

From Threads to Thread Pool to Tasks

In the beginning there were threads. Threads are the most rudimentary means of parallelizing applications and distributing asynchronous work; they are the most low-level abstraction available to user-mode programs. Threads offer little in the way of structure and control, and programming threads directly resembles strongly the long-gone days of unstructured programming, before subroutines and objects and agents have gained in popularity.

Consider the following simple task: you are given a large range of natural numbers, and are required to find all the prime numbers in the range and store them in a collection. This is a purely CPU-bound job, and has the appearance of an easily parallelizable one. First, let's write a naïve version of the code that runs on a single CPU thread:

```
//Returns all the prime numbers in the range [start, end)
public static IEnumerable < uint > PrimesInRange(uint start, uint end) {
    List < uint > primes = new List < uint > ();
    for (uint number = start; number < end; ++number) {
        if (IsPrime(number)) {
            primes.Add(number);
        }
    }
    return primes;
}

private static bool IsPrime(uint number) {
    //This is a very inefficient O(n) algorithm, but it will do for our expository purposes
    if (number == 2) return true;
    if (number % 2 == 0) return false;
    for (uint divisor = 3; divisor < number; divisor += 2) {
        if (number % divisor == 0) return false;
    }
    return true;
}
```

Is there anything to improve here? Mayhap the algorithm is so quick that there is nothing to gain by trying to optimize it? Well, for a reasonably large range, such as [100, 200000), the code above runs for several seconds on a modern processor, leaving ample room for optimization.

You may have significant reservations about the efficiency of the algorithm (e.g., there is a trivial optimization that makes it run in $O(\sqrt{n})$ time instead of linear time), but regardless of algorithmic optimality, it seems very likely to yield well to parallelization. After all, discovering whether 4977 is prime is independent of discovering whether 3221 is prime, so an apparently easy way to parallelize the above code is by dividing the range into a number of chunks and creating a separate thread to deal with each chunk (as illustrated in [Figure 6-1](#)). Clearly, we will have to synchronize access to the collection of primes to protect it against corruption by multiple threads. A naïve approach is along the following lines:

```
public static IEnumerable < uint > PrimesInRange(uint start, uint end) {
    List < uint > primes = new List < uint > ();
    uint range = end - start;
    uint numThreads = (uint)Environment.ProcessorCount; //is this a good idea?
    uint chunk = range / numThreads; //hopefully, there is no remainder
    Thread[] threads = new Thread[numThreads];
    for (uint i = 0; i < numThreads; ++i) {
        uint chunkStart = start + i*chunk;
        uint chunkEnd = chunkStart + chunk;
        threads[i] = new Thread(() => {
            for (uint number = chunkStart; number < chunkEnd; ++number) {
                if (IsPrime(number)) {
                    lock(primes) {
                        primes.Add(number);
                    }
                }
            }
        });
        threads[i].Start();
    }
    foreach (Thread thread in threads) {
        thread.Join();
    }
    return primes;
}
```

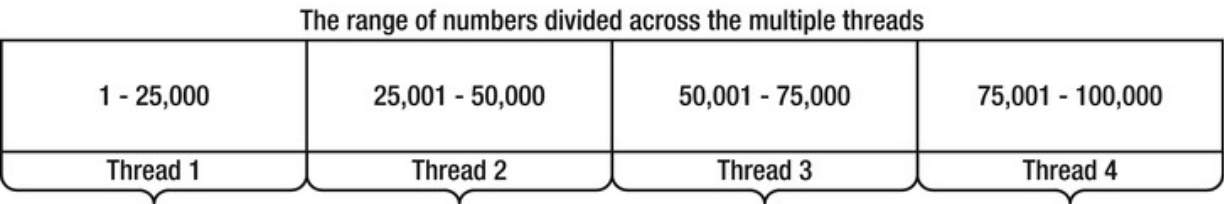


Figure 6-1 . Dividing the range of prime numbers across multiple threads

On an Intel i7 system, the sequential code took ~ 2950 ms on average to traverse the range [100, 200000], and the parallelized version took ~ 950 ms on average to do the same. From a system with 8 CPU cores you expect better results, but this particular strain of i7 processors uses HyperThreading, which means there are only 4 physical cores (each physical core hosts two logical cores). A 4× speedup is more reasonable to expect, and we gained a 3× speedup, which is still non-negligible. However, as the Concurrency Profiler's report in Figures 6-2 and 6-3 shows, some threads finish faster than others, bringing the overall CPU utilization to much lower than 100% (to run the Concurrency Profiler on your applications, consult Chapter 2).

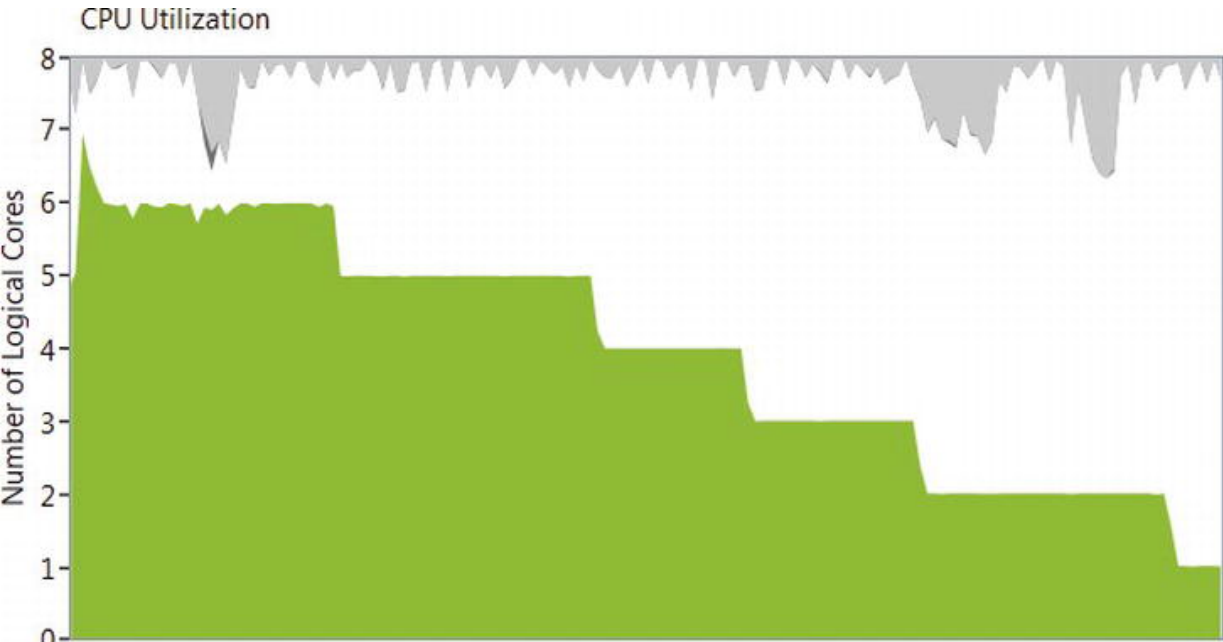


Figure 6-2 . Overall CPU utilization rose to almost 8 logical cores (100%) and then dropped to only one logical core at the end of the run

