PATTERNS OF PARALLEL PROGRAMMING

UNDERSTANDING AND APPLYING PARALLEL PATTERNS
WITH THE .NET FRAMEWORK 4 AND VISUAL C#

Stephen Toub
Parallel Computing Platform
Microsoft Corporation

Abstract

This document provides an in-depth tour of support in the Microsoft® .NET Framework 4 for parallel programming. This includes an examination of common parallel patterns and how they're implemented without and with this new support, as well as best practices for developing parallel components utilizing parallel patterns.

Last Updated:

February 16, 2010

TABLE OF CONTENTS

INTRODUCTION

Patterns are everywhere, yielding software development best practices and helping to seed new generations of developers with immediate knowledge of established directions on a wide array of problem spaces. Patterns represent successful (or in the case of anti-patterns, unsuccessful) repeated and common solutions developers have applied time and again in particular architectural and programming domains. Over time, these tried and true practices find themselves with names, stature, and variations, helping further to proliferate their application and to jumpstart many a project.

Patterns don't just manifest at the macro level. Whereas design patterns typically cover architectural structure or methodologies, coding patterns and building blocks also emerge, representing typical ways of implementing a specific mechanism. Such patterns typically become ingrained in our psyche, and we code with them on a daily basis without even thinking about it. These patterns represent solutions to common tasks we encounter repeatedly.

Of course, finding good patterns can happen only after many successful and failed attempts at solutions. Thus for new problem spaces, it can take some time for them to gain a reputation. Such is where our industry lies today with regards to patterns for parallel programming. While developers in high-performance computing have had to develop solutions for supercomputers and clusters for decades, the need for such experiences has only recently found its way to personal computing, as multi-core machines have become the norm for everyday users. As we move forward with multi-core into the manycore era, ensuring that all software is written with as much parallelism and scalability in mind is crucial to the future of the computing industry. This makes patterns in the parallel computing space critical to that same future.

> *"In general, a 'multi-core' chip refers to eight or fewer homogeneous cores in one microprocessor package, whereas a 'manycore' chip has more than eight possibly heterogeneous cores in one microprocessor package. In a manycore system, all cores share the resources and services, including memory and disk access, provided by the operating system." –The Manycore Shift, (Microsoft Corp., 2007)*

In the .NET Framework 4, a slew of new support has been added to handle common needs in parallel programming, to help developers tackle the difficult problem that is programming for multi-core and manycore. Parallel programming is difficult for many reasons and is fraught with perils most developers haven't had to experience. Issues of races, deadlocks, livelocks, priority inversions, two-step dances, and lock convoys typically have no place in a sequential world, and avoiding such issues makes quality patterns all the more important. This new support in the .NET Framework 4 provides support for key parallel patterns along with building blocks to help enable implementations of new ones that arise.

To that end, this document provides an in-depth tour of support in the .NET Framework 4 for parallel programming, common parallel patterns and how they're implemented without and with this new support, and best practices for developing parallel components in this brave new world.

This document only minimally covers the subject of asynchrony for scalable, I/O-bound applications: instead, it focuses predominantly on applications of CPU-bound workloads and of workloads with a balance of both CPU and I/O activity. This document also does not cover Visual F# in Visual Studio 2010, which includes language-based support for several key parallel patterns.

DELIGHTFULLY PARALLEL LOOPS

Arguably the most well-known parallel pattern is that befitting “Embarrassingly Parallel” algorithms. Programs that fit this pattern are able to run well in parallel because the many individual operations being performed may operate in relative independence, with few or no dependencies between operations such that they can be carried out in parallel efficiently. It’s unfortunate that the “embarrassing” moniker has been applied to such programs, as there’s nothing at all embarrassing about them. In fact, if more algorithms and problem domains mapped to the embarrassing parallel domain, the software industry would be in a much better state of affairs. For this reason, many folks have started using alternative names for this pattern, such as “conveniently parallel,” “pleasantly parallel,” and “delightfully parallel,” in order to exemplify the true nature of these problems. If you find yourself trying to parallelize a problem that fits this pattern, consider yourself fortunate, and expect that your parallelization job will be much easier than it otherwise could have been, potentially even a “delightful” activity.

A significant majority of the work in many applications and algorithms is done through loop control constructs. Loops, after all, often enable the application to execute a set of instructions over and over, applying logic to discrete entities, whether those entities are integral values, such as in the case of a **for** loop, or sets of data, such as in the case of a **for each** loop. Many languages have built-in control constructs for these kinds of loops, Microsoft Visual C#® and Microsoft Visual Basic® being among them, the former with **for** and **foreach** keywords, and the latter with **For** and **For Each** keywords. For problems that may be considered delightfully parallel, the entities to be processed by individual iterations of the loops may execute concurrently: thus, we need a mechanism to enable such parallel processing.

IMPLEMENTING A PARALLEL LOOPING CONSTRUCT

As delightfully parallel loops are such a predominant pattern, it’s really important to understand the ins and outs of how they work, and all of the tradeoffs implicit to the pattern. To understand these concepts further, we’ll build a simple parallelized loop using support in the .NET Framework 3.5, prior to the inclusion of the more comprehensive parallelization support introduced in the .NET Framework 4.

First, we need a signature. To parallelize a **for** loop, we’ll implement a method that takes three parameters: a lower-bound, an upper-bound, and a delegate for the loop body that accepts as a parameter an integral value to represent the current iteration index (that delegate will be invoked once for each iteration). Note that we have several options for the behavior of these parameters. With C# and Visual Basic, the vast majority of **for** loops are written in a manner similar to the following:

C#

```
for (int i = 0; i < upperBound; i++)
{
    // ... loop body here
}
```

Visual Basic

```
For i As Integer = 0 To upperBound
    ' ... loop body here
Next
```

Contrary to what a cursory read may tell you, these two loops are not identical: the Visual Basic loop will execute one more iteration than will the C# loop. This is because Visual Basic treats the supplied upper-bound as inclusive, whereas we explicitly specified it in C# to be exclusive through our use of the less-than operator. For our purposes here, we'll follow suit to the C# implementation, and we'll have the upper-bound parameter to our parallelized loop method represent an exclusive upper-bound:

C#

```
public static void MyParallelFor(
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body);
```

Our implementation of this method will invoke the body of the loop once per element in the range **[inclusiveLowerBound,exclusiveUpperBound)**, and will do so with as much parallelization as it can muster. To accomplish that, we first need to understand how much parallelization is possible.

Wisdom in parallel circles often suggests that a good parallel implementation will use one thread per core. After all, with one thread per core, we can keep all cores fully utilized. Any more threads, and the operating system will need to context switch between them, resulting in wasted overhead spent on such activities; any fewer threads, and there's no chance we can take advantage of all that the machine has to offer, as at least one core will be guaranteed to go unutilized. This logic has some validity, at least for certain classes of problems. But the logic is also predicated on an idealized and theoretical concept of the machine. As an example of where this notion may break down, to do anything useful threads involved in the parallel processing need to access data, and accessing data requires trips to caches or main memory or disk or the network or other stores that can cost considerably in terms of access times; while such activities are in flight, a CPU may be idle. As such, while a good parallel implementation may assume a default of one-thread-per-core, an open mindedness to other mappings can be beneficial. For our initial purposes here, however, we'll stick with the one-thread-per core notion.

With the .NET Framework, retrieving the number of logical processors is achieved using the ***System.Environment*** *class, and in particular its* ***ProcessorCount*** *property. Under the covers, .NET retrieves the corresponding value by delegating to the* ***GetSystemInfo*** *native function exposed from kernel32.dll.*

This value doesn't necessarily correlate to the number of physical processors or even to the number of physical cores in the machine. Rather, it takes into account the number of hardware threads available. As an example, on a machine with two sockets, each with four cores, each with two hardware threads (sometimes referred to as hyperthreads), ***Environment.ProcessorCount*** *would return 16.*

Starting with Windows 7 and Windows Server 2008 R2, the Windows operating system supports greater than 64 logical processors, and by default (largely for legacy application reasons), access to these cores is exposed to applications through a new concept known as "processor groups." The .NET Framework does not provide managed access to the processor group APIs, and thus ***Environment.ProcessorCount*** *will return a value capped at 64 (the maximum size of a processor group), even if the machine has a larger number of processors. Additionally, in a 32-bit process,* ***ProcessorCount*** *will be capped further to 32, in order to map well to the 32-bit mask used to represent processor affinity (a requirement that a particular thread be scheduled for execution on only a specific subset of processors).*

Once we know the number of processors we want to target, and hence the number of threads, we can proceed to create one thread per core. Each of those threads will process a portion of the input range, invoking the supplied **Action<int>** delegate for each iteration in that range. Such processing requires another fundamental operation of parallel programming, that of data partitioning. This topic will be discussed in greater depth later in this document; suffice it to say, however, that partitioning is a distinguishing concept in parallel implementations, one that separates it from the larger, containing paradigm of concurrent programming. In concurrent programming, a set of independent operations may all be carried out at the same time. In parallel programming, an operation must first be divided up into individual sub-operations so that each sub-operation may be processed concurrently with the rest; that division and assignment is known as partitioning. For the purposes of this initial implementation, we'll use a simple partitioning scheme: statically dividing the input range into one range per thread.

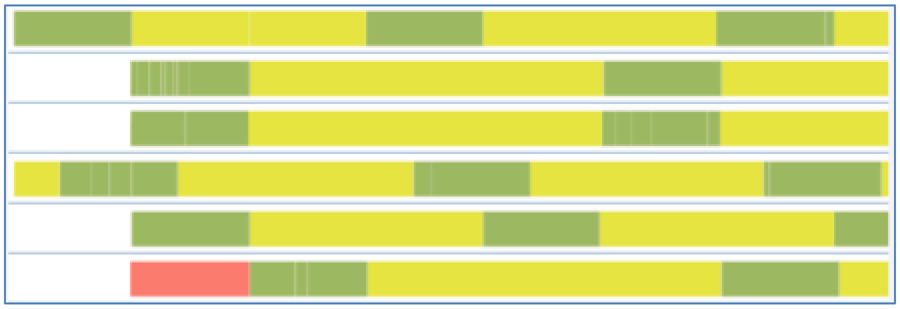
Here is our initial implementation:

C#

```
public static void MyParallelFor(
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
    // Determine the number of iterations to be processed, the number of
    // cores to use, and the approximate number of iterations to process
    // in each thread.
    int size = exclusiveUpperBound - inclusiveLowerBound;
    int numProcs = Environment.ProcessorCount;
    int range = size / numProcs;

    // Use a thread for each partition. Create them all,
    // start them all, wait on them all.
    var threads = new List<Thread>(numProcs);
    for (int p = 0; p < numProcs; p++)
    {
        int start = p * range + inclusiveLowerBound;
        int end = (p == numProcs - 1) ?
            exclusiveUpperBound : start + range;
        threads.Add(new Thread(() => {
            for (int i = start; i < end; i++) body(i);
        }));
    }
    foreach (var thread in threads) thread.Start();
    foreach (var thread in threads) thread.Join();
}
```

There are several interesting things to note about this implementation. One is that for each range, a new thread is utilized. That thread exists purely to process the specified partition, and then it terminates. This has several positive and negative implications. The primary positive to this approach is that we have dedicated threading resources for this loop, and it is up to the operating system to provide fair scheduling for these threads across the system. This positive, however, is typically outweighed by several significant negatives. One such negative is the cost of a thread. By default in the .NET Framework 4, a thread consumes a megabyte of stack space, whether or not that space is used for currently executing functions. In addition, spinning up a new thread and tearing one down are relatively costly actions, especially if compared to the cost of a small loop doing relatively few iterations and little work per iteration. Every time we invoke our loop implementation, new threads will be spun up and torn down.

There's another, potentially more damaging impact: oversubscription. As we move forward in the world of multi-core and into the world of manycore, parallelized components will become more and more common, and it's quite likely that such components will themselves be used concurrently. If such components each used a loop like the above, and in doing so each spun up one thread per core, we'd have two components each fighting for the machine's resources, forcing the operating system to spend more time context switching between components. Context switching is expensive for a variety of reasons, including the need to persist details of a thread's execution prior to the operating system context switching out the thread and replacing it with another. Potentially more importantly, such context switches can have very negative effects on the caching subsystems of the machine. When threads need data, that data needs to be fetched, often from main memory. On modern architectures, the cost of accessing data from main memory is relatively high compared to the cost of running a few instructions over that data. To compensate, hardware designers have introduced layers of caching, which serve to keep small amounts of frequently-used data in hardware significantly less expensive to access than main memory. As a thread executes, the caches for the core on which it's executing tend to fill with data appropriate to that thread's execution, improving its performance. When a thread gets context switched out, the caches will shift to containing data appropriate to that new thread. Filling the caches requires more expensive trips to main memory. As a result, the more context switches there are between threads, the more expensive trips to main memory will be required, as the caches thrash on the differing needs of the threads using them. Given these costs, oversubscription can be a serious cause of performance issues. Luckily, the new concurrency profiler views in Visual Studio 2010 can help to identify these issues, as shown here:



In this screenshot, each horizontal band represents a thread, with time on the x-axis. Green is execution time, red is time spent blocked, and yellow is time where the thread could have run but was preempted by another thread. The more yellow there is, the more oversubscription there is hurting performance.

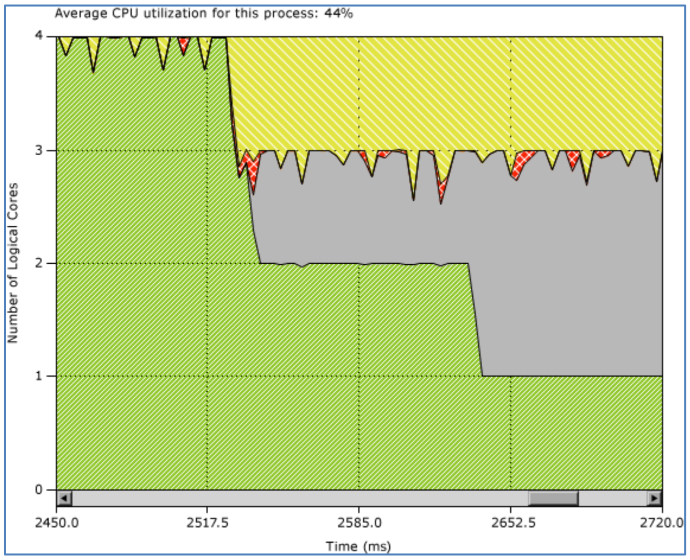
To compensate for these costs associated with using dedicated threads for each loop, we can resort to pools of threads. The system can manage the threads in these pools, dispatching the threads to access work items queued for their processing, and then allowing the threads to return to the pool rather than being torn down. This addresses many of the negatives outlined previously. As threads aren't constantly being created and torn down, the cost of their life cycle is amortized over all the work items they process. Moreover, the manager of the thread pool can enforce an upper-limit on the number of threads associated with the pool at any one time, placing a limit on the amount of memory consumed by the threads, as well as on how much oversubscription is allowed.

Ever since the .NET Framework 1.0, the **System.Threading.ThreadPool** class has provided just such a thread pool, and while the implementation has changed from release to release (and significantly so for the .NET Framework 4), the core concept has remained constant: the .NET Framework maintains a pool of threads that service work items provided to it. The main method for doing this is the static **QueueUserWorkItem**. We can use that support in a revised implementation of our parallel **for** loop:

C#

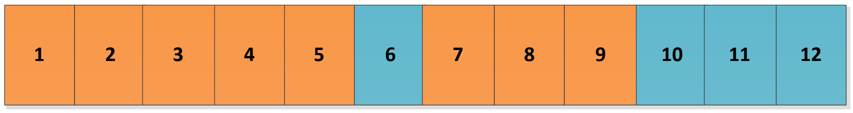
```
public static void MyParallelFor(
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
    // Determine the number of iterations to be processed, the number of
    // cores to use, and the approximate number of iterations to process in
    // each thread.
    int size = exclusiveUpperBound - inclusiveLowerBound;
    int numProcs = Environment.ProcessorCount;
    int range = size / numProcs;

    // Keep track of the number of threads remaining to complete.
    int remaining = numProcs;
    using (ManualResetEvent mre = new ManualResetEvent(false))
    {
        // Create each of the threads.
        for (int p = 0; p < numProcs; p++)
        {
            int start = p * range + inclusiveLowerBound;
            int end = (p == numProcs - 1) ?
                exclusiveUpperBound : start + range;
            ThreadPool.QueueUserWorkItem(delegate {
                for (int i = start; i < end; i++) body(i);
                if (Interlocked.Decrement(ref remaining) == 0) mre.Set();
            });
        }
        // Wait for all threads to complete.
        mre.WaitOne();
    }
}
```

This removes the inefficiencies in our application related to excessive thread creation and tear down, and it minimizes the possibility of oversubscription. However, this inefficiency was just one problem with the implementation: another potential problem has to do with the static partitioning we employed. For workloads that entail the same approximate amount of work per iteration, and when running on a relatively “quiet” machine (meaning a machine doing little else besides the target workload), static partitioning represents an effective and efficient way to partition our data set. However, if the workload is not equivalent for each iteration, either due to the nature of the problem or due to certain partitions completing more slowly due to being preempted by other significant work on the system, we can quickly find ourselves with a load imbalance. The pattern of a load-imbalance is very visible in the following visualization as rendered by the concurrency profiler in Visual Studio 2010.

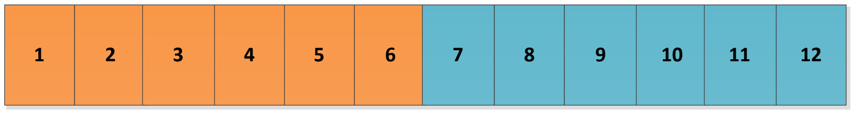


In this output from the profiler, the x-axis is time and the y-axis is the number of cores utilized at that time in the application's executions. Green is utilization by our application, yellow is utilization by another application, red is utilization by a system process, and grey is idle time. This trace resulted from the unfortunate assignment of different amounts of work to each of the partitions; thus, some of those partitions completed processing sooner than the others. Remember back to our assertions earlier about using fewer threads than there are cores to do work? We've now degraded to that situation, in that for a portion of this loop's execution, we were executing with fewer cores than were available.

By way of example, let's consider a parallel loop from 1 to 12 (inclusive on both ends), where each iteration does N seconds of work with N defined as the loop iteration value (that is, iteration #1 will require 1 second of computation, iteration #2 will require two seconds, and so forth). All in all, this loop will require ((12*13)/2) == 78 seconds of sequential processing time. In an ideal loop implementation on a dual core system, we could finish this loop's processing in 39 seconds. This could be accomplished by having one core process iterations 6, 10, 11, and 12, with the other core processing the rest of the iterations.

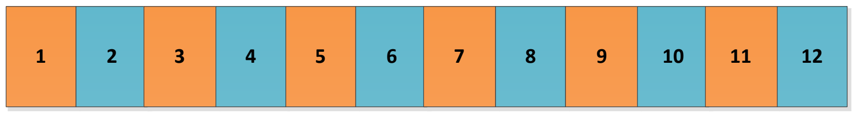


However, with the static partitioning scheme we've employed up until this point, one core will be assigned the range [1,6] and the other the range [7,12].

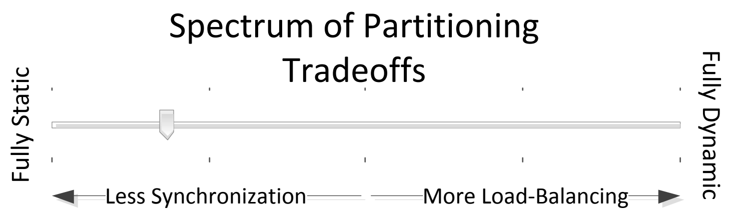


As such, the first core will have 21 seconds worth of work, leaving the latter core 57 seconds worth of work. Since the loop isn't finished until all iterations have been processed, our loop's processing time is limited by the maximum processing time of each of the two partitions, and thus our loop completes in 57 seconds instead of the aforementioned possible 39 seconds. This represents an approximate 50 percent decrease in potential performance, due solely to an inefficient partitioning. Now you can see why partitioning has such a fundamental place in parallel programming.

Different variations on static partitioning are possible. For example, rather than assigning ranges, we could use a form of round-robin, where each thread has a unique identifier in the range [0,# of threads), and where each thread processes indices from the loop where the index mod the number of threads matches the thread's identifier. For example, with the iteration space [0,12) and with four threads, thread #0 would process iteration values 0, 3, 6, and 9; thread #1 would process iteration values 1, 4, 7, and 10; and so on. If we were to apply this kind of round-robin partitioning to the previous example, instead of one thread taking 21 seconds and the other taking 57 seconds, one thread would require 36 seconds and the other 42 seconds, resulting in a much smaller discrepancy from the optimal runtime of 38 seconds.



To do the best static partitioning possible, you need to be able to accurately predict ahead of time how long all the iterations will take. That's rarely feasible, resulting in a need for a more dynamic partitioning, where the system can adapt to changing workloads quickly. We can address this by shifting to the other end of the partitioning tradeoffs spectrum, with as much load-balancing as possible.



To do that, rather than pushing to each of the threads a given set of indices to process, we can have the threads compete for iterations. We employ a pool of the remaining iterations to be processed, which initially starts filled with all iterations. Until all of the iterations have been processed, each thread goes to the iteration pool, removes an iteration value, processes it, and then repeats. In this manner, we can achieve in a greedy fashion an approximation for the optimal level of load-balancing possible (the true optimum could only be achieved with a priori knowledge of exactly how long each iteration would take). If a thread gets stuck processing a particular long iteration, the other threads will compensate by processing work from the pool in the meantime. Of course, even with this scheme you can still find yourself with a far from optimal partitioning (which could occur if one thread happened to get stuck with several pieces of work significantly larger than the rest), but without knowledge of how much processing time a given piece of work will require, there's little more that can be done.

Here's an example implementation that takes load-balancing to this extreme. The pool of iteration values is maintained as a single integer representing the next iteration available, and the threads involved in the processing "remove items" by atomically incrementing this integer:

C#

```csharp
public static void MyParallelFor(
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
    // Get the number of processors, initialize the number of remaining
    // threads, and set the starting point for the iteration.
    int numProcs = Environment.ProcessorCount;
    int remainingWorkItems = numProcs;
    int nextIteration = inclusiveLowerBound;

    using (ManualResetEvent mre = new ManualResetEvent(false))
    {
        // Create each of the work items.
        for (int p = 0; p < numProcs; p++)
        {
            ThreadPool.QueueUserWorkItem(delegate
            {
                int index;
                while ((index = Interlocked.Increment(
                    ref nextIteration) - 1) < exclusiveUpperBound)
                {
                    body(index);
                }
                if (Interlocked.Decrement(ref remainingWorkItems) == 0)
                    mre.Set();
            });
        }

        // Wait for all threads to complete.
        mre.WaitOne();
    }
}
```

This is not a panacea, unfortunately. We've gone to the other end of the spectrum, trading quality load-balancing for additional overheads. In our previous static partitioning implementations, threads were assigned ranges and were then able to process those ranges completely independently from the other threads. There was no need to synchronize with other threads in order to determine what to do next, because every thread could determine independently what work it needed to get done. For workloads that have a lot of work per iteration, the cost of synchronizing between threads so that each can determine what to do next is negligible. But for workloads that do very little work per iteration, that synchronization cost can be so expensive (relatively) as to overshadow the actual work being performed by the loop. This can make it more expensive to execute in parallel than to execute serially.

Consider an analogy: shopping with some friends at a grocery store. You come into the store with a grocery list, and you rip the list into one piece per friend, such that every friend is responsible for retrieving the elements on his or her list. If the amount of time required to retrieve the elements on each list is approximately the same as on every other list, you've done a good job of partitioning the work amongst your team, and will likely find that your time at the store is significantly less than if you had done *all of the shopping yourself. But now suppose that each list is not well balanced, with all of the items on one friend's list spread out over the entire store, while all of the items on another friend's list are concentrated in the same aisle. You could address this inequity by assigning out one element at a time. Every time a friend retrieves a food item, he or she brings it back to you at the front of the store and determines in conjunction with you which food item to retrieve next. If a particular food item takes a particularly long time to retrieve, such as ordering a custom cut piece of meat at the deli counter, the overhead of having to go back and forth between you and the merchandise may be negligible. For simply retrieving a can from a shelf, however, the overhead of those trips can be dominant, especially if multiple items to be retrieved from a shelf were near each other and could have all been retrieved in the same trip with minimal additional time. You could spend so much time (relatively) parceling out work to your friends and determining what each should buy next that it would be faster for you to just grab all of the food items in your list yourself.*

Of course, we don't need to pick one extreme or the other. As with most patterns, there are variations on themes. For example, in the grocery store analogy, you could have each of your friends grab several items at a time, rather than grabbing one at a time. This amortizes the overhead across the size of a batch, while still having some amount of dynamism:

C#

```
public static void MyParallelFor(
    int inclusiveLowerBound, int exclusiveUpperBound, Action<int> body)
{
    // Get the number of processors, initialize the number of remaining
    // threads, and set the starting point for the iteration.
    int numProcs = Environment.ProcessorCount;
    int remainingWorkItems = numProcs;
    int nextIteration = inclusiveLowerBound;
    const int batchSize = 3;

    using (ManualResetEvent mre = new ManualResetEvent(false)) {
        // Create each of the work items.
        for (int p = 0; p < numProcs; p++) {
            ThreadPool.QueueUserWorkItem(delegate {
                int index;
                while ((index = Interlocked.Add(
                    ref nextIteration, batchSize) - batchSize)
                    < exclusiveUpperBound)
                {
                    // In a real implementation, we'd need to handle
                    // overflow on this arithmetic.
                    int end = index + batchSize;
                    if (end >= exclusiveUpperBound) end = exclusiveUpperBound;
                    for (int i = index; i < end; i++) body(i);
                }
                if (Interlocked.Decrement(ref remainingWorkItems) == 0)
                    mre.Set();
            });
        }

        // Wait for all threads to complete
        mre.WaitOne();
        }
    }
```

No matter what tradeoffs you make between overheads and load-balancing, they are tradeoffs. For a particular problem, you might be able to code up a custom parallel loop algorithm mapping to this pattern that suits your particular problem best. That could result in quite a bit of custom code, however. In general, a good solution is one that provides quality results for most problems, minimizing overheads while providing sufficient load-balancing, and the .NET Framework 4 includes just such an implementation in the new **System.Threading.Tasks.Parallel** class.

PARALLEL.FOR

As delightfully parallel problems represent one of the most common patterns in parallel programming, it's natural that when support for parallel programming is added to a mainstream library, support for delightfully parallel loops is included. The .NET Framework 4 provides this in the form of the static **Parallel** class in the new **System.Threading.Tasks** namespace in **mscorlib.dll**. The **Parallel** class provides just three methods, albeit each with several overloads. One of these methods is **For**, providing multiple signatures, one of which is almost identical to the signature for **MyParallelFor** shown previously:

C#

```
public static ParallelLoopResult For(
    int fromInclusive, int toExclusive, Action<int> body);
```

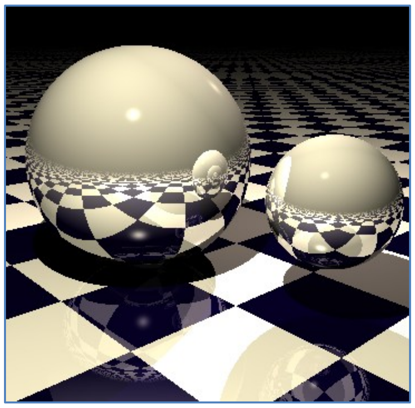
As with our previous implementations, the **For** method accepts three parameters: an inclusive lower-bound, an exclusive upper-bound, and a delegate to be invoked for each iteration. Unlike our implementations, it also returns a **ParallelLoopResult** value type, which contains details on the completed loop; more on that later.

Internally, the **For** method performs in a manner similar to our previous implementations. By default, it uses work queued to the .NET Framework **ThreadPool** to execute the loop, and with as much parallelism as it can muster, it invokes the provided delegate once for each iteration. However, **Parallel.For** and its overload set provide a whole lot more than this:

- **Exception handling.** If one iteration of the loop throws an exception, all of the threads participating in the loop attempt to stop processing as soon as possible (by default, iterations currently executing will not be interrupted, but the loop control logic tries to prevent additional iterations from starting). Once all processing has ceased, all unhandled exceptions are gathered and thrown in aggregate in an **AggregateException** instance. This exception type provides support for multiple "inner exceptions," whereas most .NET Framework exception types support only a single inner exception. For more information about **AggregateException**, see http://msdn.microsoft.com/magazine/ee321571.aspx.
- **Breaking out of a loop early.** This is supported in a manner similar to the **break** keyword in C# and the **Exit For** construct in Visual Basic. Support is also provided for understanding whether the current iteration should abandon its work because of occurrences in other iterations that will cause the loop to end early. This is the primary reason for the **ParallelLoopResult** return value, shown in the **Parallel.For** signature, which helps a caller to understand if a loop ended prematurely, and if so, why.
- **Long ranges.** In addition to overloads that support working with **Int32**-based ranges, overloads are provided for working with **Int64**-based ranges.
- **Thread-local state.** Several overloads provide support for thread-local state. More information on this support will be provided later in this document in the section on aggregation patterns.

- **Configuration options.** Multiple aspects of a loop's execution may be controlled, including limiting the number of threads used to process the loop.
- **Nested parallelism.** If you use a **Parallel.For** loop within another **Parallel.For** loop, they coordinate with each other to share threading resources. Similarly, it's ok to use two **Parallel.For** loops concurrently, as they'll work together to share threading resources in the underlying pool rather than both assuming they own all cores on the machine.
- **Dynamic thread counts.** **Parallel.For** was designed to accommodate workloads that change in complexity over time, such that some portions of the workload may be more compute-bound than others. As such, it may be advantageous to the processing of the loop for the number of threads involved in the processing to change over time, rather than being statically set, as was done in all of our implementations shown earlier.
- **Efficient load balancing.** Parallel.For supports load balancing in a very sophisticated manner, much more so than the simple mechanisms shown earlier. It takes into account a large variety of potential workloads and tries to maximize efficiency while minimizing overheads. The partitioning implementation is based on a chunking mechanism where the chunk size increases over time. This helps to ensure quality load balancing when there are only a few iterations, while minimizing overhead when there are many. In addition, it tries to ensure that most of a thread's iterations are focused in the same region of the iteration space in order to provide high cache locality.

