource, ParallelLoopState, TLocal, TLocal> delegate. This delegate will receive three arguments: our original element from the collection (TSource), a ParallelLoopState which we can use for early termination and the instance of our local state we created (TLocal). It should do whatever processing you wish to occur per element, then **return the value of the local state** after processing is completed.

The third delegate (the localFinally argument) is defined as Action<TLocal>. This delegate is passed our local state after it’s been processed by all of the elements this thread will handle. This is where you can merge your final results together. This may require synchronization, but now, instead of synchronizing once per element (potentially millions of times), you’ll only have to synchronize once per thread, which is an ideal situation.

Now that I’ve explained how this works, lets look at the code:

```
// Safe and fast!
double min = double.MaxValue;
// Make a "lock" object
object syncObject = new object();
Parallel.ForEach(
    collection,
    // First, we provide a local state initialization delegate.
    () => double.MaxValue,
    // Next, we supply the body, which takes the original item, loop state,
    // and local state and returns a new local state
    (item, loopState, localState) =>
    {
        double value = item.PerformComputation();
        return System.Math.Min(localState, value);
    },
    // Finally, we provide an Action<TLocal>, to "merge" results together
    localState =>
    {
        // This requires locking, but it's only once per used thread
        lock(syncObj)
            min = System.Math.Min(min, localState);
    }
);
```

Although this is a bit more complicated than the previous version, it is now both thread-safe and has minimal locking.

This same approach can be used by Parallel.For, although now, it's Parallel.For<TLocal>. When working with Parallel.For<TLocal>, you use the same triplet of delegates, with the same purpose and results.

Also, many times, you can completely avoid locking by using a method of the Interlocked class to perform the final aggregation in an atomic operation. The MSDN example demonstrating this same technique using Parallel.For uses the Interlocked class instead of a lock, since they are doing a sum operation on a long variable, which is possible via Interlocked.Add.

By taking advantage of local state, we can use the Parallel class methods to parallelize algorithms such as aggregation, which, at first, may seem like poor candidates for parallelization. Doing so requires careful consideration and often requires a slight redesign of the algorithm, but the performance gains can be significant if handled in a way to avoid excessive synchronization.

Part 5, Partitioning of Work

When parallelizing any routine, we start by decomposing the problem. Once the problem is understood, we need to break our work into separate tasks, so each task can be run on a different processing element. This process is called partitioning.

Partitioning our tasks is a challenging feat. There are opposing forces at work here: too many partitions adds overhead, too few partitions leaves processors idle. Trying to work the perfect balance between the two extremes is the goal for which we should aim. Luckily, the Task Parallel Library automatically handles much of this process. However, there are situations where the default partitioning may not be appropriate and knowledge of our routines may allow us to guide the framework to making better decisions.

First off, I'd like to say that this is a more advanced topic. It is perfectly acceptable to use the parallel constructs in the framework without considering the partitioning taking place. The default behavior in the Task Parallel Library is very well-behaved, even for unusual work loads and should rarely be adjusted. I have found few situations where the default partitioning behavior in the TPL is not as good or better than my own hand-written partitioning routines and recommend using the defaults unless there is a strong, measured and profiled reason to avoid using them. However, understanding partitioning and how the TPL partitions your data, helps in understanding the proper usage of the TPL.

I indirectly mentioned partitioning while discussing aggregation. Typically, our systems will have a limited number of Processing Elements (PE), which is the terminology used for hardware capable of processing a stream of instructions. For example, in a standard Intel i7 system, there are four processor cores, each of which has two potential hardware threads due to Hyperthreading. This gives us a total of 8 PEs – theoretically, we can have up to eight operations occurring concurrently within our system.

In order to fully exploit this power, we need to partition our work into Tasks. A task is a simple set of instructions that can be run on a PE. Ideally, we want to have at least one task per PE in the system, since fewer tasks means that some of our processing power will be sitting idle. A naive implementation would be to just take our data and partition it with one element in our collection being treated as one task. When we loop through our collection in parallel, using this approach, we'd just process one item at a time, then reuse that thread to process the next, etc. There's a flaw in this approach, however. It will tend to be slower than necessary, often slower than processing the data serially.

The problem is that there is overhead associated with each task. When we take a simple foreach loop body and implement it using the TPL, we add overhead. First, we change the body from a simple statement to a delegate, which must be invoked. In order to invoke the delegate on a separate thread, the delegate gets added to the ThreadPool's current work queue and the ThreadPool must pull this off the queue, assign it to a free thread, then execute it. If our collection had one million elements, the overhead of trying to spawn one million tasks would destroy our performance.

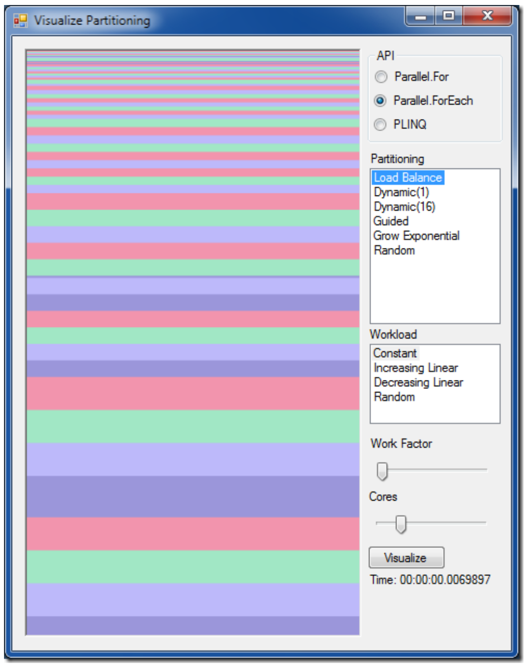
The answer, here, is to partition our collection into groups and have each group of elements treated as a single task. By adding a partitioning step, we can break our total work into small enough tasks to keep our processors busy, but large enough tasks to avoid overburdening the ThreadPool. There are two clear, opposing goals here:

Always try to keep each processor working, but also try to keep the individual partitions as large as possible.

When using Parallel.For, the partitioning is always handled automatically. At first, partitioning here seems simple. A naive implementation would merely split the total element count up by the number of PEs in the system and assign a chunk of data to each processor. Many hand-written partitioning schemes work in this exactly manner. This perfectly balanced, static partitioning scheme works very well if the amount of work is constant for each element. However, this is rarely the case. Often, the length of time required to process an element grows as we progress through the collection, especially if we're doing numerical computations. In this case, the first PEs will finish early and sit idle waiting on the last chunks to finish. Sometimes, work can decrease as we progress, since previous computations may be used to speed up later computations. In this situation, the first chunks will be working far longer than the last chunks. In order to balance the workload, many implementations create many small chunks and reuse threads. This adds overhead, but does provide better load balancing, which in turn improves performance.

The Task Parallel Library handles this more elaborately. Chunks are determined at runtime and start small. They grow slowly over time, getting larger and larger. This tends to lead to a near optimum load balancing, even in odd cases such as increasing or decreasing workloads.

Parallel.ForEach is a bit more complicated, however. When working with a generic IEnumerable<T>, the number of items required for processing is not known in advance and must be discovered at runtime. In addition, since we don't have direct access to each element, the scheduler must enumerate the collection to process it. Since IEnumerable<T> is not thread safe, it must lock on elements as it enumerates, create temporary collections for each chunk to process and schedule this out. By default, it uses a partitioning method similar to the one described above. We can see this directly by looking at the **Visual Partitioning** sample shipped by the Task Parallel Library team and available as part of the Samples for Parallel Programming. When we run the sample, with four cores and the default, Load Balancing partitioning scheme, we see this:



API
( ) Parallel.For
(●) Parallel.ForEach
( ) PLINQ

Partitioning
[✓ Load Balance]
Dynamic(1)
Dynamic(16)
Guided
Grow Exponential
Random

Workload
Constant
Increasing Linear
Decreasing Linear
Random

Work Factor

Cores

Time: 00:00:00.0069897

The colored bands represent each processing core. You can see that, when we started (at the top), we begin with very small bands of color. As the routine progresses through the Parallel.ForEach, the chunks get larger and larger (seen by larger and larger stripes).

Most of the time, this is fantastic behavior and most likely will out perform any custom written partitioning. However, if your routine is not scaling well, it may be due to a failure in the default partitioning to handle your specific case. With prior knowledge about your work, it may be possible to partition data more meaningfully than the default Partitioner.

There is the option to use an overload of Parallel.ForEach which takes a Partitioner<T> instance. The Partitioner<T> class is an abstract class which allows for both static and dynamic partitioning. By overriding Partitioner<T>.SupportsDynamicPartitions, you can specify whether a dynamic approach is available. If not, your custom Partitioner<T> subclass would override GetPartitions(int), which returns a list of IEnumerator<T> instances. These are then used by the Parallel class to split work up amongst processors. When dynamic partitioning is available, GetDynamicPartitions() is used, which returns an IEnumerable<T> for each partition. If you do decide to implement your own Partitioner<T>, keep in mind the goals and tradeoffs of different partitioning strategies and design appropriately.

The Samples for Parallel Programming project includes a ChunkPartitioner class in the **ParallelExtensionsExtras** project. This provides example code for implementing your own, custom allocation strategies, including a static allocator of a given chunk size. Although implementing your own Partitioner<T> is possible, as I mentioned above, this is rarely required or useful in practice. The default behavior of the TPL is very good, often better than any hand-written partitioning strategy.

Part 6, Declarative Data Parallelism

When working with a problem that can be decomposed by data, we have a collection and some operation being performed upon the collection. I've demonstrated how this can be parallelized using the Task Parallel Library and imperative programming using imperative data parallelism via the Parallel class. While this provides a huge step forward in terms of power and capabilities, in many cases, special care must still be given for relative common scenarios.