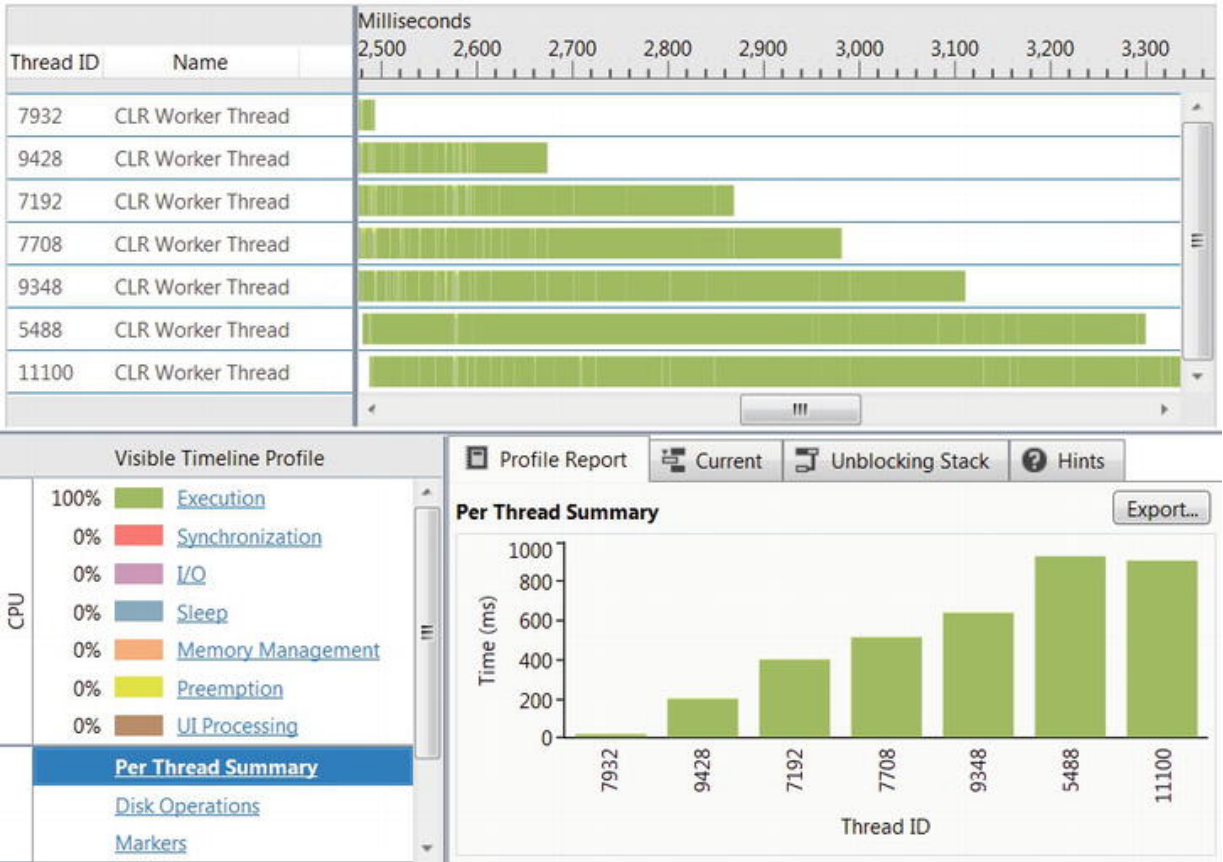


Figure 6-3 . Some threads finished much faster than others. While thread 9428 ran for less than 200ms, thread 5488 ran for more than 800ms

Indeed, this program might run faster than the sequential version (although will not scale linearly), especially if you throw a lot of cores into the mix. This begets several questions, however:

- How many threads are optimal? If the system has eight CPU cores, should we create eight threads?
- How do we make sure that we don't monopolize system resources or create oversubscription? For example, what if there is another thread in our process that needs to calculate prime numbers, and tries to run the same parallelized algorithm as we do?
- How do the threads synchronize access to the resulting collection? Accessing a `List <uint>` from multiple threads is unsafe, and will result in data corruption, as we shall see in a subsequent section. However, taking a lock every time we add a prime number to the collection (which is what the naïve solution above does) will prove to be extremely expensive and throttle our ability to scale the algorithm to a further increasing number of processing cores.
- For a small range of numbers, is it worthwhile to spawn several new threads, or perhaps it would be a better idea to execute the entire operation synchronously on a single thread? (Creating and destroying a thread is cheap on Windows, but not as cheap as finding out whether 20 small numbers are prime or composite.)

- How do we make sure that all the threads have an equal amount of work? Some threads might finish more quickly than others, especially those that operate on smaller numbers. For the range [100, 100000) divided into four equal parts, the thread responsible for the range [100, 25075) will finish more than twice as fast as the thread responsible for the range [75025, 100000), because our primality testing algorithm becomes increasingly slower as it encounters large prime numbers.
- How should we deal with exceptions that might arise from the other threads? In this particular case, it would appear that there are no possible errors to come out of the `IsPrime` method, but in real-world examples the parallelized work could be ridden with potential pitfalls and exceptional conditions. (The CLR's default behavior is to terminate the entire process when a thread fails with an unhandled exception, which is generally a good idea—fail-fast semantics—but won't allow the caller of `PrimesInRange` to deal with the exception at all.)

Good answers to these questions are far from trivial, and developing a framework that allows concurrent work execution without spawning too many threads, that avoids oversubscription and makes sure work is evenly distributed across all threads, that reports errors and results reliably, and that cooperates with other sources of parallelism within the process was precisely the task for the designers of the Task Parallel Library, which we shall deal with next.

From manual thread management, the natural first step was towards thread pools. A thread pool is a component that manages a bunch of threads available for work item execution. Instead of creating a thread to perform a certain task, you queue that task to the thread pool, which selects an available thread and dispatches that task for execution. Thread pools help address some of the problems highlighted above—they mitigate the costs of creating and destroying threads for extremely short tasks, help avoid monopolization of resources and oversubscription by throttling the total number of threads used by the application, and automate decisions pertaining to the optimal number of threads for a given task.

In our particular case, we may decide to break the number range into a significantly larger number of chunks (at the extreme, a chunk per loop iteration) and queue them to the thread pool. An example of this approach for a chunk size of 100 is below.

```
public static IEnumerable < uint > PrimesInRange(uint start, uint end) {
    List < uint > primes = new List < uint > ();
    const uint ChunkSize = 100;
    int completed = 0;
    ManualResetEvent allDone = new ManualResetEvent(initialState: false);
    uint chunks = (end - start) / ChunkSize; //again, this should divide evenly
    for (uint i = 0; i < chunks; ++i) {
        uint chunkStart = start + i*ChunkSize;
        uint chunkEnd = chunkStart + ChunkSize;
        ThreadPool.QueueUserWorkItem(_ => {
            for (uint number = chunkStart; number < chunkEnd; ++number) {
                if (IsPrime(number)) {
                    lock(primes) {
                        primes.Add(number);
                    }
                }
            }
            if (Interlocked.Increment(ref completed) == chunks) {
                allDone.Set();
            }
        });
    }
    allDone.WaitOne();
    return primes;
}
```

This version of the code is significantly more scalable, and executes faster than the previous versions we have considered. It improves upon the ~ 950 ms (for the range [100, 300000)) required for the unsophisticated thread-based version and completes within ~ 800 ms on average (which is almost a 4x speedup compared to the sequential version). What's more, CPU usage is at a consistent level of close to 100%, as the Concurrency Profiler report in [Figure 6-4](#) indicates.