Parallel.For is applicable to a wide-range of delightfully parallel problems, serving as an implementation of this quintessential pattern. As an example of its application, the parallel programming samples for the .NET Framework 4 (available at http://code.msdn.microsoft.com/ParExtSamples) include a ray tracer. Here's a screenshot:



Ray tracing is fundamentally a delightfully parallel problem. Each individual pixel in the image is generated by firing an imaginary ray of light, examining the color of that ray as it bounces off of and through objects in the scene, and storing the resulting color. Every pixel is thus independent of every other pixel, allowing them all to be processed in parallel. Here are the relevant code snippets from that sample:

C#

```
void RenderSequential(Scene scene, Int32[] rgb)
{
    Camera camera = scene.Camera;
    for (int y = 0; y < screenHeight; y++)
    {
        int stride = y * screenWidth;
        for (int x = 0; x < screenWidth; x++)
```

```
            {
                Color color = TraceRay(
                    new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
                rgb[x + stride] = color.ToInt32();
            }
        }
    }

    void RenderParallel(Scene scene, Int32[] rgb)
    {
        Camera camera = scene.Camera;
        Parallel.For(0, screenHeight, y =>
        {
            int stride = y * screenWidth;
            for (int x = 0; x < screenWidth; x++)
            {
                Color color = TraceRay(
                    new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
                rgb[x + stride] = color.ToInt32();
            }
        });
    }
```

Notice that there are very few differences between the sequential and parallel implementation, limited only to changing the C# **for** and Visual Basic **For** language constructs into the **Parallel.For** method call.

PARALLEL.FOREACH

A **for** loop is a very specialized loop. Its purpose is to iterate through a specific kind of data set, a data set made up of numbers that represent a range. The more generalized concept is iterating through any data set, and constructs for such a pattern exist in C# with the **foreach** keyword and in Visual Basic with the **For Each** construct.

Consider the following for loop:

C#

```
for(int i=0; i<10; i++)
{
    // ... Process i.
}
```

Using the **Enumerable** class from LINQ, we can generate an **IEnumerable<int>** that represents the same range, and iterate through that range using a **foreach**:

C#

```
foreach(int i in Enumerable.Range(0, 10))
{
    // ... Process i.
}
```

We can accomplish much more complicated iteration patterns by changing the data returned in the enumerable. Of course, as it is a generalized looping construct, we can use a **foreach** to iterate through any enumerable data set. This makes it very powerful, and a parallelized implementation is similarly quite powerful in the parallel realm. As with a parallel **for**, a parallel **for each** represents a fundamental pattern in parallel programming.

Implementing a parallel **for each** is similar in concept to implementing a parallel **for**. You need multiple threads to process data in parallel, and you need to partition the data, assigning the partitions to the threads doing the processing. In our dynamically partitioned **MyParallelFor** implementation, the data set remaining was represented by a single integer that stored the next iteration. In a **for each** implementation, we can store it as an **IEnumerator<T>** for the data set. This enumerator must be protected by a critical section so that only one thread at a time may mutate it. Here is an example implementation:

C#

```csharp
public static void MyParallelForEach<T>(
    IEnumerable<T> source, Action<T> body)
{
    int numProcs = Environment.ProcessorCount;
    int remainingWorkItems = numProcs;

    using (var enumerator = source.GetEnumerator())
    {
        using (ManualResetEvent mre = new ManualResetEvent(false))
        {
            // Create each of the work items.
            for (int p = 0; p < numProcs; p++)
            {
                ThreadPool.QueueUserWorkItem(delegate
                {
                    // Iterate until there's no more work.
                    while (true)
                    {
                        // Get the next item under a lock,
                        // then process that item.
                        T nextItem;
                        lock (enumerator)
                        {
                            if (!enumerator.MoveNext()) break;
                            nextItem = enumerator.Current;
                        }
                        body(nextItem);
                    }
                    if (Interlocked.Decrement(ref remainingWorkItems) == 0)
                        mre.Set();
                });
            }

            // Wait for all threads to complete.
            mre.WaitOne();
        }
    }
}
```

As with the **MyParallelFor** implementations shown earlier, there are lots of implicit tradeoffs being made in this implementation, and as with the **MyParallelFor**, they all come down to tradeoffs between simplicity, overheads, and load balancing. Taking locks is expensive, and this implementation is taking and releasing a lock for each element in the enumerable; while costly, this does enable the utmost in load balancing, as every thread only grabs one item at a time, allowing other threads to assist should one thread run into an unexpectedly expensive element. We could tradeoff some cost for some load balancing by retrieving multiple items (rather than just one) while holding the lock. By acquiring the lock, obtaining multiple items from the enumerator, and then releasing the lock, we amortize the cost of acquisition and release over multiple elements, rather than paying the cost for each element. This benefit comes at the expense of less load balancing, since once a thread has grabbed several items, it is responsible for processing all of those items, even if some of them happen to be more expensive than the bulk of the others.

We can decrease costs in other ways, as well. For example, the implementation shown previously always uses the enumerator's **MoveNext**/**Current** support, but it might be the case that the source input **IEnumerable<T>** also implements the **IList<T>** interface, in which case the implementation could use less costly partitioning, such as that employed earlier by **MyParallelFor**:

C#

```
public static void MyParallelForEach<T>(IEnumerable<T> source, Action<T> body)
{
    IList<T> sourceList = source as IList<T>;
    if (sourceList != null)
    {
        // This assumes the IList<T> implementation's indexer is safe
        // for concurrent get access.
        MyParallelFor(0, sourceList.Count, i => body(sourceList[i]));
    }
    else
    {
        // ...
    }
}
```

As with **Parallel.For**, the .NET Framework 4's **Parallel** class provides support for this pattern, in the form of the **ForEach** method. Overloads of **ForEach** provide support for many of the same things for which overloads of **For** provide support, including breaking out of loops early, sophisticated partitioning, and thread count dynamism. The simplest overload of **ForEach** provides a signature almost identical to the signature shown above:

C#

```
public static ParallelLoopResult ForEach<TSource>(
    IEnumerable<TSource> source, Action<TSource> body);
```

As an example application, consider a **Student** record that contains a settable **GradePointAverage** property as well as a readable collection of **Test** records, each of which has a grade and a weight. We have a set of such student records, and we want to iterate through each, calculating each student's grades based on the associated tests. Sequentially, the code looks as follows:

C#

```
foreach (var student in students)
{
    student.GradePointAverage =
        student.Tests.Select(test => test.Grade * test.Weight).Sum();
}
```

To parallelize this, we take advantage of **Parallel.ForEach**:

C#

```
Parallel.ForEach(students, student =>
{
    student.GradePointAverage =
        student.Tests.Select(test => test.Grade * test.Weight).Sum();
});
```

PROCESSING NON-INTEGRAL RANGES

The **Parallel** class in the .NET Framework 4 provides overloads for working with ranges of **Int32** and **Int64** values. However, **for** loops in languages like C# and Visual Basic can be used to iterate through non-integral ranges.

Consider a type **Node<T>** that represents a linked list:

C#
```
class Node<T>
{
    public Node<T> Prev, Next;
    public T Data;
}
```

Given an instance **head** of such a **Node<T>**, we can use a **for** loop to iterate through the list:

C#
```
for(Node<T> i = head; i != null; i = i.Next)
{
    // ... Process node i.
}
```

Parallel.For does not contain overloads for working with **Node<T>**, and **Node<T>** does not implement **IEnumerable<T>**, preventing its direct usage with **Parallel.ForEach**. To compensate, we can use C# iterators to create an **Iterate** method which will yield an **IEnumerable<T>** to iterate through the **Node<T>**:

C#
```
public static IEnumerable<Node<T>> Iterate(Node<T> head)
{
    for (Node<T> i = head; i != null; i = i.Next)
    {
        yield return i;
    }
}
```

With such a method in hand, we can now use a combination of **Parallel.ForEach** and **Iterate** to approximate a **Parallel.For** implementation that does work with **Node<T>**:

C#
```
Parallel.ForEach(Iterate(head), i =>
{
    // ... Process node i.
});
```

This same technique can be applied to a wide variety of scenarios. Keep in mind, however, that the **IEnumerator<T>** interface isn't thread-safe, which means that **Parallel.ForEach** needs to take locks when accessing the data source. While **ForEach** internally uses some smarts to try to amortize the cost of such locks over the processing, this is still overhead that needs to be overcome by more work in the body of the **ForEach** in order for good speedups to be achieved.

Parallel.ForEach has optimizations used when working on data sources that can be indexed into, such as lists and arrays, and in those cases the need for locking is decreased (this is similar to the example implementation shown previously, where **MyParallelForEach** was able to use **MyParallelFor** in processing an **IList<T>**). Thus, even though there is both time and memory cost associated with creating an array from an enumerable, performance may actually be improved in some cases by transforming the iteration space into a list or an array, which can be done using LINQ. For example:

C#

```
Parallel.ForEach(Iterate(head).ToArray(), i =>
{
    // ... Process node i.
});
```

The format of a **for** construct in C# and a **For** in Visual Basic may also be generalized into a generic **Iterate** method:

C#

```
public static IEnumerable<T> Iterate<T>(
    Func<T> initialization, Func<T, bool> condition, Func<T, T> update)
{
    for (T i = initialization(); condition(i); i = update(i))
    {
        yield return i;
    }
}
```

While incurring extra overheads for all of the delegate invocations, this now also provides a generalized mechanism for iterating. The **Node<T>** example can be re-implemented as follows:

C#

```
Parallel.ForEach(Iterate(() => head, i => i != null, i => i.Next), i =>
{
    // ... Process node i.
});
```

BREAKING OUT OF LOOPS EARLY

Exiting out of loops early is a fairly common pattern, one that doesn't go away when parallelism is introduced. To help simplify these use cases, the **Parallel.For** and **Parallel.ForEach** methods support several mechanisms for breaking out of loops early, each of which has different behaviors and targets different requirements.

PLANNED EXIT

Several overloads of **Parallel.For** and **Parallel.ForEach** pass a **ParallelLoopState** instance to the body delegate. Included in this type's surface area are four members relevant to this discussion: methods **Stop** and **Break**, and properties **IsStopped** and **LowestBreakIteration**.

When an iteration calls **Stop**, the loop control logic will attempt to prevent additional iterations of the loop from starting. Once there are no more iterations executing, the loop method will return successfully (that is, without an exception). The return type of **Parallel.For** and **Parallel.ForEach** is a **ParallelLoopResult** value type: if **Stop** caused the loop to exit early, the result's **IsCompleted** property will return **false**.

C#

```
ParallelLoopResult loopResult =
Parallel.For(0, N, (int i, ParallelLoopState loop) =>
{
    // ...
    if (someCondition)
    {
        loop.Stop();
        return;
    }
    // ...
});
Console.WriteLine("Ran to completion: " + loopResult.IsCompleted);
```

For long running iterations, the **IsStopped** property enables one iteration to detect when another iteration has called **Stop** in order to bail earlier than it otherwise would:

C#

```
ParallelLoopResult loopResult =
Parallel.For(0, N, (int i, ParallelLoopState loop) =>
{
    // ...
    if (someCondition)
    {
        loop.Stop();
        return;
    }
    // ...
    while (true)
    {
        if (loop.IsStopped) return;
        // ...
    }
});
```

Break is very similar to **Stop**, except **Break** provides additional guarantees. Whereas **Stop** informs the loop control logic that no more iterations need be run, **Break** informs the control logic that no iterations after the current one need be run (for example, where the iteration number is higher or where the data comes after the current element in the data source), but that iterations prior to the current one still need to be run. It doesn't guarantee that iterations after the current one haven't already run or started running, though it will try to avoid more starting after the current one. **Break** may be called from multiple iterations, and the lowest iteration from which **Break** was called is the one that takes effect; this iteration number can be retrieved from the **ParallelLoopState**'s **LowestBreakIteration** property, a nullable value. **ParallelLoopResult** offers a similar **LowestBreakIteration** property.

This leads to a decision matrix that can be used to interpret a **ParallelLoopResult**:

- **IsCompleted** == **true**
  - All iterations were processed.
  - If **IsCompleted** == **true**, **LowestBreakIteration.HasValue** will be **false**.
- **IsCompleted** == **false** && **LowestBreakIteration.HasValue** == **false**
  - Stop was used to exit the loop early
- **IsCompleted** == **false** && **LowestBreakIteration.HasValue** == **true**
  - Break was used to exit the loop early, and **LowestBreakIteration.Value** contains the lowest iteration from which **Break** was called.

Here is an example of using **Break** with a loop:

C#

```
var output = new TResult[N];
var loopResult = Parallel.For(0, N, (int i, ParallelLoopState loop) =>
{
    if (someCondition)
    {
        loop.Break();
        return;
    }
    output[i] = Compute(i);
});
long completedUpTo = N;
if (!loopResult.IsCompleted && loopResult.LowestBreakIteration.HasValue)
{
    completedUpTo = loopResult.LowestBreakIteration.Value;
}
```

Stop is typically useful for unordered search scenarios, where the loop is looking for something and can bail as soon as it finds it. **Break** is typically useful for ordered search scenarios, where all of the data up until some point in the source needs to be processed, with that point based on some search criteria.

UNPLANNED EXIT

The previously mentioned mechanisms for exiting a loop early are based on the body of the loop performing an action to bail out. Sometimes, however, we want an entity external to the loop to be able to request that the loop terminate; this is known as cancellation.

Cancellation is supported in parallel loops through the new **System.Threading.CancellationToken** type introduced in the .NET Framework 4. Overloads of all of the methods on **Parallel** accept a **ParallelOptions** instance, and one of the properties on **ParallelOptions** is a **CancellationToken**. Simply set this **CancellationToken** property to the **CancellationToken** that should be monitored for cancellation, and provide that options instance to the loop's invocation. The loop will monitor the token, and if it finds that cancellation has been requested, it will again stop launching more iterations, wait for all existing iterations to complete, and then throw an **OperationCanceledException**.

C#

```
private CancellationTokenSource _cts = new CancellationTokenSource();
```

```csharp
// ...
var options = new ParallelOptions { CancellationToken = _cts.Token };
try
{
    Parallel.For(0, N, options, i =>
    {
        // ...
    });
}
catch(OperationCanceledException oce)
{
    // ... Handle loop cancellation.
}
```

Stop and **Break** allow a loop itself to proactively exit early and successfully, and cancellation allows an external entity to the loop to request its early termination. It's also possible for something in the loop's body to go wrong, resulting in an early termination of the loop that was not expected.

In a sequential loop, an unhandled exception thrown out of a loop causes the looping construct to immediately cease. The parallel loops in the .NET Framework 4 get as close to this behavior as is possible while still being reliable and predictable. This means that when an exception is thrown out of an iteration, the **Parallel** methods attempt to prevent additional iterations from starting, though already started iterations are not forcibly terminated. Once all iterations have ceased, the loop gathers up any exceptions that have been thrown, wraps them in a **System.AggregateException**, and throws that aggregate out of the loop.

As with **Stop** and **Break**, for cases where individual operations may run for a long time (and thus may delay the loop's exit), it may be advantageous for iterations of a loop to be able to check whether other iterations have faulted. To accommodate that, **ParallelLoopState** exposes an **IsExceptional** property (in addition to the aforementioned **IsStopped** and **LowestBreakIteration** properties), which indicates whether another iteration has thrown an unhandled exception. Iterations may cooperatively check this property, allowing a long-running iteration to cooperatively exit early when it detects that another iteration failed.

While this exception logic does support exiting out of a loop early, it is not the recommended mechanism for doing so. Rather, it exists to assist in exceptional cases, cases where breaking out early wasn't an intentional part of the algorithm. As is the case with sequential constructs, exceptions should not be relied upon for control flow.

Note, too, that this exceptions behavior isn't optional. In the face of unhandled exceptions, there's no way to tell the looping construct to allow the entire loop to complete execution, just as there's no built-in way to do that with a serial **for** loop. If you wanted that behavior with a serial **for** loop, you'd likely end up writing code like the following:

C#

```csharp
var exceptions = new Queue<Exception>();
for (int i = 0; i < N; i++)
{
    try
    {
        // ... Loop body goes here.
    }
    catch (Exception exc) { exceptions.Enqueue(exc); }
}
if (exceptions.Count > 0) throw new AggregateException(exceptions);
```

If this is the behavior you desire, that same manual handling is also possible using **Parallel.For**:

C#

```
var exceptions = new ConcurrentQueue<Exception>();
Parallel.For(0, N, i =>
{
    try
    {
        // ... Loop body goes here.
    }
    catch (Exception exc) { exceptions.Enqueue(exc); }
});
if (!exceptions.IsEmpty) throw new AggregateException(exceptions);
```

EMPLOYING MULTIPLE EXIT STRATEGIES

It's possible that multiple exit strategies could all be employed together, concurrently; we're dealing with parallelism, after all. In such cases, exceptions always win: if unhandled exceptions have occurred, the loop will always propagate those exceptions, regardless of whether **Stop** or **Break** was called or whether cancellation was requested.

If no exceptions occurred but the **CancellationToken** was signaled and either **Stop** or **Break** was used, there's a potential race as to whether the loop will notice the cancellation prior to exiting. If it does, the loop will exit with an **OperationCanceledException**. If it doesn't, it will exit due to the **Stop**/**Break** as explained previously.

However, **Stop** and **Break** may not be used together. If the loop detects that one iteration called **Stop** while another called **Break**, the invocation of whichever method ended up being invoked second will result in an exception being thrown. This is enforced due to the conflicting guarantees provided by **Stop** and **Break**.

For long running iterations, there are multiple properties an iteration might want to check to see whether it should bail early: **IsStopped**, **LowestBreakIteration**, **IsExceptional**, and so on. To simplify this, **ParallelLoopState** also provides a **ShouldExitCurrentIteration** property, which consolidates all of those checks in an efficient manner. The loop itself checks this value prior to invoking additional iterations.

PARALLELENUMERABLE.FORALL

Parallel LINQ (PLINQ), exposed from **System.Core.dll** in the .NET Framework 4, provides a parallelized implementation of all of the .NET Framework standard query operators. This includes **Select** (projections), **Where** (filters), **OrderBy** (sorting), and a host of others. PLINQ also provides several additional operators not present in its serial counterpart. One such operator is **AsParallel**, which enables parallel processing of a LINQ-to-Objects query. Another such operator is **ForAll**.

Partitioning of data has already been discussed to some extent when discussing **Parallel.For** and **Parallel.ForEach**, and merging will be discussed in greater depth later in this document. Suffice it to say, however, that to process an input data set in parallel, portions of that data set must be distributed to each thread partaking in the processing, and when all of the processing is complete, those partitions typically need to be merged back together to form the single output stream expected by the caller:

C#

```
List<InputData> inputData = ...;
foreach (var o in inputData.AsParallel().Select(i => new OutputData(i)))
{
    ProcessOutput(o);
}
```

Both partitioning and merging incur costs, and in parallel programming, we strive to avoid such costs as they're pure overhead when compared to a serial implementation. Partitioning can't be avoided if data must be processed in parallel, but in some cases we can avoid merging, such as if the work to be done for each resulting item can be processed in parallel with the work for every other resulting item. To accomplish this, PLINQ provides the **ForAll** operator, which avoids the merge and executes a delegate for each output element:

C#

```
List<InputData> inputData = ...;
inputData.AsParallel().Select(i => new OutputData(i)).ForAll(o =>
{
    ProcessOutput(o);
});
```

ANTI-PATTERNS

Superman has his kryptonite. Matter has its anti-matter. And patterns have their anti-patterns. Patterns prescribe good ways to solve certain problems, but that doesn't mean they're not without potential pitfalls. There are several potential problems to look out for with **Parallel.For**, **Parallel.ForEach**, and **ParallelEnumerable.ForAll**.

SHARED DATA

The new parallelism constructs in the .NET Framework 4 help to alleviate most of the boilerplate code you'd otherwise have to write to parallelize delightfully parallel problems. As you saw earlier, the amount of code necessary just to implement a simple and naïve **MyParallelFor** implementation is vexing, and the amount of code required to do it well is reams more. These constructs do not, however, automatically ensure that your code is thread-safe. Iterations within a parallel loop must be independent, and if they're not independent, you must ensure that the iterations are safe to execute concurrently with each other by doing the appropriate synchronization.

ITERATION VARIANTS

In managed applications, one of the most common patterns used with a **for/For** loop is iterating from 0 inclusive to some upper bound (typically exclusive in C# and inclusive in Visual Basic). However, there are several variations on this pattern that, while not nearly as common, are still not rare.

DOWNWARD ITERATION

It’s not uncommon to see loops iterating down from an upper-bound exclusive to 0 inclusive:

C#
```
for(int i=upperBound-1; i>=0; --i) { /*...*/ }
```

Such a loop is typically (though not always) constructed due to dependencies between the iterations; after all, if all of the iterations are independent, why write a more complex form of the loop if both the upward and downward iteration have the same results?

Parallelizing such a loop is often fraught with peril, due to these likely dependencies between iterations. If there are no dependencies between iterations, the **Parallel.For** method may be used to iterate from an inclusive lower bound to an exclusive upper bound, as directionality shouldn’t matter: in the extreme case of parallelism, on a machine with **upperBound** number of cores, all iterations of the loop may execute concurrently, and direction is irrelevant.

When parallelizing downward-iterating loops, proceed with caution. Downward iteration is often a sign of a less than delightfully parallel problem.

STEPPED ITERATION

Another pattern of a **for** loop that is less common than the previous cases, but still is not rare, is one involving a step value other than one. A typical **for** loop may look like this:

C#
```
for (int i = 0; i < upperBound; i++) { /*...*/ }
```

But it’s also possible for the update statement to increase the iteration value by a different amount: for example to iterate through only the even values between the bounds:

C#
```
for (int i = 0; i < upperBound; i += 2) { /*...*/ }
```

Parallel.For does not provide direct support for such patterns. However, **Parallel** can still be used to implement such patterns. One mechanism for doing so is through an iterator approach like that shown earlier for iterating through linked lists:

C#
```
private static IEnumerable<int> Iterate(
    int fromInclusive, int toExclusive, int step)
{
    for (int i = fromInclusive; i < toExclusive; i += step) yield return i;
}
```

A **Parallel.ForEach** loop can now be used to perform the iteration. For example, the previous code snippet for iterating the even values between 0 and **upperBound** can be coded as:

C#
```
Parallel.ForEach(Iterate(0, upperBound, 2), i=> { /*...*/ });
```

As discussed earlier, such an implementation, while straightforward, also incurs the additional costs of forcing the **Parallel.ForEach** to takes locks while accessing the iterator. This drives up the per-element overhead of parallelization, demanding that more work be performed per element to make up for the increased overhead in order to still achieve parallelization speedups.

Another approach is to do the relevant math manually. Here is an implementation of a **ParallelForWithStep** loop that accepts a step parameter and is built on top of **Parallel.For**:

C#

```
public static void ParallelForWithStep(
    int fromInclusive, int toExclusive, int step, Action<int> body)
{
    if (step < 1)
    {
        throw new ArgumentOutOfRangeException("step");
    }
    else if (step == 1)
    {
        Parallel.For(fromInclusive, toExclusive, body);
    }
    else // step > 1
    {
        int len = (int)Math.Ceiling((toExclusive - fromInclusive) / (double)step);
        Parallel.For(0, len, i => body(fromInclusive + (i * step)));
    }
}
```

This approach is less flexible than the iterator approach, but it also involves significantly less overhead. Threads are not bottlenecked serializing on an enumerator; instead, they need only pay the cost of a small amount of math plus an extra delegate invocation per iteration.

VERY SMALL LOOP BODIES

As previously mentioned, the **Parallel** class is implemented in a manner so as to provide for quality load balancing while incurring as little overhead as possible. There is still overhead, though. The overhead incurred by **Parallel.For** is largely centered around two costs:

1) **Delegate invocations.** If you squint at previous examples of **Parallel.For**, a call to **Parallel.For** looks a lot like a C# **for** loop or a Visual Basic **For** loop. Don't be fooled: it's still a method call. One consequence of this is that the "body" of the **Parallel.For** "loop" is supplied to the method call as a delegate. Invoking a delegate incurs approximately the same amount of cost as a virtual method call.
2) **Synchronization between threads for load balancing.** While these costs are minimized as much as possible, any amount of load balancing will incur some cost, and the more load balancing employed, the more synchronization is necessary.

For medium to large loop bodies, these costs are largely negligible. But as the size of the loop's body decreases, the overheads become more noticeable. And for very small bodies, the loop can be completely dominated by this overhead's cost. To support parallelization of very small loop bodies requires addressing both #1 and #2 above. One pattern for this involves chunking the input into ranges, and then instead of replacing a sequential loop with a parallel loop, wrapping the sequential loop with a parallel loop.

The **System.Concurrent.Collections.Partitioner** class provides a **Create** method overload that accepts an integral range and returns an **OrderablePartitioner<Tuple<Int32,Int32>>** (a variant for **Int64** instead of **Int32** is also available):

C#
```
public static OrderablePartitioner<Tuple<long, long>> Create(
    long fromInclusive, long toExclusive);
```

Overloads of **Parallel.ForEach** accept instances of **Partitioner<T>** and **OrderablePartitioner<T>** as sources, allowing you to pass the result of a call to **Partitioner.Create** into a call to **Parallel.ForEach**. For now, think of both **Partitioner<T>** and **OrderablePartitioner<T>** as an **IEnumerable<T>**.

The **Tuple<Int32,Int32>** represents a range from an inclusive value to an exclusive value. Consider the following sequential loop:

C#
```
for (int i = from; i < to; i++)
{
    // ... Process i.
}
```

We could use a **Parallel.For** to parallelize it as follows:

C#
```
Parallel.For(from, to, i =>
{
    // ... Process i.
});
```

Or, we could use **Parallel.ForEach** with a call to **Partitioner.Create**, wrapping a sequential loop over the range provided in the **Tuple<Int32, Int32>**, where the **inclusiveLowerBound** is represented by the tuple's **Item1** and where the **exclusiveUpperBound** is represented by the tuple's **Item2**:

C#
```
Parallel.ForEach(Partitioner.Create(from, to), range =>
{
    for (int i = range.Item1; i < range.Item2; i++)
    {
        // ... process i
    }
});
```

While more complex, this affords us the ability to process very small loop bodies by eschewing some of the aforementioned costs. Rather than invoking a delegate for each body invocation, we're now amortizing the cost of the delegate invocation across all elements in the chunked range. Additionally, as far as the parallel loop is concerned, there are only a few elements to be processed: each range, rather than each index. This implicitly decreases the cost of synchronization because there are fewer elements to load-balance.

While **Parallel.For** should be considered the best option for parallelizing for loops, if performance measurements show that speedups are not being achieved or that they're smaller than expected, you can try an approach like the one shown using **Parallel.ForEach** in conjunction with **Partitioner.Create**.

TOO FINE-GRAINED, TOO COURSE GRAINED

The previous anti-pattern outlined the difficulties that arise from having loop bodies that are too small. In addition to problems that implicitly result in such small bodies, it's also possible to end up in this situation by decomposing the problem to the wrong granularity.

Earlier in this section, we demonstrated a simple parallelized ray tracer:

C#

```csharp
void RenderParallel(Scene scene, Int32[] rgb)
{
    Camera camera = scene.Camera;
    Parallel.For(0, screenHeight, y =>
    {
        int stride = y * screenWidth;
        for (int x = 0; x < screenWidth; x++)
        {
            Color color = TraceRay(
                new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
            rgb[x + stride] = color.ToInt32();
        }
    });
}
```

Note that there are two loops here, both of which are actually safe to parallelize:

C#

```csharp
void RenderParallel(Scene scene, Int32[] rgb)
{
    Camera camera = scene.Camera;
    Parallel.For(0, screenHeight, y =>
    {
        int stride = y * screenWidth;
        Parallel.For(0, screenWidth, x =>
        {
            Color color = TraceRay(
                new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
            rgb[x + stride] = color.ToInt32();
        });
    });
}
```

The question then arises: why and when someone would choose to parallelize one or both of these loops? There are multiple, competing principles. On the one hand, the idea of writing parallelized software that scales to any number of cores you throw at it implies that you should decompose as much as possible, so that regardless of the number of cores available, there will always be enough work to go around. This principle suggests both loops should be parallelized. On the other hand, we've already seen the performance implications that can result if there's not enough work inside of a parallel loop to warrant its parallelization, implying that only the outer loop should be parallelized in order to maintain a meaty body.

The answer is that the best balance is found through performance testing. If the overheads of parallelization are minimal as compared to the work being done, parallelize as much as possible: in this case, that would mean parallelizing both loops. If the overheads of parallelizing the inner loop would degrade performance on most systems, think twice before doing so, as it'll likely be best only to parallelize the outer loop.

There are of course some caveats to this (in parallel programming, there are caveats to everything; there are caveats to the caveats). Parallelization of only the outer loop demands that the outer loop has enough work to saturate enough processors. In our ray tracer example, what if the image being ray traced was very wide and short, such that it had a small height? In such a case, there may only be a few iterations for the outer loop to parallelize, resulting in too coarse-grained parallelization, in which case parallelizing the inner loop could actually be beneficial, even if the overheads of parallelizing the inner loop would otherwise not warrant its parallelization.

Another option to consider in such cases is flattening the loops, such that you end up with one loop instead of two. This eliminates the cost of extra partitions and merges that would be incurred on the inner loop's parallelization:

C#
```
void RenderParallel(Scene scene, Int32[] rgb)
{
    int totalPixels = screenHeight * screenWidth;
    Camera camera = scene.Camera;
    Parallel.For(0, totalPixels, i =>
    {
        int y = i / screenWidth, x = i % screenWidth;
        Color color = TraceRay(
            new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
        rgb[i] = color.ToInt32();
    });
}
```

If in doing such flattening the body of the loop becomes too small (which given the cost of **TraceRay** in this example is unlikely), the pattern presented earlier for very small loop bodies may also be employed:

C#
```
void RenderParallel(Scene scene, Int32[] rgb)
{
    int totalPixels = screenHeight * screenWidth;
    Camera camera = scene.Camera;
    Parallel.ForEach(Partitioner.Create(0, totalPixels), range =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
            int y = i / screenWidth, x = i % screenWidth;
            Color color = TraceRay(
                new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
            rgb[i] = color.ToInt32();
        }
    });
}
```

NON-THREAD-SAFE ILIST<T> IMPLEMENTATIONS

Both PLINQ and **Parallel.ForEach** query their data sources for several interface implementations. Accessing an **IEnumerable<T>** incurs significant cost, due to needing to lock on the enumerator and make virtual methods calls to **MoveNext** and **Current** for each element. In contrast, getting an element from an **IList<T>** can be done without locks, as elements of an **IList<T>** are independent. Thus, both PLINQ and **Parallel.ForEach** automatically use a source's **IList<T>** implementation if one is available.