C# 3.0 and Visual Basic 9.0 introduced a new, declarative programming model to .NET via the LINQ Project. When working with collections, we can now write software that describes what we want to occur without having to explicitly state how the program should accomplish the task. By taking advantage of LINQ, many operations become much shorter, more elegant and easier to understand and maintain. Version 4.0 of the .NET framework extends this concept into the parallel computation space by introducing Parallel LINQ.

Before we delve into PLINQ, let's begin with a short discussion of LINQ. LINQ, the extensions to the .NET Framework which implement language integrated query, set and transform operations, is implemented in many flavors. For our purposes, we are interested in LINQ to Objects. When dealing with parallelizing a routine, we typically are dealing with in-memory data storage. More data-access oriented LINQ variants, such as LINQ to SQL and LINQ to Entities in the Entity Framework fall outside of our concern, since the parallelism there is the concern of the data base engine processing the query itself.

LINQ (LINQ to Objects in particular) works by implementing a series of extension methods, most of which work on IEnumerable<T>. The language enhancements use these extension methods to create a very concise, readable alternative to using traditional foreach statement. For example, let's revisit our minimum aggregation routine we wrote in Part 4:

```
double min = double.MaxValue;
foreach(var item in collection)
{
    double value = item.PerformComputation();
    min = System.Math.Min(min, value);
}
```

Here, we're doing a very simple computation, but writing this in an imperative style. This can be loosely translated to English as:

```
Create a very large number and save it in min
Loop through each item in the collection.
For every item:
    Perform some computation and save the result
    If the computation is less than min, set min to the computation
```

Although this is fairly easy to follow, it's quite a few lines of code and it requires us to read through the code, step by step, line by line, in order to understand the intention of the developer.
We can rework this same statement, using LINQ:

```
double min = collection.Min(item => item.PerformComputation());
```

Here, we're after the same information. However, this is written using a declarative programming style. When we see this code, we'd naturally translate this to English as:

```
Save the Min value of collection, determined via calling item.PerformComputation()
```

That's it – instead of multiple logical steps, we have one single, declarative request. This makes the developer's intentions very clear and very easy to follow. The system is free to implement this using whatever method required. Parallel LINQ (PLINQ) extends LINQ to Objects to support parallel operations. This is a perfect fit in many cases when you have a problem that can be decomposed by data. To show this, let's again refer to our minimum aggregation routine from Part 4, but this time, let's review our final, parallelized version:

```
// Safe and fast!
```

```
double min = double.MaxValue;
// Make a "lock" object
object syncObject = new object();
Parallel.ForEach(
    collection,
    // First, we provide a local state initialization delegate.
    () => double.MaxValue,
    // Next, we supply the body, which takes the original item, loop state,
    // and local state and returns a new local state
    (item, loopState, localState) =>
    {
        double value = item.PerformComputation();
        return System.Math.Min(localState, value);
    },
    // Finally, we provide an Action<TLocal>, to "merge" results together
    localState =>
    {
        // This requires locking, but it's only once per used thread
        lock(syncObj)
            min = System.Math.Min(min, localState);
    }
);
```

Here, we're doing the same computation as above, but fully parallelized. Describing this in English becomes quite a feat:

```
Create a very large number and save it in min
Create a temporary object we can use for locking
Call Parallel.ForEach, specifying three delegates
    For the first delegate:
        Initialize a local variable to hold the local state to a very large number
    For the second delegate:
        For each item in the collection, perform some computation, save the result
        If the result is less than our local state, save the result in local state
    For the final delegate:
        Take a lock on our temporary object to protect our min variable
        Save the min of our min and local state variables
```

Although this solves our problem and does it in a very efficient way, we've created a set of code that is quite a bit more difficult to understand and maintain.

PLINQ provides us with a very nice alternative. In order to use PLINQ, we need to learn one new extension method that works on IEnumerable<T> – ParallelEnumerable.AsParallel().

That's all we need to learn in order to use PLINQ**: one single method**. We can write our minimum aggregation in PLINQ very simply:

```
double min = collection.AsParallel().Min(item => item.PerformComputation());
```

By simply adding ".AsParallel()" to our LINQ to Objects query, we converted this to using PLINQ and running this computation in parallel!
This can be loosely translated into English easily, as well:

```
Process the collection in parallel
Get the Minimum value, determined by calling PerformComputation on each item
```

Here, our intention is very clear and easy to understand. We just want to perform the same operation we did in serial, but run it "as parallel". PLINQ completely extends LINQ to Objects: the entire functionality of LINQ to Objects is available. By simply adding a call to AsParallel(), we can specify that a collection should be processed in parallel. This is simple, safe and incredibly useful.

Part 7, Some Differences between PLINQ and LINQ to Objects

In my previous post on Declarative Data Parallelism, I mentioned that PLINQ extends LINQ to Objects to support parallel operations. Although nearly all of the same operations are supported, there are some differences between PLINQ and LINQ to Objects. By introducing Parallelism to our declarative model, we add some extra complexity. This, in turn, adds some extra requirements that must be addressed.

In order to illustrate the main differences and why they exist, let's begin by discussing some differences in how the two technologies operate and look at the underlying types involved in LINQ to Objects and PLINQ .

LINQ to Objects is mainly built upon a single class: Enumerable. The Enumerable class is a static class that defines a large set of extension methods, nearly all of which work upon an IEnumerable<T>. Many of these methods return a new IEnumerable<T>, allowing the methods to be chained together into a fluent style interface. This is what allows us to write statements that chain together and lead to the nice declarative programming model of LINQ:

```
double min = collection
                .Where(item => item.SomeProperty > 6 && item.SomeProperty < 24)
                .Min(item => item.PerformComputation());
```

Other LINQ variants work in a similar fashion. For example, most data-oriented LINQ providers are built upon an implementation of IQueryable<T>, which allows the database provider to turn a LINQ statement into an underlying SQL query, to be performed directly on the remote database.

PLINQ is similar, but instead of being built upon the Enumerable class, most of PLINQ is built upon a new static class: ParallelEnumerable. When using PLINQ, you typically begin with any collection which implements IEnumerable<T> and convert it to a new type using an extension method defined on ParallelEnumerable: AsParallel(). This method takes any IEnumerable<T> and converts it into a ParallelQuery<T>, the core class for PLINQ. There is a similar ParallelQuery class for working with non-generic IEnumerable implementations.

This brings us to our first subtle, but important difference between PLINQ and LINQ –

PLINQ always works upon specific types, which must be explicitly created.

Typically, the type you'll use with PLINQ is ParallelQuery<T>, but it can sometimes be a ParallelQuery or an OrderedParallelQuery<T>. Instead of dealing with an interface, implemented by an unknown class, we're dealing with a specific class type. This works seamlessly from a usage standpoint – ParallelQuery<T> implements IEnumerable<T>, so you can always "switch back" to an IEnumerable<T>.

The difference only arises at the beginning of our parallelization. When we're using LINQ and we want to process a normal collection via PLINQ, we need to explicitly convert the collection into a ParallelQuery<T> by calling AsParallel(). There is an important consideration here – AsParallel() does not need to be called on your specific collection, but rather any IEnumerable<T>. This allows you to place it anywhere in the chain of methods involved in a LINQ statement, not just at the beginning. This can be useful if you have an operation which will not parallelize well or is not thread safe. For example, the following is perfectly valid and similar to our previous examples:

```
double min = collection
                .AsParallel()
                .Select(item => item.SomeOperation())
                .Where(item => item.SomeProperty > 6 && item.SomeProperty < 24)
                .Min(item => item.PerformComputation());
```

However, if SomeOperation() is not thread safe, we could just as easily do:

```
double min = collection
                .Select(item => item.SomeOperation())
                .AsParallel()
                .Where(item => item.SomeProperty > 6 && item.SomeProperty < 24)
                .Min(item => item.PerformComputation());
```

In this case, we're using standard LINQ to Objects for the Select(…) method, then converting the results of that map routine to a ParallelQuery<T> and processing our filter (the Where method) and our aggregation (the Min method) in parallel.

PLINQ also provides us with a way to convert a ParallelQuery<T> back into a standard IEnumerable<T>, forcing sequential processing via standard LINQ to Objects. If SomeOperation() was thread-safe, but PerformComputation()was not thread-safe, we would need to handle this by using the AsEnumerable() method:

```
double min = collection
                .AsParallel()
                .Select(item => item.SomeOperation())
                .Where(item => item.SomeProperty > 6 && item.SomeProperty < 24)
                .AsEnumerable()
                .Min(item => item.PerformComputation());
```

Here, we're converting our collection into a ParallelQuery<T>, doing our map operation (the Select(…) method) and our filtering in parallel, then converting the collection back into a standard IEnumerable<T>, which causes our aggregation via Min() to be performed sequentially.

This could also be written as two statements, as well, which would allow us to use the language integrated syntax for the first portion:

```
var tempCollection = from item in collection.AsParallel()
                     let e = item.SomeOperation()
                     where (e.SomeProperty > 6 && e.SomeProperty < 24)
                     select e;
double min = tempCollection.AsEnumerable().Min(item => item.PerformComputation());
```

This allows us to use the standard LINQ style language integrated query syntax, but control whether it's performed in parallel or serial by adding AsParallel() and AsEnumerable() appropriately.

The second important difference between PLINQ and LINQ deals with order preservation.

PLINQ, by default, does not preserve the order of of source collection.

This is by design. In order to process a collection in parallel, the system needs to naturally deal with multiple elements at the same time. Maintaining the original ordering of the sequence adds overhead, which is, in many cases, unnecessary. Therefore, by default, the system is allowed to completely change the order of your sequence during processing. If you are doing a standard query operation, this is usually not an issue. However, there are times when keeping a specific ordering in place is important. If this is required, you can explicitly request the ordering be preserved throughout all operations done on a ParallelQuery<T> by using the AsOrdered() extension method. This will cause our sequence ordering to be preserved.

