A Guide to Parallel Programming

Design Patterns for Decomposition, Coordination and Scalable Sharing

PRELIMINARY

May 26, 2010

Preface

This book describes patterns for parallel programming, with code examples, that use the new parallel programming support in the Microsoft® .NET Framework 4. This support is commonly referred to as the Parallel Extensions. You can use the patterns described in this book to improve your application's performance on multicore computers. Adopting the patterns in your code makes your application run faster today and also helps prepare for future hardware environments, which are expected to have an increasingly parallel computing architecture.

Who This Book Is For

The book is intended for programmers who write managed code for the .NET Framework on the Microsoft Windows® operating system platform. This includes programmers who write in Microsoft Visual C#®, Microsoft Visual Basic®, and F#. No prior knowledge of parallel programming techniques is assumed. However, readers need to be familiar with features of C# such as delegate methods, lambda expressions, generic types, and Language Integrated Query (LINQ) expressions. Readers should also have at least a basic familiarity with the concepts of processes and threads of execution.

> **Note:** Although the material in this book can be used with other languages, it focuses on C#. Particular attention is given to relevant features in .NET Framework 4.
>
> The examples in this book are written in C# and use the features of the .NET Framework 4, including the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). However, you can use the concepts presented here with other frameworks and libraries. For example, F# now has language support for concurrency.
>
> Complete code solutions are posted on CodePlex. See http://parallelpatterns.codeplex.com. There is a C# or PLINQ version for every example. In addition to the C# example code, there are also versions of the examples in Visual Basic.

Why This Book Is Pertinent Now

The advanced parallel programming features that are delivered with Visual Studio 2010 make it easier than ever to get started with parallel programming.

The Task Parallel Library (TPL) is for .NET programmers who want to write parallel programs. It simplifies the process of adding parallelism and concurrency to applications. The TPL dynamically scales the degree of parallelism to most efficiently use all the processors that are available. In addition, the TPL assists in the partitioning of work and the scheduling of tasks in the .NET thread pool. The library provides cancellation support, state management, and other services.

Parallel LINQ (PLINQ) is a parallel implementation of LINQ to Objects. PLINQ implements the full set of LINQ standard query operators as extension methods for the **System.Linq** namespace and has additional operators for parallel operations. PLINQ is a declarative, high-level interface with query capabilities for operations such as filtering, projection, and aggregation.

Visual Studio 2010 includes tools for debugging parallel applications. The Parallel Stacks window shows call stack information for all the threads in your application. It lets you navigate between threads and stack frames on those threads. The Parallel Tasks window resembles the Threads window, except that it shows information about each task instead of each thread. The Concurrency Visualizer views in the Visual Studio profiler enable you to see how your application interacts with the hardware, the operating system, and other processes on the computer. You can use the Concurrency Visualizer to locate performance bottlenecks, CPU underutilization, thread contention, cross-core thread migration, synchronization delays, areas of overlapped I/O, and other information.

What You Need to Use the Code

The code that is used as examples in this book is at http://parallelpatterns.codeplex.com/. These are the system requirements:

- Microsoft Windows Vista SP1, Windows 7, or Microsoft Windows Server® 2008 (32-bit or 64-bit)

- Microsoft Visual Studio 2010 (Ultimate or Premium edition is required for the Concurrency Visualizer, which allows you to analyze the performance of your application); this includes the .NET 4 Framework, which is required to run the samples

How to Use This Book

This book presents parallel programming techniques in terms of particular patterns. Figure 1 shows the different patterns and their relationships to each other. The numbers refer to the chapters in this book where the patterns are described.

Parallel programming patterns
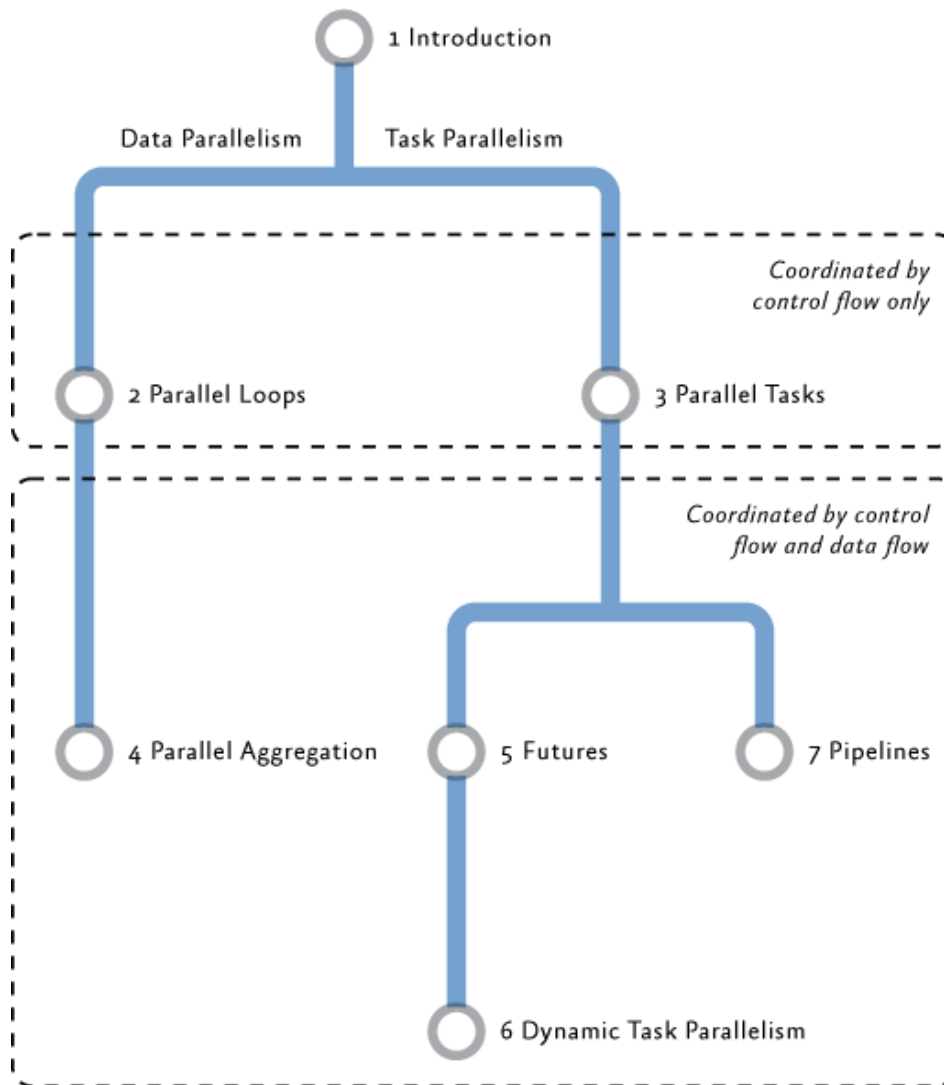
Figure 1
After the introduction, the book has two branches:

- One branch discusses data parallelism.

- One branch discusses task parallelism.

Both parallel loops and parallel tasks use only the program's control flow as the means to coordinate and order tasks. The other patterns use both control flow and data flow for coordination.

Introduction

Chapter 1 introduces the common problems faced by developers who want to use parallelism to make their applications run faster. It explains basic concepts and prepares you for the remaining chapters.

Parallelism with Control Dependencies Only

Chapters 2 and 3 deal with cases where asynchronous operations are ordered only by control flow constraints:

- **Chapter 2, "Parallel Loops**." Use parallel loops when you want to perform the same calculation on each member of a collection or for a range of indices, and where there are no dependencies between the members of the collection. For loops with dependencies, see Chapter 4, "Parallel Aggregation."

- **Chapter 3, "Parallel Tasks**." Use parallel tasks when you have several distinct asynchronous operations to perform. This chapter explains why tasks and threads serve two distinct purposes.

Parallelism with Control and Data Dependencies

Chapters 4 and 5 show patterns for concurrent operations that are constrained by both control flow and data flow:

- **Chapter 4, "Parallel Aggregation**." Patterns for parallel aggregation are appropriate when the body of a parallel loop includes data dependencies, such as when calculating a sum or searching a collection for a maximum value.

- **Chapter 5, "Futures**." The futures pattern occurs when operations produce some outputs that are needed as inputs to other operations. The order of operations is constrained by a directed graph of data dependencies. Some operations are performed in parallel and some serially, depending on when inputs become available.

Dynamic Task Parallelism and Pipelines

Chapters 6 and 7 discuss some more advanced scenarios:

- **Chapter 6, "Dynamic Task Parallelism**." In some cases, operations are dynamically added to the backlog of work as the computation proceeds. This pattern applies to several domains, including graph algorithms and sorting.

- **Chapter 7, "Pipelines**." Use pipelines to feed successive outputs of one component to the input queue of another component, in the style of an assembly line. Parallelism results when the pipeline fills, and when more than one component is simultaneously active.

Supporting Material

In addition to the patterns, there are several appendices:

- **Appendix A, "Supporting Patterns**." This appendix gives tips for adapting some of the common object-oriented patterns such as facades, decorators, and repositories to multicore architectures. For example, it shows how lazy data access, where an object is initialized only

after it's needed, can promote concurrency in a repository. Event-based coordination (named agents) is another familiar pattern that is also often adapted for parallel scenarios.

- **Appendix B, "Debugging and Profiling Parallel Applications**." This appendix gives you an overview of how to debug and profile parallel applications in Visual Studio 2010.

- **Appendix C, "Technology Roadmap**." A technology roadmap helps you place the various Microsoft technologies and frameworks for parallel programming in context.

- **Appendix D, "QuickStart Examples**." This appendix contains very brief code examples for common tasks, such as "How to parallelize a loop over a collection." It is meant as a shortcut reference.

A glossary contains definitions of terms used in this book.

Everyone should read Chapters 1, 2, and 3 for an introduction and overview of the basic principles. Although the succeeding material is presented in a logical order, each chapter, from Chapter 4 on, can be read independently.

Callouts show things you should watch out for. These are sometimes known as "anti-patterns."

| Anti-pattern: Don't apply the patterns in this book blindly to your applications. |

The hammer and the nail

Figure 2

What Is Not Covered

This book focuses more on CPU-bound workloads than on I/O-bound workloads. The goal is to make computationally intensive applications run faster by making better use of the computer's available cores. As a result, the book does not focus as much on the issue of I/O latency. Nonetheless, there is some discussion of balanced workloads that are both CPU intensive and have large amounts of I/O (see Chapter 7, "Pipelines"). There is also an important example for user interfaces in Chapter 5, "Futures," that illustrates concurrency for tasks with I/O.