In most cases, this is the right decision. However, in very rare cases, an implementation of **IList<T>** may not be thread-safe for reading due to the get accessor for the list's indexer mutating shared state. There are two predominant reasons why an implementation might do this:

1. **The data structures stores data in a non-indexible manner, such that it must traverse the data structure to find the requested index.** In such a case, the data structure may try to amortize the cost of access by keeping track of the last element accessed, assuming that accesses will occur in a largely sequential manner, making it cheaper to start a search from the previously accessed element than starting from scratch. Consider a theoretical linked list implementation as an example. A linked list does not typically support direct indexing; rather, if you want to access the 42nd element of the list, you need to start at the beginning, prior to the head, and move to the next element 42 times. As an optimization, the list could maintain a reference to the most recently accessed element. If you accessed element 42 and then element 43, upon accessing 42 the list would cache a reference to the 42nd element, thus making access to 43 a single move next rather than 43 of them from the beginning. If the implementation doesn't take thread-safety into account, these mutations are likely not thread-safe.

2. **Loading the data structure is expensive.** In such cases, the data can be lazy-loaded (loaded on first access) to defer or avoid some of the initialization costs. If getting data from the list forces initialization, then mutations could occur due to indexing into the list.

> *There are only a few, obscure occurrences of this in the .NET Framework. One example is* ***System.Data.Linq.EntitySet<TEntity>****. This type implements* ***IList<TEntity>*** *with support for lazy loading, such that the first thing its indexer's* ***get*** *accessor does is load the data into the* ***EntitySet<TEntity>*** *if loading hasn't already occurred.*

To work around such cases if you do come across them, you can force both PLINQ and **Parallel.ForEach** to use the **IEnumerable<T>** implementation rather than the **IList<T>** implementation. This can be achieved in two ways:

1) **Use System.Collections.Concurrent.Partitioner's Create method.** There is an overload specific to **IEnumerable<T>** that will ensure this interface implementation (not one for **IList<T>**) is used. **Partitioner.Create** returns an instance of a **Partitioner<T>**, for which there are overloads on **Parallel.ForEach** and in PLINQ.

C#
```csharp
// Will use IList<T> implementation if source implements it.
IEnumerable<T> source = ...;
Parallel.ForEach(source, item => { /*...*/ });

// Will use source's IEnumerable<T> implementation.
IEnumerable<T> source = ...;
Parallel.ForEach(Partitioner.Create(source), item => { /*...*/ });
```

2) **Append onto the data source a call to Enumerable.Select.** The **Select** simply serves to prevent PLINQ and **Parallel.ForEach** from finding the original source's **IList<T>** implementation.

C#

```
// Will use IList<T> implementation if source implements it.
IEnumerable<T> source = ...;
Parallel.ForEach(source, item => { /*...*/ });

// Will only provide an IEnumerable<T> implementation.
IEnumerable<T> source = ...;
Parallel.ForEach(source.Select(t => t), item => { /*...*/ });
```

PARALLEL.FOREACH OVER A PARALLELQUERY<T>

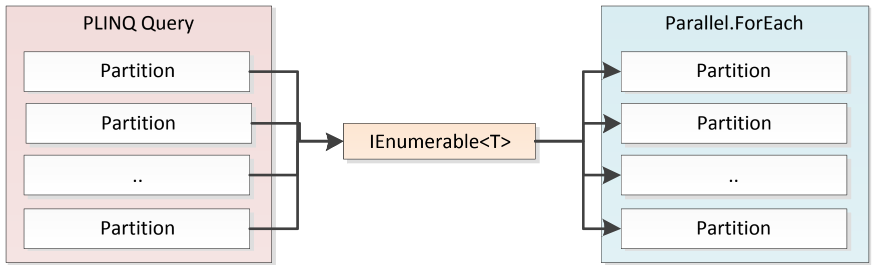
PLINQ's **ParallelEnumerable** type operates in terms of **ParallelQuery<T>** objects. Such objects are returned from the **AsParallel** extension method, and all of PLINQ's operators consume and generate instances of **ParallelQuery<T>**. **ParallelQuery<T>** is itself an **IEnumerable<T>**, which means it can be iterated over and may be consumed by anything that understands how to work with an **IEnumerable<T>**.

Parallel.ForEach is one such construct that works with **IEnumerable<T>**. As such, it may be tempting to write code that follows a pattern similar to the following:

C#

```
var q = from d in data.AsParallel() ... select d;
Parallel.ForEach(q, item => { /* Process item. */ });
```

While this works correctly, it incurs unnecessary costs. In order for PLINQ to stream its output data into an **IEnumerable<T>**, PLINQ must merge the data being generated by all of the threads involved in query processing so that the multiple sets of data can be consumed by code expecting only one. Conversely, when accepting an input **IEnumerable<T>**, **Parallel.ForEach** must consume the single data stream and partition it into multiple data streams for processing in parallel. Thus, by passing a **ParallelQuery<T>** to a **Parallel.ForEach**, in the .NET Framework 4 the data from the PLINQ query will be merged and will then be repartitioned by the **Parallel.ForEach**. This can be costly.

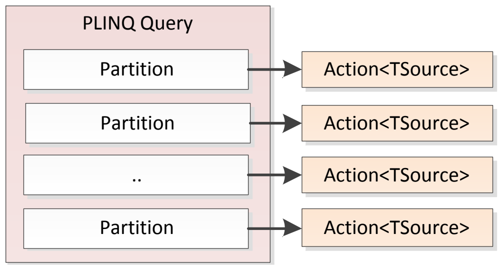


Instead, PLINQ's **ParallelEnumerable.ForAll** method should be used. Rewriting the previous code as follows will avoid the spurious merge and repartition:

C#

```
var q = (from d in data.AsParallel() ... select d);
q.ForAll(item => { /* Process item. */ });
```

This allows the output of all partitions to be processed in parallel, as discussed in the previous section on **ParallelEnumerable.ForAll**.



THREAD AFFINITY IN ACCESSING SOURCE DATA

Both **Parallel.ForEach** and **ParallelEnumerable.ForAll** rely on each of the threads participating in the loop to pull data from the source enumerator. While both **ForEach** and **ForAll** ensure that the enumerator is accessed in a thread-safe manner (only one thread at a time will use **MoveNext** and **Current**, and will do so atomically with respect to other threads in the loop), it's still the case that multiple threads may use **MoveNext** over time. In general, this shouldn't be a problem. However, in some rare cases the implementation of **MoveNext** may have thread affinity, meaning that for correctness purposes it should always be accessed from the same thread, and perhaps even from a specific thread. An example of this could be if **MoveNext** were accessing a user interface (UI) control in Windows Forms or Windows Presentation Foundation in order to retrieve its data, or if the control were pulling data from the object model of one of the Microsoft Office applications. While such thread affinity is not recommended, avoiding it may not be possible.

In such cases, the consuming implementation needs to change to ensure that the data source is only accessed by the thread making the call to the loop. That can be achieved with a producer/consumer pattern (many more details on that pattern are provided later in this document), using code similar in style to the following:

C#

```
static void ForEachWithEnumerationOnMainThread<T>(
     IEnumerable<T> source, Action<T> body)
{
    var collectedData = new BlockingCollection<T>();
    var loop = Task.Factory.StartNew(() =>
        Parallel.ForEach(collectedData.GetConsumingEnumerable(), body));
    try
    {
        foreach (var item in source) collectedData.Add(item);
    }
    finally { collectedData.CompleteAdding(); }
    loop.Wait();
    }
```

The **Parallel.ForEach** executes in the background by pulling the data from a shared collection that is populated by the main thread enumerating the data source and copying its contents into the shared collection. This solves the issue of thread affinity with the data source by ensuring that the data source is only accessed on the main thread. If, however, all access to the individual elements must also be done only on the main thread, parallelization is infeasible.

PARALLEL LOOPS FOR I/O-BOUND WORKLOADS IN SCALABLE APPLICATIONS

It can be extremely tempting to utilize the delightfully parallel looping constructs in the .NET Framework 4 for I/O-bound workloads. And in many cases, it's quite reasonable to do so as a quick-and-easy approach to getting up and running with better performance.

Consider the need to ping a set of machines. We can do this quite easily using the **System.Net.NetworkInformation.Ping** class, along with LINQ:

C#

```
var addrs = new[] { addr1, addr2, ..., addrN };
var pings = from addr in addrs
            select new Ping().Send(addr);
foreach (var ping in pings)
    Console.WriteLine("{0}: {1}", ping.Status, ping.Address);
```

By adding just a few characters, we can easily parallelize this operation using PLINQ:

C#

```
var pings = from addr in addrs.AsParallel()
            select new Ping().Send(addr);
foreach (var ping in pings)
    Console.WriteLine("{0}: {1}", ping.Status, ping.Address);
```

Rather than using a single thread to ping these machines one after the other, this code uses multiple threads to do so, typically greatly decreasing the time it takes to complete the operation. Of course, in this case, the work I'm doing is not at all CPU-bound, and yet by default PLINQ uses a number of threads equal to the number of logical processors, an appropriate heuristic for CPU-bound workloads but not for I/O-bound. As such, we can utilize PLINQ's **WithDegreeOfParallelism** method to get the work done even faster by using more threads (assuming there are enough addresses being pinged to make good use of all of these threads):

C#

```
var pings = from addr in addrs.AsParallel().WithDegreeOfParallelism(16)
            select new Ping().Send(addr);
foreach (var ping in pings)
    Console.WriteLine("{0}: {1}", ping.Status, ping.Address);
```

For a client application on a desktop machine doing just this one operation, using threads in this manner typically does not lead to any significant problems. However, if this code were running in an ASP.NET application, it could be deadly to the system. Threads have a non-negligible cost, a cost measurable in both the memory required for their associated data structures and stack space, and in the extra impact it places on the operating system and its scheduler. When threads are doing real work, this cost is justified. But when threads are simply sitting around blocked waiting for an I/O operation to complete, they're dead weight. Especially in Web applications, where thousands of users may be bombarding the system with requests, that extra and unnecessary weight can bring a server to a crawl. For applications where scalability in terms of concurrent users is at a premium, it's imperative not to write code like that shown above, even though it's really simple to write. There are other solutions, however.

> ***WithDegreeOfParallelism*** *changes the number of threads required to execute and complete the PLINQ query, but it does not force that number of threads into existence. If the number is larger than the number of threads available in the* ***ThreadPool****, it may take some time for the* ***ThreadPool*** *thread-injection logic to inject enough threads to complete the processing of the query. To force it to get there faster, you can employ the* ***ThreadPool.SetMinThreads*** *method.*

The **System.Threading.Tasks.Task** class will be discussed later in this document. In short, however, note that a **Task** instance represents an asynchronous operation. Typically these are computationally-intensive operations, but the **Task** abstraction can also be used to represent I/O-bound operations and without tying up a thread in the process. As an example of this, the samples available at http://code.msdn.microsoft.com/ParExtSamples include extension methods for the **Ping** class that provide asynchronous versions of the **Send** method to return a **Task<PingReply>**. Using such methods, we can rewrite our previous method as follows:

C#

```
var pings = (from addr in addrs
             select new Ping().SendTask(addr, null)).ToArray();
Task.WaitAll(pings);
foreach (Task<PingReply> ping in pings)
    Console.WriteLine("{0}: {1}", ping.Result.Status, ping.Result.Address);
```

This new solution will asynchronously send a ping to all of the addresses, but no threads (other than the main thread waiting on the results) will be blocked in the process; only when the pings complete will threads be utilized briefly to process the results, the actual computational work. This results in a much more scalable solution, one that may be used in applications that demand scalability. Note, too, that taking advantage of **Task.Factory.ContinueWhenAll** (to be discussed later), the code can even avoid blocking the main iteration thread, as illustrated in the following example:

C#

```
var pings = (from addr in addrs
             select new Ping().SendTask(addr, null)).ToArray();
Task.Factory.ContinueWhenAll(pings, _ =>
{
    Task.WaitAll(pings);
    foreach (var ping in pings)
        Console.WriteLine("{0}: {1}", ping.Result.Status, ping.Result.Address);
});
```

The example here was shown utilizing the **Ping** class, which implements the Event-based Asynchronous Pattern (EAP). This pattern for asynchronous operation was introduced in the .NET Framework 2.0, and is based on .NET events that are raised asynchronously when an operation completes.

A more prevalent pattern throughout the .NET Framework is the Asynchronous Programming Model (APM) pattern, which has existed in the .NET Framework since its inception. Sometimes referred to as the "begin/end" pattern, this pattern is based on a pair of methods: a "begin" method that starts the asynchronous operation, and an "end" method that joins with it, retrieving any results of the invocation or the exception from the operation.

To help integrate with this pattern, the aforementioned **Task** class can also be used to wrap an APM invocation, which can again help with the scalability, utilizing the **Task.Factory.FromAsync** method. This support can then be used to build an approximation of asynchronous methods, as is done in the **Task.Factory.Iterate** extension method available in the samples at samples available at http://code.msdn.microsoft.com/ParExtSamples. For more information, see http://blogs.msdn.com/pfxteam/9809774.aspx. Through its asynchronous workflow functionality, F# in Visual Studio 2010 also provides first-class language support for writing asynchronous methods. For more information, see http://msdn.microsoft.com/en-us/library/dd233182(VS.100).aspx. The incubation language Axum, available for download at http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx, also includes first-class language support for writing asynchronous methods.

FORK/JOIN

The patterns employed for delightfully parallel loops are really a subset of a larger set of patterns centered around "fork/join." In fork/join patterns, work is "forked" such that several pieces of work are launched asynchronously. That forked work is later joined with in order to ensure that all of the processing has completed, and potentially to retrieve the results of that processing if it wasn't utilized entirely for side-effecting behavior. Loops are a prime example of this: we fork the processing of loop iterations, and we join such that the parallel loop invocation only completes when all concurrent processing is done.

The new **System.Threading.Tasks** namespace in the .NET Framework 4 contains a significant wealth of support for fork/join patterns. In addition to the **Parallel.For**, **Parallel.ForEach**, and PLINQ constructs already discussed, the .NET Framework provides the **Parallel.Invoke** method, as well as the new **Task** and **Task<TResult>** types. The new **System.Threading.CountdownEvent** type also helps with fork/join patterns, in particular for when dealing with concurrent programming models that don't provide built-in support for joins.

COUNTING DOWN

A primary component of fork/join pattern implementations is keeping track of how much still remains to be completed. We saw this in our earlier **MyParallelFor** and **MyParallelForEach** implementations, with the loop storing a count for the number of work items that still remained to be completed, and a **ManualResetEvent** that would be signaled when this count reached 0. Support for this pattern is codified into the new **System.Threading.CountdownEvent** type in the .NET Framework 4. Below is a code snippet from earlier for implementing the sample **MyParallelFor**, now modified to use **CountdownEvent**.

C#

```
static void MyParallelFor(
    int fromInclusive, int toExclusive, Action<int> body)
{
    int numProcs = Environment.ProcessorCount;
    int nextIteration = fromInclusive;

    using (CountdownEvent ce = new CountdownEvent(numProcs))
    {
        for (int p = 0; p < numProcs; p++)
        {
            ThreadPool.QueueUserWorkItem(delegate
            {
                int index;
                while ((index = Interlocked.Increment(
                    ref nextIteration) - 1) < toExclusive)
                {
                    body(index);
                }
                ce.Signal();
            });
        }
        ce.Wait();
    }
}
```

Using **CountdownEvent** frees us from having to manage a count manually. Instead, the event is initialized with the expected number of signals, each thread signals the event when the thread completes its processing, and the main thread waits on the event for all signals to be received.

COUNTING UP AND DOWN

Counting down is often employed in parallel patterns, but so is incorporating some amount of counting up. If the remaining count represents the number of work items to be completed, and we end up adding more work items after setting the initial count, the count will need to be increased.

Here is an example of implementing a **MyParallelForEach** that launches one asynchronous work item per element to be processed. Since we don't know ahead of time how many elements there will be, we add a count of 1 for each element before launching it, and when the work item completes we signal the event.

C#
```
static void MyParallelForEach<T>(IEnumerable<T> source, Action<T> body)
{
    using (CountdownEvent ce = new CountdownEvent(1))
    {
        foreach (var item in source)
        {
            ce.AddCount(1);
            ThreadPool.QueueUserWorkItem(state =>
            {
                try { body((T)state); }
                finally { ce.Signal(); }
            }, item);
        }
        ce.Signal();
        ce.Wait();
    }
}
```

Note that the event is initialized with a count of 1. This is a common pattern in these scenarios, as we need to ensure that the event isn't set prior to all work items completing. If the count instead started at 0, and the first work item started and completed prior to our adding count for additional elements, the **CountdownEvent** would transition to a set state prematurely. By initializing the count to 1, we ensure that the event has no chance of reaching 0 until we remove that initial count, which is done in the above example by calling **Signal** after all elements have been queued.

PARALLEL.INVOKE

As shown previously, the **Parallel** class provides support for delightfully parallel loops through the **Parallel.For** and **Parallel.ForEach** methods. **Parallel** also provides support for patterns based on parallelized regions of code, where every statement in a region may be executed concurrently. This support, provided through the **Parallel.Invoke** method, enables a developer to easily specify multiple statements that should execute in parallel, and as with **Parallel.For** and **Parallel.ForEach**, **Parallel.Invoke** takes care of issues such as exception handling, synchronous invocation, scheduling, and the like:

C#

```
Parallel.Invoke(
    () => ComputeMean(),
    () => ComputeMedian(),
    () => ComputeMode());
```

Invoke itself follows patterns internally meant to help alleviate overhead. As an example, if you specify only a few delegates to be executed in parallel, **Invoke** will likely spin up one **Task** per element. However, if you specify many delegates, or if you specify **ParallelOptions** for how those delegates should be invoked, **Invoke** will likely instead choose to execute its work in a different manner. Looking at the signature for **Invoke**, we can see how this might happen:

C#

```
static void Invoke(params Action[] actions);
```

Invoke is supplied with an array of delegates, and it needs to perform an action for each one, potentially in parallel. That sounds like a pattern to which **ForEach** can be applied, doesn't it? In fact, we could implement a **MyParallelInvoke** using the **MyParallelForEach** we previously coded:

C#

```
static void MyParallelInvoke(params Action[] actions)
{
    MyParallelForEach(actions, action => action());
}
```

We could even use **MyParallelFor**:

C#

```
static void MyParallelInvoke(params Action[] actions)
{
    MyParallelFor(0, actions.Length, i => actions[i]());
}
```

This is very similar to the type of operation **Parallel.Invoke** will perform when provided with enough delegates. The overhead of a parallel loop is more than that of a few tasks, and thus when running only a few delegates, it makes sense for **Invoke** to simply use one task per element. But after a certain threshold, it's more efficient to use a parallel loop to execute all of the actions, as the cost of the loop is amortized across all of the delegate invocations.

ONE TASK PER ELEMENT

Parallel.Invoke represents a prototypical example of the fork/join pattern. Multiple operations are launched in parallel and then joined with such that only when they're all complete will the entire operation be considered complete. If we think of each individual delegate invocation from **Invoke** as being its own asynchronous operation, we can use a pattern of applying one task per element, where in this case the element is the delegate:

C#

```
static void MyParallelInvoke(params Action[] actions)
{
    var tasks = new Task[actions.Length];
    for (int i = 0; i < actions.Length; i++)
```

```
        {
            tasks[i] = Task.Factory.StartNew(actions[i]);
        }
        Task.WaitAll(tasks);
    }
```

This same pattern can be applied for variations, such as wanting to invoke in parallel a set of functions that return values, with the **MyParallelInvoke** method returning an array of all of the results. Here are several different ways that could be implemented, based on the patterns shown thus far (do note these implementations each have subtle differences in semantics, particularly with regards to what happens when an individual function fails with an exception):

C#

```csharp
// Approach #1: One Task per element
static T[]MyParallelInvoke<T>(params Func<T>[] functions)
{
    var tasks = (from function in functions
                 select Task.Factory.StartNew(function)).ToArray();
    Task.WaitAll(tasks);
    return tasks.Select(t => t.Result).ToArray();
}

// Approach #2: One Task per element, using parent/child Relationships
static T[] MyParallelInvoke<T>(params Func<T>[] functions)
{
    var results = new T[functions.Length];
    Task.Factory.StartNew(() =>
    {
        for (int i = 0; i < functions.Length; i++)
        {
            int cur = i;
            Task.Factory.StartNew(
                () => results[cur] = functions[cur](),
                TaskCreationOptions.AttachedToParent);
        }
    }).Wait();
    return results;
}

// Approach #3: Using Parallel.For
static T[] MyParallelInvoke<T>(params Func<T>[] functions)
{
    T[] results = new T[functions.Length];
    Parallel.For(0, functions.Length, i =>
    {
        results[i] = functions[i]();
    });
    return results;
}

// Approach #4: Using PLINQ
static T[] MyParallelInvoke<T>(params Func<T>[] functions)
{
    return functions.AsParallel().Select(f => f()).ToArray();
}
```

As with the Action-based **MyParallelInvoke**, for just a handful of delegates the first approach is likely the most efficient. Once the number of delegates increases to a plentiful amount, however, the latter approaches of using **Parallel.For** or PLINQ are likely more efficient. They also allow you to easily take advantage of additional functionality built into the **Parallel** and PLINQ APIs. For example, placing a limit on the degree of parallelism employed with tasks directly requires a fair amount of additional code. Doing the same with either **Parallel** or PLINQ requires only minimal additions. For example, if I want to use at most two threads to run the operations, I can do the following:

C#
```
static T[] MyParallelInvoke<T>(params Func<T>[] functions)
{
    T[] results = new T[functions.Length];
    var options = new ParallelOptions { MaxDegreeOfParallelism = 2 };
    Parallel.For(0, functions.Length, options, i =>
    {
        results[i] = functions[i]();
    });
    return results;
}
```

For fork/join operations, the pattern of creating one task per element may be particularly useful in the following situations:

1) **Additional work may be started only when specific subsets of the original elements have completed processing.** As an example, in the Strassen's matrix multiplication algorithm, two matrices are multiplied by splitting each of the matrices into four quadrants. Seven intermediary matrices are generated based on operations on the eight input submatrices. Four output submatrices that make up the larger output matrix are computed from the intermediary seven. These four output matrices each only require a subset of the previous seven, so while it's correct to wait for all of the seven prior to computing the following four, some potential for parallelization is lost as a result.
2) **All elements should be given the chance to run even if one invocation fails.** With solutions based on **Parallel** and PLINQ, the looping and query constructs will attempt to stop executing as soon as an exception is encountered; this can be solved using manual exception handling with the loop, as demonstrated earlier, however by using **Task**s, each operation is treated independently, and such custom code isn't needed.

RECURSIVE DECOMPOSITION

One of the more common fork/join patterns deals with forks that themselves fork and join. This recursive nature is known as recursive decomposition, and it applies to parallelism just as it applies to serial recursive implementations.

Consider a **Tree<T>** binary tree data structure:

C#
```
class Tree<T>
{
    public T Data;
    public Tree<T> Left, Right;
}
```

A tree walk function that executes an action for each node in the tree might look like the following:

C#
```
static void Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return;
    action(root.Data);
    Walk(root.Left, action);
    Walk(root.Right, action);
}
```

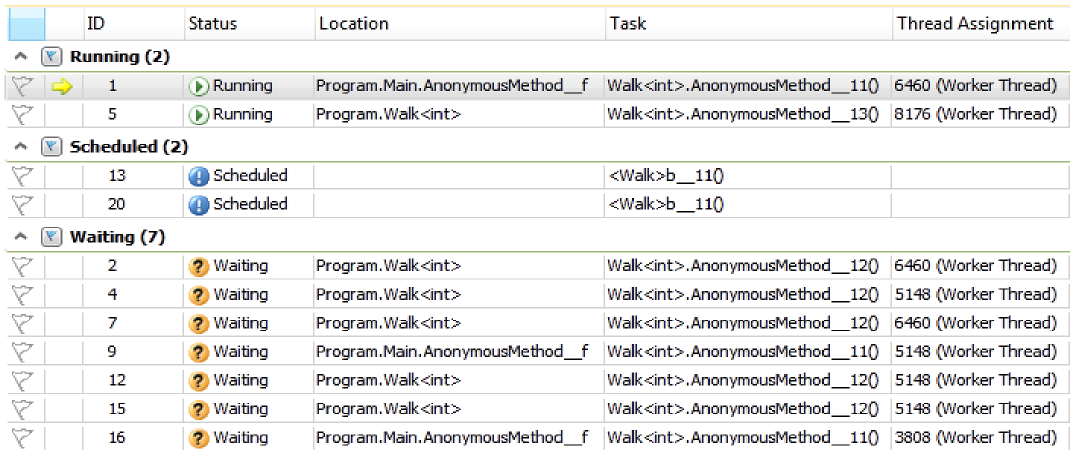
Parallelizing this may be accomplished by fork/join'ing on at least the two recursive calls, if not also on the action invocation:

C#
```
static void Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return;
    Parallel.Invoke(
        () => action(root.Data),
        () => Walk(root.Left, action),
        () => Walk(root.Right, action));
}
```

The recursive calls to **Walk** themselves fork/join as well, leading to a logical tree of parallel invocations. This can of course also be done using **Task** objects directly:

C#
```
static void Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return;
    var t1 = Task.Factory.StartNew(() => action(root.Data));
    var t2 = Task.Factory.StartNew(() => Walk(root.Left, action));
    var t3 = Task.Factory.StartNew(() => Walk(root.Right, action));
    Task.WaitAll(t1, t2, t3);
}
```

We can see all of these **Tasks** in Visual Studio using the Parallel Tasks debugger window, as shown in the following screenshot:



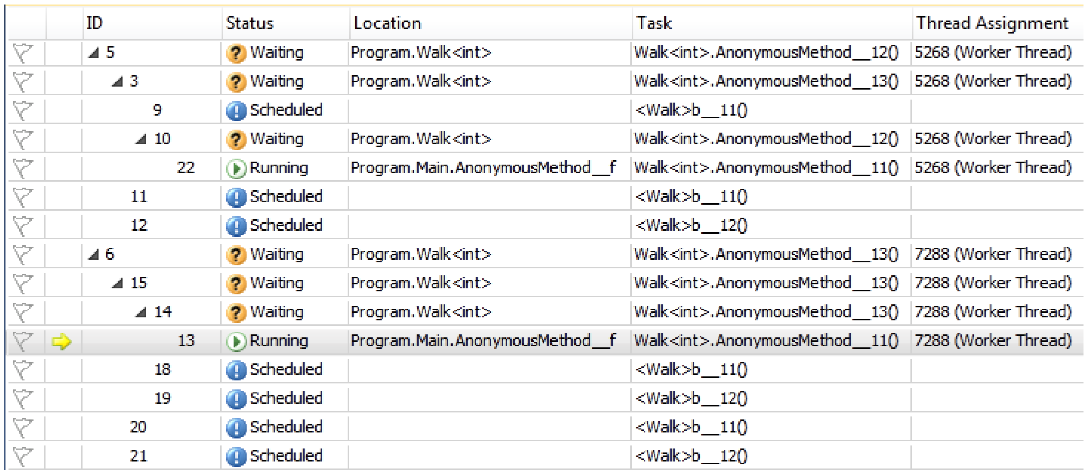
ID	Status	Location	Task	Thread Assignment
Running (2)				
1	Running	Program.Main.AnonymousMethod__f	Walk<int>.AnonymousMethod__11()	6460 (Worker Thread)
5	Running	Program.Walk<int>	Walk<int>.AnonymousMethod__13()	8176 (Worker Thread)
Scheduled (2)				
13	Scheduled		<Walk>b__11()	
20	Scheduled		<Walk>b__11()	
Waiting (7)				
2	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	6460 (Worker Thread)
4	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	5148 (Worker Thread)
7	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	6460 (Worker Thread)
9	Waiting	Program.Main.AnonymousMethod__f	Walk<int>.AnonymousMethod__11()	5148 (Worker Thread)
12	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	5148 (Worker Thread)
15	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	5148 (Worker Thread)
16	Waiting	Program.Main.AnonymousMethod__f	Walk<int>.AnonymousMethod__11()	3808 (Worker Thread)

We can further take advantage of parent/child relationships in order to see the associations between these **Task**s in the debugger. First, we can modify our code by forcing all tasks to be attached to a parent, which will be the **Task** currently executing when the child is created. This is done with the **TaskCreationOptions.AttachedToParent** option:

C#

```csharp
static void Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return;
    var t1 = Task.Factory.StartNew(() => action(root.Data),
        TaskCreationOptions.AttachedToParent);
    var t2 = Task.Factory.StartNew(() => Walk(root.Left, action),
        TaskCreationOptions.AttachedToParent);
    var t3 = Task.Factory.StartNew(() => Walk(root.Right, action),
        TaskCreationOptions.AttachedToParent);
    Task.WaitAll(t1, t2, t3);
}
```

Re-running the application, we can now see the following parent/child hierarchy in the debugger:



		ID	Status	Location	Task	Thread Assignment
		◢ 5	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	5268 (Worker Thread)
		◢ 3	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__13()	5268 (Worker Thread)
		9	Scheduled		<Walk>b__11()	
		◢ 10	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__12()	5268 (Worker Thread)
		22	Running	Program.Main.AnonymousMethod__f	Walk<int>.AnonymousMethod__11()	5268 (Worker Thread)
		11	Scheduled		<Walk>b__11()	
		12	Scheduled		<Walk>b__12()	
		◢ 6	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__13()	7288 (Worker Thread)
		◢ 15	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__13()	7288 (Worker Thread)
		◢ 14	Waiting	Program.Walk<int>	Walk<int>.AnonymousMethod__13()	7288 (Worker Thread)
	⇨	13	Running	Program.Main.AnonymousMethod__f	Walk<int>.AnonymousMethod__11()	7288 (Worker Thread)
		18	Scheduled		<Walk>b__11()	
		19	Scheduled		<Walk>b__12()	
		20	Scheduled		<Walk>b__11()	
		21	Scheduled		<Walk>b__12()	

CONTINUATION CHAINING

The previous example of walking a tree utilizes blocking semantics, meaning that a particular level won't complete until its children have completed. **Parallel.Invoke**, and the Task Wait functionality on which it's based, attempt what's known as inlining, where rather than simply blocking waiting for another thread to execute a **Task**, the waiter may be able to run the waitee on the current thread, thereby improving resource reuse, and improving performance as a result. Still, there may be some cases where tasks are not inlinable, or where the style of development is better suited towards a more asynchronous model. In such cases, task completions can be chained.