For example, suppose we wanted to take a collection, perform an expensive operation which converts it to a new type and display the first 100 elements. In LINQ to Objects, our code might look something like:

```
// Using IEnumerable<SourceClass> collection
IEnumerable<ResultClass> results = collection
                                       .Select(e => e.CreateResult())
                                       .Take(100);
```

If we just converted this to a parallel query naively, like so:

```
IEnumerable<ResultClass> results = collection
                                       .AsParallel()
                                       .Select(e => e.CreateResult())
                                       .Take(100);
```

We could very easily get a very different and non-reproducable, set of results, since the ordering of elements in the input collection is not preserved. To get the same results as our original query, we need to use:

```
IEnumerable<ResultClass> results = collection
                                       .AsParallel()
                                       .AsOrdered()
                                       .Select(e => e.CreateResult())
                                       .Take(100);
```

This requests that PLINQ process our sequence in a way that verifies that our resulting collection is ordered as if it were processed serially. This will cause our query to run slower, since there is overhead involved in maintaining the ordering. However, in this case, it is required, since the ordering is required for correctness.

PLINQ is incredibly useful. It allows us to easily take nearly any LINQ to Objects query and run it in parallel, using the same methods and syntax we've used previously. There are some important differences in operation that must be considered, however – it is not a free pass to parallelize everything. When using PLINQ in order to parallelize your routines declaratively, the same guideline I mentioned before still applies:

Parallelization is something that should be handled with care and forethought, added by design and not just introduced casually.

Part 8, PLINQ's ForAll Method

Parallel LINQ extends LINQ to Objects and is typically very similar. However, as I previously discussed, there are some differences. Although the standard way to handle simple Data Parellelism is via Parallel.ForEach, it's possible to do the same thing via PLINQ.

PLINQ adds a new method unavailable in standard LINQ which provides new functionality…

LINQ is designed to provide a much simpler way of handling querying, including filtering, ordering, grouping and many other benefits. Reading the description in LINQ to Objects on MSDN, it becomes clear that the thinking behind LINQ deals with retrieval of data. LINQ works by adding a functional programming style on top of .NET, allowing us to express filters in terms of predicate functions, for example.

PLINQ is, generally, very similar. Typically, when using PLINQ, we write declarative statements to filter a dataset or perform an aggregation. However, PLINQ adds one new method, which provides a very different purpose: ForAll.

The ForAll method is defined on ParallelEnumerable and will work upon any ParallelQuery<T>. Unlike the sequence operators in LINQ and PLINQ, ForAll is **intended to cause side effects**. It does not filter a collection, but rather invokes an action on each element of the collection.

At first glance, this seems like a bad idea. For example, Eric Lippert clearly explained two philosophical objections to providing an IEnumerable<T>.ForEach extension method, one of which still applies when parallelized. The sole purpose of this method is to cause side effects and as such, I agree that the ForAll method "violates the functional programming principles that all the other sequence operators are based upon", in exactly the same manner an IEnumerable<T>.ForEach extension method would violate these principles. Eric Lippert's second reason for disliking a ForEach extension method does not necessarily apply to ForAll – replacing ForAll with a call to Parallel.ForEach has the same closure semantics, so there is no loss there.

Although ForAll may have philosophical issues, there is a pragmatic reason to include this method. Without ForAll, we would take a fairly serious performance hit in many situations. Often, we need to perform some filtering or grouping, then perform an action using the results of our filter.

Using a standard foreach statement to perform our action would avoid this philosophical issue:

```
// Filter our collection
var filteredItems = collection.AsParallel().Where( i => i.SomePredicate() );

// Now perform an action
foreach (var item in filteredItems)
{
    // These will now run serially
    item.DoSomething();
}
```

This would cause a loss in performance, since we lose any parallelism in place and cause all of our actions to be run serially.

We could easily use a Parallel.ForEach instead, which adds parallelism to the actions:

```
// Filter our collection
var filteredItems = collection.AsParallel().Where( i => i.SomePredicate() );

// Now perform an action once the filter completes
Parallel.ForEach(filteredItems, item =>
{
    // These will now run in parallel
    item.DoSomething();
});
```

This is a noticeable improvement, since both our filtering and our actions run parallelized. However, there is still a large bottleneck in place here.

The problem lies with my comment "perform an action once the filter completes". Here, we're parallelizing the filter, then collecting all of the results, blocking until the filter completes. Once the filtering of every element is completed, we then repartition the results of the filter, reschedule into multiple threads and perform the action on each element. By moving this into two separate statements, we potentially double our parallelization overhead, since we're forcing the work to be partitioned and scheduled twice as many times.

This is where the pragmatism comes into play. By violating our functional principles, we gain the ability to avoid the overhead and cost of rescheduling the work:

```
// Perform an action on the results of our filter
collection
    .AsParallel()
    .Where( i => i.SomePredicate() )
    .ForAll( i => i.DoSomething() );
```

The ability to avoid the scheduling overhead is a compelling reason to use ForAll. This really goes back to one of the key points I discussed in data parallelism: Partition your problem in a way to place the most work possible into each task. Here, this means leaving the statement attached to the expression, even though it causes side effects and is not standard usage for LINQ.
This leads to my one guideline for using ForAll:

The ForAll extension method should only be used to process the results of a parallel query, as returned by a PLINQ expression.

Any other usage scenario should use Parallel.ForEach, instead.

Part 9, Configuration in PLINQ and TPL

Parallel LINQ and the Task Parallel Library contain many options for configuration. Although the default configuration options are often ideal, there are times when customizing the behavior is desirable. Both frameworks provide full configuration support.

When working with Data Parallelism, there is one primary configuration option we often need to control – the number of threads we want the system to use when parallelizing our routine. By default, PLINQ and the TPL both use the ThreadPool to schedule tasks. Given the major improvements in the ThreadPool in CLR 4, this default behavior is often ideal.

However, there are times that the default behavior is not appropriate. For example, if you are working on multiple threads simultaneously and want to schedule parallel operations from within both threads, you might want to consider restricting each parallel operation to using a subset of the processing cores of the system. Not doing this might over-parallelize your routine, which leads to inefficiencies from having too many context switches.

In the Task Parallel Library, configuration is handled via the ParallelOptions class. All of the methods of the Parallel class have an overload which accepts a ParallelOptions argument.

We configure the Parallel class by setting the ParallelOptions.MaxDegreeOfParallelism property. For example, let's revisit one of the simple data parallel examples from Part 2:

```
Parallel.For(0, pixelData.GetUpperBound(0), row =>
{
    for (int col=0; col < pixelData.GetUpperBound(1); ++col)
    {
        pixelData[row, col] = AdjustContrast(pixelData[row, col], minPixel, maxPixel);
    }
});
```

Here, we're looping through an image and calling a method on each pixel in the image. If this was being done on a separate thread and we knew another thread within our system was going to be doing a similar operation, we likely would want to restrict this to using half of the cores on the system. This could be accomplished easily by doing:

```
var options = new ParallelOptions();
options.MaxDegreeOfParallelism = Math.Max(Environment.ProcessorCount / 2, 1);

Parallel.For(0, pixelData.GetUpperBound(0), options, row =>
{
    for (int col=0; col < pixelData.GetUpperBound(1); ++col)
    {
        pixelData[row, col] = AdjustContrast(pixelData[row, col], minPixel, maxPixel);
    }
});
```

Now, we're restricting this routine to using no more than half the cores in our system. Note that I included a check to prevent a single core system from supplying zero; without this check, we'd potentially cause an exception. I also did not hard code a specific value for the MaxDegreeOfParallelism property. One of our goals when parallelizing a routine is allowing it to scale on better hardware. Specifying a hard-coded value would contradict that goal.

Parallel LINQ also supports configuration and in fact, has quite a few more options for configuring the system. The main configuration option we most often need is the same as our TPL option: we need to supply the maximum number of processing threads. In PLINQ, this is done via a new extension method on ParallelQuery<T>: ParallelEnumerable.WithDegreeOfParallelism.

Let's revisit our declarative data parallelism sample from Part 6:

```
double min = collection.AsParallel().Min(item => item.PerformComputation());
```

Here, we're performing a computation on each element in the collection and saving the minimum value of this operation. If we wanted to restrict this to a limited number of threads, we would add our new extension method:

```
int maxThreads = Math.Max(Environment.ProcessorCount / 2, 1);
double min = collection
                 .AsParallel()
                 .WithDegreeOfParallelism(maxThreads)
                 .Min(item => item.PerformComputation());
```

This automatically restricts the PLINQ query to half of the threads on the system.

PLINQ provides some additional configuration options. By default, PLINQ will occasionally revert to processing a query in parallel. This occurs because many queries, if parallelized, typically actually cause an overall slowdown compared to a serial processing equivalent. By analyzing the "shape" of the query, PLINQ often decides to run a query serially instead of in parallel. This can occur for (taken from MSDN):

- Queries that contain a Select, indexed Where, indexed SelectMany, or ElementAt clause after an ordering or filtering operator that has removed or rearranged original indices.
- Queries that contain a Take, TakeWhile, Skip, SkipWhile operator and where indices in the source sequence are not in the original order.
- Queries that contain Zip or SequenceEquals, unless one of the data sources has an originally ordered index and the other data source is indexable (i.e. an array or IList(T)).
- Queries that contain Concat, unless it is applied to indexable data sources.
- Queries that contain Reverse, unless applied to an indexable data source.

If the specific query follows these rules, PLINQ will run the query on a single thread. However, none of these rules look at the specific work being done in the delegates, only at the "shape" of the query. There are cases where running in parallel may still be beneficial, even if the shape is one where it typically parallelizes poorly. In these cases, you can override the default behavior by using the WithExecutionMode extension method. This would be done like so:

```
var reversed = collection
                  .AsParallel()
                  .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                  .Select(i => i.PerformComputation())
                  .Reverse();
```

Here, the default behavior would be to not parallelize the query unless collection implemented IList<T>. We can force this to run in parallel by adding the WithExecutionMode extension method in the method chain.