7

The book describes parallelism within a single multicore node with shared memory instead of the cluster, High Performance Computing (HPC) Server approach that uses networked nodes with distributed memory. However, cluster programmers who want to take advantage of parallelism within a node may find the examples in this book helpful, because each node of a cluster can have multiple processing units.

1

Introduction

The CPU meter shows the problem. One core is running at 100 percent, but all the other cores are idle. Your application is CPU-bound, but you are using only a fraction of the computing power of your multicore system. What next?

The answer, in a nutshell, is *parallel programming*. Where you once would have written the kind of sequential code that is familiar to all programmers, you now find that this no longer meets your performance goals. To use your system's CPU resources efficiently, you need to split your application into pieces that can run at the same time.

Parallel programming lets your application use more than one core at the same time. The goal is to improve the application's speed.

This is easier said than done. Parallel programming has a reputation for being the domain of experts and a minefield of subtle, hard-to-reproduce software defects. Everyone seems to have a favorite story about a parallel program that did not behave as expected because of a mysterious bug.

These stories should inspire a healthy respect for the difficulty of the problems you face in writing your own parallel programs. Fortunately, help has arrived. Microsoft® Visual Studio® 2010 introduces a new programming model for parallelism that significantly simplifies the job. Behind the scenes are supporting libraries with sophisticated algorithms that dynamically distribute computations on multicore architectures.

Writing parallel programs has the reputation of being hard.

Proven design patterns are another source of help. This guide introduces you to the most important and frequently used patterns of parallel programming and gives executable code samples for them, using the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). When thinking about where to begin, a good place to start is to review the patterns in this book. See if your problem has any attributes that match the seven patterns presented in the chapters. If it does, then delve more deeply into the relevant pattern or patterns and study the sample code.

Stated another way, parallel programming can be hard, but in some cases, it's easy, as the patterns in this guide show. If you can't use these patterns, you have probably encountered one of the more difficult cases. In these difficult cases, you'll need to hire an expert or consult the academic literature.

The code examples for this guide are online at http://parallelpatterns.codeplex.com.

The Importance of Potential Parallelism

The patterns in this book are ways to express *potential parallelism*. This means that your program is written so that it runs faster when parallel hardware is available and roughly the same as an equivalent sequential program when it's not. If you correctly structure your code, the run-time environment can automatically adapt to the workload on a particular computer. This is why the patterns in this book only express potential parallelism. They do not guarantee it. This concept is a central organizing principle behind the parallel programming model of Visual Studio 2010. It deserves some explanation.

> Declaring the potential parallelism of your program allows the execution environment to run it on all available cores, whether one or many.

Some parallel applications can be written for specific hardware. For example, creators of programs for a console gaming platform have detailed knowledge about the hardware resources that will be available at run time. They know the number of cores and the details of the memory architecture in advance. The game can be written to exploit the exact level of parallelism provided by the platform. Complete knowledge of the hardware environment is also a characteristic of some embedded applications, such as industrial control. The life cycle of such programs matches the life cycle of the specific hardware they were designed to use.

In contrast, when you write programs that run on general-purpose computing platforms, such as desktop workstations and servers, there is less predictability about the hardware features. You may not always know how many cores will be available. You may not know whether memory access times will be uniform across all cores or whether the memory system of the computer optimizes memory access by grouping cores and memory regions into nodes.

> In most cases it's a bad idea to hard code the degree of parallelism in an application. You can't always predict how many cores will be available at run time.

In addition to uncertainty about the hardware capabilities of the computer that will run your application, you also may be unable to predict what other software could be running at the same time as your application.

Even if you initially know about your application's environment, it can change over time. Planning for the future adds yet another layer of uncertainty, especially if the application will outlast the current generation of hardware. In the past, programmers assumed that their applications would automatically run faster on later generations of hardware. You could rely on this because processor clock speeds kept increasing. With multicore processors, clock speeds may not increase with newer hardware as much as they have in the past. Instead, the trend in processor design is toward more cores and more distribution of memory systems. If you want your application to benefit from hardware advances in the multicore world, you need to adapt your programming model.

Finally, you must plan for all of these contingencies in a way that does not penalize users who might not have access to the latest hardware. You want your parallel application to run as fast on a single-core computer as an application that was written using only sequential code. In other words, you want *scalable performance* from one to many cores.

Allowing your application to adapt to varying hardware capabilities, both now and in the future, is the motivation for potential parallelism.

An example of potential parallelism is the parallel loop pattern described in Chapter 2, "Parallel Loops." If you have a **for** loop that performs a million independent iterations, it makes sense to divide those iterations among the available cores and do the work in parallel. It's easy to see that how you divide the work should depend on the number of cores. If you do this, for many common scenarios, the speed of the loop will be approximately proportional to the number of cores.

Decomposition, Coordination and Scalable Sharing

The patterns in this book contain some common themes. You will see that the process of designing and implementing a parallel application involves three aspects: *decomposing* the work into discrete units known as tasks, ways of *coordinating* these tasks as they run in parallel, and scalable techniques for *sharing* the data needed to perform the tasks.

You can apply these guidelines at the architectural level, when you think about the overall structure of your application, and when you design and implement your algorithms. The patterns described in this guide are considered to be design patterns and the example applications are relatively small, but the principles they demonstrate apply equally well to the architectures of large applications.