As an example of this, we'll revisit the **Walk** method. Rather than returning void, the **Walk** method can return a **Task**. That **Task** can represent the completion of all child tasks. There are two primary ways to accomplish this. One way is to take advantage of **Task** parent/child relationships briefly mentioned previously. With parent/child relationships, a parent task won't be considered completed until all of its children have completed.

C#

```
static Task Walk<T>(Tree<T> root, Action<T> action)
{
    return Task.Factory.StartNew(() =>
    {
        if (root == null) return;
        Walk(root.Left, action);
        Walk(root.Right, action);
        action(root.Data);
    }, TaskCreationOptions.AttachedToParent);
}
```

Every call to **Walk** creates a new **Task** that's attached to its parent and immediately returns that **Task**. That **Task**, when executed, recursively calls Walk (thus creating **Task**s for the children) and executes the relevant action. At the root level, the initial call to **Walk** will return a **Task** that represents the entire tree of processing and that won't complete until the entire tree has completed.

Another approach is to take advantage of continuations:

C#

```
static Task Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return _completedTask;
    Task t1 = Task.Factory.StartNew(() => action(root.Data));
    Task<Task> t2 = Task.Factory.StartNew(() => Walk(root.Left, action));
    Task<Task> t3 = Task.Factory.StartNew(() => Walk(root.Right, action));
    return Task.Factory.ContinueWhenAll(
        new Task[] { t1, t2.Unwrap(), t3.Unwrap() },
        tasks => Task.WaitAll(tasks));
}
```

As we've previously seen, this code uses a task to represent each of the three operations to be performed at each node: invoking the action for the node, walking the left side of the tree, and walking the right side of the tree. However, we now have a predicament, in that the **Task** returned for walking each side of the tree is actually a **Task<Task>** rather than simply a **Task**. This means that the result will be signaled as completed when the **Walk** call has returned, but not necessarily when the Task it returned has completed. To handle this, we can take advantage of the Unwrap method, which converts a **Task<Task>** into a **Task**, by "unwrapping" the internal Task into a top-level **Task** that represents it (another overload of **Unwrap** handles unwrapping a **Task<Task<TResult>>** into a **Task<TResult>**). Now with our three tasks, we can employ the **ContinueWhenAll** method to create and return a **Task** that represents the total completion of this node and all of its descendants. In order to ensure exceptions are propagated correctly, the body of that continuation explicitly waits on all of the tasks; it knows they're completed by this point, so this is simply to utilize the exception propagation logic in **WaitAll**.

The parent-based approach has several advantages, including that the Visual Studio 2010 Parallel Tasks toolwindow can highlight the parent/child relationship involved, showing the task hierarchy graphically during a debugging session, and exception handling is simplified, as all exceptions will bubble up to the root parent. However, the continuation approach may have a memory benefit for deep hierarchies or long-chains of tasks, since with the parent/child relationships, running children prevent the parent nodes from being garbage collected.

To simplify this, you can consider codifying this into an extension method for easier implementation:

C#
```
static Task ContinueWhenAll(
    this TaskFactory factory, params Task[] tasks)
{
    return factory.ContinueWhenAll(
        tasks, completed => Task.WaitAll(completed));
}
```

With that extension method in place, the previous snippet may be rewritten as:

C#
```
static Task Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return _completedTask;
    var t1 = Task.Factory.StartNew(() => action(root.Data));
    var t2 = Task.Factory.StartNew(() => Walk(root.Left, action));
    var t3 = Task.Factory.StartNew(() => Walk(root.Right, action));
    return Task.Factory.ContinueWhenAll(t1, t2.Unwrap(), t3.Unwrap());
}
```

One additional thing to notice is the **_completedTask** returned if the root node is null. Both **WaitAll** and **ContinueWhenAll** will throw an exception if the array of tasks passed to them contains a null element. There are several ways to work around this, one of which is to ensure that a null element is never provided. To do that, we can return a valid **Task** from **Walk** even if there is no node to be processed. Such a **Task** should be already completed so that little additional overhead is incurred. To accomplish this, we can create a single **Task** using a **TaskCompletionSource<TResult>**, resolve the **Task** into a completed state, and cache it for all code that needs a completed **Task** to use:

C#
```
private static Task _completedTask = ((Func<Task>)(() => {
    var tcs = new TaskCompletionSource<object>();
    tcs.SetResult(null);
    return tcs.Task;
}))();
```

ANTI-PATTERNS

FALSE SHARING

Data access patterns are important for serial applications, and they're even more important for parallel applications. One serious performance issue that can arise in parallel applications occurs where unexpected sharing happens at the hardware level.

For performance reasons, memory systems use groups called cache lines, typically of 64 bytes or 128 bytes. A cache line, rather than an individual byte, is moved around the system as a unit, a classic example of chunky instead of chatty communication. If multiple cores attempt to access two different bytes on the same cache line, there's no correctness sharing conflict, but only one will be able to have exclusive access to the cache line at the hardware level, thus introducing the equivalent of a lock at the hardware level that wasn't otherwise present in the code. This can lead to unforeseen and serious performance problems.

As an example, consider the following method, which uses a **Parallel.Invoke** to initialize two arrays to random values:

C#
```
void WithFalseSharing()
{
    Random rand1 = new Random(), rand2 = new Random();
    int[] results1 = new int[20000000], results2 = new int[20000000];
    Parallel.Invoke(
        () => {
            for (int i = 0; i < results1.Length; i++)
                results1[i] = rand1.Next();
        },
        () => {
            for (int i = 0; i < results2.Length; i++)
                results2[i] = rand2.Next();
        });
}
```

The code initializes two distinct **System.Random** instances and two distinct arrays, such that each thread involved in the parallelization touches its own non-shared state. However, due to the way these two **Random** instances were allocated, they're likely on the same cache line in memory. Since every call to **Next** modifies the **Random** instance's internal state, multiple threads will now be contending for the same cache line, leading to seriously impacted performance. Here's a version that addresses the issue:

C#
```
void WithoutFalseSharing()
{
    int[] results1, results2;
    Parallel.Invoke(
        () => {
            Random rand1 = new Random();
            results1 = new int[20000000];
            for (int i = 0; i < results1.Length; i++)
                results1[i] = rand1.Next();
        },
        () => {
            Random rand2 = new Random();
            results2 = new int[20000000];
            for (int i = 0; i < results2.Length; i++)
                results2[i] = rand2.Next();
        });
}
```

On my dual-core system, when comparing the performance of these two methods, the version with false sharing typically ends up running slower than the serial equivalent, whereas the version without false sharing typically ends up running almost twice as fast as the serial equivalent.

False sharing is a likely source for investigation if you find that parallelized code operating with minimal synchronization isn't obtaining the parallelized performance improvements you expected. More information is available in the MSDN Magazine article .NET Matters: False Sharing.

RECURSION WITHOUT THRESHOLDS

In a typical introductory algorithms course, computer science students learn about various algorithms for sorting, often culminating in quicksort. Quicksort is a recursive divide-and-conquer algorithm, where the input array to be sorted is partitioned into two contiguous chunks, one with values less than a chosen pivot and one with values greater than or equal to a chosen pivot. Once the array has been partitioned, the quicksort routine may be used recursively to sort each chunk. The recursion ends when the size of a chunk is one element, since one element is implicitly sorted.

Students learn that quicksort has an average algorithmic complexity of O(N log N), which for large values of N is much faster than other algorithms like insertion sort which have a complexity of O(N^2). They also learn, however, that big-O notation focuses on the limiting behavior of functions and ignores constants, because as the value of N grows, the constants aren't relevant. Yet when N is small, those constants can actually make a difference.

It turns out that constants involved in quicksort are larger than those involved in insertion sort, and as such, for small values of N, insertion sort is often faster than quicksort. Due to quicksort's recursive nature, even if the operation starts out operating on a large N, at some point in the recursion the value of N for that particular call is small enough that it's actually better to use insertion sort. Thus, many quality implementations of quicksort won't stop the recursion when a chunk size is one, but rather will choose a higher value, and when that threshold is reached, the algorithm will switch over to a call to insertion sort to sort the chunk, rather than continuing with the recursive quicksort routine.

As has been shown previously, quicksort is a great example for recursive decomposition with task-based parallelism, as it's easy to recursively sort the left and right partitioned chunks in parallel, as shown in the following example:

C#
```
static void QuickSort<T>(T[] data, int fromInclusive, int toExclusive)
    where T : IComparable<T>
{
    if (toExclusive - fromInclusive <= THRESHOLD)
        InsertionSort(data, fromInclusive, toExclusive);
    else
    {
        int pivotPos = Partition(data, fromInclusive, toExclusive);
        Parallel.Invoke(
            () => QuickSort(data, fromInclusive, pivotPos),
            () => QuickSort(data, pivotPos, toExclusive));
    }
}
```

You'll note, however, that in addition to the costs associated with the quicksort algorithm itself, we now have additional overheads involved with creating tasks for each half of the sort. If the computation is completely balanced, at some depth into the recursion we will have saturated all processors. For example, on a dual-core machine, the first level of recursion will create two tasks, and thus theoretically from that point forward we're saturating the machine and there's no need to continue to bear the overhead of additional tasks. This implies that we now may benefit from a second threshold: in addition to switching from quicksort to insertion sort at some threshold, we now also want to switch from parallel to serial at some threshold. That threshold may be defined in a variety of ways.

As with the insertion sort threshold, a simple parallel threshold could be based on the amount of data left to be processed:

C#

```
static void QuickSort<T>(T[] data, int fromInclusive, int toExclusive)
    where T : IComparable<T>
{
    if (toExclusive - fromInclusive <= THRESHOLD)
        InsertionSort(data, fromInclusive, toExclusive);
    else
    {
        int pivotPos = Partition(data, fromInclusive, toExclusive);
        if (toExclusive - fromInclusive <= PARALLEL_THRESHOLD)
        {
            // NOTE: PARALLEL_THRESHOLD is chosen to be greater than THRESHOLD.
            QuickSort(data, fromInclusive, pivotPos);
            QuickSort(data, pivotPos, toExclusive);
        }
        else Parallel.Invoke(
            () => QuickSort(data, fromInclusive, pivotPos),
            () => QuickSort(data, pivotPos, toExclusive));
    }
}
```

Another simple threshold may be based on depth. We can initialize the depth to the max depth we want to recur to in parallel, and decrement the depth each time we recur… when it reaches 0, we fall back to serial.

C#

```
static void QuickSort<T>(T[] data, int fromInclusive, int toExclusive, int depth)
    where T : IComparable<T>
{
    if (toExclusive - fromInclusive <= THRESHOLD)
        InsertionSort(data, fromInclusive, toExclusive);
    else
    {
        int pivotPos = Partition(data, fromInclusive, toExclusive);
        if (depth > 0)
        {
            Parallel.Invoke(
                () => QuickSort(data, fromInclusive, pivotPos, depth-1),
                () => QuickSort(data, pivotPos, toExclusive, depth-1));
        }
        else
        {
            QuickSort(data, fromInclusive, pivotPos, 0);
            QuickSort(data, pivotPos, toExclusive, 0);
        }
    }
}
```

If you assume that the parallelism will be completely balanced due to equal work resulting from all partition operations, you might then base the initial depth on the number of cores in the machine:

C#

```
QuickSort(data, 0, data.Length, Math.Log(Environment.ProcessorCount, 2));
```

Alternatively, you might provide a bit of extra breathing room in case the problem space isn't perfectly balanced:

C#

```
QuickSort(data, 0, data.Length, Math.Log(Environment.ProcessorCount, 2) + 1);
```

Of course, the partitioning may result in very unbalanced workloads. And quicksort is just one example of an algorithm; many other algorithms that are recursive in this manner will frequently result in very unbalanced workloads.

Another approach is to keep track of the number of outstanding work items, and only "go parallel" when the number of outstanding items is below a threshold. An example of this follows:

C#

```
class Utilities
{
    static int CONC_LIMIT = Environment.ProcessorCount * 2;
    volatile int _invokeCalls = 0;

    public void QuickSort<T>(T[] data, int fromInclusive, int toExclusive)
        where T : IComparable<T>
    {
        if (toExclusive - fromInclusive <= THRESHOLD)
            InsertionSort(data, fromInclusive, toExclusive);
        else
        {
            int pivotPos = Partition(data, fromInclusive, toExclusive);
            if (_invokeCalls < CONC_LIMIT)
            {
                Interlocked.Increment(ref _invokeCalls);
                Parallel.Invoke(
                    () => QuickSort(data, fromInclusive, pivotPos),
                    () => QuickSort(data, pivotPos, toExclusive));
                Interlocked.Decrement(ref _invokeCalls);
            }
            else
            {
                QuickSort(data, fromInclusive, pivotPos);
                QuickSort(data, pivotPos, toExclusive);
            }
        }
    }
}
```

Here, we're keeping track of the number of **Parallel.Invoke** calls active at any one time. When the number is below a predetermined limit, we recur using **Parallel.Invoke**; otherwise, we recur serially. This adds the additional expense of two interlocked operations per recursive call (and is only an approximation, as the **_invokeCalls** field is compared to the threshold outside of any synchronization), forcing synchronization where it otherwise wasn't needed, but it also allows for more load-balancing. Previously, once a recursive path was serial, it would remain serial. With this modification, a serial path through **QuickSort** may recur and result in a parallel path.

PASSING DATA

There are several common patterns in the .NET Framework for passing data to asynchronous work.

CLOSURES

Since support for them was added to C# and Visual Basic, closures represent the easiest way to pass data into background operations. By creating delegates that refer to state outside of their scope, the compiler transforms the accessed variables in a way that makes them accessible to the delegates, “closing over” those variables. This makes it easy to pass varying amounts of data into background work:

C#

```csharp
int data1 = 42;
string data2 = "The Answer to the Ultimate Question of " +
               "Life, the Universe, and Everything";
Task.Factory.StartNew(()=>
{
    Console.WriteLine(data2 + ": " + data1);
});
```

For applications in need of the utmost in performance and scalability, it’s important to keep in mind that under the covers the compiler may actually be allocating an object in which to store the variables (in the above example, data1 and data2) that are accessed by the delegate.

STATE OBJECTS

Dating back to the beginning of the .NET Framework, many APIs that spawn asynchronous work accept a state parameter and pass that state object into the delegate that represents the body of work. The **ThreadPool.QueueUserWorkItem** method is a quintessential example of this:

C#

```csharp
public static bool QueueUserWorkItem(WaitCallback callBack, object state);
...
public delegate void WaitCallback(object state);
```

We can take advantage of this state parameter to pass a single object of data into the **WaitCallback**:

C#

```csharp
ThreadPool.QueueUserWorkItem(state => {
    Console.WriteLine((string)state);
}, data2);
```

The **Task** class in the .NET Framework 4 also supports this pattern:

C#

```csharp
Task.Factory.StartNew(state => {
    Console.WriteLine((string)state);
}, data2);
```

Note that in contrast to the closures approach, this typically does not cause an extra object allocation to handle the state, unless the state being supplied is a value type (value types must be boxed to supply them as the object state parameter).

To pass in multiple pieces of data with this approach, those pieces of data must be wrapped into a single object. In the past, this was typically a custom class to store specific pieces of information. With the .NET Framework 4, the new **Tuple<>** classes may be used instead:

C#

```
Tuple<int,string> data = Tuple.Create(data1, data2);
Task.Factory.StartNew(state => {
    Tuple<int,string> d = (Tuple<int,string>)data;
    Console.WriteLine(d.Item2 + ": " + d.Item1);
}, data);
```

As with both closures and working with value types, this requires an object allocation to support the creation of the tuple to wrap the data items. The built-in tuple types in the .NET Framework 4 also support a limited number of contained pieces of data.

STATE OBJECTS WITH MEMBER METHODS

Another approach, similar to the former, is to pass data into asynchronous operations by representing the work to be done asynchronously as an instance method on a class. This allows data to be passed in to that method implicitly through the “this” reference.

C#

```
class Work
{
    public int Data1;
    public string Data2;
    public void Run()
    {
        Console.WriteLine(Data1 + ": " + Data2);
    }
}
// ...
Work w = new Work();
w.Data1 = 42;
w.Data2 = "The Answer to the Ultimate Question of " +
          "Life, the Universe, and Everything";
Task.Factory.StartNew(w.Run);
```

As with the previous approaches, this approach requires an object allocation for an object (in this case, of class **Work**) to store the state. Such an allocation is still required if **Work** is a struct instead of a class; this is because the creation of a delegate referring to **Work** must reference the object on which to invoke the instance method **Run**, and that reference is stored as an object, thus boxing the struct.

As such, which of these approaches you choose is largely a matter of preference. The closures approach typically leads to the most readable code, and it allows the compiler to optimize the creation of the state objects. For example, if the anonymous delegate passed to **StartNew** doesn’t access any local state, the compiler may be able to avoid the object allocation to store the state, as it will already be stored as accessible instance or static fields.

ANTI-PATTERNS

CLOSING OVER INAPPROPRIATELY SHARED DATA

Consider the following code, and hazard a guess for what it outputs:

C#

```
static void Main()
{
    for (int i = 0; i < 10; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate { Console.WriteLine(i); });
    }
}
```

If you guessed that this outputs the numbers 0 through 9 inclusive, you'd likely be wrong. While that might be the output, more than likely this will actually output ten "10"s. The reason for this has to do with the language's rules for scoping and how it captures variables into anonymous methods, which here were used to represent the work provided to **QueueUserWorkItem**. The variable **i** is shared by both the main thread queuing the work items and the **ThreadPool** threads printing out the value of **i**. The main thread is continually updating the value of **i** as it iterates from 0 through 9, and thus each output line will contain the value of **i** at whatever moment the **Console.WriteLine** call occurs on the background thread. (Note that unlike the C# compiler, the Visual Basic compiler kindly warns about this issue: "warning BC42324: Using the iteration variable in a lambda expression may have unexpected results. Instead, create a local variable within the loop and assign it the value of the iteration variable.")

This phenomenon isn't limited to parallel programming, though the prominence of anonymous methods and lambda expressions in the the .NET Framework parallel programming model does exacerbate the issue. For a serial example, consider the following code:

C#

```
static void Main()
{
    var actions = new List<Action>();
    for (int i = 0; i < 10; i++)
        actions.Add(() => Console.WriteLine(i));
    actions.ForEach(action => action());
}
```

This code will reliably output ten "10"s, as by the time the **Action** delegates are invoked, the value of **i** is already 10, and all of the delegates are referring to the same captured **i** variable.

To address this issue, we can create a local copy of the iteration variable in scope inside the loop (as was recommended by the Visual Basic compiler). This will cause each anonymous method to gain its own variable, rather than sharing them with other delegates. The sequential code shown earlier can be fixed with a small alteration:

C#

```
static void Main()
{
    var actions = new List<Action>();
    for (int i = 0; i < 10; i++)
    {
        int tmp = i;
        actions.Add(() => Console.WriteLine(tmp));
    }
    actions.ForEach(action => action());
}
```

This will reliably print out the sequence “0” through “9” as expected. The parallel code can be fixed in a similar manner:

C#

```
static void Main()
{
    for (int i = 0; i < 10; i++)
    {
        int tmp = i;
        ThreadPool.QueueUserWorkItem(delegate { Console.WriteLine(tmp); });
    }
}
```

This will also reliably print out the values “0” through “9”, although the order in which they’re printed is not guaranteed.

Another similar case where closure semantics can lead you astray is if you’re in the habit of declaring your variables at the top of your function, and then using them later on. For example:

C#

```
static void Main(string[] args)
{
    int j;
    Parallel.For(0, 10000, i =>
    {
        int total = 0;
        for (j = 1; j <= 10000; j++) total += j;
    });
}
```

Due to closure semantics, the **j** variable will be shared by all iterations of the parallel loop, thus wreaking havoc on the inner serial loop. To address this, the variable declarations should be moved as close to their usage as possible:

C#

```
static void Main(string[] args)
{
    Parallel.For(0, 10000, i =>
    {
        int total = 0;
        for (int j = 1; j <= 10000; j++) total += j;
    });
}
```

PRODUCER/CONSUMER

The real world revolves around the "producer/consumer" pattern. Individual entities are responsible for certain functions, where some entities generate material that ends up being consumed by others. In some cases, those consumers are also producers for even further consumers. Sometimes there are multiple producers per consumer, sometimes there are multiple consumers per producer, and sometimes there's a many-to-many relationship. We live and breathe producer/consumer, and the pattern similarly has a very high value in parallel computing.

Often, producer/consumer relationships are applied to parallelization when there's no ability to parallelize an individual operation, but when multiple operations may be carried out concurrently, with one having a dependency on the other. For example, consider the need to both compress and encrypt a particular file. This can be done sequentially, with a single thread reading in a chunk of data, compressing it, encrypting the compressed data, writing out the encrypted data, and then repeating the process for more chunks until the input file has been completely processed. Depending on the compression and encryption algorithms utilized, there may not be the ability to parallelize an individual compression or encryption, and the same data certainly can't be compressed concurrently with it being encrypted, as the encryption algorithm must run over the compressed data rather than over the uncompressed input. Instead, multiple threads may be employed to form a pipeline. One thread can read in the data. That thread can hand the read data off to another thread that compresses it, and in turn hands the compressed data off to a third thread. The third thread can then encrypt it, and pass it off to a fourth thread, which writes the encrypted data to the output file. Each processing "agent", or "actor", in this scheme is serial in nature, churning its input into output, and as long as the hand-offs between agents don't introduce any reordering operations, the output data from the entire process will emerge in the same order the associated data was input.

Those hand-offs can be managed with the new **BlockingCollection<>** type, which provides key support for this pattern in the .NET Framework 4.

PIPELINES

Hand-offs between threads in a parallelized system require shared state: the producer needs to put the output data somewhere, and the consumer needs to know where to look to get its input data. More than just having access to a storage location, however, there is additional communication that's necessary. A consumer is often prevented from making forward progress until there's some data to be consumed. Additionally, in some systems, a producer needs to be throttled so as to avoid producing data much faster than consumers can consume it. In both of these cases, a notification mechanism must also be incorporated. Additionally, with multiple producers and multiple consumers, participants must not trample on each other as they access the storage location.

We can build a simple version of such a hand-off mechanism using a **Queue<T>** and a **SemaphoreSlim**:

C#

```
class BlockingQueue<T>
{
    private Queue<T> _queue = new Queue<T>();
    private SemaphoreSlim _semaphore = new SemaphoreSlim(0, int.MaxValue);

    public void Enqueue(T data)
    {
        if (data == null) throw new ArgumentNullException("data");
        lock (_queue) _queue.Enqueue(data);
        _semaphore.Release();
    }

    public T Dequeue()
    {
        _semaphore.Wait();
        lock (_queue) return _queue.Dequeue();
    }
}
```

Here we have a very simple "blocking queue" data structure. Producers call **Enqueue** to add data into the queue, which adds the data to an internal **Queue<T>** and notifies consumers using a semaphore that another element of data is available. Similarly, consumers use **Dequeue** to wait for an element of data to be available and then remove that data from the underlying **Queue<T>**. Note that because multiple threads could be accessing the data structure concurrently, a lock is used to protect the non-thread-safe **Queue<T>** instance.

Another similar implementation makes use of **Monitor**'s notification capabilities instead of using a semaphore:

C#

```
class BlockingQueue<T>
{
    private Queue<T> _queue = new Queue<T>();

    public void Enqueue(T data)
    {
        if (data == null) throw new ArgumentNullException("data");
        lock (_queue)
        {
            _queue.Enqueue(data);
            Monitor.Pulse(_queue);
        }
    }

    public T Dequeue()
    {
        lock (_queue)
        {
            while (_queue.Count == 0) Monitor.Wait(_queue);
            return _queue.Dequeue();
        }
    }
}
```

Such implementations provide basic support for data hand-offs between threads, but they also lack several important things. How do producers communicate that there will be no more elements produced? With this blocking behavior, what if a consumer only wants to block for a limited amount of time before doing something else? What if producers need to be throttled, such that if the underlying **Queue<T>** is full they're blocked from adding to it? What if you want to pull from one of several blocking queues rather than from a single one? What if semantics others than first-in-first-out (FIFO) are required of the underlying storage? What if producers and consumers need to be canceled? And so forth.

All of these questions have answers in the new .NET Framework 4 **System.Collections.Concurrent.BlockingCollection<T>** type in **System.dll**. It provides the same basic behavior as shown in the naïve implementation above, sporting methods to add to and take from the collection. But it also supports throttling both consumers and producers, timeouts on waits, support for arbitrary underlying data structures, and more. It also provides built-in implementations of typical coding patterns related to producer/consumer in order to make such patterns simple to utilize.

As an example of a standard producer/consumer pattern, consider the need to read in a file, transform each line using a regular expression, and write out the transformed line to a new file. We can implement that using a **Task** to run each step of the pipeline asynchronously, and **BlockingCollection<string>** as the hand-off point between each stage.

C#

```
static void ProcessFile(string inputPath, string outputPath)
{
    var inputLines = new BlockingCollection<string>();
    var processedLines = new BlockingCollection<string>();

    // Stage #1
    var readLines = Task.Factory.StartNew(() =>
    {
        try
        {
            foreach (var line in File.ReadLines(inputPath)) inputLines.Add(line);
        }
        finally { inputLines.CompleteAdding(); }
    });

    // Stage #2
    var processLines = Task.Factory.StartNew(() =>
    {
        try
        {
            foreach(var line in inputLines.GetConsumingEnumerable()
                .Select(line => Regex.Replace(line, @"\s+", ", ")))
            {
                processedLines.Add(line);
            }
        }
        finally { processedLines.CompleteAdding(); }
    });

    // Stage #3
    var writeLines = Task.Factory.StartNew(() =>
    {
        File.WriteAllLines(outputPath, processedLines.GetConsumingEnumerable());
    });

    Task.WaitAll(readLines, processLines, writeLines);
}
```

With this basic structure coded up, we have a lot of flexibility and room for modification. For example, what if we discover from performance testing that we're reading from the input file much faster than the processing and outputting can handle it? One option is to limit the speed at which the input file is read, which can be done by modifying how the **inputLines** collection is created:

C#

```
var inputLines = new BlockingCollection<string>(boundedCapacity:20);
```

By adding the **boundedCapacity** parameter (shown here for clarity using named parameter functionality, which is now supported by both C# and Visual Basic in Visual Studio 2010), a producer attempting to add to the collection will block until there are less than 20 elements in the collection, thus slowing down the file reader. Alternatively, we could further parallelize the solution. For example, let's assume that through testing you found the real problem to be that the **processLines Task** was heavily compute bound. To address that, you could parallelize it using PLINQ in order to utilize more cores:

C#

```
foreach(var line in inputLines.GetConsumingEnumerable()
    .AsParallel().AsOrdered()
    .Select(line => Regex.Replace(line, @"\s+", ", ")))
```

Note that by specifying ".AsOrdered()" after the ".AsParallel()", we're ensuring that PLINQ maintains the same ordering as in the sequential solution.

DECORATOR TO PIPELINE

The decorator pattern is one of the original Gang Of Four design patterns. A decorator is an object that has the same interface as another object it contains. In object-oriented terms, it is an object that has an "is-a" and a "has-a" relationship with a specific type. Consider the **CryptoStream** class in the **System.Security.Cryptography** namespace. **CryptoStream** derives from **Stream** (it "is-a" Stream), but it also accepts a Stream to its constructor and stores that **Stream** internally (it "has-a" stream); that underlying stream is where the encrypted data is stored. **CryptoStream** is a decorator.

With decorators, we typically chain them together. For example, as alluded to in the introduction to this section on producer/consumer, a common need in software is to both compress and encrypt data. The .NET Framework contains two decorator stream types to make this feasible: the **CryptoStream** class already mentioned, and the **GZipStream** class. We can compress and encrypt an input file into an output file with code like the following:

C#

```
static void CompressAndEncrypt(string inputFile, string outputFile)
{
    using (var input = File.OpenRead(inputFile))
    using (var output = File.OpenWrite(outputFile))
    using (var rijndael = new RijndaelManaged())
    using (var transform = rijndael.CreateEncryptor())
    using (var encryptor =
             new CryptoStream(output, transform, CryptoStreamMode.Write))
    using (var compressor =
             new GZipStream(encryptor, CompressionMode.Compress, true))
        input.CopyTo(compressor);
}
```

The input file stream is copied to a **GZipStream**, which wraps a **CryptoStream**, which wraps the output stream. The data flows from one stream to the other, with its data modified along the way.