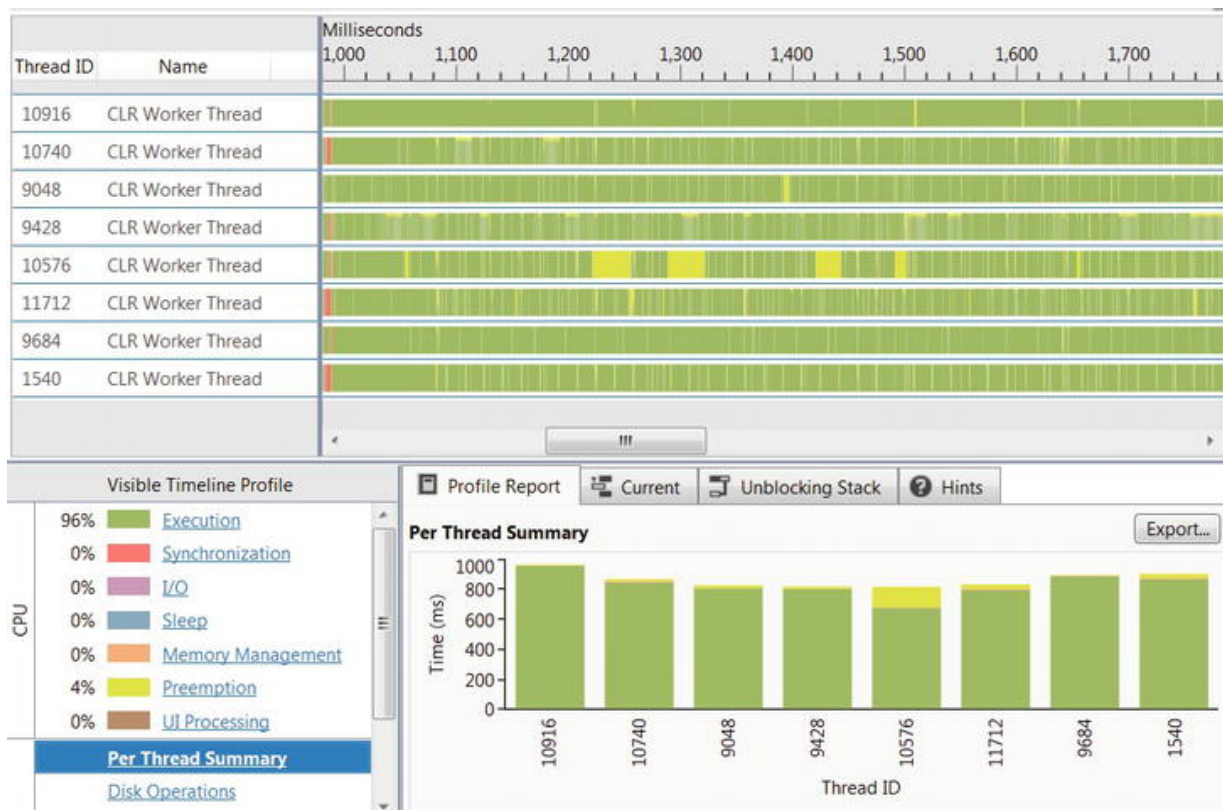


Figure 6-4 . The CLR thread pool used 8 threads (one per logical core) during the program's execution. Each thread ran for almost the entire duration

As of CLR 4.0, the CLR thread pool consists of several cooperating components. When a thread that doesn't belong to the thread pool (such as the application's main thread) dispatches work items to the thread pool, they are enqueued into a global FIFO (first-in-first-out) queue. Each thread pool thread has a local LIFO (last-in-first-out) queue, to which it will enqueue work items created on that thread (see Figure 6-5). When a thread pool thread is looking for work, it begins by consulting its own LIFO queue, and executes work items from it as long as they are available. If a thread's LIFO queue is exhausted, it will attempt *work stealing*—consulting the local queues of other threads and taking work items from them, in FIFO order. Finally, if all the local queues are empty, threads will consult the global (FIFO) queue and execute work items from there.

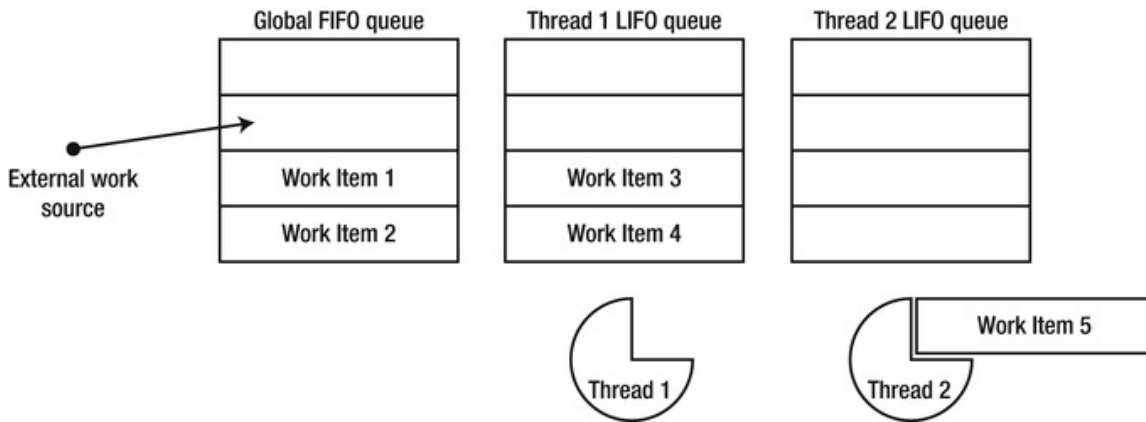


Figure 6-5 . Thread #2 is currently executing work item #5; after completing its execution, it will borrow work from the global FIFO queue. Thread #1 will drain its local queue before tending to any other work

THREAD POOL FIFO AND LIFO SEMANTICS

The reason behind the apparently eccentric FIFO and LIFO queue semantics is the following: when work is enqueued to the global queue, no particular thread has any preference to executing that work, and fairness is the only criterion by which work is selected for execution. This is why FIFO semantics are suitable for the global queue. However, when a thread pool thread queues a work item for execution, it is likely to use the same data and the same instructions as the currently executing work item; that's why it makes sense to enqueue it in a LIFO queue that belongs to the same thread—it will be executed shortly after the currently executing work item, and take advantage of the CPU data and instruction caches.

Furthermore, accessing work items on the thread's local queue requires less synchronization and is less likely to encounter contention from other threads than when accessing the global queue. Similarly, when a thread steals work from another thread's queue, it steals it in FIFO order, so that the LIFO optimization with respect to CPU caches on the original thread's processor is maintained. This thread pool structure is very friendly towards work item hierarchies, where a single work item enqueued into the global queue will spawn off dozens of additional work items and provide work for several thread pool threads.

As with any abstraction, thread pools take some of the granular control over thread lifetime and work item scheduling away from the application developer. Although the CLR thread pool has some control APIs, such as `ThreadPool.SetMinThreads` and `ThreadPool.SetMaxThreads` that control the number of threads, it does not have built-in APIs to control the *priority* of its threads or tasks. More often than not, however, this loss of control is more than compensated by the application's ability to scale automatically on more powerful systems, and the performance gain from not having to create and destroy threads for short-lived tasks.

Work items queued to the thread pool are extremely inept; they do not have state, can't carry exception information, don't have support for asynchronous continuations and cancellation, and don't feature any mechanism for obtaining a result from a task that has completed. The Task Parallel Library in .NET 4.0 introduces *tasks*, which are a powerful abstraction on top of thread pool work items. Tasks are the structured alternative to threads and thread pool work items, much in the same way that objects and subroutines were the structured alternative to `goto`-based assembly language programming.