Understanding Tasks

Tasks are sequential operations that work together to perform a larger operation. When you think about how to structure a parallel program, it's important to identify tasks at a level of granularity that results in efficient use of the CPU resources. If the chosen granularity is too fine, the overhead of managing tasks will dominate. If it is too coarse, opportunities for parallelism may be lost as cores that could otherwise be used remain idle.

In general, tasks should be as large as possible, but they should remain independent of each other, and there should be enough tasks to keep the cores busy. Meeting these goals sometimes involves design tradeoffs. Decomposing a problem into tasks requires a good understanding of the algorithmic and structural aspects of your application.

Tasks are sequential units of work. They should be as independent of each other as possible, and there should be enough tasks to keep all cores busy.

An example of these guidelines is a parallel ray tracing application. A ray tracer constructs a synthetic image by simulating the path of each ray of light in a scene. The individual ray simulations are a good level of granularity for parallelism. Breaking the tasks into smaller units, for example, by trying to decompose the ray simulation itself into independent tasks, only adds overhead, because the number of ray simulations is already large enough to keep all cores occupied. A good rule of thumb is that the number of tasks should be from two to ten times the number of cores that are expected to be available at run time.

Another advantage to grouping work into larger (fewer) tasks is that they often are more independent of each other than smaller (more numerous) tasks. Larger tasks are less likely to share local variables or fields than smaller tasks. Unfortunately, in applications that rely on large mutable object graphs, such as applications that expose a large object model with many public classes, methods, and properties, the opposite may be true. In these cases, the larger the task, the more chance there is for unexpected sharing of data or other side effects.

> **Note:** Tasks are not threads. The distinction between tasks and threads is covered in Chapter 3, "Parallel Tasks."

Coordinating Tasks

It's often possible that more than one task can run at the same time. Tasks that are independent of one another can run in parallel, while some tasks can begin only after other tasks complete. The order of execution and the degree of parallelism are constrained by the application's underlying algorithms. Constraints can arise from control flow (the steps of the algorithm) or data flow (the availability of inputs and outputs).

Various mechanisms for coordinating tasks are possible. How tasks are coordinated depends on which parallel pattern you use. For example, the pipeline pattern described in Chapter 7, "Pipelines," is distinguished by its use of concurrent queues to coordinate tasks. Regardless of the mechanism you choose for coordinating tasks, in order to have a successful design, you must understand the dependencies between tasks.

Scalable Sharing of Data

Tasks often need to share data. There are a number of techniques that allow data to be shared that don't degrade performance or make your program prone to error. These techniques include the use of immutable, read-only data, carefully isolating mutable state, limiting your program's reliance on shared variables and introducing new steps in your algorithm that merge mutable state at appropriate checkpoints. In other words, scalable sharing may involve changes to an existing algorithm. It's not just a matter of adding locks. A better alternative is using immutable data, independent functions, and concurrent collections that are provided by the framework.

Object-Oriented Designs and Data Sharing

Conventional object-oriented designs can have complex and highly interconnected in-memory graphs of object references. As a result, traditional object-oriented programming styles can be very difficult to adapt to scalable parallel execution. Your first impulse might be to consider all fields of a large, interconnected object graph as mutable shared state, and to wrap access to these fields in serializing locks whenever there is the possibility that they may be shared by multiple tasks. However, this is not a scalable approach to sharing. Locks have side effects that negatively affect the performance of all cores. Locks force cores to pause and communicate, which takes time, and they introduce serial regions in the code, which reduces the potential for parallelism. As the number of cores gets larger, the cost of lock contention can increase. As more and more tasks are added that share the same data, the overhead associated with locks can dominate the computation.

In addition to performance problems, programs that rely on complex synchronization are prone to deadlock and other software defects. Most of the horror stories about parallel programming are actually horror stories about the incorrect use of shared mutable state or locking protocols.

No one is advocating the removal, in the name of performance, of synchronization that's necessary for correctness. First and foremost, the code still needs to be correct. However, it's important to use design principles that avoid the need for synchronization. This is part of the design process; it can't be added as an afterthought later.

Thus, synchronizing (that is, locking) elements in an object graph plays a legitimate if limited role in scalable parallel programs. This book uses synchronization sparingly. You should, too. Locks are the **goto** statements of parallel programming: error prone, occasionally necessary and usually best left to compilers and libraries.

Locks are the **goto** statements of parallel programming; use design principles that avoid the need for locks.

Design Approaches

It's common for developers to identify one problem area, parallelize the code to improve performance, and then repeat the process for the next bottleneck. This is a particularly tempting approach when you parallelize an existing application. Although this may give you some initial improvements in performance, it has many pitfalls and it may not produce the best results. A far better approach is to understand your problem or application and look for potential parallelism across the entire application. In other words, look at your application and its potential to express parallelism holistically. This may lead you to adopt a different architecture or algorithm that better exposes the areas of potential parallelism in your application. Don't simply identify bottlenecks and parallelize them.

Think in terms of data structures and algorithms; don't just identify bottlenecks.

Techniques for decomposition, coordination, and scalable sharing are interrelated. There's a circular dependency. You need to consider all of these aspects together when choosing your approach for a particular application.

After reading the preceding description, you might complain that it all seems vague. How specifically do you divide your problem into tasks? Exactly what kinds of coordination techniques should you use?