Both compression and encryption are computationally intense operations, and as such it can be beneficial to parallelize this operation. However, given the nature of the problem, it's not just as simple as running both the compression and encryption in parallel on the input stream, since the encryption operates on the output of the compression. Instead, we can form a pipeline, with the output of the compression being fed as the input to the encryption, such that while the encryption is processing data block N, the compression routine can have already moved on to be processing N+1 or greater. To make this simple, we'll implement it with another decorator, a **TransferStream**. The idea behind this stream is that writes are offloaded to another thread, which sequentially writes to the underlying stream all of the writes to the transfer stream. That way, when code calls **Write** on the transfer stream, it's not blocked waiting for the whole chain of decorators to complete their processing: **Write** returns immediately after queuing the work, and the caller can go on to do additional work. A simple implementation of **TransferStream** is shown below (relying on a custom **Stream** base type, which simply implements the abstract **Stream** class with default implementations of all abstract members, in order to keep the code shown here small), taking advantage of both **Task** and **BlockingCollection**:

C#

```
public sealed class TransferStream : AbstractStreamBase
{
    private Stream _writeableStream;
    private BlockingCollection<byte[]> _chunks;
    private Task _processingTask;

    public TransferStream(Stream writeableStream)
    {
        // ... Would validate arguments here
        _writeableStream = writeableStream;
        _chunks = new BlockingCollection<byte[]>();
        _processingTask = Task.Factory.StartNew(() =>
        {
            foreach (var chunk in _chunks.GetConsumingEnumerable())
                _writeableStream.Write(chunk, 0, chunk.Length);
        }, TaskCreationOptions.LongRunning);
    }

    public override bool CanWrite { get { return true; } }

    public override void Write(byte[] buffer, int offset, int count)
    {
        // ... Would validate arguments here
        var chunk = new byte[count];
        Buffer.BlockCopy(buffer, offset, chunk, 0, count);
        _chunks.Add(chunk);
    }

    public override void Close()
    {
        _chunks.CompleteAdding();
        try { _processingTask.Wait(); }
        finally { base.Close(); }
    }
}
```

The constructor stores the underlying stream to be written to. It then sets up the necessary components of the parallel pipeline. First, it creates a **BlockingCollection<byte[]>** to store all of the data chunks to be written. Then, it launches a long-running **Task** that continually pulls from the collection and writes each chunk out to the underlying stream. The **Write** method copies the provided input data into a new array which it enqueues to the **BlockingCollection**; by default, **BlockingCollection** uses a queue data structure under the covers, maintaining first-in-first-out (FIFO) semantics, so the data will be written to the underlying stream in the same order it's added to the collection, a property important for dealing with streams which have an implicit ordering. Finally, closing the stream marks the **BlockingCollection** as complete for adding, which will cause the consuming loop in the **Task** launched in the constructor to cease as soon as the collection is empty, and then waits for the **Task** to complete; this ensures that all data is written to the underlying stream before the underlying stream is closed, and it propagates any exceptions that may have occurred during processing.

With our **TransferStream** in place, we can now use it to parallelize our compression/encryption snippet shown earlier:

C#

```
static void CompressAndEncrypt(string inputFile, string outputFile)
{
    using (var input = File.OpenRead(inputFile))
    using (var output = File.OpenWrite(outputFile))
    using (var rijndael = new RijndaelManaged())
    using (var transform = rijndael.CreateEncryptor())
    using (var encryptor =
        new CryptoStream(output, transform, CryptoStreamMode.Write))
    using (var threadTransfer = new TransferStream(encryptor))
    using (var compressor =
        new GZipStream(threadTransfer, CompressionMode.Compress, true))
        input.CopyTo(compressor);
}
```

With those simple changes, we've now modified the operation so that both the compression and the encryption may run in parallel. Of course, it's important to note here that there are implicit limits on how much speedup I can achieve from this kind of parallelization. At best the code is doing only two elements of work concurrently, overlapping the compression with encryption, and thus even on a machine with more than two cores, the best speedup I can hope to achieve is 2x. Note, too, that I could use additional transfer streams in order to read concurrently with compressing and to write concurrently with encrypting, as such:

C#

```
static void CompressAndEncrypt(string inputFile, string outputFile)
{
    using (var input = File.OpenRead(inputFile))
    using (var output = File.OpenWrite(outputFile))
    using (var t2 = new TransferStream(output))
    using (var rijndael = new RijndaelManaged())
    using (var transform = rijndael.CreateEncryptor())
    using (var encryptor =
        new CryptoStream(t2, transform, CryptoStreamMode.Write))
    using (var t1 = new TransferStream(encryptor))
    using (var compressor =
        new GZipStream(t1, CompressionMode.Compress, true))
    using (var t0 = new TransferStream(compressor))
        input.CopyTo(t0);
}
```

Benefits of doing this might manifest if I/O is a bottleneck.

IPRODUCERCONSUMERCOLLECTION<T>

As mentioned, **BlockingCollection<T>** defaults to using a queue as its storage mechanism, but arbitrary storage mechanisms are supported. This is done utilizing a new interface in the .NET Framework 4, passing an instance of an implementing type to the **BlockingCollection**'s constructor:

C#

```
public interface IProducerConsumerCollection<T> :
    IEnumerable<T>, ICollection, IEnumerable
{
    bool TryAdd(T item);
    bool TryTake(out T item);
    T[] ToArray();
    void CopyTo(T[] array, int index);
}

public class BlockingCollection<T> : //...
{
    //...
    public BlockingCollection(
        IProducerConsumerCollection<T> collection);
    public BlockingCollection(
        IProducerConsumerCollection<T> collection, int boundedCapacity);
    //...
}
```

Aptly named to contain the name of this pattern, **IProducerConsumerCollection<T>** represents a collection used in producer/consumer implementations, where data will be added to the collection by producers and taken from it by consumers. Hence, the primary two methods on the interface are **TryAdd** and **TryTake**, both of which must be implemented in a thread-safe and atomic manner.

*The .NET Framework 4 provides three concrete implementations of this interface: **ConcurrentQueue<T>**, **ConcurrentStack<T>**, and **ConcurrentBag<T>**. **ConcurrentQueue<T>** is the implementation of the interface used by default by **BlockingCollection<T>**, providing first-in-first-out (FIFO) semantics. **ConcurrentStack<T>** provides last-in-first-out (LIFO) behavior, and **ConcurrentBag<T>** eschews ordering guarantees in favor of improved performance in various use cases, in particular those in which the same thread will be acting as both a producer and a consumer.*

In addition to **BlockingCollection<T>**, other data structures may be built around **IProducerConsumerCollection<T>**. For example, an object pool is a simple data structure that's meant to allow object reuse. We could build a concurrent object pool by tying it to a particular storage type, or we can implement one in terms of **IProducerConsumerCollection<T>**.

C#

```
public sealed class ObjectPool<T>
{
    private Func<T> _generator;
    private IProducerConsumerCollection<T> _objects;
```

```
        public ObjectPool(Func<T> generator)
            : this(generator, new ConcurrentQueue<T>()) { }

        public ObjectPool(
            Func<T> generator, IProducerConsumerCollection<T> storage)
        {
            if (generator == null) throw new ArgumentNullException("generator");
            if (storage == null) throw new ArgumentNullException("storage");
            _generator = generator;
            _objects = storage;
        }

        public T Get()
        {
            T item;
            if (!_objects.TryTake(out item)) item = _generator();
            return item;
        }

        public void Put(T item) { _objects.TryAdd(item); }
    }
```

By parameterizing the storage in this manner, we can adapt our **ObjectPool<T>** based on use cases and the associated strengths of the collection implementation. For example, for doing a graphics-intensive UI application, we may want to render to buffers on background threads and then “bitblip” those buffers onto the UI on the UI thread. Given the likely size of these buffers, rather than continually allocating large objects and forcing the garbage collector to clean up after me, we can pool them. In this case, a **ConcurrentQueue<T>** is a likely choice for the underlying storage. Conversely, if the pool were being used in a concurrent memory allocator to cache objects of varying sizes, I don’t need the FIFO-ness of **ConcurrentQueue<T>**, and I would be better off with a data structure that minimizes synchronization between threads; for this purpose, **ConcurrentBag<T>** might be ideal.

*Under the covers, **ConcurrentBag<T>** utilizes a list of instances of T per thread. Each thread that accesses the bag is able to add and remove data in relative isolation from other threads accessing the bag. Only when a thread tries to take data out and its local list is empty will it go in search of items from other threads (the implementation makes the thread-local lists visible to other threads for only this purpose). This might sound familiar: **ConcurrentBag<T>** implements a pattern very similar to the work-stealing algorithm employed by the the .NET Framework 4 ThreadPool.*

*While accessing the local list is relatively inexpensive, stealing from another thread’s list is relatively quite expensive. As a result, **ConcurrentBag<T>** is best for situations where each thread only needs its own local list the majority of the time. In the object pool example, to assist with this it could be worthwhile for every thread to initially populate the pool with some objects, such that when it later gets and puts objects, it will be dealing predominantly with its own queue.*

PRODUCER/CONSUMER EVERYWHERE

If you've written a Windows-based application, it's extremely likely you've used the producer/consumer pattern, potentially without even realizing it. Producer/consumer has many prominent implementations.

THREAD POOLS

If you've used a thread pool, you've used a quintessential implementation of the producer/consumer pattern. A thread pool is typically engineered around a data structure containing work to be performed. Every thread in the pool monitors this data structure, waiting for work to arrive. When work does arrive, a thread retrieves the work, processes it, and goes back to wait for more. In this capacity, the work that's being produced is consumed by the threads in the pool and executed. Utilizing the **BlockingCollection<T>** type we've already seen, it's straightforward to build a simple, no-frills thread pool:

C#

```
public static class SimpleThreadPool
{
    private static BlockingCollection<Action> _work =
        new BlockingCollection<Action>();

    static SimpleThreadPool()
    {
        for (int i = 0; i < Environment.ProcessorCount; i++)
        {
            new Thread(() =>
            {
                foreach (var action in _work.GetConsumingEnumerable())
                {
                    action();
                }
            }) { IsBackground = true }.Start();
        }
    }

    public static void QueueWorkItem(Action workItem) { _work.Add(workItem); }
}
```

In concept, this is very similar to how the **ThreadPool** type in the .NET Framework 3.5 and earlier operated. In the .NET Framework 4, the data structure used to store the work to be executed is more distributed. Rather than maintaining a single global queue, as is done in the above example, the **ThreadPool** in .NET Framework 4 maintains not only a global queue but also a queue per thread. Work generated outside of the pool goes into the global queue as it always did, but threads in the pool can put their generated work into the thread-local queues rather than into the global queues. When threads go in search of work to be executed, they first examine their local queue, and only if they don't find anything there, they then check the global queue. If the global queue is found to be empty, the threads are then also able to check the queues of their peers, "stealing" work from other threads in order to stay busy. This work-stealing approach can provide significant benefits in the form of both minimized contention and synchronization between threads (in an ideal workload, threads can spend most of their time working on their own local queues) as well as cache utilization. (You can approximate this behavior with the **SimpleThreadPool** by instantiating the **BlockingCollection<Action>** with an underlying **ConcurrentBag<Action>** rather than utilizing the default **ConcurrentQueue<Action>**.)

> *In the previous paragraph, we said that "threads in the pool can put their generated work into the thread-local queues," not that they necessarily do. In fact, the* ***ThreadPool.QueueUserWorkItem*** *method is unable to take advantage of this work-stealing support. The functionality is only available through* ***Task****s, for which it is turned on by default. This behavior can be disabled on a per-****Task*** *basis using* ***TaskCreationOptions.PreferFairness****.*

By default, **Task**s execute in the **ThreadPool** using these internal work-stealing queues. This functionality isn't hardwired into **Task**s, however. Rather, the functionality is abstracted through the **TaskScheduler** type. **Task**s execute on **TaskScheduler**s, and the .NET Framework 4 comes with a built-in **TaskScheduler** that targets this functionality in the **ThreadPool**; this implementation is what's returned from the **TaskScheduler.Default** property, and as this property's name implies, this is the default scheduler used by **Task**s. As with anything where someone talks about a "default," there's usually a mechanism to override the default, and that does in fact exist for **Task** execution. It's possible to write custom **TaskScheduler** implementations to execute **Task**s in whatever manner is needed by the application.

TaskScheduler itself embodies the concept of producer/consumer. As an abstract class, it provides several abstract methods that must be overridden and a few virtual methods that may be. The primary abstract method is called **QueueTask**, and is used by the rest of the .NET Framework infrastructure, acting as the producer, to queue tasks into the scheduler. The scheduler implementation then acts as the consumer, executing those tasks in whatever manner it sees fit. We can build a very simple, no frills **TaskScheduler**, based on the previously shown **SimpleThreadPool**, simply by delegating from **QueueTask** to **QueueWorkItem**, using a delegate that executes the task:

C#

```
public sealed class SimpleThreadPoolTaskScheduler : TaskScheduler
{
    protected override void QueueTask(Task task)
    {
        SimpleThreadPool.QueueWorkItem(() => base.TryExecuteTask(task));
    }

    protected override bool TryExecuteTaskInline(
        Task task, bool taskWasPreviouslyQueued)
    {
        return base.TryExecuteTask(task);
    }

    protected override IEnumerable<Task> GetScheduledTasks()
    {
        throw new NotSupportedException();
    }
}
```

We can then produce tasks to be run on an instance of this scheduler:

C#

```
var myScheduler = new SimpleThreadPoolTaskScheduler();
var t = new Task(() => Console.WriteLine("hello, world"));
t.Start(myScheduler);
```

The **TaskFactory** class, a default instance of which is returned from the static **Task.Factory** property, may also be instantiated with a **TaskScheduler** instance. This then allows us to easily utilize all of the factory methods while targeting a custom scheduler:

C#

```
var factory = new TaskFactory(new SimpleThreadPoolTaskScheduler());
factory.StartNew(() => Console.WriteLine("hello, world"));
```

UI MARSHALING

If you've written a responsive Windows-based application, you've already taken advantage of the producer/consumer pattern. With both Windows Forms and Windows Presentation Foundation (WPF), UI controls must only be accessed from the same thread that created them, a form of thread affinity. This is problematic for several reasons, one of the most evident having to do with UI responsiveness. To write a response application, it's typically necessary to offload work from the UI thread to a background thread, in order to allow that UI thread to continue processing Windows messages that cause the UI to repaint, to respond to mouse input, and so on. That processing occurs with code referred to as a Windows message loop. While the work is executing in the background, it may need to update visual progress indication in the UI, and when it completes, it may need to refresh the UI in some manner. Those interactions often require the manipulation of controls that were created on the UI thread, and as a result, the background thread must marshal calls to those controls to the UI thread.

Both Windows Forms and WPF provide mechanisms for doing this. Windows Forms provides the instance **Invoke** method on the **Control** class. This method accepts a delegate, and marshals the execution of that delegate to the right thread for that **Control**, as demonstrated in the following Windows-based application that updates a label on the UI thread every second:

C#

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;

static class Program
{
    [STAThread]
    static void Main(string[] args)
    {
        var form = new Form();
        var lbl = new Label()
        {
            Dock = DockStyle.Fill,
            TextAlign = ContentAlignment.MiddleCenter
        };
        form.Controls.Add(lbl);
        var handle = form.Handle;

        ThreadPool.QueueUserWorkItem(_ =>
        {
            while (true)
            {
                lbl.Invoke((Action)delegate
                {
                    lbl.Text = DateTime.Now.ToString();
                });
                Thread.Sleep(1000);
            }
        });

        form.ShowDialog();
    }
}
```

The **Invoke** call is synchronous, in that it won't return until the delegate has completed execution. There is also a **BeginInvoke** method, which runs the delegate asynchronously.

This mechanism is itself a producer/consumer implementation. Windows Forms maintains a queue of delegates to be processed by the UI thread. When **Invoke** or **BeginInvoke** is called, it puts the delegate into this queue, and sends a Windows message to the UI thread. The UI thread's message loop eventually processes this message, which tells it to dequeue a delegate from the queue and execute it. In this manner, the thread calling **Invoke** or **BeginInvoke** is the producer, the UI thread is the consumer, and the data being produced and consumed is the delegate.

*The particular pattern of producer/consumer employed by **Invoke** has a special name, "rendezvous," which is typically used to signify multiple threads that meet to exchange data bidirectionally. The caller of **Invoke** is providing a delegate and is potentially getting back the result of that delegate's invocation. The UI thread is receiving a delegate and is potentially handing over the delegate's result. Neither thread may progress past the rendezvous point until the data has been fully exchanged.*

This producer/consumer mechanism is available for WPF as well, through the **Dispatcher** class, which similarly provides **Invoke** and **BeginInvoke** methods. To abstract away this functionality and to make it easier to write components that need to marshal to the UI and that must be usable in multiple UI environments, the .NET Framework provides the **SynchronizationContext** class. **SynchronizationContext** provides **Send** and **Post** methods, which map to **Invoke** and **BeginInvoke**, respectively. Windows Forms provides an internal **SynchronizationContext**-derived type called **WindowsFormsSynchronizationContext**, which overrides **Send** to call **Control.Invoke** and which overrides **Post** to call **Control.BeginInvoke**. WPF provides a similar type. With this in hand, a library can be written in terms of **SynchronizationContext**, and can then be supplied with the right **SynchronziationContext** at runtime to ensure it's able to marshal appropriately to the UI in the current environment.

***SynchronizationContext** may also be used for other purposes, and in fact there are other implementations of it provided in the .NET Framework for non-UI related purposes. For this discussion, however, we'll continue to refer to **SynchronizationContext** pertaining only to UI marshaling.*

To facilitate this, the static **SynchronizationContext.Current** property exists to help code grab a reference to a **SynchronizationContext** that may be used to marshal to the current thread. Both Windows Forms and WPF set this property on the UI thread to the relevant **SynchronizationContext** instance. Code may then get the value of this property and use it to marshal work back to the UI. As an example, I can rewrite the previous example by using **SynchronizationContext.Send** rather than explicitly using **Control.Invoke**:

C#
```
[STAThread]
static void Main(string[] args)
{
    var form = new Form();
    var lbl = new Label()
    {
        Dock = DockStyle.Fill,
        TextAlign = ContentAlignment.MiddleCenter
    };
    form.Controls.Add(lbl);
    var handle = form.Handle;
    var sc = SynchronizationContext.Current;

    ThreadPool.QueueUserWorkItem(_ =>
    {
        while (true)
        {
            sc.Send(delegate
            {
                lbl.Text = DateTime.Now.ToString();
            }, null);
            Thread.Sleep(1000);
        }
    });

    form.ShowDialog();
}
```

As mentioned in the previous section, custom **TaskScheduler** types may be implemented to supply custom consumer implementations for **Task**s being produced. In addition to the default implementation of **TaskScheduler** that targets the .NET Framework **ThreadPool**'s internal work-stealing queues, the .NET Framework 4 also includes the **TaskScheduler.FromCurrentSynchronizationContext** method, which generates a **TaskScheduler** that targets the current synchronization context. We can then take advantage of that functionality to further abstract the previous example:

C#
```
[STAThread]
static void Main(string[] args)
{
    var form = new Form();
    var lbl = new Label()
    {
        Dock = DockStyle.Fill,
        TextAlign = ContentAlignment.MiddleCenter
    };
    form.Controls.Add(lbl);
    var handle = form.Handle;
    var ui = new TaskFactory(
        TaskScheduler.FromCurrentSynchronizationContext());

    ThreadPool.QueueUserWorkItem(_ =>
    {
        while (true)
        {
            ui.StartNew(() => lbl.Text = DateTime.Now.ToString());
            Thread.Sleep(1000);
        }
    });

    form.ShowDialog();
}
```

This ability to execute **Task**s in various contexts also integrates very nicely with continuations and dataflow, for example:

C#

```
Task.Factory.StartNew(() =>
{
    // Run in the background a long computation which generates a result
    return DoLongComputation();
}).ContinueWith(t =>
{
    // Render the result on the UI
    RenderResult(t.Result);
}, TaskScheduler.FromCurrentSynchronizationContext());
```

SYSTEM EVENTS

The **Microsoft.Win32.SystemEvents** class exposes a plethora of static events for being notified about happenings in the system, for example:

C#

```
public static event EventHandler DisplaySettingsChanged;
public static event EventHandler DisplaySettingsChanging;
public static event EventHandler EventsThreadShutdown;
public static event EventHandler InstalledFontsChanged;
public static event EventHandler PaletteChanged;
public static event PowerModeChangedEventHandler PowerModeChanged;
public static event SessionEndedEventHandler SessionEnded;
public static event SessionEndingEventHandler SessionEnding;
public static event SessionSwitchEventHandler SessionSwitch;
public static event EventHandler TimeChanged;
public static event TimerElapsedEventHandler TimerElapsed;
public static event UserPreferenceChangedEventHandler UserPreferenceChanged;
public static event UserPreferenceChangingEventHandler UserPreferenceChanging;
```

The Windows operating system notifies applications of the conditions that lead to most of these events through Windows messages, as discussed in the previous section. To receive these messages, the application must make sure it has a window to which the relevant messages can be broadcast, and a message loop running to process them. Thus, if you subscribe to one of these events, even in an application without UI, **SystemEvents** ensures that a broadcast window has been created and that a thread has been created to run a message loop for it. That thread then waits for messages to arrive and consumes them by translating them into the proper .NET Framework objects and invoking the relevant event. When you register an event handler with an event on **SystemEvents**, in a strong sense you're then implementing the consumer side of this multithreaded, producer/consumer implementation.

AGGREGATIONS

Combining data in one way or another is very common in applications, and aggregation is an extremely common need in parallel applications. In parallel systems, work is divided up, processed in parallel, and the results of these intermediate computations are then combined in some manner to achieve a final output.

In some cases, no special work is required for the last step. For example, if a parallel **for** loop iterates from 0 to N, and the **i**th result is stored into the resulting array's **i**th slot, the aggregation of results into the output array can be done in parallel with no additional work: the locations in the output array may all be written to independently, and no two parallel iterations will attempt to store into the same index.

In many cases, however, special work is required to ensure that the results are aggregated safely. There are several common patterns for achieving such aggregations.

OUTPUTTING A SET OF RESULTS

A common coding pattern in sequential code is of the following form, where some input data is processed, and the results are stored into an output collection:

C#

```
var output = new List<TOutput>();
foreach (var item in input)
{
    var result = Compute(item);
    output.Add(result);
}
```

If the size of the input collection is known in advance, this can be converted into an instance of the aforementioned example, where the results are stored directly into the corresponding slots in the output:

C#

```
var output = new TOutput[input.Count];
for (int i = 0; i < input.Count; i++)
{
    var result = Compute(input[i]);
    output[i] = result;
}
```

This then makes parallelization straightforward, at least as it pertains to aggregation of the results:

C#

```
var output = new TOutput[input.Count];
Parallel.For(0, input.Count, i =>
{
    var result = Compute(input[i]);
    output[i] = result;
});
```

However, this kind of transformation is not always possible. In cases where the input size is not known or where the input collection may not be indexed into, an output collection is needed that may be modified from multiple threads. This may be done using explicit synchronization to ensure the output collection is only modified by a single thread at a time:

C#
```
var output = new List<TOutput>();
Parallel.ForEach(input, item =>
{
    var result = Compute(item);
    lock (output) output.Add(result);
});
```

If the amount of computation done per item is significant, the cost of this locking is likely to be negligible. However, as the amount of computation per item decreases, the overhead of taking and releasing a lock becomes more relevant, and contention on the lock increases as more threads are blocked waiting to acquire it concurrently. To decrease these overheads and to minimize contention, the new thread-safe collections in the .NET Framework 4 may be used. These collections reside in the **System.Collections.Concurrent** namespace, and are engineered to be scalable, minimizing the impact of contention. Some of these collections are implemented with lock-free techniques, while others are implemented using fine-grained locking.

Amongst these new collections, there's no direct corollary to the **List<T>** type. However, there are several collections that address many of the most common usage patterns for **List<T>**. If you reexamine the previous code snippet, you'll notice that the output ordering from the serial code is not necessarily maintained in the parallel version. This is because the order in which the data is stored into the output list is no longer based solely on the order of the data in the input, but also on the order in which the parallel loop chooses to process the elements, how partitioning occurs, and how long each element takes to process. Once we've accepted this issue and have coded the rest of the application to not rely on the output ordering, our choices expand for what collection to use to replace the list. Here I'll use the new **ConcurrentBag<T>** type:

C#
```
var output = new ConcurrentBag<TOutput>();
Parallel.ForEach(input, item =>
{
    var result = Compute(item);
    output.Add(result);
});
```

All of the synchronization necessary to ensure the consistency of the output data structure is handled internally by the **ConcurrentBag**.

OUTPUTTING A SINGLE RESULT

Many algorithms output a single result, rather than a single collection. For example, consider the following serial routine to estimate the value of Pi:

C#
```
const int NUM_STEPS = 100000000;

static double SerialPi()
{
    double sum = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
```

```
        for (int i = 0; i < NUM_STEPS; i++)
        {
            double x = (i + 0.5) * step;
            double partial = 4.0 / (1.0 + x * x);
            sum += partial;
        }
        return step * sum;
    }
```

The output of this operation is a single double value. This value is the sum of millions of independent operations, and thus should be parallelizable. Here is a naïve parallelization:

C#

```
static double NaiveParallelPi()
{
    double sum = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    object obj = new object();
    Parallel.For(0, NUM_STEPS, i =>
    {
        double x = (i + 0.5) * step;
        double partial = 4.0 / (1.0 + x * x);
        lock (obj) sum += partial;
    });
    return step * sum;
}
```

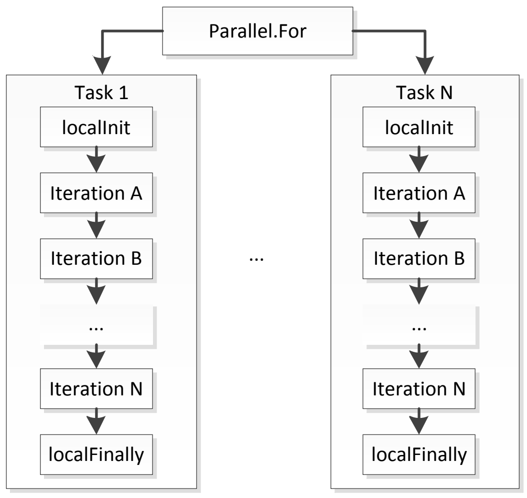
We say "naïve" here, because while this solution is correct, it will also be extremely slow. Every iteration of the parallel loop does only a few real cycles worth of work, made up of a few additions, multiplications, and divisions, and then takes a lock to accumulate that iteration's result into the overall result. The cost of that lock will dominate all of the other work happening in the parallel loop, largely serializing it, such that parallel version will likely run significantly slower than the sequential.

To fix this, we need to minimize the amount of synchronization necessary. That can be achieved by maintaining local sums. We know that certain iterations will never be in conflict with each other, namely those running on the same underlying thread (since a thread can only do one thing at a time), and thus we can maintain a local sum per thread or task being used under the covers in **Parallel.For**. Given the prevalence of this pattern, **Parallel.For** actually bakes in support for it. In addition to passing to **Parallel.For** a delegate for the body, you can also pass in a delegate that represents an initialization routine to be run on each task used by the loop, and a delegate that represents a finalization routine that will be run at the end of the task when no more iterations will be executed in it.

C#

```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive, int toExclusive,
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally);
```

The result of the initialization routine is passed to the first iteration run by that task, the output of that iteration is passed to the next iteration, the output of that iteration is passed to the next, and so on, until finally the last iteration passes its result to the localFinally delegate.



In this manner, a partial result can be built up on each task, and only combined with the partials from other tasks at the end. Our Pi example can thusly be implemented as follows:

C#

```
static double ParallelPi()
{
    double sum = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    object obj = new object();
    Parallel.For(0, NUM_STEPS,
        () => 0.0,
        (i, state, partial) =>
        {
            double x = (i + 0.5) * step;
            return partial + 4.0 / (1.0 + x * x);
        },
        partial => { lock (obj) sum += partial; });
    return step * sum;
}
```

The **localInit** delegate returns an initialized value of **0.0**. The body delegate calculates its iteration's result, adds it to the partial result it was passed in (which either directly from the result of **localInit** or from the previous iteration on the same task), and returns the updated partial. The **localFinally** delegate takes the completed partial, and only then synchronizes with other threads to combine the partial sum into the total sum.

Earlier in this document we saw the performance ramifications of having a very small delegate body. This Pi calculation is an example of that case, and thus we can likely achieve better performance using the batching pattern described previously.

C#

```
static double ParallelPartitionerPi()
{
    double sum = 0.0;
```

```
        double step = 1.0 / (double)NUM_STEPS;
        object obj = new object();
        Parallel.ForEach(Partitioner.Create(0, NUM_STEPS),
            () => 0.0,
            (range, state, partial) =>
            {
                for (int i = range.Item1; i < range.Item2; i++)
                {
                    double x = (i + 0.5) * step;
                    partial += 4.0 / (1.0 + x * x);
                }
                return partial;
            },
            partial => { lock (obj) sum += partial; });
        return step * sum;
    }
```

PLINQ AGGREGATIONS

Any time you find yourself needing to aggregate, think PLINQ. For many problems, aggregation is one of several areas in which PLINQ excels, with a plethora of aggregation support built-in.

TOARRAY / TOLIST / TODICTIONARY / TOLOOKUP

As does LINQ to Objects, PLINQ provides four "To*" methods that may be used to aggregate all of the output from a query into a single data structure. PLINQ internally handles all of the relevant synchronization. For example, here is the previous example of storing all results into a **List<T>**:

C#

```
var output = new List<TOutput>();
foreach (var item in input)
{
    var result = Compute(item);
    output.Add(result);
}
```

This may be converted to a LINQ implementation as follows:

C#

```
var output = input
             .Select(item => Compute(item))
             .ToList();
```

And then it can be parallelized with PLINQ:

C#

```
var output = input.AsParallel()
             .Select(item => Compute(item))
             .ToList();
```

In fact, not only does PLINQ handle all of the synchronization necessary to do this aggregation safely, it can also be used to automatically regain the ordering we lost in our parallelized version when using **Parallel.ForEach**:

C#

```
var output = input.AsParallel().AsOrdered()
              .Select(item => Compute(item))
              .ToList();
```

SINGLE-VALUE AGGREGATIONS

Just as LINQ and PLINQ are useful for aggregating sets of output, they are also quite useful for aggregating down to a single value, with operators including but not limited to **Average**, **Sum**, **Min**, **Max**, and **Aggregate**. As an example, the same Pi calculation can be done using LINQ:

C#

```
static double SerialLinqPi()
{
    double step = 1.0 / (double)NUM_STEPS;
    return Enumerable.Range(0, NUM_STEPS).Select(i =>
    {
        double x = (i + 0.5) * step;
        return 4.0 / (1.0 + x * x);
    }).Sum() * step;
}
```

With a minimal modification, PLINQ can be used to parallelize this:

C#

```
static double ParallelLinqPi()
{
    double step = 1.0 / (double)NUM_STEPS;
    return ParallelEnumerable.Range(0, NUM_STEPS).Select(i =>
    {
        double x = (i + 0.5) * step;
        return 4.0 / (1.0 + x * x);
    }).Sum() * step;
}
```

This parallel implementation does scale nicely as compared to the serial LINQ version. However, if you test the serial LINQ version and compare its performance against the previously shown serial **for** loop version, you'll find that the serial LINQ version is significantly more expensive; this is largely due to all of the extra delegate invocations involved in its execution. We can create a hybrid solution that utilizes PLINQ to creation partitions and sum partial results but creates the individual partial results on each partition using a **for** loop:

C#

```
static double ParallelPartitionLinqPi()
{
    double step = 1.0 / (double)NUM_STEPS;
    return Partitioner.Create(0, NUM_STEPS).AsParallel().Select(range =>
    {
        double partial = 0.0;
        for (int i = range.Item1; i < range.Item2; i++)
        {
            double x = (i + 0.5) * step;
            partial += 4.0 / (1.0 + x * x);
        }
        return partial;
    }).Sum() * step;
}
```

AGGREGATE

Both LINQ and PLINQ may be used for arbitrary aggregations using the **Aggregate** method. **Aggregate** has several overloads, including several unique to PLINQ that provide more support for parallelization. PLINQ assumes that the aggregation delegates are both associative and commutative; this limits the kinds of operations that may be performed, but also allows PLINQ to optimize its operation in ways that wouldn't otherwise be possible if it couldn't make these assumptions.

The most advanced PLINQ overload of **Aggregate** is very similar in nature and purpose to the **Parallel.ForEach** overload that supports **localInit** and **localFinally** delegates:

C#

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this ParallelQuery<TSource> source,
    Func<TAccumulate> seedFactory,
    Func<TAccumulate, TSource, TAccumulate> updateAccumulatorFunc,
    Func<TAccumulate, TAccumulate, TAccumulate> combineAccumulatorsFunc,
    Func<TAccumulate, TResult> resultSelector);
```

The **seedFactory** delegate is the logical equivalent of **localInit**, executed once per partition to provide a seed for the aggregation accumulator on that partition. The **updateAccumulatorFunc** is akin to the body delegate, provided with the current value of the accumulator and the current element, and returning the updated accumulator value based on incorporating the current element. The **combineAccumulatorsFunc** is logically equivalent to the **localFinally** delegate, combining the results from multiple partitions (unlike **localFinally**, which is given the current task's final value and may do with it what it chooses, this delegate accepts two accumulator values and returns the aggregation of the two). And finally, the **resultSelector** takes the total accumulation and processes it into a result value. In many scenarios, **TAccumulate** will be **TResult**, and this **resultSelector** will simply return its input.