Task Parallelism

Task parallelism is a paradigm and set of APIs for breaking down a large task into a set of smaller ones, and executing them on multiple threads. The Task Parallel Library (TPL) has first-class APIs for managing millions of tasks simultaneously (through the CLR thread pool). At the heart of the TPL is the `System.Threading.Tasks.Task` class, which represents a task. The `Task` class provides the following capabilities:

- Scheduling work for independent execution on an unspecified thread. (The specific thread to execute a given task is determined by a *task scheduler*; the default task scheduler enqueues tasks to the CLR thread pool, but there are schedulers that send tasks to a particular thread, such as the UI thread.)
- Waiting for a task to complete and obtaining the result of its execution.
- Providing a continuation that should run as soon as the task completes. (This is often called a callback, but we shall use the term *continuation* throughout this chapter.)
- Handling exceptions that arise in a single task or even a hierarchy of tasks on the original thread that scheduled them for execution, or any other thread that is interested in the task results.
- Canceling tasks that haven't started yet, and communicating cancellation requests to tasks that are in the middle of executing.

Because we can think of tasks as a higher-level abstraction on top of threads, we could rewrite the code we had for prime number calculation to use tasks instead of threads. Indeed, it would make the code shorter—at the very least, we wouldn't need the completed task counter and the `ManualResetEvent` object to keep track of task execution. However, as we shall see in the next section, the data parallelism APIs provided by the TPL are even more suitable for parallelizing a loop that finds all prime numbers in a range. Instead, we shall consider a different problem.

There is a well-known recursive comparison-based sorting algorithm called QuickSort that yields itself quite easily to parallelization (and has an average case runtime complexity of $O(n \log(n))$, which is optimal—although scarcely any large framework uses QuickSort to sort anything these days). The QuickSort algorithm proceeds as follows:

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

private static void QuickSort < T > (T[] items, int left, int right) where T : IComparable < T > {
    if (left == right) return;
    int pivot = Partition(items, left, right);
    QuickSort(items, left, pivot);
    QuickSort(items, pivot + 1, right);
}

private static int Partition < T > (T[] items, int left, int right) where T : IComparable < T > {
    int pivotPos = . . .; //often a random index between left and right is used
    T pivotValue = items[pivotPos];
    Swap(ref items[right-1], ref items[pivotPos]);
    int store = left;
    for (int i = left; i < right - 1; ++i) {
```

```

    if (items[i].CompareTo(pivotValue) < 0) {
        Swap(ref items[i], ref items[store]);
        ++store;
    }
}
Swap(ref items[right-1], ref items[store]);
return store;
}

private static void Swap < T > (ref T a, ref T b) {
    T temp = a;
    a = b;
    b = temp;
}

```

Figure 6-6 is an illustration of a single step of the Partition method. The fourth element (whose value is 5) is chosen as the pivot. First, it's moved to the far right of the array. Next, all elements larger than the pivot are propagated towards the right side of the array. Finally, the pivot is positioned such that all elements to its right are strictly larger than it, and all elements to its left are either smaller than or equal to it.

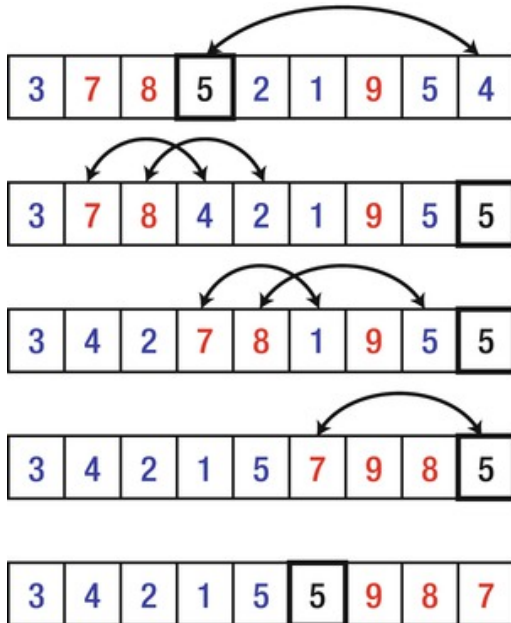


Figure 6-6 . Illustration of a single invocation of the Partition method.

The recursive calls taken by QuickSort at every step must set the parallelization alarm. Sorting the left and right parts of the array are independent tasks, which require no synchronization among them, and the `Task` class is ideal for expressing this. Below is a first attempt at parallelizing QuickSort using `Tasks`:

```

public static void QuickSort < T > (T[] items) where T : IComparable < T > {
    QuickSort(items, 0, items.Length);
}

private static void QuickSort < T > (T[] items, int left, int right) where T : IComparable < T > {
    if (right - left < 2) return;
    int pivot = Partition(items, left, right);
    Task leftTask = Task.Run(() => QuickSort(items, left, pivot));
    Task rightTask = Task.Run(() => QuickSort(items, pivot + 1, right));
    Task.WaitAll(leftTask, rightTask);
}

private static int Partition < T > (T[] items, int left, int right) where T : IComparable < T > {
    //Implementation omitted for brevity
}

```

The `Task.Run` method creates a new task (equivalent to calling `new Task()`) and schedules it for execution (equivalent to the newly created task's `Start` method). The `Task.WaitAll` static method waits for both tasks to complete and then returns. Note that we don't have to deal with specifying how to wait for tasks to complete, nor when to create threads and when to destroy them.

There is a helpful utility method called `Parallel.Invoke`, which executes a set of tasks provided to it and returns when all the tasks have completed. This would allow us to rewrite the core of the QuickSort method body with the following:

```

Parallel.Invoke(
    () => QuickSort(items, left, pivot),
    () => QuickSort(items, pivot + 1, right)
);

```

Regardless of whether we use `Parallel.Invoke` or create tasks manually, if we try to compare this version with the straightforward sequential one, we will find that it runs significantly *slower*, even though it seems to take advantage of all the available processor resources. Indeed, using an array of 1,000,000 random integers, the sequential version ran (on our test system) for ~ 250 ms and the parallelized version took nearly ~ 650 ms to complete on average!

The problem is that parallelism needs to be sufficiently coarse-grained; attempting to parallelize the sorting of a three-element array is futile, because the overhead introduced by creating `Task` objects, scheduling work items to the thread pool, and waiting for them to complete execution overwhelms completely the handful of comparison operations required.

Throttling Parallelism in Recursive Algorithms

How do you propose to throttle the parallelism to prevent this overhead from diminishing any returns from our optimization? There are several viable approaches:

- Use the parallel version as long as the size of the array to be sorted is bigger than a certain threshold (say, 500 items), and switch to the sequential version as soon as it is smaller.
- Use the parallel version as long as the recursion depth is smaller than a certain threshold, and switch to the sequential version as soon as the recursion is very deep. (This option is somewhat inferior to the previous one, unless the pivot is always positioned exactly in the middle of the array.)
- Use the parallel version as long as the number of outstanding tasks (which the method would have to maintain manually) is smaller than a certain threshold, and switch to the sequential version otherwise. (This is the only option when there are no other criteria for limiting parallelism, such as recursion depth or input size.)

Indeed, in the case above, limiting the parallelization for arrays larger than 500 elements produces excellent results on the author's Intel i7 processor, yielding a 4× improvement in execution time compared to the sequential version. The code changes are quite simple, although the threshold should not be hardcoded in a production-quality implementation:

```
private static void QuickSort < T > (T[] items, int left, int right) where T : IComparable < T > {
    if (right - left < 2) return;
    int pivot = Partition(items, left, right);
    if (right - left > 500) {
        Parallel.Invoke(
            () => QuickSort(items, left, pivot),
            () => QuickSort(items, pivot + 1, right)
        );
    } else {
        QuickSort(items, left, pivot);
        QuickSort(items, pivot + 1, right);
    }
}
```

More Examples of Recursive Decomposition

There are many additional algorithms that can be parallelized by applying similar recursive decomposition. In fact, almost all recursive algorithms that split their input into several parts are *designed* to execute independently on each part and combine the results afterwards. Later in this chapter we shall consider examples that do not succumb so easily for parallelization, but first let's take a look at a few that do:

- Strassen's algorithm for matrix multiplication (see http://en.wikipedia.org/wiki/Strassen_algorithm for an overview). This algorithm for matrix multiplication offers better performance than the naïve cubic algorithm we shall see later in this chapter. Strassen's algorithm recursively decomposes a matrix of size $2^n \times 2^n$ into four equal block matrices of size $2^{n-1} \times 2^{n-1}$, and uses a clever trick that relies on *seven* multiplications instead of eight to obtain asymptotic running time of $\sim O(n^{2.807})$. As in the QuickSort example, practical implementations of Strassen's algorithm often fall back to the standard cubic algorithm for small-enough matrices; when parallelizing Strassen's algorithm using its recursive decomposition, it is even more important to put a threshold on parallelization for smaller matrices.
- Fast Fourier Transform (Cooley-Tukey algorithm, see http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm). This algorithm computes the DFT (Discrete Fourier Transform) of a vector of length 2^n using a recursive decomposition of the vector into two vectors of size 2^{n-1} . Parallelizing this computation is fairly easy, but it is again important to be wary of placing a threshold to the parallelization for sufficiently small vectors.
- Graph traversal (Depth-First Search or Breadth-First Search). As we have seen in [Chapter 4](#), the CLR garbage collector traverses a graph in which objects are vertices and references between objects are edges. Graph traversal using DFS or BFS can benefit greatly from parallelization as well as other recursive algorithms we have considered; however, unlike QuickSort or FFT, when parallelizing branches of the graph traversal it is difficult to estimate in advance the amount of work a recursive call represents. This difficulty requires heuristics to decide how the search space should be partitioned to multiple threads: we have seen that the server GC flavor performs this partitioning rather crudely, based on the separate heaps from which each processor allocates objects.

If you are looking for more examples to practice your parallel programming skills, consider also Karatsuba's multiplication algorithm that relies on recursive decomposition to multiply n -digit numbers in $\sim O(n^{1.585})$ operations; merge sort that relies on recursive decomposition for sorting, similarly to QuickSort; and numerous dynamic programming algorithms, which often require advanced tricks to employ memoization in different branches of the parallel computation (we will examine one example later).

Exceptions and Cancellation

We haven't tapped into the full power of the `Task` class yet. Suppose we wanted to handle exceptions that could arise from the recursive invocations of QuickSort down the line, and provide support for canceling the entire sort operation if it hasn't completed yet.

The task execution environment provides the infrastructure for marshaling exceptions that arise within the task back to any thread deemed appropriate to receive it. Suppose that one of the recursive invocations of the QuickSort tasks encountered an exception, perhaps because we didn't consider the array bounds carefully and introduced an off-by-one error to either side of the array. This exception would arise on a thread pool thread, a thread that is not under our explicit control and that does not allow any overarching exception handling behavior. Fortunately, the TPL will catch the exception and store it within the `Task` object for later propagation.

The exception that arose within a task will be rethrown (wrapped in an `AggregateException` object) when the program attempts to wait for the task to complete (using the `Task.Wait` instance method) or to retrieve its result (using the `Task.Result` property). This allows automatic and centralized exception handling within the code that created the task, and does not require manual propagation of errors to a central location and synchronization of error-reporting activities. The following minimal code example demonstrates the exception-handling paradigm in the TPL:

```
int i = 0;
Task < int > divideTask = Task.Run(() => { return 5/i; });
try {
    Console.WriteLine(divideTask.Result); //accessing the Result property eventually throws
} catch (AggregateException ex) {
    foreach (Exception inner in ex.InnerExceptions) {
        Console.WriteLine(inner.Message);
    }
}
```

Note When creating a task from within the body of an existing task, the `TaskCreationOptions.AttachedToParent` enumeration value establishes a relationship between the new child task and its parent task in which it was created. We will see later in this chapter that parent-child relationships between tasks affect cancellation, continuations, and debugging aspects of task execution. As far as exception handling is concerned, however, waiting for the parent task to complete implies waiting for all the child tasks to complete, and any exceptions from the child tasks are propagated to the parent task as well. This is why the TPL throws an `AggregateException` instance, which contains a hierarchy of exceptions that may have arisen from a hierarchy of tasks.

Cancellation of existing work is another matter to consider. Suppose that we have a hierarchy of tasks, such as the hierarchy created by QuickSort if we used the `TaskCreationOptions.AttachedToParent` enumeration value. Even though there may be hundreds of tasks running simultaneously, we might want to provide the user with cancellation semantics, e.g. if the sorted data is no longer required. In other scenarios, cancellation of outstanding work might be an integral part of the task execution. For example, consider a parallelized algorithm that looks up a node in a graph using DFS or BFS. When the desired node is found, the entire hierarchy of tasks performing the lookup should be

recalled.

Cancelling tasks involves the `CancellationTokenSource` and `CancellationToken` types, and is performed cooperatively. In other words, if a task's execution is already underway, it cannot be brutally terminated using TPL's cancellation mechanisms. Cancellation of already executing work requires cooperation from the code executing that work. However, tasks that have not begun executing yet can be cancelled completely without any malignant consequences.

The following code demonstrates a binary tree lookup where each node contains a potentially long array of elements that needs to be linearly traversed; the entire lookup can be cancelled by the caller using the TPL's cancellation mechanisms. On the one hand, unstarted tasks will be cancelled automatically by the TPL; on the other hand, tasks that have already started will periodically monitor their cancellation token for cancellation instructions and stop cooperatively when required.

```
public class TreeNode < T > {
    public TreeNode < T > Left, Right;
    public T[] Data;
}

public static void TreeLookup < T > (
    TreeNode < T > root, Predicate < T > condition, CancellationTokenSource cts) {
    if (root == null) {
        return;
    }
    //Start the recursive tasks, passing to them the cancellation token so that they are
    //cancelled automatically if they haven't started yet and cancellation is requested
    Task.Run(() => TreeLookup(root.Left, condition, cts), cts.Token);
    Task.Run(() => TreeLookup(root.Right, condition, cts), cts.Token);
    foreach (T element in root.Data) {
        if (cts.IsCancellationRequested) break; //abort cooperatively
        if (condition(element)) {
            cts.Cancel(); //cancels all outstanding work
            //Do something with the interesting element
        }
    }
}

//Example of calling code:
CancellationTokenSource cts = new CancellationTokenSource();
Task.Run(() => TreeLookup(treeRoot, i => i % 77 == 0, cts);
//After a while, e.g. if the user is no longer interested in the operation:
cts.Cancel();
```

Inevitably, there will be examples of algorithms where an easier way of expressing parallelism should be desirable. Consider the primality testing example with which we started. We could break the range manually into chunks, create a task for each chunk, and wait for all the tasks to complete. In fact, there is an entire family of algorithms in which there is a range of data to which a certain operation is applied. These algorithms mandate a higher-level abstraction than task parallelism. We now turn to this abstraction.

Data Parallelism

Whilst task parallelism dealt primarily with tasks, data parallelism aims to remove tasks from direct view and replace them by a higher-level abstraction—parallel loops. In other words, the source of parallelism is not the algorithm's *code*, but rather the *data* on which it operates. The Task Parallel Library offers several APIs providing data parallelism.

Parallel.For and Parallel.ForEach

`for` and `foreach` loops are often excellent candidates for parallelization. Indeed, since the dawn of parallel computing, there have been attempts to parallelize such loops automatically. Some attempts have gone the way of language changes or language extensions, such as the OpenMP standard (which introduced directives such as `#pragma omp parallel for` to parallelize `for` loops). The Task Parallel Library provides loop parallelism through explicit APIs, which are nonetheless very close to their language counterparts. These APIs are `Parallel.For` and `Parallel.ForEach`, matching as closely as possible the behavior of `for` and `foreach` loops in the language.

Returning to the example of parallelizing primality testing, we had a loop iterating over a large range of numbers, checking each one for primality and inserting it into a collection, as follows:

```
for (int number = start; number < end; ++number) {
    if (IsPrime(number)) {
        primes.Add(number);
    }
}
```

Converting this code to use `Parallel.For` is almost a mechanical task, although synchronizing access to the collection of primes warrants some caution (and there exist much better approaches, such as aggregation, that we consider later):

```
Parallel.For(start, end, number => {
    if (IsPrime(number)) {
        lock(primes) {
            primes.Add(number);
        }
    }
});
```

By replacing the language-level loop with an API call we gain automatic parallelization of the loop's iterations. Moreover, the `Parallel.For` API is not a straightforward loop that generates a task per iteration, or a task for each hard-coded chunk-sized part of the range. Instead, `Parallel.For` adapts slowly to the execution pace of individual iterations, takes into account the number of tasks currently executing, and prevents too-granular behavior by dividing the iteration range dynamically. Implementing these optimizations manually is not trivial, but you can apply specific customizations (such as controlling the maximum number of concurrently executing tasks) using another overload of `Parallel.For` that takes a `ParallelOptions` object or using a custom partitioner to determine how the iteration ranges should be divided across different tasks.

A similar API works with `foreach` loops, where the data source may not be fully enumerated when the loop begins, and in fact may not be finite. Suppose that we need to download from the Web a set of RSS feeds, specified as an `IEnumerable < string >`. The skeleton of the loop would have the following shape:

```
IEnumerable < string > rssFeeds = . . .;
WebClient webClient = new WebClient();
foreach (string url in rssFeeds) {
    Process(webClient.DownloadString(url));
}
```

```
}
```

This loop can be parallelized by the mechanical transformation where the `foreach` loop is replaced by an API call to `Parallel.ForEach`. Note that the data source (the `rssFeeds` collection) need not be thread-safe, because `Parallel.ForEach` will use synchronization when accessing it from several threads.

```
IEnumerable < string > rssFeeds = . . .; //The data source need
not be thread-safe
WebClient webClient = new WebClient();
Parallel.ForEach(rssFeeds, url => {
    Process(webClient.DownloadString(url));
});
```

Note You can voice a concern about performing an operation on an infinite data source. It turns out, however, that it is quite convenient to begin such an operation and expect to terminate it early when some condition is satisfied. For example, consider an infinite data source such as all the natural numbers (specified in code by a method that returns `IEnumerable < BigInteger >`). We can write and parallelize a loop that looks for a number whose digit sum is 477 but is not divisible by 133. Hopefully, there is such a number, and our loop will terminate.

Parallelizing loops is not as simple as it may seem from the above discussion. There are several "missing" features we need to consider before we fasten this tool assuredly to our belt. For starters, C# loops have the `break` keyword, which can terminate a loop early. How can we terminate a loop that has been parallelized across multiple threads, where we don't even know which iteration is currently executing on threads other than our own?

The `ParallelLoopState` class represents the state of a parallel loop's execution, and allows breaking early from a loop. Here is a simple example:

```
int invitedToParty = 0;
Parallel.ForEach(customers, (customer, loopState) => {
    if (customer.Orders.Count > 10 && customer.City == "Portland") {
        if (Interlocked.Increment(ref invitedToParty) >= 25) {
            loopState.Stop(); //no attempt will be made to execute any additional iterations
        }
    }
});
```

Note that the `Stop` method does not guarantee that the last iteration to execute is the one that called it—iterations that have already started executing will run to completion (unless they poll the `ParallelLoopState.ShouldExitCurrentIteration` property). However, no additional iterations that have been queued will begin to execute.

One of the drawbacks of `ParallelLoopState.Stop` is that it does not guarantee that all iterations up to a certain one have executed. For example, if there are 1,000 customers, it is possible that customers 1–100 have been processed completely, customers 101–110 have not been processed at all, and customer 111 was the last to be processed before `Stop` was called. If you would like to guarantee that all iterations before a certain iteration will have executed (even if they haven't started yet!), you should use the `ParallelLoopState.Break` method instead.

Parallel LINQ (PLINQ)

Possibly the highest level of abstraction for parallel computation is that where you declare: "I want this code to run in parallel", and leave the rest for the framework to implement. This is what Parallel LINQ is about. But first, a short refresher on LINQ is due. LINQ (Language Integrated Query) is a framework and a set of language extensions introduced in C# 3.0 and .NET 3.5, blurring the line between imperative and declarative programming where iterating over data is concerned. For example, the following LINQ query retrieves from a data source called `customers`—which might be an in-memory collection, a database table, or a more exotic origin—the names and ages of the Washington-based customers who have made at least three over \$10 purchases over the last ten months, and prints them to the console:

```
var results = from customer in customers
               where customer.State == "WA"
               let custOrders = (from order in orders
                                where customer.ID == order.ID
                                select new { order.Date, order.Amount })
               where custOrders.Count(co => co.Amount >= 10 &&
                                       co.Date >= DateTime.Now.AddMonths(-10)) >= 3
               select new { customer.Name, customer.Age };
foreach (var result in results) {
    Console.WriteLine("{0} {1}", result.Name, result.Age);
}
```

The primary thing to note here is that most of the query is specified declaratively—quite like an SQL query. It doesn't use loops to filter out objects or to group together objects from different data sources. Often enough, you shouldn't worry about synchronizing different iterations of the query, because most LINQ queries are purely functional and have no side effects—they convert one collection (`IEnumerable < T >`) to another without modifying any additional objects in the process.

To parallelize the execution of the above query, the only code change required is to modify the source collection from a general `IEnumerable < T >` to a `ParallelQuery < T >`. The `AsParallel` extension method takes care of this, and allows the following elegant syntax:

```
var results = from customer in customers.AsParallel()
               where customer.State == "WA"
               let custOrders = (from order in orders
                                where customer.ID == order.ID
                                select new { order.Date, order.Amount })
               where custOrders.Count(co => co.Amount >= 10 &&
                                       co.Date >= DateTime.Now.AddMonths(-10)) >= 3
               select new { customer.Name, customer.Age };
foreach (var result in results) {
    Console.WriteLine("{0} {1}", result.Name, result.Age);
}
```

PLINQ uses a three-stage processing pipeline to execute parallel queries, as illustrated in Figure 6-7. First, PLINQ decides how many threads should be used to parallelize the query's execution. Next, the worker threads retrieve chunks of work from the source collection, ensuring that it is accessed under a lock. Each thread proceeds to execute its work items independently, and the results are queued locally within each thread. Finally, all the local results are buffered into a single result collection, which is polled by a `foreach` loop in the above example.

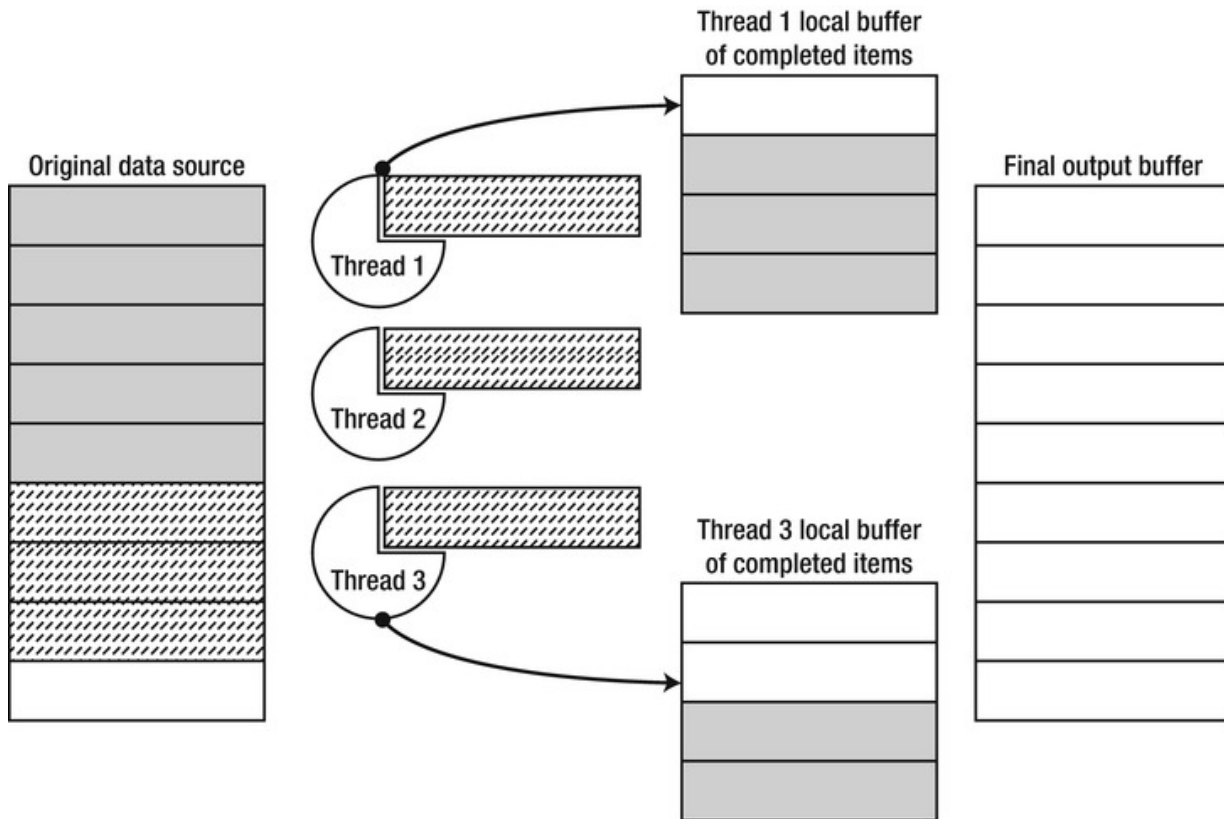


Figure 6-7 . Work item execution in PLINQ. Solid grey work items have been completed and placed in thread-local buffers, from which they are subsequently moved to the final output buffer available to the caller. Dashed work items are currently being executed

The primary advantage of PLINQ compared to `Parallel.ForEach` stems from the fact that PLINQ automatically handles aggregation of temporary processing results locally within each thread that executes the query. When using `Parallel.ForEach` to find prime numbers, we had to access a global collection of prime numbers to aggregate the results (and later in this chapter we will consider an optimization that uses aggregation). This global access required continuous synchronization and introduced a significant overhead. We could accomplish the same result by using PLINQ, as follows:

```
List < int > primes = (from n in Enumerable.Range(3, 200000)).AsParallel()

    where IsPrime(n)
    select n).ToList();
//Could have used ParallelEnumerable.Range instead of Enumerable.Range(. . .).AsParallel()
```

CUSTOMIZING PARALLEL LOOPS AND PLINQ

Parallel loops (`Parallel.For` and `Parallel.ForEach`) and PLINQ have several customization APIs, which make them extremely flexible and close in richness and expressiveness to the explicit task parallelism APIs we have considered previously. The parallel loop APIs accept a `ParallelOptions` object with various properties, whereas PLINQ relies on additional methods of `ParallelQuery < T >`. These options include:

- Limiting the degree of parallelism (the number of tasks that would be allowed to execute concurrently)
- Providing a cancellation token for canceling the parallel execution
- Forcing output ordering of a parallel query
- Controlling output buffering (merge mode) of a parallel query

With parallel loops, it is most common to limit the degree of parallelism using the `ParallelOptions` class, whereas with PLINQ, you would often customize the query's merge mode and ordering semantics. For more information on these customization options, consult the MSDN documentation.

C# 5 Async Methods

So far, we considered rich APIs that allow a variety of parallelism solutions to be expressed using the classes and methods of the Task Parallel Library. However, other parallel programming environments sometimes rely on language extensions to obtain even better expressiveness where APIs are clumsy or insufficiently concise. In this section we will see how C# 5 adapts to the challenges of the concurrent programming world by providing a language extension to express continuations more easily. But first, we must consider continuations in the asynchronous programming world.

Often enough, you would want to associate a *continuation* (or callback) with a specific task; the continuation should be executed when the task completes. If you have control of the task—i.e., you schedule it for execution—you can embed the callback in the task itself, but if you receive the task from another method, an explicit continuation API is desirable. The TPL offers the `ContinueWith` instance method and `ContinueWhenAll/ContinueWhenAny` static methods (self-explanatory) to control continuations in several settings. The continuation may be scheduled only in specific circumstances (e.g., only when the task ran to completion or only when the task has encountered an exception), and may be scheduled on a particular thread or thread group using the `TaskScheduler` API. Below are some examples of the various APIs:

```
Task < string > weatherTask = DownloadWeatherInfoAsync(. . .);
weatherTask.ContinueWith(_ => DisplayWeather(weatherTask.Result), TaskScheduler.Current);
Task left = ProcessLeftPart(. . .);
Task right = ProcessRightPart(. . .);
TaskFactory.ContinueWhenAll(
    new Task[] { left, right },
    CleanupResources
);
TaskFactory.ContinueWhenAny(
    new Task[] { left, right },
```

```

        HandleError,
        TaskContinuationOptions.OnlyOnFaulted
    );

```

Continuations are a reasonable way to program asynchronous applications, and are very valuable when performing asynchronous I/O in a GUI setting. For example, to ensure that Windows 8 Metro-style applications maintain a responsive user interface, the WinRT (Windows Runtime) APIs in Windows 8 offer only asynchronous versions of all operations that might run for longer than 50 milliseconds. With multiple asynchronous calls chained together, nested continuations become somewhat clumsy, as the following example may demonstrate:

```

//Synchronous version:
private void updateButton_Clicked(. . .) {
    using (LocationService location = new LocationService())
    using (WeatherService weather = new WeatherService()) {
        Location loc = location.GetCurrentLocation();
        Forecast forecast = weather.GetForecast(loc.City);
        MessageDialog msg = new MessageDialog(forecast.Summary);
        msg.Display();
    }
}

//Asynchronous version:
private void updateButton_Clicked(. . .) {
    TaskScheduler uiScheduler = TaskScheduler.Current;
    LocationService location = new LocationService();
    Task < Location > locTask = location.GetCurrentLocationAsync();
    locTask.ContinueWith(_ => {
        WeatherService weather = new WeatherService();
        Task < Forecast > forTask = weather.GetForecastAsync(locTask.Result.City);
        forTask.ContinueWith(__ => {
            MessageDialog message = new MessageDialog(forTask.Result.Summary);
            Task msgTask = message.DisplayAsync();
            msgTask.ContinueWith(____ => {
                weather.Dispose();
                location.Dispose();
            });
        }, uiScheduler);
    });
}

```

This deep nesting is not the only peril of explicit continuation-based programming. Consider the following synchronous loop that requires conversion to an asynchronous version:

```

//Synchronous version:
private Forecast[] GetForecastForAllCities(City[] cities) {
    Forecast[] forecasts = new Forecast[cities.Length];
    using (WeatherService weather = new WeatherService()) {
        for (int i = 0; i < cities.Length; ++i) {
            forecasts[i] = weather.GetForecast(cities[i]);
        }
    }
    return forecasts;
}

//Asynchronous version:
private Task < Forecast[] > GetForecastsForAllCitiesAsync(City[] cities) {
    if (cities.Length == 0) {
        return Task.Run(() => new Forecast[0]);
    }
    WeatherService weather = new WeatherService();
    Forecast[] forecasts = new Forecast[cities.Length];
    return GetForecastHelper(weather, 0, cities, forecasts).ContinueWith(_ => forecasts);
}

private Task GetForecastHelper( WeatherService weather, int i, City[] cities, Forecast[] forecasts)
    if (i >= cities.Length) return Task.Run(() => { });
    Task < Forecast > forecast = weather.GetForecastAsync(cities[i]);
    forecast.ContinueWith(task => {
        forecasts[i] = task.Result;
        GetForecastHelper(weather, i + 1, cities, forecasts);
    });
    return forecast;
}

```

Converting this loop requires completely rewriting the original method and scheduling a continuation that essentially executes the next iteration in a fairly unintuitive and recursive manner. This is something the C# 5 designers have chosen to address on the language level by introducing two new keywords, `async` and `await`.

An `async` method must be marked with the `async` keyword and may return `void`, `Task`, or `Task < T >`. Within an `async` method, the `await` operator can be used to express a continuation without using the `ContinueWith` API. Consider the following example:

```
private async void updateButton_Clicked(. . .) {
    using (LocationService location = new LocationService()) {
        Task < Location > locTask = location.GetCurrentLocationAsync();
        Location loc = await locTask;

        cityTextBox.Text = loc.City.Name;
    }
}
```

In this example, the `await locTask` expression provides a continuation to the task returned by `GetCurrentLocationAsync`. The continuation's body is the rest of the method (starting from the assignment to the `loc` variable), and the `await` expression evaluates to what the task returns, in this case—a `Location` object. Moreover, the continuation is implicitly scheduled on the UI thread, which is something we had to explicitly take care of earlier using the `TaskScheduler` API.

The C# compiler takes care of all the relevant syntactic features associated with the method's body. For example, in the method we just wrote, there is a `try. . .finally` block hidden behind the `using` statement. The compiler rewrites the continuation such that the `Dispose` method on the `location` variable is invoked regardless of whether the task completed successfully or an exception occurred.

This smart rewriting allows almost trivial conversion of synchronous API calls to their asynchronous counterparts. The compiler supports exception handling, complex loops, recursive method invocation—language constructs that are *hard* to combine with the explicit continuation-passing APIs. For example, here is the asynchronous version of the forecast-retrieval loop that caused us trouble earlier:

```
private async Task < Forecast[] > GetForecastForAllCitiesAsync(City[] cities) {
    Forecast[] forecasts = new Forecast[cities.Length];
    using (WeatherService weather = new WeatherService()) {
        for (int i = 0; i < cities.Length; ++i) {
            forecasts[i] = await weather.GetForecastAsync(cities[i]);
        }
    }
    return forecasts;
}
```

Note that the changes are minimal, and the compiler handles the details of taking the `forecasts` variable (of type `Forecast[]`) our method returns and creating the `Task < Forecast[] >` scaffold around it.

With only two simple language features (whose implementation is everything but simple!), C# 5 dramatically decreases the barrier of entry for asynchronous programming, and makes it easier to work with APIs that return and manipulate tasks. Furthermore, the language implementation of the `await` operator is not wed to the Task Parallel Library; native WinRT APIs in Windows 8 return `IAsyncOperation < T >` and not `Task` instances (which are a managed concept), but can still be awaited, as in the following example, which uses a real WinRT API:

```
using Windows.Devices.Geolocation;
. . .
private async void updateButton_Clicked(. . .) {
    Geolocator locator = new Geolocator();
    Geoposition position = await locator.GetGeopositionAsync();
    statusTextBox.Text = position.CivicAddress.ToString();
}
```

Advanced Patterns in the TPL

So far in this chapter, we have considered fairly simple examples of algorithms that were subjected to parallelization. In this section, we will briefly inspect a few advanced tricks that you may find useful when dealing with real-world problems; in several cases we may be able to extract a performance gain from very surprising places.

The first optimization to consider when parallelizing loops with shared state is *aggregation* (sometimes called *reduction*). When using shared state in a parallel loop, scalability is often lost because of synchronization on the shared state access; the more CPU cores are added to the mix, the smaller the gains because of the synchronization (this phenomenon is a direct corollary of Amdahl's Law, and is often called *The Law of Diminishing Returns*). A big performance boost is often available from aggregating local state within each thread or task that executes the parallel loop, and combining the local states to obtain the eventual result at the end of the loop's execution. TPL APIs that deal with loop execution come equipped with overloads to handle this kind of local aggregation.

For example, consider the prime number computation we implemented earlier. One of the primary hindrances to scalability was the need to insert newly discovered prime numbers into a shared list, which required synchronization. Instead, we can use a local list in each thread, and aggregate the lists together when the loop completes:

```
List < int > primes = new List < int > ();
Parallel.For(3, 200000,
    () => new List < int > (), //initialize the local copy
    (i, pls, localPrimes) => { //single computation step, returns new local state
        if (IsPrime(i)) {
            localPrimes.Add(i); //no synchronization necessary, thread-local state
        }
        return localPrimes;
    },
    localPrimes => { //combine the local lists to the global one
        lock(primes) { //synchronization is required
            primes.AddRange(localPrimes);
        }
    }
);
```

In the example above, the number of locks taken is significantly smaller than earlier—we only need to take a lock once per thread that executes the parallel loop, instead of having to take it per each prime number we discovered. We *did* introduce an additional cost of combining the lists together, but this cost is negligible compared to the scalability gained by local aggregation.

Another source of optimization is loop iterations that are too small to be parallelized effectively. Even though the data parallelism APIs chunk multiple iterations together, there may be loop bodies that are so quick to complete that they are dominated by the delegate invocation required to call the loop body for each iteration. In this case, the `Partitioner` API can be used to extract manually chunks of iterations, minimizing the number of delegate invocations:

```
Parallel.For(Partitioner.Create(3, 200000), range => { //range is a Tuple < int,int>
    for (int i = range.Item1; i < range.Item2; ++i) . . . //loop body with no delegate invocation
});
```

For more information on custom partitioning, which is as well an important optimization available to data-parallel programs, consult the MSDN article "Custom Partitioners for PLINQ and TPL", at <http://msdn.microsoft.com/en-us/library/dd997411.aspx>.

Finally, there are applications which can benefit from custom task schedulers. Some examples include scheduling work on the UI thread (something we have already done using `TaskScheduler.Current` to queue continuations to the UI thread), prioritizing tasks by scheduling them to a higher-priority scheduler, and affinizing tasks to a particular CPU by scheduling them to a scheduler that uses threads with a specific CPU affinity. The `TaskScheduler` class can be extended to create custom task schedulers. For an example of a custom task scheduler, consult the MSDN article "How to: Create a Task Scheduler That Limits the Degree of Concurrency", at <http://msdn.microsoft.com/en-us/library/ee789351.aspx>.