<div align="center">Use patterns.</div>

Questions like these are best answered by the patterns described in this book. Patterns are a true shortcut to understanding. As you begin to see the design motivations behind the patterns in this book, you will also develop your intuition about how the patterns can be applied to your own applications. The following section gives more details about how you can use the parallel programming patterns described in this book.

Selecting the Right Pattern

To select the relevant pattern, use the following table.

Application characteristic	Relevant pattern
Do you have sequential loops where there's no communication among the steps of each iteration?	The Parallel Loop pattern (Chapter 2). Parallel loops apply an independent operation to multiple inputs simultaneously.
Do you have specific units of works with well-defined control dependencies?	The Parallel Task pattern (Chapter 3) Parallel tasks allow you to establish parallel control flow in the style of fork and join.
Do you need to summarize data by applying some kind of combination operator? Do you have loops with steps that are not fully independent?	The Parallel Aggregation pattern (Chapter 4) Parallel aggregation introduces special steps in the algorithm for merging partial results. This pattern expresses a reduction operation and includes map/reduce as one of its variations.
Does the ordering of steps in your algorithm depend on data flow constraints?	The Futures pattern (Chapter 5) Futures make the data flow dependencies between tasks explicit. This pattern is also referred to as the Task Graph pattern.
Does your algorithm divide the problem domain dynamically during the run? Do you operate on recursive data structures such as graphs?	The Dynamic Task Parallelism pattern (Chapter 6) This pattern takes a divide-and-conquer approach and spawns new tasks on demand.
Does your application perform a sequence of operations repetitively? Does the input data have streaming characteristics?	The Pipelines pattern (Chapter 7) Pipelines consist of components that are connected by queues, in the style of produces and consumers. All the components run in parallel.

One way to familiarize yourself with the possibilities is to read the first page or two of each chapter. This gives you an overview of approaches that have been proven to work in a wide variety of applications. Then go back and more deeply explore patterns that may apply in your situation.

A Word About Terminology

You will often hear the words *parallelism* and *concurrency* used as synonyms. This book makes a distinction between the two terms.

Concurrency is a concept related to multitasking and asynchronous I/O. It usually refers to the existence of multiple threads of execution that may each get a slice of time to execute before being preempted by another thread, which also gets a slice of time. Concurrency can exist on a computer with a single core.

With *parallelism*, concurrent threads execute at the same time on multiple cores. Parallel programming focuses on improving the performance of CPU-bound applications when multiple cores are available.

The goals of concurrency and parallelism are distinct. The main goal of concurrency is to reduce latency by never allowing long periods of time to go by without at least some computation being performed by each unblocked thread. In other words, the goal of concurrency is to prevent *thread starvation*.

Concurrency is required operationally. For example, an operating system with a graphical user interface must support concurrency if more than one window at a time can be displayed on a single-core computer. Parallelism, on the other hand, is only about throughput. It is an optimization, not a functional requirement. Its goal is to maximize CPU usage across all available cores; to do this, it uses scheduling algorithms that are not preemptive, such as algorithms that process queues or stacks of work to be done.

The Limits of Parallelism

A theoretical result known as Amdahl's law says that the amount of performance improvement that parallelism provides is limited by the amount of sequential processing in your application. This may, at first, seem counterintuitive.

Amdahl's law says that no matter how many cores you have, the maximum speedup you can ever achieve is (1 / percent of time spent in sequential processing). Figure 1 illustrates this.

Amdahl's law for an application with 25 percent sequential processing
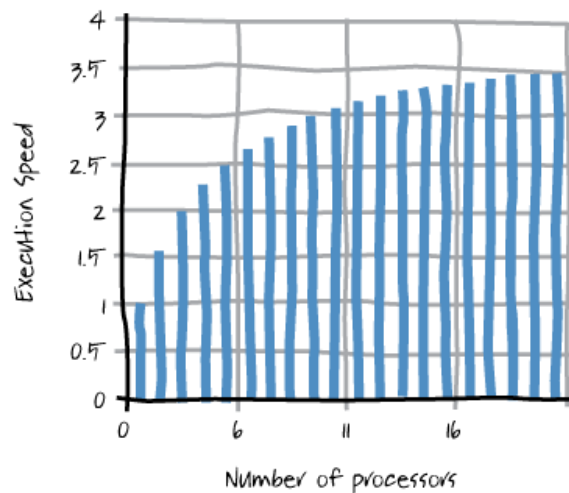
Figure 1

For example, with 11 processors, the application runs slightly more than three times faster than it would if it were entirely sequential.

Even with fewer cores, you can see that the expected speedup is not linear. Figure 2 illustrates this.

Per-core performance improvement for a 25 percent sequential application
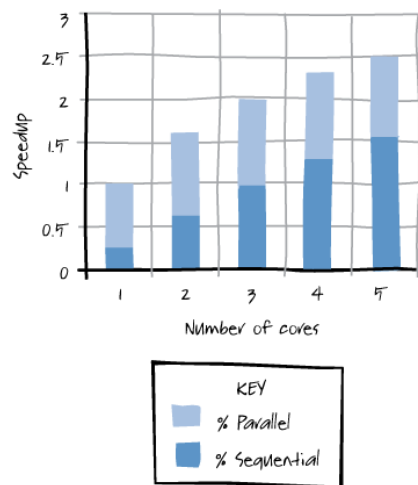
Figure 2

Figure 2 shows that as the number of cores (and overall application speed) increases the percentage of time spent in the sequential part of the application increases. (The elapsed time spent in sequential processing is constant.) The figure also shows why you might be satisfied with a 2x speedup on a four-core computer for actual applications, as opposed to sample programs. The important question is always how scalable the application is; this depends on the amount of time spent doing work that is inherently sequential in nature.