As a concrete case for where this aggregation operator is useful, consider a common pattern: the need to take the best N elements output from a query. An example of this might be in a spell checker. Given an input word list, compare the input text against each word in the dictionary and compute a distance metric between the two. We then want to select out the best results to be displayed to the user as options. One approach to implementing this with PLINQ would be as follows:

C#

```
var bestResults = dictionaryWordList
    .Select(word => new { Word = word, Distance = GetDistance(word, text) })
    .TakeTop(p => -p.Distance, NUM_RESULTS_TO_RETURN)
    .Select(p => p.Word)
    .ToList();
```

In the previous example, **TakeTop** is implemented as:

C#

```
public static IEnumerable<TSource> TakeTop<TSource, TKey>(
    this ParallelQuery<TSource> source,
    Func<TSource, TKey> keySelector,
    int count)
{
    return source.OrderBy(keySelector).Take(count);
}
```

The concept of “take the top N” here is implemented by first sorting all of the result using **OrderBy** and then taking the first N results. This may be overly expensive, however. For a large word list of several hundred thousand words, we’re forced to sort the entire result set, and sorting has relatively high computational complexity. If we’re only selecting out a handful of results, we can do better. For example, in a sequential implementation we could simply walk the result set, keeping track of the top N along the way. We can implement this in parallel by walking each partition in a similar manner, keeping track of the best N from each partition. An example implementation of this approach is included in the Parallel Extensions samples at http://code.msdn.microsoft.com/ParExtSamples, and the relevant portion is shown here:

C#

```
public static IEnumerable<TSource> TakeTop<TSource, TKey>(
    this ParallelQuery<TSource> source,
    Func<TSource, TKey> keySelector,
    int count)
{
    return source.Aggregate(
        // seedFactory
        () => new SortedTopN<TKey,TSource>(count),

        // updateAccumulatorFunc
        (accum, item) =>
        {
            accum.Add(keySelector(item), item);
            return accum;
        },

        // combineAccumulatorsFunc
        (accum1, accum2) =>
        {
            foreach (var item in accum2) accum1.Add(item);
            return accum1;
        },

        // resultSelector
        (accum) => accum.Values);
}
```

The **seedFactory** delegate, called once for each partition, generates a new data structure to keep track of the top **count** items added to it. Up until **count** items, all items added to the collection get stored. Beyond that, every time a new item is added, it’s compared against the least item currently stored, and if it’s greater than it, the least item is bumped out and the new item is stored in its place. The **updateAccumulatorFunc** simply adds the current item to the data structure accumulator (according to the rules of only maintaining the top N). The **combineAccumulatorsFunc** combines two of these data structures by adding all of the elements from one to the other and then returning that end result. And the **resultSelector** simply returns the set of values from the ultimate resulting accumulator.

MAPREDUCE

The “MapReduce” pattern was introduced to handle large-scale computations across a cluster of servers, often involving massive amounts of data. The pattern is relevant even for a single multi-core machine, however. Here is a description of the pattern’s core algorithm:

> “The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: **Map** and **Reduce**.
>
> **Map**, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the **Reduce** function.
>
> The **Reduce** function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per **Reduce** invocation. The intermediate values are supplied to the user's **Reduce** function via an iterator.”
>
> *Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (Jan. 2008), 107-113. DOI= http://doi.acm.org/10.1145/1327452.1327492*

IMPLEMENTING MAPREDUCE WITH PLINQ

The core MapReduce pattern (and many variations on it) is easily implemented with LINQ, and thus with PLINQ. To see how, we’ll break apart the description of the problem as shown previously.

The description of the **Map** function is that it takes a single input value and returns a set of mapped values: this is the purpose of LINQ’s **SelectMany** operator, which is defined as follows:

C#
```
public static ParallelQuery<TResult> SelectMany<TSource, TResult>(
    this ParallelQuery<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector);
```

Moving on, the MapReduce problem description highlights that results are then grouped according to an intermediate key. That grouping operation is the purpose of the LINQ **GroupBy** operator:

C#
```
public static ParallelQuery<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this ParallelQuery<TSource> source,
    Func<TSource, TKey> keySelector);
```

Finally, a reduction is performed by a function that takes each intermediate key and a set of values for that key, and produces any number of outputs per key. Again, that’s the purpose of **SelectMany**.

We can put all of this together to implement MapReduce in LINQ:

C#
```
public static IEnumerable<TResult> MapReduce<TSource, TMapped, TKey, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce)
{
    return source.SelectMany(map)
                 .GroupBy(keySelector)
                 .SelectMany(reduce);
}
```

Parallelizing this new combined operator with PLINQ is as simply as changing the input and output types to work with PLINQ's **ParallelQuery<>** type instead of with LINQ's **IEnumerable<>**:

C#

```csharp
public static ParallelQuery<TResult> MapReduce<TSource, TMapped, TKey, TResult>(
    this ParallelQuery<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce)
{
    return source.SelectMany(map)
                 .GroupBy(keySelector)
                 .SelectMany(reduce);
}
```

USING MAPREDUCE

The typical example used to demonstrate a MapReduce implementation is a word counting routine, where a bunch of documents are parsed, and the frequency of all of the words across all of the documents is summarized. For this example, the map function takes in an input document and outputs all of the words in that document. The grouping phase groups all of the identical words together, such that the reduce phase can then count the words in each group and output a word/count pair for each grouping:

C#

```csharp
var files = Directory.EnumerateFiles(dirPath, "*.txt").AsParallel();
var counts = files.MapReduce(
    path => File.ReadLines(path).SelectMany(line => line.Split(delimiters)),
    word => word,
    group => new[] { new KeyValuePair<string, int>(group.Key, group.Count()) });
```

The tokenization here is done in a naïve fashion using the **String.Split** function, which accepts the list of characters to use as delimiters. For this example, that list was generated using another LINQ query that generates an array of all of the ASCII white space and punctuation characters:

C#

```csharp
static char[] delimiters =
    Enumerable.Range(0, 256).Select(i => (char)i)
    .Where(c => Char.IsWhiteSpace(c) || Char.IsPunctuation(c))
    .ToArray();
```

DEPENDENCIES

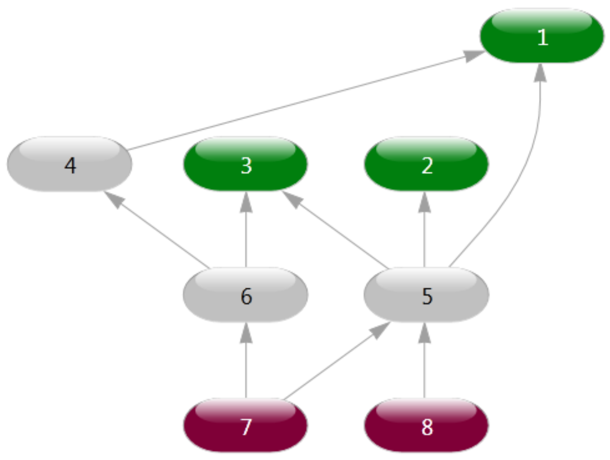
A dependency is the Achilles heel of parallelism. A dependency between two operations implies that one operation can't run until the other operation has completed, inhibiting parallelism. Many real-world problems have implicit dependencies, and thus it's important to be able to accommodate them and extract as much parallelism as is possible. With the producer/consumer pattern, we've already explored one key solution to specific kinds of dependencies. Here we'll examine others.

DIRECTED ACYCLIC GRAPHS

It's very common in real-world problems to see patterns where dependencies between components form a directed acyclic graph (DAG). As an example of this, consider compiling a solution of eight code projects. Some projects have references to other projects, and thus depend on those projects being built first. The dependencies are as follows:

- Components 1, 2, and 3 depend on nothing else in the solution.
- Component 4 depends on 1.
- Component 5 depends on 1, 2, and 3.
- Component 6 depends on 3 and 4.
- Component 7 depends on 5 and 6 and has no dependencies on it.
- Component 8 depends on 5 and has no dependencies on it.

This set of dependencies forms the following DAG (as rendered by the new Architecture tools in Visual Studio 2010):



If building each component is represented as a **Task**, we can take advantage of continuations to express as much parallelism as is possible:

C#
```
var f = Task.Factory;
var build1 = f.StartNew(() => Build(project1));
var build2 = f.StartNew(() => Build(project2));
var build3 = f.StartNew(() => Build(project3));
var build4 = f.ContinueWhenAll(new[] { build1 },
    _ => Build(project4));
var build5 = f.ContinueWhenAll(new[] { build1, build2, build3 },
    _ => Build(project5));
var build6 = f.ContinueWhenAll(new[] { build3, build4 },
    _ => Build(project6));
var build7 = f.ContinueWhenAll(new[] { build5, build6 },
    _ => Build(project7));
var build8 = f.ContinueWhenAll(new[] { build5 },
    _ => Build(project8));
Task.WaitAll(build1, build2, build3, build4, build5, build6, build7, build8);
```

With this code, we immediately queue up work items to build the first three projects. As those projects complete, projects with dependencies on them will be queued to build as soon as all of their dependencies are satisfied.

ITERATING IN LOCK STEP

A common pattern in many algorithms is to have a series of operations that need to be done, from 0 to N, where step **i**+1 can't realistically be processed until step **i** has completed. This often occurs in image processing algorithms, where processing one scan line of the image depends on the previous scan line having already been processed. This also frequently occurs in analysis of a system over time, where each iteration represents another step forward in time, and the world at iteration **i**+1 depends on the state of the world after iteration **i**.

An example of the latter is in simple modeling of the dissipation of heat across a metal plate, exemplified by the following sequential code:

C#

```
float[,] SequentialSimulation(int plateSize, int timeSteps)
{
    // Initial plates for previous and current time steps, with
    // heat starting on one side
    var prevIter = new float[plateSize, plateSize];
    var currIter = new float[plateSize, plateSize];
    for (int y = 0; y < plateSize; y++) prevIter[y, 0] = 255.0f;

    // Run simulation
    for (int step = 0; step < timeSteps; step++)
    {
        for (int y = 1; y < plateSize - 1; y++)
        {
            for (int x = 1; x < plateSize - 1; x++)
            {
                currIter[y, x] =
                   ((prevIter[y, x - 1] +
                     prevIter[y, x + 1] +
                     prevIter[y - 1, x] +
                     prevIter[y + 1, x]) * 0.25f);
            }
        }
        Swap(ref prevIter, ref currIter);
    }

    // Return results
```

```
        return prevIter;
    }

    private static void Swap<T>(ref T one, ref T two)
    {
        T tmp = one; one = two; two = tmp;
    }
```

On close examination, you'll see that this can actually be expressed as a DAG, since the cell **[y,x]** for time step **i**+1 can be computed as soon as the cells **[y,x-1]**, **[y,x+1]**, **[y-1,x]**, and **[y+1,x]** from time step i are completed. However, attempting this kind of parallelization can lead to significant complications. For one, the amount of computation required per cell is very small, just a few array accesses, additions, and multiplications; creating a new **Task** for such an operation is respectively a lot of overhead. Another significant complication is around memory management. In the serial scheme shown, we only need to maintain two plate arrays, one storing the previous iteration and one storing the current. Once we start expressing the problem as a DAG, we run into issues of potentially needing plates (or at least portions of plates) for many generations.

An easier solution is simply to parallelize one or more of the inner loops, but not the outer loop. In effect, we can parallelize each step of the simulation, just not all time steps of the simulation concurrently:

C#

```
// Run simulation
for (int step = 0; step < timeSteps; step++)
{
    Parallel.For(1, plateSize - 1, y =>
    {
        for (int x = 1; x < plateSize - 1; x++)
        {
            currIter[y, x] =
                ((prevIter[y, x - 1] +
                  prevIter[y, x + 1] +
                  prevIter[y - 1, x] +
                  prevIter[y + 1, x]) * 0.25f);
        }
    });
    Swap(ref prevIter, ref currIter);
}
```

Typically, this approach will be sufficient. For some kinds of problems, however, it can be more efficient (largely for reasons of cache locality) to ensure that the same thread processes the same sections of iteration space on each time step. We can accomplish that by using **Task**s directly, rather than by using **Parallel.For**. For this heated plate example, we spin up one **Task** per processor and assign each a portion of the plate's size; each **Task** is responsible for processing that portion at each time step. Now, we need some way of ensuring that each **Task** does not go on to process its portion of the plate at iteration **i**+1 until all tasks have completed processing iteration **i**. For that purpose, we can use the **System.Threading.Barrier** class that's new to the .NET Framework 4:

C#

```
// Run simulation
int numTasks = Environment.ProcessorCount;
var tasks = new Task[numTasks];
var stepBarrier = new Barrier(numTasks, _ => Swap(ref prevIter, ref currIter));
int chunkSize = (plateSize - 2) / numTasks;
for (int i = 0; i < numTasks; i++)
```

```
{
    int yStart = 1 + (chunkSize * i);
    int yEnd = (i == numTasks - 1) ? plateSize - 1 : yStart + chunkSize;
    tasks[i] = Task.Factory.StartNew(() =>
    {
        for (int step = 0; step < timeSteps; step++)
        {
            for (int y = yStart; y < yEnd; y++)
            {
                for (int x = 1; x < plateSize - 1; x++)
                {
                    currIter[y, x] =
                       ((prevIter[y, x - 1] +
                         prevIter[y, x + 1] +
                         prevIter[y - 1, x] +
                         prevIter[y + 1, x]) * 0.25f);
                }
            }
            stepBarrier.SignalAndWait();
        }
    });
}
Task.WaitAll(tasks);
```

Each **Task** calls the **Barrier**'s **SignalAndWait** method at the end of each time step, and the **Barrier** ensures that no tasks progress beyond this point in a given iteration until all tasks have reached this point for that iteration. Further, because we need to swap the previous and current plates at the end of every time step, we register that swap code with the **Barrier** as a post-phase action delegate; the **Barrier** will run that code on one thread once all **Task**s have reached the **Barrier** in a given iteration and before it releases any **Task**s to the next iteration.

DYNAMIC PROGRAMMING

Not to be confused with dynamic languages or with Visual Basic's and C#'s for dynamic invocation, "dynamic programming" in computer science is a classification for optimization algorithms that break down problems recursively into smaller problems, caching (or "memoizing") the results of those subproblems for future use, rather than recomputing them every time they're needed. Common dynamic programming problems include longest common subsequence, matrix-chain multiplication, string edit distance, and sequence alignment. Dynamic programming problems are ripe with dependencies, but these dependencies can be bested and typically don't prevent parallelization.

To demonstrate parallelization of a dynamic programming program, consider a simple implementation of the Levenshtein edit distance algorithm:

C#

```
static int EditDistance(string s1, string s2)
{
    int[,] dist = new int[s1.Length + 1, s2.Length + 1];
    for (int i = 0; i <= s1.Length; i++) dist[i, 0] = i;
    for (int j = 0; j <= s2.Length; j++) dist[0, j] = j;

    for (int i = 1; i <= s1.Length; i++)
    {
        for (int j = 1; j <= s2.Length; j++)
        {
            dist[i, j] = (s1[i - 1] == s2[j - 1]) ?
                dist[i - 1, j - 1] :
                1 + Math.Min(dist[i - 1, j],
                    Math.Min(dist[i, j - 1],
                             dist[i - 1, j - 1]));
        }
    }
    return dist[s1.Length, s2.Length];
}
```

This algorithm builds up a distance matrix, where the **[i,j]** entry represents the number of operations it would take to transform the first **i** characters of string **s1** into the first **j** characters of **s2**; an operation is defined as a single character substitution, insertion, or deletion. To see how this works in action, consider computing the distance between two strings, going from “PARALLEL" to “STEPHEN”. We start by initializing the first row to the values 0 through 8… these represent deletions (going from “P” to “” requires 1 deletion, going from “PA” to “” requires 2 deletions, going from “PAR” to “” requires 3 deletions, and so on). We also initialize the first column to the values 0 through 7… these represent additions (going from “” to “STEP” requires 4 additions, going from “” to “STEPHEN” requires 7 additions, and so on).

		P	A	R	A	L	L	E	L
	0	1	2	3	4	5	6	7	8
S	1								
T	2								
E	3								
P	4								
H	5								
E	6								
N	7								

Now starting from cell **[1,1]** we walk down each column, calculating each cell’s value in order. Let’s call the two strings **s1** and **s2**. A cell’s value is based on two potential options:

1. **The two characters corresponding with that cell are the same.** The value for this cell is the same as the value for the diagonally previous cell, which represents comparing each of the two strings without the current letter (for example, if we already know the value for comparing “STEPH” and “PARALL”, the value for “STEPHE” and “PARALLE” is the same, as we added the same letter to the end of both strings, and thus the distance doesn’t change).
2. **The two characters corresponding with that cell are different.** The value for this cell is the minimum of three potential operations: a deletion, a substitution, or an insertion. These are represented by adding 1 to the value retrieved from the cells immediately above, diagonally to the upper-left, and to the left.

As an exercise, try filling in the table. The completed table for “PARALLEL” and “STEPHEN” is as follows:

		P	A	R	A	L	L	E	L
	0	1	2	3	4	5	6	7	8
S	1	1	2	3	4	5	6	7	8
T	2	2	2	3	4	5	6	7	8
E	3	3	3	3	4	5	6	6	7
P	4	3	4	4	4	5	6	7	7
H	5	4	4	5	5	5	6	7	8
E	6	5	5	5	6	6	6	6	7
N	7	6	6	6	6	7	7	7	**7**

As you filled it in, you should have noticed that the numbers were filled in almost as if a wavefront were moving through the table, since a cell **[i,j]** can be filled in as soon as the three cells **[i-1,j-1]**, **[i-1,j]**, and **[i,j-1]** are completed (and in fact, the completion of the cell above and to the left implies that the diagonal cell was also completed). From a parallel perspective, this should sound familiar, harkening back to our discussion of DAGs. We could, in fact, parallelize this problem using one **Task** per cell and multi-task continuations, but as with previous examples on dependencies, there's very little work being done per cell, and the overhead of creating a task for each cell would significantly outweigh the value of doing so.

You'll notice, however, that there are macro versions of these micro problems: take any rectangular subset of the cells in the grid, and that rectangular subset can be completed when the rectangular block above it and to its left have completed. This presents a solution: we can block the entire matrix up into rectangular regions, run the algorithm over each block, and use continuations for dependencies between blocks. This amortizes the cost of the parallelization with tasks across all of the cells in each block, making a **Task** worthwhile as long as the block is big enough.

Since the macro problem is the same as the micro, we can write one routine to work with this general pattern, dubbed the "wavefront" pattern; we can then write a small routine on top of it to deal with blocking as needed.

Here's an implementation based on **Task**s and continuations:

C#

```
static void Wavefront(
    int numRows, int numColumns, Action<int, int> processRowColumnCell)
{
    // ... Would validate arguments here

    // Store the previous row of tasks as well as the previous task
    // in the current row.
    Task[] prevTaskRow = new Task[numColumns];
    Task prevTaskInCurrentRow = null;
    var dependencies = new Task[2];

    // Create a task for each cell.
    for (int row = 0; row < numRows; row++)
    {
        prevTaskInCurrentRow = null;
        for (int column = 0; column < numColumns; column++)
        {
            // In-scope locals for being captured in the task closures.
            int j = row, i = column;

            // Create a task with the appropriate dependencies.
            Task curTask;
            if (row == 0 && column == 0)
            {
                // Upper-left task kicks everything off,
                // having no dependencies.
                curTask = Task.Factory.StartNew(() =>
                    processRowColumnCell(j, i));
            }
            else if (row == 0 || column == 0)
            {
                // Tasks in the left-most column depend only on the task
                // above them, and tasks in the top row depend only on
                // the task to their left.
                var antecedent = column == 0 ?
                    prevTaskRow[0] : prevTaskInCurrentRow;
                curTask = antecedent.ContinueWith(p =>
                {
                    p.Wait(); // Necessary only to propagate exceptions.
                    processRowColumnCell(j, i);
                });
            }
            else // row > 0 && column > 0
            {
                // All other tasks depend on both the tasks above
                // and to the left.
                dependencies[0] = prevTaskInCurrentRow;
                dependencies[1] = prevTaskRow[column];
                curTask = Task.Factory.ContinueWhenAll(dependencies, ps =>
                {
                    Task.WaitAll(ps); // Necessary to propagate exceptions
                    processRowColumnCell(j, i);
                });
            }

            // Keep track of the task just created for future iterations.
            prevTaskRow[column] = prevTaskInCurrentRow = curTask;
        }
    }

    // Wait for the last task to be done.
    prevTaskInCurrentRow.Wait();
}
```

While a non-trivial amount of code, it's actually quite straightforward. We maintain an array of **Task**s represented the previous row, and a **Task** represented the previous **Task** in the current row. We start by launching a **Task** to process the initial **Task** in the **[0,0]** slot, since it has no dependencies. We then walk each cell in each row, creating a continuation **Task** for each cell. In the first row or the first column, there is just one dependency, the previous cell in that row or the previous cell in that column, respectively. For all other cells, the continuation is based on the previous cell in both the current row and the current column. At the end, we just wait for the last **Task** to complete.

With that code in place, we now need to support blocks, and we can layer another Wavefront function on top to support that:

C#

```
static void Wavefront(
    int numRows, int numColumns,
    int numBlocksPerRow, int numBlocksPerColumn,
    Action<int, int, int, int> processBlock)
{
    // ... Would validate arguments here

    // Compute the size of each block.
    int rowBlockSize = numRows / numBlocksPerRow;
    int columnBlockSize = numColumns / numBlocksPerColumn;

    Wavefront(numBlocksPerRow, numBlocksPerColumn, (row, column) =>
    {
        int start_i = row * rowBlockSize;
        int end_i = row < numBlocksPerRow - 1 ?
            start_i + rowBlockSize : numRows;

        int start_j = column * columnBlockSize;
        int end_j = column < numBlocksPerColumn - 1 ?
            start_j + columnBlockSize : numColumns;

        processBlock(start_i, end_i, start_j, end_j);
    });
}
```

This code is much simpler. The function accepts the number of rows and number of columns, but also the number of blocks to use. The delegate now accepts four values, the starting and ending position of the block for both row and column. The function validates parameters, and then computes the size of each block. From there, it delegates to the Wavefront overload we previously implemented. Inside the delegate, it uses the provided row and column number along with the block size to compute the starting and ending row and column positions, and then passes those values down to the user-supplied delegate.

With this Wavefront pattern implementation in place, we can now parallelize our **EditDistance** function with very little additional code:

C#

```
static int ParallelEditDistance(string s1, string s2)
{
    int[,] dist = new int[s1.Length + 1, s2.Length + 1];
    for (int i = 0; i <= s1.Length; i++) dist[i, 0] = i;
    for (int j = 0; j <= s2.Length; j++) dist[0, j] = j;
    int numBlocks = Environment.ProcessorCount * 2;

    Wavefront(s1.Length, s2.Length, numBlocks, numBlocks,
        (start_i, end_i, start_j, end_j) =>
    {
        for (int i = start_i + 1; i <= end_i; i++)
```

```
            {
                for (int j = start_j + 1; j <= end_j; j++)
                {
                    dist[i, j] = (s1[i - 1] == s2[j - 1]) ?
                        dist[i - 1, j - 1] :
                        1 + Math.Min(dist[i - 1, j],
                            Math.Min(dist[i, j - 1],
                                     dist[i - 1, j - 1]));
                }
            }
        });

        return dist[s1.Length, s2.Length];
    }
```

For small strings, the parallelization overheads will outweigh any benefits. But for large strings, this parallelization approach can yield significant benefits.

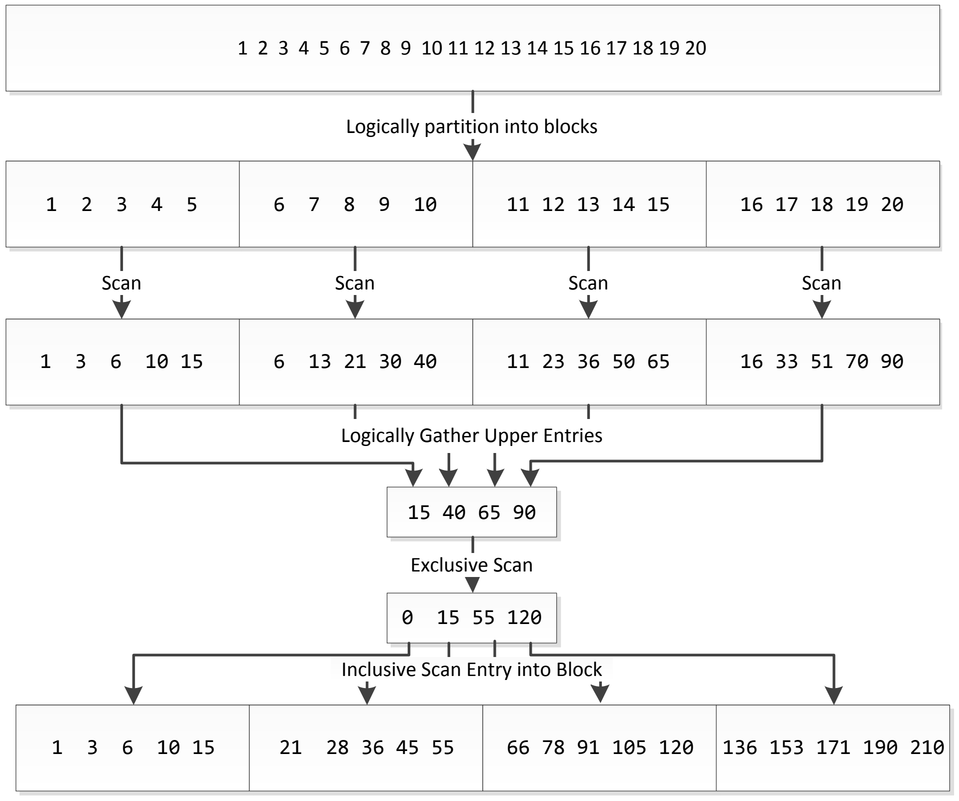
FOLD AND SCAN

Sometimes a dependency is so significant, there is seemingly no way around it. One such example of this is a "fold" operation. A fold is typically of the following form:

C#
```
b[0] = a[0];
for (int i = 1; i < N; i++)
{
    b[i] = f(b[i - 1], a[i]);
}
```

As an example, if the function **f** is addition and the input array is 1,2,3,4,5, the result of the fold will be 1,3,6,10,15. Each iteration of the fold operation is entirely dependent on the previous iteration, leaving little room for parallelism. However, as with aggregations, we can make an accommodation: if we guarantee that the **f** function is associative, that enables enough wiggle room to introduce some parallelism (many operations are associative, including the addition operation used as an example). With this restriction on the operation, it's typically called a "scan," or sometimes "prefix scan."

There are several ways a scan may be parallelized. An approach we'll show here is based on blocking. Consider wanting to parallel scan the input sequence of the numbers 1 through 20 using the addition operator on a quad-core machine. We can split the input into four blocks, and then in parallel, scan each block individually. Once that step has completed, we can pick out the top element from each block, and do a sequential, exclusive scan on just those four entries; in an exclusive scan, element **b[i]** is what element **b[i-1]** would have been in a regular (inclusive) scan, with **b[0]** initialized to 0. The result of this exclusive scan is that, for each block, we now have the accumulated value for the entry just before the block, and thus we can fold that value in to each element in the block. For that latter fold, again each block may be processed in parallel.



Here is an implementation of this algorithm. As with the heated plate example shown previously, we're using one **Task** per block with a **Barrier** to synchronize all tasks across the three stages:

1. Scan each block in parallel.
2. Do the exclusive scan of the upper value from each block serially.
3. Scan the exclusive scan results into the blocks in parallel.