Finally, PLINQ has the ability to configure how results are returned. When a query is filtering or selecting an input collection, the results will need to be streamed back into a single IEnumerable<T> result. For example, the method above returns a new, reversed collection. In this case, the processing of the collection will be done in parallel, but the results need to be streamed back to the caller serially, so they can be enumerated on a single thread.

This streaming introduces overhead. IEnumerable<T> isn't designed with thread safety in mind, so the system needs to handle merging the parallel processes back into a single stream, which introduces synchronization issues. There are two extremes of how this could be accomplished, but both extremes have disadvantages.

The system could watch each thread and whenever a thread produces a result, take that result and send it back to the caller. This would mean that the calling thread would have access to the data as soon as data is available, which is the benefit of this approach. However, it also means that every item is introducing synchronization overhead, since each item needs to be merged individually.

On the other extreme, the system could wait until all of the results from all of the threads were ready, then push all of the results back to the calling thread in one shot. The advantage here is that the least amount of synchronization is added to the system, which means the query will, on a whole, run the fastest. However, the calling thread will have to wait for all elements to be processed, so this could introduce a long delay between when a parallel query begins and when results are returned.

The default behavior in PLINQ is actually between these two extremes. By default, PLINQ maintains an internal buffer and chooses an optimal buffer size to maintain. Query results are accumulated into the buffer, then returned in the IEnumerable<T> result in chunks. This provides reasonably fast access to the results, as well as good overall throughput, in most scenarios.

However, if we know the nature of our algorithm, we may decide we would prefer one of the other extremes. This can be done by using the WithMergeOptions extension method. For example, if we know that our PerformComputation() routine is very slow, but also variable in runtime, we may want to retrieve results as they are available, with no bufferring. This can be done by changing our above routine to:

```
var reversed = collection
                 .AsParallel()
                 .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                 .WithMergeOptions(ParallelMergeOptions.NotBuffered)
                 .Select(i => i.PerformComputation())
                 .Reverse();
```

On the other hand, if are already on a background thread and we want to allow the system to maximize its speed, we might want to allow the system to fully buffer the results:

```
var reversed = collection
                 .AsParallel()
                 .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                 .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
                 .Select(i => i.PerformComputation())
                 .Reverse();
```

Notice, also, that you can specify multiple configuration options in a parallel query. By chaining these extension methods together, we generate a query that will always run in parallel and will always complete before making the results available in our IEnumerable<T>.

Part 10, Cancellation in PLINQ and the Parallel class

Many routines are parallelized because they are long running processes. When writing an algorithm that will run for a long period of time, its typically a good practice to allow that routine to be cancelled. I previously discussed terminating a parallel loop from within but have not demonstrated how a routine can be cancelled from the caller's perspective. Cancellation in PLINQ and the Task Parallel Library is handled through a new, unified cooperative cancellation model introduced with .NET 4.0.

Cancellation in .NET 4 is based around a new, lightweight struct called CancellationToken. A CancellationToken is a small, thread-safe value type which is generated via a CancellationTokenSource. There are many goals which led to this design. For our purposes, we will focus on a couple of specific design decisions:

- **Cancellation is cooperative**. A calling method can request a cancellation, but it's up to the processing routine to terminate – it is not forced.
- **Cancellation is consistent.** A single method call requests a cancellation on every copied CancellationToken in the routine.

Let's begin by looking at how we can cancel a PLINQ query. Supposed we wanted to provide the option to cancel our query from Part 6:

```
double min = collection
                .AsParallel()
                .Min(item => item.PerformComputation());
```

We would rewrite this to allow for cancellation by adding a call to ParallelEnumerable.WithCancellation as follows:

```
var cts = new CancellationTokenSource();

// Pass cts here to a routine that could,
// in parallel, request a cancellation

try
{
    double min = collection
                    .AsParallel()
                    .WithCancellation(cts.Token)
                    .Min(item => item.PerformComputation());
}
catch (OperationCanceledException e)
{
    // Query was cancelled before it finished
}
```

Here, if the user calls cts.Cancel() before the PLINQ query completes, the query will stop processing and an OperationCanceledException will be raised.

Be aware, however, that cancellation will not be instantaneous. When cts.Cancel() is called, the query will only stop after the current item.PerformComputation() elements all finish processing. cts.Cancel() will prevent PLINQ from scheduling a new task for a new element, but will not stop items which are currently being processed. This goes back to the first goal I mentioned – Cancellation is cooperative. Here, we're requesting the cancellation, but it's up to PLINQ to terminate.

If we wanted to allow cancellation to occur within our routine, we would need to change our routine to accept a CancellationToken and modify it to handle this specific case:

```
public void PerformComputation(CancellationToken token)
{
    for (int i=0; i<this.iterations; ++i)
    {
        // Add a check to see if we've been canceled
        // If a cancel was requested, we'll throw here
        token.ThrowIfCancellationRequested();

        // Do our processing now
        this.RunIteration(i);
    }
}
```

With this overload of PerformComputation, each internal iteration checks to see if a cancellation request was made and will throw an OperationCanceledException at that point, instead of waiting until the method returns. This is good, since it allows us, as developers, to plan for cancellation and terminate our routine in a clean, safe state.

This is handled by changing our PLINQ query to:

```
try
{
    double min = collection
                    .AsParallel()
                    .WithCancellation(cts.Token)
                    .Min(item => item.PerformComputation(cts.Token));
}
catch (OperationCanceledException e)
{
    // Query was cancelled before it finished
}
```

PLINQ is very good about handling this exception, as well. There is a very good chance that multiple items will raise this exception, since the entire purpose of PLINQ is to have multiple items be processed concurrently. PLINQ will take all of the OperationCanceledException instances raised within these methods and merge them into a single OperationCanceledException in the call stack. This is done internally because we added the call to ParallelEnumerable.WithCancellation.

If, however, a different exception is raised by any of the elements, the OperationCanceledException as well as the other Exception will be merged into a single AggregateException.

The Task Parallel Library uses the same cancellation model, as well. Here, we supply our CancellationToken as part of the configuration. The ParallelOptions class contains a property for the CancellationToken. This allows us to cancel a Parallel.For or Parallel.ForEach routine in a very similar manner to our PLINQ query. As an example, we could rewrite our Parallel.ForEach loop from Part 2 to support cancellation by changing it to:

```
try
{
    var cts = new CancellationTokenSource();
    var options = new ParallelOptions()
                    {
                        CancellationToken = cts.Token
                    };
    Parallel.ForEach(customers, options, customer =>
    {
        // Run some process that takes some time...
        DateTime lastContact = theStore.GetLastContact(customer);
        TimeSpan timeSinceContact = DateTime.Now - lastContact;

        // Check for cancellation here
        options.CancellationToken.ThrowIfCancellationRequested();

        // If it's been more than two weeks, send an email and update...
        if (timeSinceContact.Days > 14)
        {
           theStore.EmailCustomer(customer);
           customer.LastEmailContact = DateTime.Now;
        }
    });
}
catch (OperationCanceledException e)
{
    // The loop was cancelled
}
```

Notice that here we use the same approach taken in PLINQ. The Task Parallel Library will automatically handle our cancellation in the same manner as PLINQ, providing a clean, unified model for cancellation of any parallel routine. The TPL performs the same aggregation of the cancellation exceptions as PLINQ, as well, which is why a single exception handler for OperationCanceledException will cleanly handle this scenario. This works because we're using the same CancellationToken provided in the ParallelOptions. If a different exception was thrown by one thread, or a CancellationToken from a different CancellationTokenSource was used to raise our exception, we would instead receive all of our individual exceptions merged into one AggregateException.

Part 11, Divide and Conquer via Parallel.Invoke

Many algorithms are easily written to work via recursion. For example, most data-oriented tasks where a tree of data must be processed are much more easily handled by starting at the root and recursively "walking" the tree. Some algorithms work this way on flat data structures, such as arrays, as well. This is a form of divide and conquer: an algorithm design which is based around breaking up a set of work recursively, "dividing" the total work in each recursive step and "conquering" the work when the remaining work is small enough to be solved easily.

Recursive algorithms, especially ones based on a form of divide and conquer, are often a very good candidate for parallelization.

This is apparent from a common sense standpoint. Since we're dividing up the total work in the algorithm, we have an obvious, built-in partitioning scheme. Once partitioned, the data can be worked upon independently, so there is good, clean isolation of data.

Implementing this type of algorithm is fairly simple. The Parallel class in .NET 4 includes a method suited for this type of operation: Parallel.Invoke. This method works by taking any number of delegates defined as an Action and operating them all in parallel. The method returns when every delegate has completed:

```
Parallel.Invoke(
    () =>
        {
            Console.WriteLine("Action 1 executing in thread {0}",

Thread.CurrentThread.ManagedThreadId);
        },
    () =>
        {
            Console.WriteLine("Action 2 executing in thread {0}",

Thread.CurrentThread.ManagedThreadId);
        },
    () =>
        {
            Console.WriteLine("Action 3 executing in thread {0}",

Thread.CurrentThread.ManagedThreadId);
        }
    );
```

Running this simple example demonstrates the ease of using this method. For example, on my system, I get three separate thread IDs when running the above code. By allowing any number of delegates to be executed directly, concurrently, the Parallel.Invoke method provides us an easy way to parallelize any algorithm based on divide and conquer. We can divide our work in each step and execute each task in parallel, recursively.

For example, suppose we wanted to implement our own quicksort routine. The quicksort algorithm can be designed based on divide and conquer. In each iteration, we pick a pivot point and use that to partition the total array. We swap the elements around the pivot, then recursively sort the lists on each side of the pivot.