Another implication of Amdahl's law is that for some problems, you may want to invest in additional features in the parts of an application that are amenable to parallel execution. For example, a developer

16

of a computer game might find that it's possible to make increasingly sophisticated graphics for newer multicore computers by using the parallel hardware, even if it's not as feasible to make the game logic (the artificial intelligence engine) run in parallel. Performance can influence the mix of application features.

The speedup you can achieve in practice is usually somewhat worse than Amdahl's law would predict. As the number of cores increases, the overhead incurred by accessing shared memory also increases. Also, parallel algorithms may include overhead for coordination that would not be necessary for the sequential case. Profiling tools, such as the Visual Studio Concurrency Visualizer, can help you understand how effective your use of parallelism is.

In summary, you will rarely get a linear increase in performance with a linear increase in the number of cores. Understanding the existing application's performance, the structure of your application, and its algorithms—that is, which parts of your application are suitable for parallel execution—is a step that can't be skipped.

A Few Tips

Always try for the simplest approach. Here are some basic precepts:

- Whenever possible, use a library that does the parallel work for you.

- Use your application server's inherent parallelism; for example, use the parallelism that is incorporated into a web server or database.

- Use an API to encapsulate parallelism, such as Microsoft Parallel Extensions for .NET (TPL and PLINQ) or Microsoft Parallel Patterns Library (PPL) for native code. These libraries were written by experts; they help you to avoid many of the common problems that arise in parallel programming.

- Consider the overall architecture of your application when thinking about how to parallelize it. It's tempting to simply look for the performance hotspots and focus on improving them. While this may improve things, it does not necessarily give you the best results.

- Use patterns, such as the ones described in this book.

- Often, restructuring your algorithm (for example, to eliminate the need for shared data) is better than making low-level improvements to code that was originally designed to run serially.

- Don't share data among concurrent tasks unless absolutely necessary. If you do share data, use one of the containers provided by the API you are using, such as a shared queue.

- Use low-level primitives, such as threads and locks, only as a last resort. Raise the level of abstraction from threads to tasks in your applications.

Exercises

1. What are some of the tradeoffs between decomposing a problem into many small tasks versus decomposing it into larger tasks?

2. What is the maximum potential speedup of a program that spends 25 percent of its time in sequential processing when you move it from one to four cores?

3. What is the difference between parallelism and concurrency?

For More Information

If you are interested in better understanding the terminology used in the text, refer to the glossary at the end of the book.

The design patterns presented in this book are consistent with classifications of parallel patterns developed by groups in both industry and academia. In the terminology of these groups, the patterns in this book would be considered to be algorithm or implementation patterns. Classification approaches for parallel patterns can be found in Berkeley [7] and Mattson [4]. This book attempts to be consistent with the terminology of these sources. In cases where this is not possible, an explanation appears in the text.

2

Parallel Loops

Use the Parallel Loop pattern when the same independent operation needs to be performed for each element of a collection or for a fixed number of iterations. Iterations of a loop are independent if they don't write to memory locations or files that are read by other iterations.

The syntax of a parallel loop is very similar to the **for** and **for each** loops you already know, but the parallel loop runs faster on a computer that has available cores.

The Parallel Loop pattern independently applies an operation to multiple data elements. This kind of computation is sometimes known as *data parallelism*.

> The Parallel Loop pattern is sometimes known as loop parallelism, the map pattern, or Single Program, Multiple Data (SPMD).

With parallel loops, the degree of parallelism doesn't need to be specified by your code. Instead, the run-time environment executes iterations of your loop as concurrently as possible on the available cores. The loop works correctly no matter how many cores are available. If there is only one core, the performance is close to (perhaps within a few percentage points of) the sequential equivalent. If there are multiple cores, performance usually improves; in many cases, performance improves proportionately with the number of cores.

The Basics

The Parallel Loop pattern includes both parallel **for** and parallel **for each** loops. Use the first to iterate over a range of integer indices and the second to iterate over user-provided values.

To make **for** or **for each** loops with independent iterations run faster on multicore computers, use their parallel counterparts.

These options are included in the .NET Framework 4 **Parallel** class.

Parallel For Loops

Here is an example of a C# sequential **for** loop.

```
for (int i = 0; i < N; i++)
{
    // ... do some work ...
}
```

To take advantage of multiple cores, simply replace the **for** keyword with a call to the **Parallel.For** method. This requires that the iterations of the loop body are independent of one another.

```
Parallel.For(0, N, i =>
{
    // ... do some work ...
});
```

Parallel.For uses multiple cores to operate over an index range.

Parallel.For is a static method, not part of the programming language.

In this example, the first two arguments specify the iteration limits. The first argument is the lowest index of the loop. The second argument is the exclusive upper bound. This is the largest index plus one. The third argument is a delegate method that is invoked once per iteration. The delegate method takes the iteration's index as its argument and executes the loop body once for each index.