*One important thing to note about this parallelization is that it incurs significant overhead. In the sequential scan implementation, we're executing the combiner function **f** approximately N times, where N is the number of entries. In the parallel implementation, we're executing **f** approximately 2N times. As a result, while the operation may be parallelized, at least two cores are necessary just to break even.*

While there are several ways to enforce the serial nature of the second step, here we're utilizing the **Barrier**'s post-phase action delegate (the complete implementation is available at http://code.msdn.microsoft.com/ParExtSamples):

C#

```
public static void InclusiveScanInPlaceParallel<T>(
    T[] arr, Func<T, T, T> function)
{
    int procCount = Environment.ProcessorCount;
    T[] intermediatePartials = new T[procCount];
    using (var phaseBarrier = new Barrier(procCount,
        _ => ExclusiveScanInPlaceSerial(
             intermediatePartials, function, 0, intermediatePartials.Length)))
    {
        // Compute the size of each range.
        int rangeSize = arr.Length / procCount, nextRangeStart = 0;

        // Create, store, and wait on all of the tasks.
        var tasks = new Task[procCount];
        for (int i = 0; i < procCount; i++, nextRangeStart += rangeSize)
        {
            // Get the range for each task, then start it.
            int rangeNum = i;
            int lowerRangeInclusive = nextRangeStart;
            int upperRangeExclusive = i < procCount - 1 ?
                nextRangeStart + rangeSize : arr.Length;
            tasks[rangeNum] = Task.Factory.StartNew(() =>
            {
                // Phase 1: Prefix scan assigned range.
                InclusiveScanInPlaceSerial(arr, function,
                    lowerRangeInclusive, upperRangeExclusive, 1);
                intermediatePartials[rangeNum] = arr[upperRangeExclusive - 1];

                // Phase 2: One thread should prefix scan intermediaries.
                phaseBarrier.SignalAndWait();

                // Phase 3: Incorporate partials.
                if (rangeNum != 0)
                {
                    for (int j = lowerRangeInclusive;
                         j < upperRangeExclusive;
                         j++)
                    {
                        arr[j] = function(
                            intermediatePartials[rangeNum], arr[j]);
                    }
                }
            });
        }
        Task.WaitAll(tasks);
    }
}
```

This demonstrates that parallelization may be achieved where dependences would otherwise appear to be an obstacle that can’t be mitigated.

DATA SETS OF UNKNOWN SIZE

Most of the examples described in this document thus far center around data sets of known sizes: input arrays, input lists, and so forth. In many real-world problems, however, the size of the data set to be processed isn't known in advance. This may be because the data is coming in from an external source and hasn't all arrived yet, or it may because the data structure storing the data doesn't keep track of the size or doesn't store the data in a manner amenable to the size being relevant. Regardless of the reason, it's important to be able to parallelize such problems.

STREAMING DATA

Data feeds are becoming more and more important in all areas of computing. Whether it's a feed of ticker data from a stock exchange, a sequence of network packets arriving at a machine, or a series of mouse clicks being entered by a user, such data can be an important input to parallel implementations.

Parallel.ForEach and PLINQ are the two constructs discussed thus far that work on data streams, in the form of enumerables. Enumerables, however, are based on a pull-model, such that both **Parallel.ForEach** and PLINQ are handed an enumerable from which they continually "move next" to get the next element. This is seemingly contrary to the nature of streaming data, where it hasn't all arrived yet, and comes in more of a "push" fashion rather than "pull". However, if we think of this pattern as a producer/consumer pattern, where the streaming data is the producer and the **Parallel.ForEach** or PLINQ query is the consumer, a solution from the .NET Framework 4 becomes clear: we can use BlockingCollection. **BlockingCollection**'s **GetConsumingEnumerable** method provides an enumerable that can be supplied to either **Parallel.ForEach** or PLINQ. **ForEach** and PLINQ will both pull data from this enumerable, which will block the consumers until data is available to be processed. Conversely, as streaming data arrives in, that data may be added to the collection so that it may be picked up by the consumers.

C#

```
private BlockingCollection<T> _streamingData = new BlockingCollection<T>();

// Parallel.ForEach
Parallel.ForEach(_streamingData.GetConsumingEnumerable(),
    item => Process(item));

// PLINQ
var q = from item in _streamingData.GetConsumingEnumerable().AsParallel()
        ...
        select item;
```

There are several caveats to be aware of here, both for **Parallel.ForEach** and for PLINQ. Parallel.ForEach and PLINQ work on slightly different threading models in the .NET Framework 4. PLINQ uses a fixed number of threads to execute a query; by default, it uses the number of logical cores in the machine, or it uses the value passed to **WithDegreeOfParallelism** if one was specified. Conversely, **Parallel.ForEach** may use a variable number of threads, based on the **ThreadPool**'s support for injecting and retiring threads over time to best accommodate current workloads. For **Parallel.ForEach**, this means that it's continually monitoring for new threads to be available to it, taking advantage of them when they arrive, and the **ThreadPool** is continually trying out injecting new threads into the pool and retiring threads from the pool to see whether more or fewer threads is beneficial. However, when passing the result of calling **GetConsumingEnumerable** as the data source to **Parallel.ForEach**, the threads used by the loop have the potential to block when the collection becomes empty. And a blocked thread may not be released by **Parallel.ForEach** back to the **ThreadPool** for retirement or other uses. As such, with the code as shown above, if there are any periods of time where the collection is empty, the thread count in the process may steadily grow; this can lead to problematic memory usage and other negative performance implications. To address this, when using **Parallel.ForEach** in a streaming scenario, it's best to place an explicit limit on the number of threads the loop may utilize: this can be done using the **ParallelOptions** type, and specifically its **MaxDegreeOfParallelism** field:

C#

```
var options = new ParallelOptions { MaxDegreeOfParallelism = 4 };
Parallel.ForEach(_streamingData.GetConsumingEnumerable(), options,
    item => Process(item));
```

By adding the bolded code above, the loop is now limited to at most four threads, avoiding the potential for significant thread consumption. Even if the collection is empty for a long period of time, the loop can block only four threads at most.

PLINQ has a different set of caveats. It already uses a fixed number of threads, so thread injection isn't a concern. Rather, in the .NET Framework 4, PLINQ has an internally hardwired limit on the number of data elements in an input data source that are supported: 2^{31}, or 2,147,483,648. This means that PLINQ should only be used for streaming scenarios where fewer than this number of elements will be processed. In most scenarios, this limit should not be problematic. Consider a scenario where each element takes one millisecond to process. It would take at least 24 days at that rate of processing to exhaust this element space. If this limit does prove troublesome, however, in many cases there is a valid mitigation. The limit of 2^{31} elements is per execution of a query, so a potential solution is to simply restart the query after a certain number of items has been fed into the query. Consider a query of the form:

C#

```
_streamingData.GetConsumingEnumerable().AsParallel()
              .OtherOperators()
              .ForAll(x => Process(x));
```

We need two things, a loop around the query so that when one query ends, we start it over again, and an operator that only yields the first N elements from the source, where N is chosen to be less than the 2^{31} limit. LINQ already provides us with the latter, in the form of the **Take** operator. Thus, a workaround would be to rewrite the query as follows:

C#

```
while (true)
{
    _streamingData.GetConsumingEnumerable().Take(2000000000).AsParallel()
                  .OtherOperators()
                  .ForAll(x => Process(x));
}
```

An additional caveat for PLINQ is that not all operators may be used in a streaming query, due to how those operators behave. For example, OrderBy performs a sort and releases items in sorted order. OrderBy has no way of knowing whether the items it has yet to consume from the source are less than the smallest item seem thus far, and thus it can't release any elements until it's seen all elements from the source. With an "infinite" source, as is the case with a streaming input, that will never happen.

PARALLELWHILENOTEMPTY

There's a fairly common pattern that emerges when processing some data structures: the processing of an element yields additional work to be processed. We can see this with the tree-walk example shown earlier in this document: processing one node of the tree may yield additional work to be processed in the form of that node's children. Similarly in processing a graph data structure, processing a node may yield additional work to be processed in the form of that node's neighbors.

Several parallel frameworks include a construct focused on processing these kinds of workloads. No such construct is included in the .NET Framework 4, however it's straightforward to build one. There are a variety of ways such a solution may be coded. Here's one:

C#

```
public static void ParallelWhileNotEmpty<T>(
    IEnumerable<T> initialValues, Action<T, Action<T>> body)
{
    var from = new ConcurrentQueue<T>(initialValues);
    var to = new ConcurrentQueue<T>();

    while (!from.IsEmpty)
    {
        Action<T> addMethod = v => to.Enqueue(v);
        Parallel.ForEach(from, v => body(v, addMethod));
        from = to;
        to = new ConcurrentQueue<T>();
    }
}
```

This solution is based on maintaining two lists of data: the data currently being processed (the "from" queue), and the data generated by the processing of the current data (the "to" queue). The initial values to be processed are stored into the first list. All those values are processed, and any new values they create are added to the second list. Then the second list is processed, and any new values that are produced go into a new list (or alternatively the first list cleared out). Then that list is processed, and... so on. This continues until the next list to be processed has no values available.

With this in place, we can rewrite our tree walk implementation shown previously:

C#

```
static void Walk<T>(Tree<T> root, Action<T> action)
{
    if (root == null) return;
    ParallelWhileNotEmpty(new[] { root }, (item, adder) =>
    {
        if (item.Left != null) adder(item.Left);
        if (item.Right != null) adder(item.Right);
        action(item.Data);
    });
}
```

ANTI-PATTERNS

BLOCKING DEPENDENCIES BETWEEN PARTITIONED CHUNKS

As mentioned, there are several ways **ParallelWhileNotEmpty** could be implemented. Another approach to implementing **ParallelWhileNotEmpty** combines two parallel patterns we've previously seen in this document: counting up and down, and streaming. Simplistically, we can use a **Parallel.ForEach** over a **BlockingCollection**'s **GetConsumingEnumerable**, and allow the body of the **ForEach** to add more items into the **BlockingCollection**. The only thing missing, then, is the ability to mark the collection as complete for adding, which we only want to do after the last element has been processed (since the last element may result in more elements being added). To accomplish that, we keep track of the number of elements remaining to complete processing; every time the adder operation is invoked, we increase this count, and every time we complete the processing of an item we decrease it. If the act of decreasing it causes it to reach 0, we're done, and we can mark the collection as complete for adding so that all threads involved in the **ForEach** will wake up.

C#

```
// WARNING: THIS METHOD HAS A BUG
static void ParallelWhileNotEmpty<T>(
    IEnumerable<T> source, Action<T, Action<T>> body)
{
    var queue = new ConcurrentQueue<T>(source);
    if (queue.IsEmpty) return;

    var remaining = new CountdownEvent(queue.Count);
    var bc = new BlockingCollection<T>(queue);
    Action<T> adder = item => {
        remaining.AddCount();
        bc.Add(item);
    };
    var options = new ParallelOptions {
        MaxDegreeOfParallelism = Environment.ProcessorCount
    };
    Parallel.ForEach(bc.GetConsumingEnumerable(), options, item =>
    {
        try { body(item, adder); }
        finally {
            if (remaining.Signal()) bc.CompleteAdding();
        }
    });
}
```

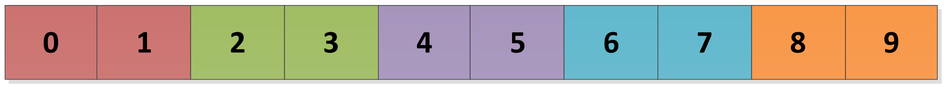
Unfortunately, this implementation has a devious bug in it, one that will likely result in deadlock close to the end of its execution such that **ParallelWhileNotEmpty** will never return. The issue has to do with partitioning. **Parallel.ForEach** uses multiple threads to process the supplied data source (in this case, the result of calling **bc.GetConsumingEnumerable**), and as such the data from that source needs to be dispensed to those threads. By default, **Parallel.ForEach** does this by having its threads take a lock, pull some number of elements from the source, release the lock, and then process those items. This is a performance optimization for the general case, where the number of trips back to the data source and the number of times the lock must be acquired and released is minimized. However, it's then also very important that the processing of elements not have dependencies between them.

Consider a very simple example:

C#

```
var mres = new ManualResetEventSlim();
Parallel.ForEach(Enumerable.Range(0, 10), i =>
{
    if (i == 7) mres.Set();
    else mres.Wait();
});
```

Theoretically, this code could deadlock. All iterations have a dependency on iteration #7 executing, and yet the same thread that executed one of the other iterations may be the one destined to execute #7. To see this, consider a potential partitioning of the input data [0,10), where every thread grabs two elements at a time:



Here, the same thread grabbed both elements 6 and 7. It then processes 6, which immediately blocks waiting for an event that will only be set when 7 is processed, but 7 won't ever be processed, because the thread that would process it is blocked processing 6.

Back to our **ParallelWhileNotEmpty** example, a similar issue exists there but is less obvious. The last element to be processed marks the **BlockingCollection** as complete for adding, which will cause any threads waiting on the empty collection to wake up, aware that no more data will be coming. However, threads are pulling multiple data elements from the source on each go around, and are not processing the elements from that chunk until the chunk contains a certain number of elements. Thus, a thread may grab what turns out to be the last element, but then continues to wait for more elements to arrive before processing it; however, only when that last element is processed will the collection signal to all waiting threads that there won't be any more data, and we have a deadlock.

We can fix this by modifying the partitioning such that every thread only goes for one element at a time. That has the downside of resulting in more overhead per element, since each element will result in a lock being taken, but it has the serious upside of not resulting in deadlock. To control that, we can supply a custom partitioner that provides this functionality. The parallel programming samples for the .NET Framework 4, available for download at http://code.msdn.microsoft.com/ParExtSamples includes a **ChunkPartitioner** capable of yielding a single element at a time. Taking advantage of that, we get the following fixed solution:

C#

```
static void ParallelWhileNotEmpty<T>(
    IEnumerable<T> source, Action<T, Action<T>> body)
{
    var queue = new ConcurrentQueue<T>(source);
    if (queue.IsEmpty) return;

    var remaining = new CountdownEvent(queue.Count);
    var bc = new BlockingCollection<T>(queue);
    Action<T> adder = t => {
        remaining.AddCount();
        bc.Add(t);
    };
    var options = new ParallelOptions {
        MaxDegreeOfParallelism = Environment.ProcessorCount
    };
    Parallel.ForEach(ChunkPartitioner.Create(bc.GetConsumingEnumerable(), 1),
        options, item =>
    {
        try { body(item, adder); }
        finally {
            if (remaining.Signal()) bc.CompleteAdding();
        }
    });
}
```

SPECULATIVE PROCESSING

Speculation is the pattern of doing something that may not be needed in case it actually is needed. This is increasing relevant to parallel computing, where we can take advantage of multiple cores to do more things in advance of their actually being needed. Speculation trades off throughput for reduced latency, by utilizing resources to do more work in case that extra work could pay dividends.

THERE CAN BE ONLY ONE

There are many scenarios where multiple mechanisms may be used to compute a result, but how long each mechanism will take can't be predicted in advance. With serial computing, you're forced to pick one and hope that it's the fastest. With parallel computing, we can theoretically run them all in parallel: once we have a winner, we can stop running the rest of the operations.

We can encapsulate this functionality into a **SpeculativeInvoke** operation. **SpeculativeInvoke** will take a set of functions to be executed, and will start executing them in parallel until at least one returns.

C#
```
public static T SpeculativeInvoke<T>(params Func<T>[] functions);
```

As mentioned earlier in the section on parallel loops, it's possible to implement **Invoke** in terms of **ForEach**… we can do the same here for **SpeculativeInvoke**:

C#
```
public static T SpeculativeInvoke<T>(params Func<T>[] functions)
{
    return SpeculativeForEach(functions, function => function());
}
```

Now all we need is a **SpeculativeForEach**.

SPECULATIVEFOREACH USING PARALLEL.FOREACH

With **ForEach**, the goal is to process every item. With **SpeculativeForEach**, the goal is to get just one result, executing as many items as we can in parallel in order to get just one to return.

C#
```
public static TResult SpeculativeForEach<TSource, TResult>(
    IEnumerable<TSource> source,
    Func<TSource, TResult> body)
{
    object result = null;
    Parallel.ForEach(source, (item, loopState) =>
    {
        result = body(item);
        loopState.Stop();
    });
    return (TResult)result;
}
```

We take advantage of **Parallel.ForEach**'s support for breaking out of a loop early, using **ParallelLoopState.Stop**. This tells the loop to try not to start any additional iterations. When we get a result from an iteration, we store it, request that the loop stop as soon as possible, and when the loop is over, return the result. A **SpeculativeParallelFor** could be implemented in a very similar manner.

> *Note that we store the result as an object, rather than as a **TResult**. This is to accommodate value types. With multiple iterations executing in parallel, it's possible that multiple iterations may try to write out a result concurrently. With reference types, this isn't a problem, as the CLR ensures that all of the data in a reference is written atomically. But with value types, we could potentially experience "torn writes," where portions of the results from multiple iterations get written, resulting in an incorrect result.*

As noted, when an iteration completes it does not terminate other currently running iterations, it only works to prevent additional iterations from starting. If we want to update the implementation to also make it possible to cancel currently running iterations, we can take advantage of the .NET Framework 4 **CancellationToken** type. The idea is that we'll pass a **CancellationToken** into all functions, and the functions themselves may monitor for cancellation, breaking out early if cancellation was experienced.

C#

```csharp
public static TResult SpeculativeForEach<TSource, TResult>(
    IEnumerable<TSource> source,
    Func<TSource, CancellationToken, TResult> body)
{
    var cts = new CancellationTokenSource();
    object result = null;
    Parallel.ForEach(source, (item, loopState) =>
    {
        try
        {
            result = body(item, cts.Token);
            loopState.Stop();
            cts.Cancel();
        }
        catch (OperationCanceledException) { }
    });
    return (TResult)result;
}
```

SPECULATIVEFOREACH USING PLINQ

We can also achieve this kind of speculative processing utilizing PLINQ. The goal of **SpeculativeForEach** is to *select* the result of the *first* function to complete, an operation which maps very nicely to PLINQ's **Select** and **First** operators. We can thus re-implement **SpeculativeForEach** with very little PLINQ-based code:

C#

```csharp
public static TResult SpeculativeForEach<TSource, TResult>(
    IEnumerable<TSource> source, Func<TSource, TResult> body)
{
    if (body == null) throw new ArgumentNullException("body");
    return source.AsParallel().Select(i => body(i)).First();
}
```

FOR THE FUTURE

The other large classification of speculative processing is around anticipation: an application can anticipate a need, and do some computation based on that guess. Prefetching, common in hardware and operating systems, is an example of this. Based on past experience and heuristics, the system anticipates that the program is going to need a particular resource and thus preloads that resource so that it's available by the time it's needed. If the system guessed correctly, the end result is improved perceived performance.

Task<TResult> in the .NET Framework 4 makes it very straightforward to implement this kind of logic. When the system anticipates a particular computation's result may be needed, it launches a **Task<TResult>** to compute the result.

C#

```
var cts = new CancellationTokenSource();
Task<int> dataForThefuture = Task.Factory.StartNew(
    () => ComputeSomeResult(), cts.Token);
```

If it turns out that result is not needed, the task may be canceled.

C#

```
// Cancel it and make sure we are made aware of any exceptions
// that occurred.
cts.Cancel();
dataForTheFuture.ContinueWith(t => LogException(dataForTheFuture),
    TaskContinuationOptions.OnlyOnFaulted);
```

If it turns out it is needed, its **Result** may be retrieved.

C#

```
// This will return the value immediately if the Task has already
// completed, or will wait for the result to be available if it's
// not yet completed.
int result = dataForTheFuture.Result;
```

LAZINESS

Programming is one of few professional areas where laziness is heralded. As we write software, we look for ways to improve performance, or at least to improve perceived performance, and laziness helps in both of these regards.

Lazy evaluation is all about delaying a computation such that it's not evaluated until it's needed. In doing so, we may actually get away with never evaluating it at all, since it may never be needed. And other times, we can make the cost of evaluating lots of computations "pay-for-play" by only doing those computations when they're needed and not before. (In a sense, this is the opposite of speculative computing, where we may start computations asynchronously as soon as we think they may be needed, in order to ensure the results are available if they're needed.)

Lazy evaluation is not something at all specific to parallel computing. LINQ is heavily based on a lazy evaluation model, where queries aren't executed until **MoveNext** is called on an enumerator for the query. Many types lazily-load data, or lazily initialize properties. Where parallelization comes into play is in making it possible for multiple threads to access lazily-evaluated data in a thread-safe manner.

ENCAPSULATING LAZINESS

Consider the extremely common pattern for lazily-initializing some property on a type:

C#

```
public class MyLazy<T> where T : class
{
    private T _value;

    public T Value
    {
        get
        {
            if (_value == null) _value = Compute();
            return _value;
        }
    }

    private static T Compute() { /*...*/ }
}
```

Here, the **_value** field needs to be initialized to the result of some function **Compute. _value** could have been initialized in the constructor of **MyLazy<T>**, but that would have forced the user to incur the cost of computing **_value**, even if the **Value** property is never accessed. Instead, the Value property's **get** accessor checks to see whether **_value** has been initialized, and if it hasn't, initializes it before returning **_value**. The initialization check happens by comparing **_value** to **null**, hence the class restriction on **T**, since a struct may never be **null**.

Unfortunately, this pattern breaks down if the **Value** property may be accessed from multiple threads concurrently. There are several common patterns for dealing with this predicament. The first is through locking:

C#

```
public class MyLazy<T> where T : class
{
    private object _syncObj = new object();
    private T _value;

    public T Value
    {
        get
        {
            lock (_syncObj)
            {
                if (_value == null) _value = Compute();
                return _value;
            }
        }
    }

    private static T Compute() { /*...*/ }
}
```

Now, the **Value** property is thread-safe, such that only one thread at a time will execute the body of the **get** accessor. Unfortunately, we also now force every caller of **Value** to accept the cost of taking a lock, even if **Value** has already previously been initialized. To work around that, there's the classic double-checked locking pattern:

C#

```
public class MyLazy<T> where T : class
{
    private object _syncObj = new object();
    private volatile T _value;

    public T Value
    {
        get
        {
            if (_value == null)
            {
                lock(_syncObj)
                {
                    if (_value == null) _value = Compute();
                }
            }
            return _value;
    }

    private static T Compute() { /*...*/ }
}
```

This is starting to get complicated, with much more code having to be written than was necessary for the initial non-thread-safe version. Moreover, we haven't factored in the complications of exception handling, supporting value types in addition to reference types (and having to deal with potential "torn reads" and "torn writes"), cases where **null** is a valid value, and more. To simplify this, all aspects of the pattern, including the synchronization to ensure thread-safety, have been codified into the new .NET Framework 4 **System.Lazy<T>** type. We can re-write the code using **Lazy<T>** as follows:

C#

```
public class MyLazy<T>
{
    private Lazy<T> _value = new Lazy<T>(Compute);

    public T Value { get { return _value.Value; } }

    private static T Compute() { /*...*/ }
}
```

Lazy<T> supports the most common form of thread-safe initialization through a simple-to-use interface. If more control is needed, the static methods on **System.Threading.LazyInitializer** may be employed.

The double-checked locking pattern supported by **Lazy<T>** is also supported by **LazyInitializer**, but through a single static method:

C#

```
public static T EnsureInitialized<T>(
    ref T target, ref bool initialized,
    ref object syncLock,
    Func<T> valueFactory);
```

This overload allows the developer to specify the target reference to be initialized as well as a Boolean value that signifies whether initialization has been completed. It also allows the developer to explicitly specify the monitor object to be used for synchronization.

> *Being able to explicitly specify the synchronization object allows multiple initialization routines and fields to be protected by the same lock.*

We can use this method to re-implement our previous examples as follows:

C#

```
public class MyLazy<T> where T : class
{
    private object _syncObj = new object();
    private bool _initialized;
    private T _value;

    public T Value
    {
        get
        {
            return LazyInitializer.EnsureInitialized(
                ref _value, ref _initialized, ref _syncObj, Compute);
        }
    }

    private static T Compute() { /*...*/ }
}
```

This is not the only pattern supported by **LazyInitializer**, however. Another less-common thread-safe initialization pattern is based on the principle that the initialization function is itself thread-safe, and thus it's okay for it to be executed concurrently with itself. Given that property, we no longer need to use a lock to ensure that only one thread at a time executes the initialization function. However, we still need to maintain the invariant that the value being initialized is only initialized once. As such, while the initialization function may be run multiple times concurrently in the case of multiple threads racing to initialize the value, one and only one of the resulting values must be published for all threads to see. If we were writing such code manually, it might look as follows:

C#

```
public class MyLazy<T> where T : class
{
    private volatile T _value;

    public T Value
    {
        get
        {
            if (_value == null)
            {
                T temp = Compute();
                Interlocked.CompareExchange(ref _value, temp, null);
            }
            return _value;
        }
    }

    private static T Compute() { /*...*/ }
}
```

LazyInitializer provides an overload to support this pattern as well:

C#

```
public static T EnsureInitialized<T>(
    ref T target, Func<T> valueFactory) where T : class;
```

With this method, we can re-implement the same example as follows:

C#

```
public class MyLazy<T> where T : class
{
    private T _value;

    public T Value
    {
        get
        {
            return LazyInitializer.EnsureInitialized(ref _value, Compute);
        }
    }

    private static T Compute() { /*...*/ }
}
```

It’s worth noting that in these cases, if the **Compute** function returns **null**, **_value** will be set to **null**, which is indistinguishable from **Compute** never having been run, and as a result the next time **Value**’s get accessor is invoked, **Compute** will be executed again.

ASYNCHRONOUS LAZINESS

Another common pattern centers around a need to lazily-initialize data asynchronously and to receive notification when the initialization has completed. This can be accomplished by marrying two types we've already seen: **Lazy<T>** and **Task<TResult>**.

Consider an initialization routine:

C#
```
T Compute();
```

We can create a **Lazy<T>** to provide the result of this function:

C#
```
Lazy<T> data = new Lazy<T>(Compute);
```

However, now when we access **data.Value**, we're blocked waiting for the **Compute** operation to complete. Instead, for asynchronous lazy initialization, we'd like to delay the computation until we know we'll need it, but once we do we also don't want to block waiting for it to complete. That latter portion should hint at using a **Task<TResult>**:

C#
```
Task<T> data = Task<T>.Factory.StartNew(Compute);
```

Combining the two, we can use a **Lazy<Task<T>>** to get both the delayed behavior and the asynchronous behavior:

C#
```
var data = new Lazy<Task<T>>(() => Task<T>.Factory.StartNew(Compute));
```

Now when we access **data.Value**, we get back a **Task<T>** that represents the running of **Compute**. No matter how many times we access **data.Value**, we'll always get back the same **Task**, even if accessed from multiple threads concurrently, thanks to support for the thread-safety patterns built into **Lazy<T>**. This means that only one **Task<T>** will be launched for **Compute**. Moreover, we can now use this result as we would any other **Task<T>**, including registering continuations with it (using **ContinueWith**) in order to be notified when the computation is complete:

C#
```
data.Value.ContinueWith(t => UseResult(t.Result));
```

This approach can also be combined with multi-task continuations to lazily-initialize multiple items, and to only do work with those items when they've all completed initialization:

C#
```
private Lazy<Task<T>> _data1 = new Lazy<Task<T>>(() =>
    Task<T>.Factory.StartNew(Compute1));
private Lazy<Task<T>> _data2 = new Lazy<Task<T>>(() =>
    Task<T>.Factory.StartNew(Compute2));
private Lazy<Task<T>> _data3 = new Lazy<Task<T>>(() =>
    Task<T>.Factory.StartNew(Compute3));
//...
```

```
Task.Factory.ContinueWhenAll(
    new [] { _data1.Value, _data2.Value, _data3.Value },
    tasks => UseResults(_data1.Value.Result, _data2.Value.Result,
                        _data3.Value.Result));
```

Such laziness is also useful for certain patterns of caching, where we want to maintain a cache of these lazily-initialized values. Consider a non-thread-safe cache like the following:

C#

```
public class Cache<TKey, TValue>
{
    private readonly Func<TKey, TValue> _valueFactory;
    private readonly Dictionary<TKey, TValue> _map;

    public Cache(Func<TKey, TValue> valueFactory)
    {
        if (valueFactory == null) throw new ArgumentNullException("loader");
        _valueFactory = valueFactory;
        _map = new Dictionary<TKey, TValue>();
    }

    public TValue GetValue(TKey key)
    {
        if (key == null) throw new ArgumentNullException("key");

        TValue val;
        if (!_map.TryGetValue(key, out val))
        {
            val = _valueFactory(key);
            _map.Add(key, val);
        }
        return val;
    }
}
```

The cache is initialized with a function that produces a value based on a key supplied to it. Whenever the value of a key is requested from the cache, the cache returns the cached value for the key if one is available in the internal dictionary, or it generates a new value using the cache's **_valueFactory** function, stores that value for later, and returns it.

We now want an asynchronous version of this cache. Just like with our asynchronous laziness functionality, we can represent this as a **Task<TValue>** rather than simply as a **TValue**. Multiple threads will be accessing the cache concurrently, so we want to use a **ConcurrentDictionary<TKey,TValue>** instead of a **Dictionary<TKey,TValue>** (**ConcurrentDictionary<>** is a new map type available in the .NET Framework 4, supporting multiple readers and writers concurrently without corrupting the data structure).

C#

```
public class AsyncCache<TKey, TValue>
{
    private readonly Func<TKey, Task<TValue>> _valueFactory;
    private readonly ConcurrentDictionary<TKey, Lazy<Task<TValue>>> _map;

    public AsyncCache(Func<TKey, Task<TValue>> valueFactory)
    {
        if (valueFactory == null) throw new ArgumentNullException("loader");
        _valueFactory = valueFactory;
        _map = new ConcurrentDictionary<TKey, Lazy<Task<TValue>>>();
    }

    public Task<TValue> GetValue(TKey key)
    {
        if (key == null) throw new ArgumentNullException("key");
        var value = new Lazy<Task<TValue>>(() => _valueFactory(key));
        return _map.GetOrAdd(key, value).Value;
    }
}
```

The function now returns a **Task<TValue>** instead of just **TValue**, and the dictionary stores **Lazy<Task<TValue>>** rather than just **TValue**. The latter is done so that if multiple threads request the value for the same key concurrently, only one task for that value will be generated.