For example, let's look at this simple, sequential implementation of quicksort:

```
public static void QuickSort<T>(T[] array) where T : IComparable<T>
{
    QuickSortInternal(array, 0, array.Length - 1);
}

private static void QuickSortInternal<T>(T[] array, int left, int right)
    where T : IComparable<T>
{
    if (left >= right)
    {
        return;
    }

    SwapElements(array, left, (left + right) / 2);
    int last = left;
    for (int current = left + 1; current <= right; ++current)
    {
        if (array[current].CompareTo(array[left]) < 0)
        {
            ++last;
            SwapElements(array, last, current);
        }
    }
    SwapElements(array, left, last);

    QuickSortInternal(array, left, last - 1);
    QuickSortInternal(array, last + 1, right);
}

static void SwapElements<T>(T[] array, int i, int j)
{
    T temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

Here, we implement the quicksort algorithm in a very common, divide and conquer approach. Running this against the built-in Array.Sort routine shows that we get the exact same answers (although the framework's sort routine is slightly faster). On my system, for example, I can use framework's sort to sort ten million random doubles in about 7.3s and this implementation takes about 9.3s on average.

Looking at this routine, though, there is a clear opportunity to parallelize. At the end of QuickSortInternal, we recursively call into QuickSortInternal with each partition of the array after the pivot is chosen. This can be rewritten to use Parallel.Invoke by simply changing it to:

```
// Code above is unchanged...
    SwapElements(array, left, last);

    Parallel.Invoke(
        () => QuickSortInternal(array, left, last - 1),
        () => QuickSortInternal(array, last + 1, right)
    );
}
```

This routine will now run in parallel. When executing, we now see the CPU usage across all cores spike while it executes.

However, there is a significant problem here – by parallelizing this routine, we took it from an execution time of 9.3s to an execution time of approximately 14 seconds! We're using more resources as seen in the CPU usage, but the overall result is a dramatic slowdown in overall processing time.

This occurs because parallelization adds overhead. Each time we split this array, we spawn two new tasks to parallelize this algorithm! This is far, far too many tasks for our cores to operate upon at a single time. In effect, we're "over-parallelizing" this routine. This is a common problem when working with divide and conquer algorithms and leads to an important observation:

When parallelizing a recursive routine, take special care not to add more tasks than necessary to fully utilize your system.

This can be done with a few different approaches, in this case. Typically, the way to handle this is to stop parallelizing the routine at a certain point and revert back to the serial approach. Since the first few recursions will all still be parallelized, our "deeper" recursive tasks will be running in parallel and can take full advantage of the machine. This also dramatically reduces the overhead added by parallelizing, since we're only adding overhead for the first few recursive calls.

There are two basic approaches we can take here. The first approach would be to look at the total work size and if it's smaller than a specific threshold, revert to our serial implementation. In this case, we could just check right-left and if it's under a threshold, call the methods directly instead of using Parallel.Invoke.

The second approach is to track how "deep" in the "tree" we are currently at and if we are below some number of levels, stop parallelizing. This approach is a more general-purpose approach, since it works on routines which parse trees as well as routines working off of a single array, but may not work as well if a poor partitioning strategy is chosen or the tree is not balanced evenly.

This can be written very easily. If we pass a maxDepth parameter into our internal routine, we can restrict the amount of times we parallelize by changing the recursive call to:

```
// Code above is unchanged...
SwapElements(array, left, last);

if (maxDepth < 1)
{
    QuickSortInternal(array, left, last - 1, maxDepth);
    QuickSortInternal(array, last + 1, right, maxDepth);
}
else
{
    --maxDepth;
    Parallel.Invoke(
        () => QuickSortInternal(array, left, last - 1, maxDepth),
        () => QuickSortInternal(array, last + 1, right, maxDepth));
}
```

We no longer allow this to parallelize indefinitely – only to a specific depth, at which time we revert to a serial implementation. By starting the routine with a maxDepth equal to Environment.ProcessorCount, we can restrict the total amount of parallel operations significantly, but still provide adequate work for each processing core.

With this final change, my timings are much better. On average, I get the following timings:

- Framework via Array.Sort: 7.3 seconds
- Serial Quicksort Implementation: 9.3 seconds
- Naive Parallel Implementation: 14 seconds
- Parallel Implementation Restricting Depth: **4.7 seconds**

Finally, we are now faster than the framework's Array.Sort implementation.

Part 12, More on Task Decomposition

Many tasks can be decomposed using a Data Decomposition approach, but often, this is not appropriate. Frequently, decomposing the problem into distinctive tasks that must be performed is a more natural abstraction.

However, as I mentioned in Part 1, Task Decomposition tends to be a bit more difficult than data decomposition and can require a bit more effort. Before we being parallelizing our algorithm based on the tasks being performed, we need to decompose our problem and take special care of certain considerations such as ordering and grouping of tasks.

Up to this point in this series, I've focused on parallelization techniques which are most appropriate when a problem space can be decomposed by data. Using PLINQ and the Parallel class, I've shown how problem spaces where there is a collection of data and each element needs to be processed, can potentially be parallelized.

However, there are many other routines where this is not appropriate. Often, instead of working on a collection of data, there is a single piece of data which must be processed using an algorithm or series of algorithms. Here, there is no collection of data, but there may still be opportunities for parallelism.

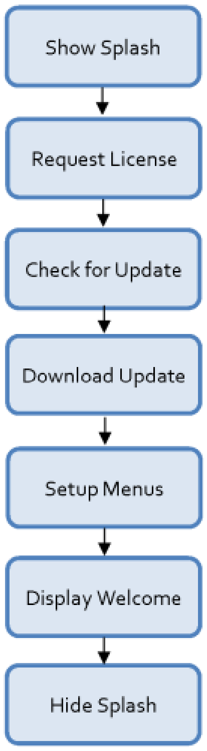
As I mentioned before, in cases like this, the approach is to look at your overall routine and decompose your problem space based on tasks. The idea here is to look for discrete "tasks," individual pieces of work which can be conceptually thought of as a single operation.

Let's revisit the example I used in Part 1, an application startup path. Say we want our program, at startup, to do a bunch of individual actions, or "tasks". The following is our list of duties we must perform right at startup:

- Display a splash screen
- Request a license from our license manager
- Check for an update to the software from our web server
- If an update is available, download it
- Setup our menu structure based on our current license
- Open and display our main, welcome Window
- Hide the splash screen

The first step in Task Decomposition is breaking up the problem space into discrete tasks.

This, naturally, can be abstracted as seven discrete tasks. In the serial version of our program, if we were to diagram this, the general process would appear as:



These tasks, obviously, provide some opportunities for parallelism. Before we can parallelize this routine, we need to analyze these tasks and find any dependencies between tasks. In this case, our dependencies include:

- The splash screen must be displayed first and as quickly as possible.
- We can't download an update before we see whether one exists.
- Our menu structure depends on our license, so we must check for the license before setting up the menus.
- Since our welcome screen will notify the user of an update, we can't show it until we've downloaded the update.
- Since our welcome screen includes menus that are customized based off the licensing, we can't display it until we've received a license.
- We can't hide the splash until our welcome screen is displayed.

By listing our dependencies, we start to see the natural ordering that must occur for the tasks to be processed correctly.

The second step in Task Decomposition is determining the dependencies between tasks and ordering tasks based on their dependencies.

Looking at these tasks and looking at all the dependencies, we quickly see that even a simple decomposition such as this one can get quite complicated. In order to simplify the problem of defining the dependencies, it's often a useful practice to group our tasks into larger, discrete tasks. The goal when grouping tasks is that you want to make each task "group" have as few dependencies as possible to other tasks or groups and then work out the dependencies within that group. Typically, this works best when any external dependency is based on the "last" task within the group when it's ordered, although that is not a firm requirement. This process is often called **Grouping Tasks**. In our case, we can easily group together tasks, effectively turning this into four discrete task groups:

1. Show our splash screen –

This needs to be left as its own task. First, multiple things depend on this task, mainly because we want this to start before any other action and start as quickly as possible.

2. Check for Update and Download the Update if it Exists –

These two tasks logically group together. We know we only download an update if the update exists, so that naturally follows. This task has one dependency as an input and other tasks only rely on the final task within this group.

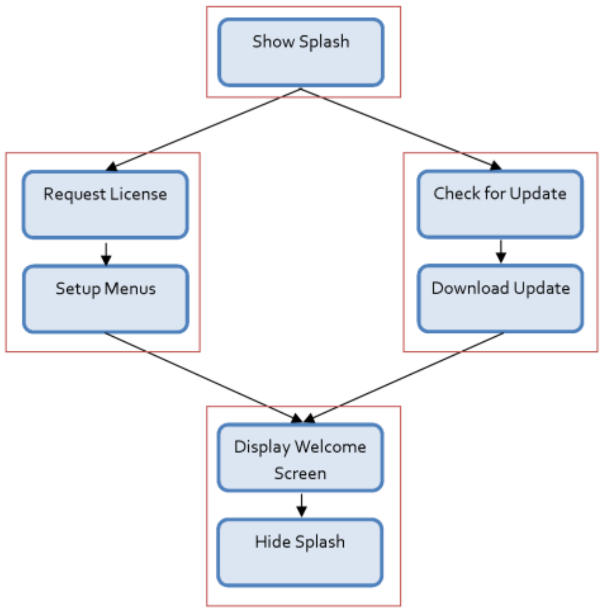
3. Request a License and then Setup the Menus –

Here, we can group these two tasks together. Although we mentioned that our welcome screen depends on the license returned, it also depends on setting up the menu, which is the final task here. Setting up our menus cannot happen until after our license is requested. By grouping these together, we further reduce our problem space.

4. Display welcome and hide splash –

Finally, we can display our welcome window and hide our splash screen. This task group depends on all three previous task groups – it cannot happen until all three of the previous groups have completed.

By grouping the tasks together, we reduce our problem space and can naturally see a pattern for how this process can be parallelized. The diagram below shows one approach:



The orange boxes show each task group, with each task represented within. We can, now, effectively take these tasks and run a large portion of this process in parallel, including the portions which may be the most time consuming. We've now created two parallel paths which our process execution can follow, hopefully speeding up the application startup time dramatically.

The main point to remember here is that, when decomposing your problem space by tasks, you need to:

- Define each discrete action as an individual Task
- Discover dependencies between your tasks
- Group tasks based on their dependencies
- Order the tasks and groups of tasks

Part 13, Introducing the Task class

Once we've used a task-based decomposition to decompose a problem, we need a clean abstraction usable to implement the resulting decomposition. Given that task decomposition is founded upon defining discrete tasks, .NET 4 has introduced a new API for dealing with task related issues, the aptly named Task class.

The Task class is a wrapper for a delegate representing a single, discrete task within your decomposition. We will go into various methods of construction for tasks later, but, when reduced to its fundamentals, an instance of a Task is nothing more than a wrapper around a delegate with some utility functionality added.

In order to fully understand the Task class within the new Task Parallel Library, it is important to realize that a task really is just a delegate – nothing more. In particular, note that I never mentioned threading or parallelism in my description of a Task. Although the Task class exists in the new System.Threading.Tasks namespace:

Tasks are not directly related to threads or multithreading.

Of course, Task instances will typically be used in our implementation of concurrency within an application, but the Task class itself does not provide the concurrency used. The Task API supports using Tasks in an entirely single threaded, synchronous manner.

Tasks are very much like standard delegates. You can execute a task synchronously via Task.RunSynchronously(), or you can use Task.Start() to schedule a task to run, typically asynchronously. This is very similar to using delegate.Invoke to execute a delegate synchronously, or using delegate.BeginInvoke to execute it asynchronously.

The Task class adds some nice functionality on top of a standard delegate which improves usability in both synchronous and multithreaded environments.

The first addition provided by Task is a means of handling cancellation via the new unified cancellation mechanism of .NET 4. If the wrapped delegate within a Task raises an OperationCanceledException during it's operation, which is typically generated via calling ThrowIfCancellationRequested on a CancellationToken, or if the CancellationToken used to construct a Task instance is flagged as canceled, the Task's IsCanceled property will be set to true automatically. This provides a clean way to determine whether a Task has been canceled, often without requiring specific exception handling.

Tasks also provide a clean API which can be used for waiting on a task. Although the Task class explicitly implements IAsyncResult, Tasks provide a nicer usage model than the traditional .NET Asynchronous Programming Model. Instead of needing to track an IAsyncResult handle, you can just directly call Task.Wait() to block until a Task has completed. Overloads exist for providing a timeout, a CancellationToken, or both to prevent waiting indefinitely. In addition, the Task class provides static methods for waiting on multiple tasks – Task.WaitAll and Task.WaitAny, again with overloads providing time out options. This provides a very simple, clean API for waiting on single or multiple tasks.

Finally, Tasks provide a much nicer model for Exception handling. If the delegate wrapped within a Task raises an exception, the exception will automatically get wrapped into an AggregateException and exposed via the Task.Exception property. This exception is stored with the Task directly and does not tear down the application. Later, when Task.Wait() (or Task.WaitAll or Task.WaitAny) is called on this task, an AggregateException will be raised at that point if any of the tasks raised an exception.

For example, suppose we have the following code:

```
Task taskOne = new Task(
                       () =>
                       {
                           throw new ApplicationException("Random Exception!");
                       });
Task taskTwo = new Task(
                       () =>
                       {
                           throw new ArgumentException("Different exception here");
                       });

// Start the tasks
taskOne.Start();
taskTwo.Start();

try
{
    Task.WaitAll(new[] { taskOne, taskTwo });
}
catch (AggregateException e)
{
    Console.WriteLine(e.InnerExceptions.Count);
    foreach (var inner in e.InnerExceptions)
        Console.WriteLine(inner.Message);
}
```

Here, our routine will print:

```
2
Different exception here
Random Exception!
```

Note that we had two separate tasks, each of which raised two distinctly different types of exceptions. We can handle this cleanly, with very little code, in a much nicer manner than the Asynchronous Programming API. We no longer need to handle TargetInvocationException or worry about implementing the Event-based Asynchronous Pattern properly by setting the AsyncCompletedEventArgs.Error property. Instead, we just raise our exception as normal and handle AggregateException in a single location in our calling code.

Part 14, The Different Forms of Task

Before discussing Task creation and actual usage in concurrent environments, I will briefly expand upon my introduction of the Task class and provide a short explanation of the distinct forms of Task. The Task Parallel Library includes four distinct, though related, variations on the Task class.

In my introduction to the Task class, I focused on the most basic version of Task. This version of Task, the standard Task class, is most often used with an Action delegate. This allows you to implement for each task within the task decomposition as a single delegate.

Typically, when using the new threading constructs in .NET 4 and the Task Parallel Library, we use lambda expressions to define anonymous methods. The advantage of using a lambda expression is that it allows the Action delegate to directly use variables in the calling scope. This eliminates the need to make separate Task classes for Action<T>, Action<T1,T2> and all of the other Action<…> delegate types. As an example, suppose we wanted to make a Task to handle the "Show Splash" task from our earlier decomposition. Even if this task required parameters, such as a message to display, we could still use an Action delegate specified via a lambda:

```
// Store this as a local variable
string messageForSplashScreen = GetSplashScreenMessage();
// Create our task
Task showSplashTask = new Task(
    () =>
        {
            // We can use variables in our outer scope,
            // as well as methods scoped to our class!
            this.DisplaySplashScreen(messageForSplashScreen);
        });
```

This provides a huge amount of flexibility. We can use this single form of task for any task which performs an operation, provided the only information we need to track is whether the task has completed successfully or not. This leads to my first observation:

Use a Task with a System.Action delegate for any task for which no result is generated.

This observation leads to an obvious corollary: we also need a way to define a task which generates a result. The Task Parallel Library provides this via the Task<TResult> class.

Task<TResult> subclasses the standard Task class, providing one additional feature – the ability to return a value back to the user of the task. This is done by switching from providing an Action delegate to providing a Func<TResult> delegate. If we decompose our problem and we realize we have one task where its result is required by a future operation, this can be handled via Task<TResult>. For example, suppose we want to make a task for our "Check for Update" task, we could do:

```
Task<bool> checkForUpdateTask = new Task<bool>(
    () =>
        {
            return this.CheckWebsiteForUpdate();
        });
```

Later, we would start this task and perform some other work. At any point in the future, we could get the value from the Task<TResult>.Result property, which will cause our thread to block until the task has finished processing:

```
// This uses Task<bool> checkForUpdateTask generated above...
// Start the task, typically on a background thread
checkForUpdateTask.Start();

// Do some other work on our current thread
this.DoSomeWork();

// Discover, from our background task, whether an update is available
// This will block until our task completes
bool updateAvailable = checkForUpdateTask.Result;
```

This leads me to my second observation:

Use a Task<TResult> with a System.Func<TResult> delegate for any task which generates a result.

Task and Task<TResult> provide a much cleaner alternative to the previous Asynchronous Programming design patterns in the .NET framework. Instead of trying to implement IAsyncResult and providing BeginXXX() and EndXXX() methods, implementing an asynchronous programming API can be as simple as creating a method that returns a Task or Task<TResult>. The client side of the pattern also is dramatically simplified – the client can call a method, then either choose to call task.Wait() or use task.Result when it needs to wait for the operation's completion.

While this provides a much cleaner model for future APIs, there is quite a bit of infrastructure built around the current Asynchronous Programming design patterns. In order to provide a model to work with existing APIs, two other forms of Task exist. There is a constructor for Task which takes an Action<Object> and a state parameter. In addition, there is a constructor for creating a Task<TResult> which takes a Func<Object, TResult> as well as a state parameter. When using these constructors, the state parameter is stored in the Task.AsyncState property.

While these two overloads exist and are usable directly, I strongly recommend avoiding this for new development. The two forms of Task which take an object state parameter exist primarily for interoperability with traditional .NET Asynchronous Programming methodologies. Using lambda expressions to capture variables from the scope of the creator is a much cleaner approach than using the untyped state parameters, since lambda expressions provide full type safety without introducing new variables.

Part 15, Making Tasks Run: The TaskScheduler

In my introduction to the Task class, I specifically made mention that the Task class does not directly provide it's own execution. In addition, I made a strong point that the Task class itself is not directly related to threads or multithreading. Rather, the Task class is used to implement our decomposition of tasks.

Once we've implemented our tasks, we need to execute them. In the Task Parallel Library, the execution of Tasks is handled via an instance of the TaskScheduler class.

The TaskScheduler class is an abstract class which provides a single function: it schedules the tasks and executes them within an appropriate context. This class is the class which actually runs individual Task instances. The .NET Framework provides two (internal) implementations of the TaskScheduler class.

Since a Task, based on our decomposition, should be a self-contained piece of code, parallel execution makes sense when executing tasks. The default implementation of the TaskScheduler class and the one most often used, is based on the ThreadPool. This can be retrieved via the TaskScheduler.Default property and is, by default, what is used when we just start a Task instance with Task.Start().

Normally, when a Task is started by the default TaskScheduler, the task will be treated as a single work item and run on a ThreadPool thread. This pools tasks and provides Task instances all of the advantages of the ThreadPool, including thread pooling for reduced resource usage and an upper cap on the number of work items. In addition, .NET 4 brings us a much improved thread pool, providing work stealing and reduced locking within the thread pool queues. By using the default TaskScheduler, our Tasks are run asynchronously on the ThreadPool.

There is one notable exception to my above statements when using the default TaskScheduler. If a Task is created with the TaskCreationOptions set to TaskCreationOptions.LongRunning, the default TaskScheduler will generate a new thread for that Task, at least in the current implementation. This is useful for Tasks which will persist for most of the lifetime of your application, since it prevents your Task from starving the ThreadPool of one of it's work threads.

The Task Parallel Library provides one other implementation of the TaskScheduler class. In addition to providing a way to schedule tasks on the ThreadPool, the framework allows you to create a TaskScheduler which works within a specified SynchronizationContext. This scheduler can be retrieved within a thread that provides a valid SynchronizationContext by calling the TaskScheduler.FromCurrentSynchronizationContext() method.

This implementation of TaskScheduler is intended for use with user interface development. Windows Forms and Windows Presentation Foundation both require any access to user interface controls to occur on the same thread that created the control. For example, if you want to set the text within a Windows Forms TextBox and you're working on a background thread, that UI call must be marshaled back onto the UI thread. The most common way this is handled depends on the framework being used. In Windows Forms, Control.Invoke or Control.BeginInvoke is most often used. In WPF, the equivelent calls are Dispatcher.Invoke or Dispatcher.BeginInvoke.

As an example, say we're working on a background thread and we want to update a TextBlock in our user interface with a status label. The code would typically look something like:

```
// Within background thread work...
string status = GetUpdatedStatus();

Dispatcher.BeginInvoke(DispatcherPriority.Normal,
    new Action( () =>
        {
            statusLabel.Text = status;
        }));

// Continue on in background method
```

This works fine, but forces your method to take a dependency on WPF or Windows Forms. There is an alternative option, however. Both Windows Forms and WPF, when initialized, setup a SynchronizationContext in their thread, which is available on the UI thread via the SynchronizationContext.Current property. This context is used by classes such as BackgroundWorker to marshal calls back onto the UI thread in a framework-agnostic manner.

The Task Parallel Library provides the same functionality via the TaskScheduler.FromCurrentSynchronizationContext()method. When setting up our Tasks, as long as we're working on the UI thread, we can construct a TaskScheduler via:

TaskScheduler uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();

We then can use this scheduler on any thread to marshal data back onto the UI thread. For example, our code above can then be rewritten as:

```
string status = GetUpdatedStatus();

(new Task(() =>
    {
        statusLabel.Text = status;
    }))
.Start(uiScheduler);

// Continue on in background method
```

This is nice since it allows us to write code that isn't tied to Windows Forms or WPF, but is still fully functional with those technologies. I'll discuss even more uses for the SynchronizationContext based TaskScheduler when I demonstrate task continuations, but even without continuations, this is a very useful construct.

In addition to the two implementations provided by the Task Parallel Library, it is possible to implement your own TaskScheduler. The ParallelExtensionsExtras project within the Samples for Parallel Programming provides nine sample TaskScheduler implementations. These include schedulers which restrict the maximum number of concurrent tasks, run tasks on a single threaded apartment thread, use a new thread per task and more.

Part 16, Creating Tasks via a TaskFactory

The Task class in the Task Parallel Library supplies a large set of features. However, when creating the task and assigning it to a TaskScheduler and starting the Task, there are quite a few steps involved. This gets even more cumbersome when multiple tasks are involved. Each task must be constructed, duplicating any options required, then started individually, potentially on a specific scheduler. At first glance, this makes the new Task class seem like more work than ThreadPool.QueueUserWorkItem in .NET 3.5.

In order to simplify this process and make Tasks simple to use in simple cases, without sacrificing their power and flexibility, the Task Parallel Library added a new class: TaskFactory.

The TaskFactory class is intended to "Provide support for creating and scheduling Task objects." Its entire purpose is to simplify development when working with Task instances. The Task class provides access to the default TaskFactory via the Task.Factory static property. By default, TaskFactory uses the default TaskScheduler to schedule tasks on a ThreadPool thread. By using Task.Factory, we can automatically create and start a task in a single “fire and forget” manner, similar to how we did with ThreadPool.QueueUserWorkItem:

```
Task.Factory.StartNew(() => this.ExecuteBackgroundWork(myData) );
```

This provides us with the same level of simplicity we had with ThreadPool.QueueUserWorkItem, but even more power. For example, we can now easily wait on the task:

```
// Start our task on a background thread
var task = Task.Factory.StartNew(() => this.ExecuteBackgroundWork(myData) );
// Do other work on the main thread,
// while the task above executes in the background
this.ExecuteWorkSynchronously();
// Wait for the background task to finish
task.Wait();
```

TaskFactory simplifies creation and startup of simple background tasks dramatically.
In addition to using the default TaskFactory, it’s often useful to construct a custom TaskFactory. The TaskFactory classincludes an entire set of constructors which allow you to specify the default configuration for every Task instance created by that factory.
This is particularly useful when using a custom TaskScheduler. For example, look at the sample code for starting a task on the UI thread in Part 15:

```
// Given the following, constructed on the UI thread
// TaskScheduler uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
// When inside a background task, we can do
string status = GetUpdatedStatus();

(new Task(() =>
    {
        statusLabel.Text = status;
    }))
.Start(uiScheduler);
```

This is actually quite a bit more complicated than necessary. When we create the uiScheduler instance, we can use that to construct a TaskFactory that will automatically schedule tasks on the UI thread. To do that, we’d create the following on our main thread, prior to constructing our background tasks:

```
// Construct a task scheduler from the current SynchronizationContext (UI thread)
var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
// Construct a new TaskFactory using our UI scheduler
var uiTaskFactory = new TaskFactory(uiScheduler);
```

If we do this, when we’re on a background thread, we can use this new TaskFactory to marshal a Task back onto the UI thread. Our previous code simplifies to:

```
// When inside a background task, we can do
string status = GetUpdatedStatus();
// Update our UI
uiTaskFactory.StartNew( () => statusLabel.Text = status);
```

Notice how much simpler this becomes! By taking advantage of the convenience provided by a custom TaskFactory, we can now marshal to set data on the UI thread in a single, clear line of code!

Part 17, Think Continuations, not Callbacks

In traditional asynchronous programming, we'd often use a callback to handle notification of a background task's completion. The Task class in the Task Parallel Library introduces a cleaner alternative to the traditional callback: continuation tasks.

Asynchronous programming methods typically required callback functions. For example, MSDN's Asynchronous Delegates Programming Sample shows a class that factorizes a number. The original method in the example has the following signature:

```
public static bool Factorize(int number, ref int primefactor1, ref int primefactor2)
{
 //...
```

However, calling this is quite "tricky", even if we modernize the sample to use lambda expressions via C# 3.0. Normally, we could call this method like so:

```
int primeFactor1 = 0;
int primeFactor2 = 0;

bool answer = Factorize(10298312, ref primeFactor1, ref primeFactor2);
Console.WriteLine("{0}/{1}  [Succeeded {2}]", primeFactor1, primeFactor2, answer);
```

If we want to make this operation run in the background and report to the console via a callback, things get tricker. First, we need a delegate definition:

```
public delegate bool AsyncFactorCaller(
      int number,
      ref int primefactor1,
      ref int primefactor2);
```

Then we need to use BeginInvoke to run this method asynchronously:

```
int primeFactor1 = 0;
int primeFactor2 = 0;

AsyncFactorCaller caller  = new AsyncFactorCaller(Factorize);
caller.BeginInvoke(10298312, ref primeFactor1, ref primeFactor2,
   result =>
       {
           int factor1 = 0;
           int factor2 = 0;
           bool answer = caller.EndInvoke(ref factor1, ref factor2, result);
           Console.WriteLine("{0}/{1}  [Succeeded {2}]", factor1, factor2, answer);
       }, null);
```

This works, but is quite difficult to understand from a conceptual standpoint. To combat this, the framework added the Event-based Asynchronous Pattern, but it isn't much easier to understand or author.
Using .NET 4's new Task<T> class and a continuation, we can dramatically simplify the implementation of the above code, as well as make it much more understandable. We do this via the Task.ContinueWith method. This method will schedule a new Task upon completion of the original task and provide the original Task (including its Result if it's a Task<T>) as an argument. Using Task, we can eliminate the delegate and rewrite this code like so:

```
var background = Task.Factory.StartNew(
     () =>
        {
            int primeFactor1 = 0;
            int primeFactor2 = 0;
            bool result = Factorize(10298312, ref primeFactor1, ref primeFactor2);
            return new {
                           Result = result,
                           Factor1 = primeFactor1,
                           Factor2 = primeFactor2
                       };
        });
background.ContinueWith(task => Console.WriteLine("{0}/{1}  [Succeeded {2}]",
                                 task.Result.Factor1,
                                 task.Result.Factor2,
                                 task.Result.Result));
```

This is much simpler to understand, in my opinion. Here, we're explicitly asking to start a new task, then **continue** the task with a resulting task. In our case, our method used ref parameters (this was from the MSDN Sample), so there is a little bit of extra boiler plate involved, but the code is at least easy to understand.

That being said, this isn't dramatically shorter when compared with our C# 3 port of the MSDN code above. However, if we were to extend our requirements a bit, we can start to see more advantages to the Task based approach. For example, supposed we need to report the results in a user interface control instead of reporting it to the Console. This would be a common operation, but now, we have to think about marshaling our calls back to the user interface. This is probably going to require calling Control.Invoke or Dispatcher.Invoke within our callback, forcing us to specify a delegate within the delegate. The maintainability and ease of understanding drops. However, just as a standard Task can be created with a TaskScheduler that uses the UI synchronization context, so too can we continue a task with a specific context. There are Task.ContinueWith method overloads which allow you to provide a TaskScheduler. This means you can schedule the continuation to run on the UI thread, by simply doing:

```
Task.Factory.StartNew(
    () =>
        {
            int primeFactor1 = 0;
            int primeFactor2 = 0;
            bool result = Factorize(10298312, ref primeFactor1, ref primeFactor2);
            return new {
                           Result = result,
                           Factor1 = primeFactor1,
                           Factor2 = primeFactor2
                       };
        }).ContinueWith(task => textBox1.Text = string.Format("{0}/{1}  [Succeeded
{2}]",
                                 task.Result.Factor1,
                                 task.Result.Factor2,
                                 task.Result.Result),
                        TaskScheduler.FromCurrentSynchronizationContext());
```

This is far more understandable than the alternative. By using Task.ContinueWith in conjunction with TaskScheduler.FromCurrentSynchronizationContext(), we get a simple way to push any work onto a background thread and update the user interface on the proper UI thread. This technique works with Windows Presentation Foundation as well as Windows Forms, with no change in methodology.

Part 18, Task Continuations with Multiple Tasks

In my introduction to Task continuations I demonstrated how the Task class provides a more expressive alternative to traditional callbacks. Task continuations provide a much cleaner syntax to traditional callbacks, but there are other reasons to switch to using continuations…

Task continuations provide a clean syntax and a very simple, elegant means of synchronizing asynchronous method results with the user interface. In addition, continuations provide a very simple, elegant means of working with collections of tasks.

Prior to .NET 4, working with multiple related asynchronous method calls was very tricky. If, for example, we wanted to run two asynchronous operations, followed by a single method call which we wanted to run when the first two methods completed, we'd have to program all of the handling ourselves. We would likely need to take some approach such as using a shared callback which synchronized against a common variable, or using a WaitHandle shared within the callbacks to allow one to wait for the second. Although this could be accomplished easily enough, it requires manually placing this handling into every algorithm which requires this form of blocking. This is error prone, difficult and can easily lead to subtle bugs.

Similar to how the Task class static methods providing a way to block until multiple tasks have completed, TaskFactorycontains static methods which allow a continuation to be scheduled upon the completion of multiple tasks: TaskFactory.ContinueWhenAll.

This allows you to easily specify a single delegate to run when a collection of tasks has completed. For example, suppose we have a class which fetches data from the network. This can be a long running operation and potentially fail in certain situations, such as a server being down. As a result, we have three separate servers which we will "query" for our information. Now, suppose we want to grab data from all three servers and verify that the results are the same from all three.

With traditional asynchronous programming in .NET, this would require using three separate callbacks and managing the synchronization between the various operations ourselves. The Task and TaskFactory classes simplify this for us, allowing us to write:

```
var server1 = Task.Factory.StartNew(
                 () => networkClass.GetResults(firstServer) );
var server2 = Task.Factory.StartNew(
                 () => networkClass.GetResults(secondServer) );
var server3 = Task.Factory.StartNew(
                 () => networkClass.GetResults(thirdServer) );

var result = Task.Factory.ContinueWhenAll( new[] {server1, server2, server3 },
                 (tasks) =>
                 {
                       // Propogate exceptions (see below)
                       Task.WaitAll(tasks);

                       return this.CompareTaskResults(
                               tasks[0].Result,
                               tasks[1].Result,
                               tasks[2].Result);
                  });
```

This is clean, simple and elegant. The one complication is the Task.WaitAll(tasks); statement.

Although the continuation will not complete until all three tasks (server1, server2 and server3) have completed, there is a potential snag. If the networkClass.GetResults method fails and raises an exception, we want to make sure to handle it cleanly. By using Task.WaitAll, any exceptions raised within any of our original tasks will get wrapped into a single AggregateException by the WaitAll method, providing us a simplified means of handling the exceptions. If we wait on the continuation, we can trap this AggregateException and handle it cleanly. Without this line, it's possible that an exception could remain uncaught and unhandled by a task, which later might trigger a nasty UnobservedTaskException. This would happen any time two of our original tasks failed.

Just as we can schedule a continuation to occur when an entire collection of tasks has completed, we can just as easily setup a continuation to run when any single task within a collection completes. If, for example, we didn't need to compare the results of all three network locations, but only use one, we could still schedule three tasks. We could then have our completion logic work on the first task which completed and ignore the others. This is done via TaskFactory.ContinueWhenAny:

```
var server1 = Task.Factory.StartNew(
                 () => networkClass.GetResults(firstServer) );
var server2 = Task.Factory.StartNew(
                 () => networkClass.GetResults(secondServer) );
var server3 = Task.Factory.StartNew(
                 () => networkClass.GetResults(thirdServer) );

var result = Task.Factory.ContinueWhenAny( new[] {server1, server2, server3 },
                 (firstTask) =>
                 {
                       return this.ProcessTaskResult(firstTask.Result);
                  });
```

Here, instead of working with all three tasks, we're just using the first task which finishes. This is very useful, as it allows us to easily work with results of multiple operations and "throw away" the others. However, you must take care when using ContinueWhenAny to properly handle exceptions. At some point, you should always wait on each task (or use the Task.Result property) in order to propogate any exceptions raised from within the task. Failing to do so can lead to an UnobservedTaskException.

Part 19, TaskContinuationOptions

My introduction to Task continuations demonstrates continuations on the Task class. In addition, I've shown how continuations allow handling of multiple tasks in a clean, concise manner. Continuations can also be used to handle exceptional situations using a clean, simple syntax.

In addition to standard Task continuations , the Task class provides some options for filtering continuations automatically. This is handled via the TaskContinationOptions enumeration, which provides hints to the TaskSchedulerthat it should only continue based on the operation of the antecedent task.

This is especially useful when dealing with exceptions. For example, we can extend the sample from our earlier continuation discussion to include support for handling exceptions thrown by the Factorize method:

```
// Get a copy of the UI-thread task scheduler up front to use later
var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();

// Start our task
var factorize = Task.Factory.StartNew(
    () =>
        {
            int primeFactor1 = 0;
            int primeFactor2 = 0;
            bool result = Factorize(10298312, ref primeFactor1, ref primeFactor2);
            return new {
                           Result = result,
                                Factor1 = primeFactor1,
                                Factor2 = primeFactor2
                            };
            });

// When we succeed, report the results to the UI
factorize.ContinueWith(task => textBox1.Text = string.Format("{0}/{1}  [Succeeded
{2}]",
                                     task.Result.Factor1,
                                     task.Result.Factor2,
                                     task.Result.Result),
                        CancellationToken.None,
                        TaskContinuationOptions.NotOnFaulted,
                        uiScheduler);

// When we have an exception, report it
factorize.ContinueWith(task =>
                             textBox1.Text = string.Format("Error: {0}",
task.Exception.Message),
                        CancellationToken.None,
                        TaskContinuationOptions.OnlyOnFaulted,
                        uiScheduler);
```

The above code works by using a combination of features. First, we schedule our task, the same way as in the previous example. However, in this case, we use a different overload of Task.ContinueWith which allows us to specify both a specific TaskScheduler (in order to have your continuation run on the UI's synchronization context) as well as a TaskContinuationOption.

In the first continuation, we tell the continuation that we only want it to run when there was not an exception by specifying TaskContinuationOptions.NotOnFaulted. When our factorize task completes successfully, this continuation will automatically run on the UI thread and provide the appropriate feedback.

However, if the factorize task has an exception – for example, if the Factorize method throws an exception due to an improper input value, the second continuation will run. This occurs due to the specification of TaskContinuationOptions.OnlyOnFaulted in the options. In this case, we'll report the error received to the user.

We can use TaskContinuationOptions to filter our continuations by whether or not an exception occurred and whether or not a task was cancelled. This allows us to handle many situations and is especially useful when trying to maintain a valid application state without ever blocking the user interface. The same concepts can be extended even further and allow you to chain together many tasks based on the success of the previous ones. Continuations can even be used to create a state machine with full error handling, all without blocking the user interface thread.

Part 20, Using Task with Existing APIs

Although the Task class provides a huge amount of flexibility for handling asynchronous actions, the .NET Framework still contains a large number of APIs that are based on the previous asynchronous programming model. While Task and Task<T> provide a much nicer syntax as well as extending the flexibility, allowing features such as continuations based on multiple tasks, the existing APIs don't directly support this workflow.

There is a method in the TaskFactory class which can be used to adapt the existing APIs to the new Task class: TaskFactory.FromAsync. This method provides a way to convert from the BeginOperation/EndOperation method pair syntax common through .NET Framework directly to a Task<T> containing the results of the operation in the task's Result parameter.

While this method does exist, it unfortunately comes at a cost – the method overloads are far from simple to decipher and the resulting code is not always as easily understood as newer code based directly on the Task class. For example, a single call to handle WebRequest.BeginGetResponse/EndGetReponse, one of the easiest "pairs" of methods to use, looks like the following:

```
var task = Task.Factory.FromAsync<WebResponse>(
                            request.BeginGetResponse,
                            request.EndGetResponse,
                            null);
```

The compiler is unfortunately unable to infer the correct type, and, as a result, the WebReponse must be explicitly mentioned in the method call. As a result, I typically recommend wrapping this into an extension method to ease use. For example, I would place the above in an extension method like:

```
public static class WebRequestExtensions
{
    public static Task<WebResponse> GetReponseAsync(this WebRequest request)
    {
        return Task.Factory.FromAsync<WebResponse>(
                        request.BeginGetResponse,
                        request.EndGetResponse,
                        null);
    }
}
```

This dramatically simplifies usage. For example, if we wanted to asynchronously check to see if this blog supported XHTML 1.0 and report that in a text box to the user, we could do:

```
var webRequest = WebRequest.Create("http://www.reedcopsey.com");
webRequest.GetReponseAsync().ContinueWith(t =>
    {
        using (var sr = new StreamReader(t.Result.GetResponseStream()))
        {
            string str = sr.ReadLine();;
            this.textBox1.Text = string.Format("Page at {0} supports XHTML 1.0: {1}",
                t.Result.ResponseUri,
                str.Contains("XHTML 1.0"));
        }
    }, TaskScheduler.FromCurrentSynchronizationContext());
```

By using a continuation with a TaskScheduler based on the current synchronization context, we can keep this request asynchronous, check based on the first line of the response string and report the results back on our UI directly.