The **Parallel.For** method does not guarantee the order of execution. Unlike a sequential loop, some higher-valued indices may be processed before lower-valued indices.

The **Parallel.For** method has additional overloaded versions. These are covered in the section, "Variations," later in this chapter and in Chapter 4, "Parallel Aggregation with Map-Reduce."

> **Note:** The code example includes a lambda expression in the form *args => body* as the third parameter to the **Parallel.For** method. Lambda expressions denote anonymous delegate methods. If you are unfamiliar with the syntax for lambda expressions, check the reference section at the end of this chapter for more information.
>
> Of course, this argument could also be a named delegate. You don't have to use lambdas if you don't want to. Examples in this book use lambda expressions because they approximate the code that would be written for the serial case and are therefore easier to read and to write.

Parallel ForEach

You can also execute a **for each** loop in parallel. Here is an example of a sequential C# **foreach** loop.

```
MyObject[] myEnumerable = ...

foreach (var obj in myEnumerable)
{
    // ... do some work ...
}
```

To take advantage of multiple cores, replace the **foreach** keyword with a call to the **Parallel.ForEach** method. This requires that iterations of the loop body are independent. This example assumes that the

loop body only makes updates to fields of the particular instance of the **MyObject** class that is passed to it. The loop body doesn't read fields that are updated by other iterations. The example also assumes that each instance appears only once in the input array.

```
MyObject[] myEnumerable = ...

Parallel.ForEach(myEnumerable, obj =>
{
   // ... do some work ...
});
```

<center>**Parallel.ForEach** runs the loop body for each element in a collection.</center>

Parallel.ForEach is a static method, not part of the programming language.

In this example, the first argument is an object that implements the **IEnumerable<T>** interface. The second argument is a delegate method that is invoked for each element of the input collection.

The **Parallel.ForEach** method does not guarantee the order of execution. Unlike a sequential for each loop, the incoming values are not always processed in order.

The **Parallel.ForEach** method has additional overloaded versions. These are covered in the section, "Variations," later in this chapter and in Chapter 4, "Parallel Aggregation with Map-Reduce."

What to Expect

By default, the degree of parallelism (that is, how many iterations run at the same time in hardware) depends on the number of available cores. In typical scenarios, the more cores you have, the faster your loop executes. How much faster depends on the kind of work your loop does. For some algorithms, you will at some point reach a point of over-decomposition and the algorithm runs slower as you use more cores.

<center>Adding cores makes your loop run faster; however, sometimes there's an upper limit.</center>

The .NET implementation of the Parallel Loop pattern ensures that exceptions that are thrown during the execution of a loop body are not lost. For both the **Parallel.For** and **Parallel.ForEach** methods, exceptions are collected into an **AggregateException** object and rethrown in the context of the calling thread. All exceptions are propagated back to you. To learn more about exception handling for parallel loops, see the section, "Variations," later in this chapter.

<center>Robust exception handling is an important aspect of parallel loop processing.</center>

So far, you've seen only the most basic parallel loops. There are many variations. There are 12 overloaded methods for **Parallel.For** and 20 overloaded methods for **Parallel.ForEach**. In addition, the Language Integrated Query (LINQ) feature of the .NET Framework includes a parallel version named PLINQ (Parallel LINQ). With close to 200 extension methods, there are many options and variations on how to express PLINQ queries. To learn about some of the most important cases, see the section, "Variations," later in this chapter.

The Parallel Loop pattern is one of the easiest parallel patterns to apply to existing code. First, look for the top-level loops in your application. These are the loops that consume most of the CPU cycles. Then check that the loop body is independent. If it is, you can replace **for** or **foreach** with **Parallel.For** or **Parallel.ForEach**.

If you convert a sequential loop to a parallel loop and you then find that your program does not behave as expected, the mostly likely problem is that the loop's iterations are not independent. Here are some common examples of dependent loop bodies:

Check carefully for dependencies between loop iterations; subtle loop dependencies are a common source of bugs.

- **Shared state**. If the body of a parallel loop writes to a shared variable, there is a loop body dependency. This is a common case that occurs when you are aggregating values. Here is an example.

```
for(int i = 1; i < N; i++)
  total += data[i];
```

If you encounter this situation, see Chapter 4, "Parallel Aggregation with Map-Reduce."

If the object being processed by a loop body exposes properties, you need to keep in mind whether those properties refer to shared state or state that is local to the object itself. For example, a property named **Parent** is likely to refer to global state. Here is an example.

```
for(int i = 0; i < N; i++)
  SomeObject[i].Parent.Update();
```

In this example, it is likely that the loop iterations are not independent.

- **Single-threaded data types**. If the body of the parallel loop uses a single-threaded data type, the loop body is not independent (there is an implicit context dependency). An example of this case, along with a solution, is shown in "Using Thread-Local State in a Loop Body" in the section, "Variations," later in this chapter.

- **Index arithmetic**. If the body of a parallel for loop performs arithmetic on the loop index, there is likely to be a loop body dependency. This is shown in the following code example.

```
for(int i = 1; i < N; i++)
  data[i] = data[i] + data[i - 1];
```

It is sometimes possible to use a parallel algorithm in cases with this kind of dependency, but this is outside of the scope of this guide. Your best bet is to look elsewhere in your program for opportunities for parallelism or analyze your algorithm and see if it matches some of the advanced parallel patterns that occur in scientific computing. Parallel scan and parallel dynamic programming are examples of these patterns.