Note the **GetOrAdd** method on **ConcurrentDictionary**. This method was added in recognition of a very common coding pattern with dictionaries, exemplified in the earlier synchronous cache example. It's quite common to want to check a dictionary for a value, returning that value if it could be found, otherwise creating a new value, adding it, and returning it, as exemplified in the following example:

C#

```
public static TValue GetOrAdd<TKey, TValue>(
    this Dictionary<TKey, TValue> dictionary,
    TKey key, Func<TKey, TValue> valueFactory)
{
    TValue value;
    if (!dictionary.TryGetValue(key, out value))
    {
        value = valueFactory(key);
        dictionary.Add(key, value);
    }
    return value;
}
```

This pattern has been codified into **ConcurrentDictionary** in a thread-safe manner in the form of the **GetOrAdd** method. Similarly, another coding pattern that's quite common with dictionaries is around checking for an existing value in the dictionary, updating that value if it could be found or adding a new one if it couldn't.

C#

```
public static TValue AddOrUpdate<TKey, TValue>(
    this Dictionary<TKey, TValue> dictionary,
    TKey key,
    Func<TKey, TValue> addValueFactory,
    Func<TKey, TValue, TValue> updateValueFactory)
{
    TValue value;
    return dictionary.TryGetValue(key, out value) ?
        updateValueFactory(key, value) : addValueFactory(key);
}
```

This pattern has been codified into **ConcurrentDictionary** in a thread-safe manner in the form of the **AddOrUpdate** method.

SHARED STATE

Dealing with shared state is arguably the most difficult aspect of building parallel applications and is one of the main sources of both correctness and performance problems. There are several ways of dealing with shared state, including synchronization, immutability, and isolation. With synchronization, the shared state is protected by mechanisms of mutual exclusion to ensure that the data remains consistent in the face of multiple threads accessing and modifying it. With immutability, shared data is read-only, and without being modified, there's no danger in sharing it. With isolation, sharing is avoided, with threads utilizing their own isolated state that's not available to other threads.

ISOLATION & THREAD-LOCAL STATE

Thread-local state is a very common mechanism for supporting isolation, and there are several reasons why you might want to use thread-local state. One is to pass information out-of-band between stack frames. For example, the **System.Transactions.TransactionScope** class is used to register some ambient information for the current thread, such that operations (for example, commands against a database) can automatically enlist in the ambient transaction. Another use of thread-local state is to maintain a cache of data per thread rather than having to synchronize on a shared data source. For example, if multiple threads need random numbers, each thread can maintain its own **Random** instance, accessing it freely and without concern for another thread accessing it concurrently; an alternative would be to share a single **Random** instance, locking on access to it.

Thread-local state is exposed in the .NET Framework 4 in three different ways. The first way, and the most efficient, is through the **ThreadStaticAttribute**. By applying **[ThreadStatic]** to a static field of a type, that field becomes a thread-local static, meaning that rather than having one field of storage per AppDomain (as you would with a traditional static), there's one field of storage per thread per AppDomain.

Hearkening back to our randomness example, you could imagine trying to initialize a **ThreadStatic Random** field as follows:

C#

```
[ThreadStatic]
static Random _rand = new Random(); // WARNING: buggy

static int GetRandomNumber()
{
    return _rand.Next();
}
```

Unfortunately, this won't work as expected. The C# and Visual Basic compilers extract initialization for static/Shared members into a static/Shared constructor for the containing type, and a static constructor is only run once. As such, this initialization code will only be executed for one thread in the system, leaving the rest of the threads with **_rand** initialized to **null**. To account for this, we need to check prior to accessing **_rand** to ensure it's been initialized, invoking the initialization code on each access if it hasn't been:

C#

```
[ThreadStatic]
static Random _rand;

static int GetRandomNumber()
```

```
{
    if (_rand == null) _rand = new Random();
    return _rand.Next();
}
```

Any thread may now call **GetRandomNumber**, and any number of threads may do so concurrently; each will end up utilizing its own instance of **Random**. Another issue with this approach is that, unfortunately, **[ThreadStatic]** may only be used with statics. Applying this attribute to an instance member is a no-op, leaving us in search of another mechanism for supporting per-thread, per-instance state.

Since the original release of the .NET Framework, thread-local storage has been supported in a more general form through the **Thread.GetData** and **Thread.SetData** static methods. The **Thread.AllocateDataSlot** and **Thread.AllocateNamedDataSlot** static methods may be used to create a new **LocalDataStoreSlot**, representing a single object of storage. The **GetData** and **SetData** methods can then be used to get and set that object for the current thread. Re-implementing our previous **Random** example could be done as follows:

C#

```
static LocalDataStoreSlot _randSlot = Thread.AllocateDataSlot();

static int GetRandomNumber()
{
    Random rand = (Random)Thread.GetData(_randSlot);
    if (rand == null)
    {
        rand = new Random();
        Thread.SetData(_randSlot, rand);
    }
    return rand.Next();
}
```

However, since our thread-local storage is now represented as an object (**LocalDataStoreSlot**) rather than as a static field, we can use this mechanism to achieve the desired per-thread, per-instance data:

C#

```
public class MyType
{
    private LocalDataStoreSlot _rand = Thread.AllocateDataSlot();

    public int GetRandomNumber()
    {
        Random r = (Random)Thread.GetData(_rand);
        if (r == null)
        {
            r = new Random();
            Thread.SetData(_rand, r);
        }
        return r.Next();
    }
}
```

While flexible, this approach also has downsides. First, **Thread.GetData** and **Thread.SetData** work with type **Object** rather than with a generic type parameter. In the best case, the data being stored is a reference type, and we only need to cast to retrieve data from a slot, knowing in advance what kind of data is stored in that slot. In the worst case, the data being stored is a value type, forcing an object allocation every time the data is modified, as the value type gets boxed when passed into the **Thread.SetData** method. Another issue is around performance. The **ThreadStaticAttribute** approach has always been significantly faster than the **Thread.GetData**/**SetData** approach, and while both mechanisms have been improved for the .NET Framework 4, the **ThreadStaticAttribute** approach is still an order of magnitude faster. Finally, with **Thread.GetData/SetData**, the reference to the storage and the capability for accessing that storage are separated out into individual APIs, rather than being exposed in a convenient manner that combines them in an object-oriented manner.

To address these shortcomings, the .NET Framework 4 introduces a third thread-local storage mechanism: **ThreadLocal<T>**. **ThreadLocal<T>** addresses the shortcomings outlined:

- **ThreadLocal<T> is generic.** It's **Value** property is typed as **T** and the data is stored in a generic manner. This eliminates the need to cast when accessing the value, and it eliminates the boxing that would otherwise occur if **T** were a value type.
- **The constructor for ThreadLocal<T> optionally accepts a Func<T> delegate.** This delegate can be used to initialize the thread-local value on every accessing thread. This alleviates the need to explicitly check on every access to **ThreadLocal<T>.Value** whether it's been initialized yet.
- **ThreadLocal<T> encapsulates both the data storage and the mechanism for accessing that storage.** This simplifies the pattern of accessing the storage, as all that's required is to utilize the **Value** property.
- **ThreadLocal<T>.Value is fast.** **ThreadLocal<T>** has a sophisticated implementation based on **ThreadStaticAttribute** that makes the Value property more efficient than **Thread.GetData/SetData**.

ThreadLocal<T> is still not as fast as **ThreadStaticAttribute**, so if **ThreadStaticAttribute** fits your needs well and if access to thread-local storage is a bottleneck on your fast path, it should still be your first choice. Additionally, a single instance of **ThreadLocal<T>** consumes a few hundred bytes, so you need to consider how many of these you want active at any one time.

Regardless of what mechanism for thread-local storage you use, if you need thread-local storage for several successive operations, it's best to work on a local copy so as to avoid accessing thread-local storage as much as possible. For example, consider adding two vectors stored in thread-local storage:

C#

```
const int VECTOR_LENGTH = 1000000;
private ThreadLocal<int[]> _vector1 =
    new ThreadLocal<int[]>(() => new int[VECTOR_LENGTH]);
private ThreadLocal<int[]> _vector2 =
    new ThreadLocal<int[]>(() => new int[VECTOR_LENGTH]);
// ...

private void DoWork()
{
    for(int i=0; i<VECTOR_LENGTH; i++)
    {
        _vector2.Value[i] += _vector1.Value[i];
    }
}
```

While the cost of accessing **ThreadLocal<T>.Value** has been minimized as best as possible in the implementation, it still has a non-negligible cost (the same is true for accessing **ThreadStaticAttribute**). As such, it's much better to rewrite this code as follows:

C#

```
private void DoWork()
{
    int [] vector1 = _vector1.Value;
    int [] vector2 = _vector2.Value;
    for(int i=0; i<VECTOR_LENGTH; i++)
    {
        vector2[i] += vector1[i];
    }
    _vector2.Value = vector2;
}
```

Returning now to our previous example of using a thread-local Random, we can take advantage of ThreadLocal<T> to implement this support in a much more concise manner:

C#

```
public class MyType
{
    private ThreadLocal<Random> _rand =
        new ThreadLocal<Random>(() => new Random());

    public int GetRandomNumber() { return _rand.Value.Next(); }
}
```

> *Earlier in this document, it was mentioned that the **ConcurrentBag<T>** data structure maintains a list of instances of **T** per thread. This is achieved internally using **ThreadLocal<>**.*

SYNCHRONIZATION

In most explicitly-threaded parallel applications, no matter how much we try, we end up with some amount of shared state. Accessing shared state from multiple threads concurrently requires that either that shared state be immutable or that the application utilize synchronization to ensure the consistency of the data.

RELIABLE LOCK ACQUISITION

By far, the most prevalent pattern for synchronization in the .NET Framework is in usage of the **lock** keyword in C# and the **SyncLock** keyword in Visual Basic. Compiling down to usage of **Monitor** under the covers, this pattern manifests as follows:

C#

```
lock (someObject)
{
    // ... critical region of code
}
```

This code ensures that the work inside the critical region is executed by at most one thread at a time. In C# 3.0 and earlier and Visual Basic 9.0 and earlier, the above code was compiled down to approximately the equivalent of the following:

C#

```
var lockObj = someObject;
Monitor.Enter(lockObj);
try
{
    // ... critical region of code
}
finally
{
    Monitor.Exit(lockObj);
}
```

This code ensures that even in the case of exception, the lock is released when the critical region is done. Or at least it’s meant to. A problem emerges due to asynchronous exceptions: external influences may cause exceptions to occur on a block of code even if that exception is not explicitly stated in the code. In the extreme case, a thread abort may be injected into a thread between any two instructions, though not within a finally block except in extreme conditions. If such an abort occurred after the call to **Monitor.Enter** but prior to entering the **try** block, the monitor would never be exited, and the lock would be “leaked.” To help prevent against this, the just-in-time (JIT) compiler ensures that, as long as the call to **Monitor.Enter** is the instruction immediately before the **try** block, no asynchronous exception will be able to sneak in between the two. Unfortunately, it’s not always the case that these instructions are immediate neighbors. For example, in debug builds, the compiler uses **nop** instructions to support setting breakpoints in places that breakpoints would not otherwise be feasible. Worse, it’s often the case that developers want to enter a lock conditionally, such as with a timeout, and in such cases there are typically branching instructions between the call and entering the **try** block:

C#

```
if (Monitor.TryEnter(someObject, 1000))
{
    try
    {
        // ... critical region of code
    }
    finally
    {
        Monitor.Exit(someObject);
    }
}
else { /*...*/ }
```

To address this, in the .NET Framework 4 new overloads of **Monitor.Enter** (and **Monitor.TryEnter**) have been added, supporting a new pattern of reliable lock acquisition and release:

C#

```
public static void Enter(object obj, ref bool lockTaken);
```

This overload guarantees that the **lockTaken** parameter is initialized by the time Enter returns, even in the face of asynchronous exceptions. This leads to the following new, reliable pattern for entering a lock:

C#

```
bool lockTaken = false;
try
{
    Monitor.Enter(someObject, ref lockTaken);
    // ... critical region of code
}
finally
{
    if (lockTaken) Monitor.Exit(someObject);
}
```

In fact, code similar to this is what the C# and Visual Basic compilers output in the .NET Framework 4 for the **lock** and **SyncLock** construct. This pattern applies equally to **TryEnter**, with only a slight modification:

C#

```
bool lockTaken = false;
try
{
    Monitor.TryEnter(someObject, 1000, ref lockTaken);
    if (lockTaken)
    {
         // ... critical region of code
    }
    else { /*...*/ }
}
finally
{
    if (lockTaken) Monitor.Exit(someObject);
}
```

Note that the new **System.Threading.SpinLock** type also follows this new pattern, and in fact provides only the reliable overloads:

C#

```
public struct SpinLock
{
    public void Enter(ref bool lockTaken);
    public void TryEnter(ref bool lockTaken);
    public void TryEnter(TimeSpan timeout, ref bool lockTaken);
    public void TryEnter(int millisecondsTimeout, ref bool lockTaken);
    // ...
}
```

With these methods, **SpinLock** is then typically used as follows:

C#

```
private static SpinLock _lock = new SpinLock(enableThreadOwnerTracking: false);
// ...
bool lockTaken = false;
try
{
    _lock.Enter(ref lockTaken);
    // ... very small critical region here
}
```

```
finally
{
    if (lockTaken) _lock.Exit(useMemoryBarrier: false);
}
```

Alternatively, **SpinLock** may be used with **TryEnter** as follows:

C#

```
bool lockTaken = false;
try
{
    _lock.TryEnter(ref lockTaken);
    if (lockTaken)
    {
        // ... very small critical region here
    }
    else { /*...*/ }
}
finally
{
    if (lockTaken) _lock.Exit(useMemoryBarrier:false);
}
```

The concept of a spin lock is that rather than blocking, it continually iterates through a loop ("spinning"), until the lock is available. This can lead to benefits in some cases, where contention on the lock is very infrequent, and where if there is contention, the lock will be available in very short order. This then allows the application to avoid costly kernel transitions and context switches, instead iterating through a loop a few times. When used at incorrect times, however, spin locks can lead to significant performance degradation in an application.

> *The constructor to **SpinLock** accepts an **enableThreadOwnerTracking** parameter, which default to **true**. This causes the **SpinLock** to keep track of which thread currently owns the lock, and can be useful for debugging purposes. This does, however, have an effect on the lock's behavior when the lock is misused. **SpinLock** is not reentrant, meaning that a thread may only acquire the lock once. If thread holding the lock tries to enter it again, and if **enableThreadOwnerTracking** is **true**, the call to **Enter** will throw an exception. If **enableThreadOwnerTracking** is **false**, however, the call will deadlock, spinning forever.*

In general, if you need a lock, start with **Monitor**. Only if after performance testing do you find that **Monitor** isn't fitting the bill should **SpinLock** be considered. If you do end up using a **SpinLock**, inside the protected region you should avoid blocking or calling anything that may block, trying to acquire another lock, calling into unknown code (including calling virtual methods, interface methods, or delegates), and allocating memory. You should be able to count the number of instructions executed under a spin lock on two hands, with the total amount of CPU utilization in the protected region amounting to only tens of cycles.

MIXING EXCEPTIONS WITH LOCKS

As described, a lot of work has gone into ensuring that locks are properly released, even if exceptions occur within the protected region. This, however, isn't always the best behavior.

Locks are used to make non-atomic sets of actions appear atomic, and that's often needed due to multiple statements making discrete changes to shared state. If an exception occurs inside of a critical region, that exception may leave shared data in an inconsistent state. All of the work we've done to ensure reliable lock release in the face of exceptions now leads to a problem: another thread may acquire the lock and expect state to be consistent, but find that it's not.

In these cases, we have a decision to make: is it better to allow threads to access potentially inconsistent state, or is it better to deadlock (which would be achievable by not releasing the lock, but by "leaking" it instead)? The answer really depends on the case in question.

If you decide that leaking a lock is the best solution, instead of using the aforementioned patterns the following may be employed:

C#

```
Monitor.Enter(someObject);
// ... critical region
Monitor.Exit(someObject);
```

Now if an exception occurs in the critical region, the lock will not be exited, and any other threads that attempt to acquire this lock will deadlock. Of course, due to the reentrancy supported by **Monitor** in the .NET Framework, if this same thread later attempts to enter the lock, it will succeed in doing so.

AVOIDING DEADLOCKS

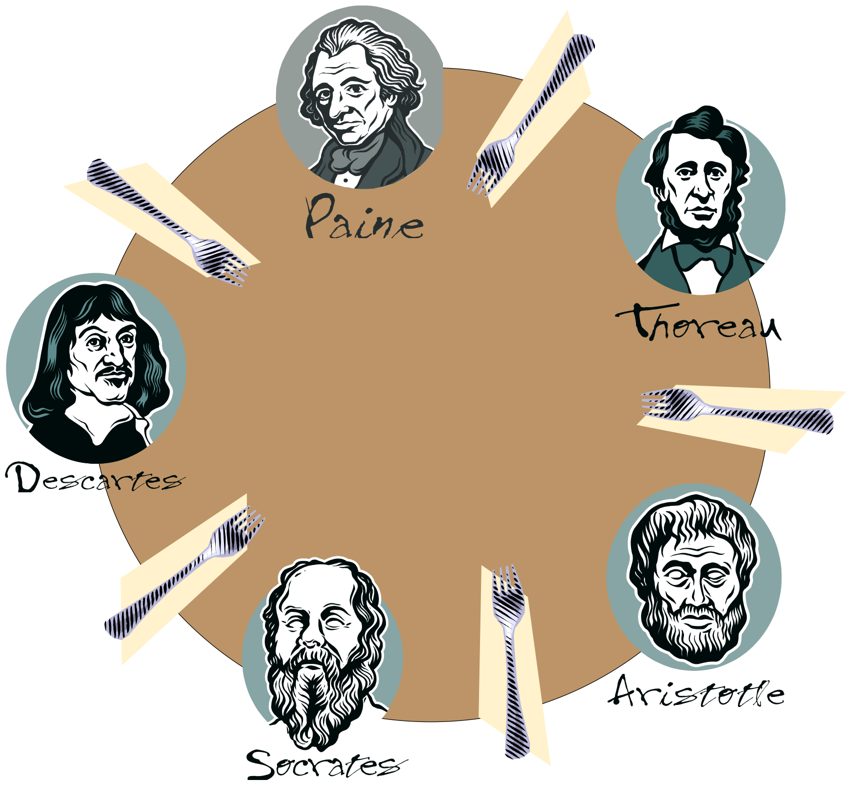
Of all of the problems that may result from incorrect synchronization, deadlocks are one of the most well-known. There are four conditions required for a deadlock to be possible:

1. **Mutual exclusion.** Only a limited number of threads may utilize a resource concurrently.
2. **Hold and wait.** A thread holding a resource may request access to other resources and wait until it gets them.
3. **No preemption.** Resources are released only voluntarily by the thread holding the resource.
4. **Circular wait.** There is a set of {T_1, …, T_N} threads, where T_1 is waiting for a resource held by T_2, T_2 is waiting for a resource held by T_3, and so forth, up through T_N waiting for a resource held by T_1.

If any one of these conditions doesn't hold, deadlock isn't possible. Thus, in order to avoid deadlock, we need to ensure that we avoid at least one of these. The most common and actionable condition to avoid in real-world code is #4, circular waits, and we can attack this condition in a variety of ways. One approach involves detecting that a cycle is about to occur. We can maintain a store of what threads hold what locks, and if a thread makes an attempt to acquire a lock that would lead to a cycle, we can prevent it from doing so; an example of this graph analysis is codified in the ".NET Matters: Deadlock Monitor" article at http://msdn.microsoft.com/en-us/magazine/cc163352.aspx. There is another example in the article "No More Hangs: Advanced Techniques To Avoid And Detect Deadlocks In .NET Apps" by Joe Duffy at http://msdn.microsoft.com/en-us/magazine/cc163618.aspx. That same article by Joe Duffy also includes an example of another approach: lock leveling. In lock leveling, locks are assigned numerical values, and the system tracks the smallest value lock held by a thread, only allowing the thread to acquire locks with smaller values than the smallest value it already holds; this prevents the potential for a cycle.

In some cases, we can avoid cycles simply by sorting the locks utilized in some consistent way, and ensuring that if multiple locks need to be taken, they're taken in sorted order (this is, in effect, a lock leveling scheme). We can see a simple example of this in an implementation of the classic "dining philosophers" problem.

The dining philosophers problem was posited by Tony Hoare, based on previous examples from Edsger Dijkstra in the 1960s. The basic idea is that five philosophers sit around a table. Every philosopher has a plate of pasta, and between every pair of philosophers is a fork. To eat the pasta, a philosopher must pick up and use the forks on both sides of him; thus, if a philosopher's neighbor is eating, the philosopher can't. Philosophers alternate between thinking and eating, typically for random periods of time.



We can represent each fork as a lock, and a philosopher must acquire both locks in order to eat. This would result in a solution like the following:

C#

```
// WARNING: THIS METHOD HAS A BUG
const int NUM_PHILOSOPHERS = 5;
object[] forks = new object[NUM_PHILOSOPHERS];
var philosophers = new Task[NUM_PHILOSOPHERS];
for (int i = 0; i < NUM_PHILOSOPHERS; i++)
{
    int id = i;
    philosophers[i] = Task.Factory.StartNew(() =>
    {
```

```
                var rand = new Random(id);
                while (true)
                {
                    // Think
                    Thread.Sleep(rand.Next(100, 1000));

                    // Get forks
                    object leftFork = forks[id];
                    object rightFork = forks[(id + 1) % NUM_PHILOSOPHERS];
                    Monitor.Enter(leftFork);
                    Monitor.Enter(rightFork);

                    // Eat
                    Thread.Sleep(rand.Next(100, 1000));

                    // Put down forks
                    Monitor.Exit(rightFork);
                    Monitor.Exit(leftFork);
                }
            }, TaskCreationOptions.LongRunning);
        }
        Task.WaitAll(philosophers);
```

Unfortunately, this implementation is problematic. If every philosopher were to pick up his left fork at the same time, all of the forks would be off the table. Each philosopher would then attempt to pick up the right fork and would need to wait indefinitely. This is a classic deadlock, following the exact circular wait condition previously described.

To fix this, we can eliminate the cycle by ensuring that a philosopher first picks up the lower numbered fork and then the higher numbered fork, even if that means picking up the right fork first:

C#

```
while (true)
{
    // Think
    Thread.Sleep(rand.Next(100, 1000));

    // Get forks in sorted order to avoid deadlock
    int firstForkId = id, secondForkId = (id + 1) % NUM_PHILOSOPHERS;
    if (secondForkId < firstForkId) Swap(ref firstForkId, ref secondForkId);
    object firstFork = forks[firstForkId];
    object secondFork = forks[secondForkId];
    Monitor.Enter(firstFork);
    Monitor.Enter(secondFork);

    // Eat
    Thread.Sleep(rand.Next(100, 1000));

    // Put down forks
    Monitor.Exit(secondFork);
    Monitor.Exit(firstFork);
}
```

Another solution is to circumvent the second deadlock requirement, hold and wait, by utilizing the operating system kernel's ability to acquire multiple locks atomically. To accomplish that, we need to forego usage of

Monitor, and instead utilize one of the .NET Framework synchronization primitives derived from **WaitHandle**, such as **Mutex**. When we want to acquire both forks, we can then utilize **WaitHandle.WaitAll** to acquire both forks atomically. Using **WaitAll**, we block until we've acquired both locks, and no other thread will see us holding one lock but not the other.

C#

```
const int NUM_PHILOSOPHERS = 5;
Mutex[] forks = Enumerable.Range(0, NUM_PHILOSOPHERS)
                          .Select(i => new Mutex())
                          .ToArray();
var philosophers = new Task[NUM_PHILOSOPHERS];
for (int i = 0; i < NUM_PHILOSOPHERS; i++)
{
    int id = i;
    philosophers[i] = Task.Factory.StartNew(() =>
    {
        var rand = new Random(id);
        while (true)
        {
            // Think
            Thread.Sleep(rand.Next(100, 1000));

            // Get forks together atomically
            var leftFork = forks[id];
            var rightFork = forks[(id + 1) % NUM_PHILOSOPHERS];
            WaitHandle.WaitAll(new[] { leftFork, rightFork });

            // Eat
            Thread.Sleep(rand.Next(100, 1000));

            // Put down forks; order of release doesn't matter
            leftFork.ReleaseMutex();
            rightFork.ReleaseMutex();
        }
    }, TaskCreationOptions.LongRunning);
}
Task.WaitAll(philosophers);
```

The .NET Framework 4 parallel programming samples at http://code.msdn.microsoft.com/ParExtSamples contain several example implementations of the dining philosophers problem.

ANTI-PATTERNS

LOCK(THIS) AND LOCK(TYPEOF(SOMETYPE))

Especially in code written early in the .NET Framework's lifetime, it was common to see synchronization done in instance members with code such as:

C#

```
void SomeMethod()
{
    lock (this)
    {
        // ... critical region here
    }
}
```

It was also common to see synchronization done in static members with code such as:

C#
```
static void SomeMethod()
{
    lock(typeof(MyType))
    {
        // ... critical region here
    }
}
```

In general, this pattern should be avoided. Good object-oriented design results in implementation details remaining private through non-public state, and yet here, the locks used to protect that state are exposed. With these lock objects then public, it becomes possible for an external entity to accidentally or maliciously interfere with the internal workings of the implementation, as well as make common multithreading problems such as deadlocks more likely. (Additionally, **Type** instances can be domain agile, and a lock on a type in one **AppDomain** may seep into another **AppDomain**, even if the state being protected is isolated within the **AppDomain**.) Instead and in general, non-public (and non-**AppDomain**-agile) objects should be used for locking purposes.

The same guidance applies to **MethodImplAttribute**. The **MethodImplAttribute** accepts a **MethodImplOptions** enumeration value, one of which is **Synchronized**. When applied to a method, this ensures that only one thread at a time may access the attributed member:

C#
```
[MethodImpl(MethodImplOptions.Synchronized)]
void SomeMethod()
{
    // ... critical region here
}
```

However, it does so using the equivalent of the explicit locking code shown previously, with a lock on the instance for instance members and with a lock on the type for static members. As such, this option should be avoided.

READONLY SPINLOCK FIELDS

The readonly keyword informs the compiler that a field should only be updated by the constructor; any attempts to modify the field from elsewhere results in a compiler error. As such, you might be tempted to write code like the following:

C#
```
private readonly SpinLock _lock; // WARNING!
```

Don't do this. Due to the nature of structs and how they interact with the readonly keyword, every access to this **_lock** field will return a copy of the **SpinLock**, rather than the original. As a result, every call to **_lock.Enter** will succeed in acquiring the lock, even if another thread thinks it owns the lock.

For the same reason, don't pass try to pass **SpinLock**s around. In most cases, when you do so, you'll be making a copy of the **SpinLock**. As an example, consider the desire to write an extension method for **SpinLock** that executes a user-provided delegate while holding the lock:

C#

```
// WARNING! DON'T DO THIS.
public static void Execute(this SpinLock sl, Action runWhileHoldingLock)
{
    bool lockWasTaken = false;
    try
    {
        sl.Enter(ref lockWasTaken);
        runWhileHoldingLock();
    }
    finally
    {
        if (lockWasTaken) sl.Exit();
    }
}
```

Theoretically, this code should allow you to write code like:

C#

```
_lock.Execute( () =>
{
    … // will be run while holding the lock
});
```

However, the code is very problematic. The **SpinLock** being targeted by the method will be passed by value, such that the method will execute on a copy of the **SpinLock** rather than the original. To write such a method correctly, you'd need to pass the **SpinLock** into the **Execute** method by reference, and C# doesn't permit an extension method to target a value passed by reference. Fortunately, Visual Basic does, and we could write this extension method correctly as follows:

C#

```
(This extension method cannot be written in C#.)
```

Visual Basic

```
<Extension()>
Public Sub Execute(ByRef sl As SpinLock, ByVal runWhileHoldingLock As Action)
    Dim lockWasTaken As Boolean
    Try
        sl.Enter(lockWasTaken)
        runWhileHoldingLock()
    Finally
        If lockWasTaken Then sl.Exit()
    End Try
End Sub
```

See the blog post at http://blogs.msdn.com/pfxteam/archive/2009/05/07/9592359.aspx for more information about this dangerous phenomenon.

CONCLUSION

Understanding design and coding patterns as they relate to parallelism will help you to find more areas of your application that may be parallelized and will help you to do so efficiently. Knowing and understanding patterns of parallelization will also help you to significantly reduce the number of bugs that manifest in your code. Finally, using the new parallelization support in the .NET Framework 4 which encapsulate these patterns will not only help to reduce the bug count further, but it should help you to dramatically decrease the amount of time and code it takes to get up and running quickly and efficiently.

Now, go forth and parallelize.

Enjoy!

ACKNOWLEDGEMENTS

The author would like to thank the following people for their feedback on drafts of this paper: Donny Amalo, John Bristowe, Tina Burden, David Callahan, Chris Dern, Joe Duffy, Ed Essey, Lisa Feigenbaum, Boby George, Scott Hanselman, Jerry Higgins, Joe Hoag, Luke Hoban, Mike Liddell, Daniela Cristina Manu, Ade Miller, Pooja Nagpal, Jason Olson, Emad Omara, Igor Ostrovsky, Josh Phillips, Danny Shih, Cindy Song, Mike Stall, Herb Sutter, Don Syme, Roy Tan, Ling Wo, and Huseyin Yildiz.

ABOUT THE AUTHOR

Stephen Toub is a Program Manager Lead on the Parallel Computing Platform team at Microsoft, where he spends his days focusing on the next generation of programming models and runtimes for concurrency. Stephen is also a Contributing Editor for MSDN® Magazine, for which he writes the .NET Matters column, and is an avid speaker at conferences such as PDC, TechEd, and DevConnections. Prior to working on the Parallel Computing Platform, Stephen designed and built enterprise applications for companies such as GE, JetBlue, and BankOne. Stephen holds degrees in computer science from Harvard University and New York University.