> **Note:** When looking for opportunities for parallelism, profiling your application is a way to deepen your understanding of where your application spends its time; however, profiling is not a substitute for understanding your application's structure and algorithms. For example, profiling doesn't tell you whether loop bodies are independent.

An Example

Here's an example of when to use a parallel loop. Fabrikam Shipping extends credit to its commercial accounts. It uses customer credit trends to identify accounts that might pose a credit risk. Each customer account includes a history of past balance-due amounts. Fabrikam has noticed that customers who don't pay their bills often have histories of steadily increasing balances over a period of several months before they default.

To identify at-risk accounts, Fabrikam uses statistical trend analysis to calculate a projected credit balance for each account. If the analysis predicts that a customer account will exceed its credit limit within three months, the account is flagged for manual review by one of Fabrikam's credit analysts.

In the application, a top-level loop iterates over customers in the account repository. The body of the loop fits a trend line to the balance history, extrapolates the projected balance, compares it to the credit limit, and assigns the warning flag if necessary.

An important aspect of this application is that each customer's credit status can be calculated independently. The credit status of one customer doesn't depend on the credit status of any other customer. Because the operations are independent, making the credit analysis application run faster is simply a matter of replacing a sequential **foreach** loop with a parallel loop.

If in doubt about how effective the parallel loop is, profile.

The complete source code for this example is online at http://parallelpatterns.codeplex.com in the Chapter2\CreditReview project.

Sequential Prediction

Here's the sequential version of the credit analysis operation.

```
static void UpdatePredictionsSequential(
                              AccountRepository accounts)
  {
    foreach (Account account in accounts.AllAccounts)
    {
      Trend trend = SampleUtilities.Fit(account.Balance);
      double prediction = trend.Predict(
                    account.Balance.Length + NumberOfMonths);
      account.SeqPrediction = prediction;
      account.SeqWarning = prediction < account.Overdraft;
  }
}
```

23

The **UpdatePredictionsSequential** method processes each account from the application's account repository. The **Fit** method is a utility function that uses the least squares method to create a trend line from an array of numbers. The prediction is a three-month projection based on the trend. If a prediction is more negative than the overdraft limit (credit balances are negative numbers in the accounting system), the account is flagged for review.

Prediction Using Parallel.ForEach

The parallel version of the credit scoring analysis is very similar to the sequential version.

```
static void UpdatePredictionsParallel(AccountRepository accounts)
{
  Parallel.ForEach(accounts.AllAccounts, account =>
  {
    Trend trend = SampleUtilities.Fit(account.Balance);
    double prediction = trend.Predict(
                        account.Balance.Length + NumberOfMonths);
    account.ParPrediction = prediction;
    account.ParWarning = prediction < account.Overdraft;
  });
}
```

The **UpdatePredictionsParallel** method is identical to the **UpdatePredictionsSequential** method, except that the **Parallel.ForEach** method replaces the **foreach** operator.

Prediction with PLINQ

You can also use PLINQ to express a parallel loop. Here is an example.

```
static void UpdatePredictionsPlinq(AccountRepository accounts)
{
  accounts.AllAccounts
  .AsParallel()
  .ForAll(account =>
  {
    Trend trend = SampleUtilities.Fit(account.Balance);
    double prediction = trend.Predict(
                        account.Balance.Length + NumberOfMonths);
    account.PlinqPrediction = prediction;
    account.PlinqWarning = prediction < account.Overdraft;
  });
}
```

Using PLINQ is almost exactly like using LINQ-to-Objects and LINQ-to-XML. PLINQ provides a **ParallelEnumerable** class that defines extension methods for various types in a manner very similar to LINQ's **Enumerable** class. One of the methods of **ParallelEnumerable** is the **AsParallel** extension method.

The **AsParallel** extension method allows you to convert a sequential collection of type **IEnumerable<T>** into a **ParallelQuery<T>** object. Applying **AsParallel** to the **accounts.GetAccountsList** collection returns an object of type **ParallelQuery<AccountRecord>**.

PLINQ's **ParallelEnumerable** class has close to 200 extension methods that provide parallel queries for **ParallelQuery<T>** objects. In addition to parallel implementations of LINQ methods, such as **Select** and **Where**, PLINQ provides a **ForAll** extension method that invokes a delegate method in parallel for each element.

In the PLINQ prediction example, the argument to **ForAll** is a lambda expression that performs the credit analysis for a specified account. The body is the same as in the sequential version.

Performance Comparison

Running the credit review example shows that on a dual-core computer, the **Parallel.ForEach** and PLINQ versions run slightly less than twice as fast as the sequential version.

Variations

The credit analysis example presents a typical way to use parallel loops, but there can be variations. This section introduces some of the most important variations of the parallel loop pattern. These discussions are condensed summaries of topics presented in a paper by Stephen Toub [Toub09]. For more information, see the chapter, "References."

Breaking Out of Loops Early

Breaking out of loops is a familiar pattern in sequential iteration. Here is a simple example.

```
for (int i = 0; i < N; i++)
{
  // ... do some work ...
  if (/* condition is true */)
    break;
}
```

The situation is more complicated with parallel loops because more than one iteration may be active at the same time, and partitioning algorithms used by the runtime may result in iterations that are not necessarily executed in order. Partitioning refers to the process of assigning the work to be done to the available cores.

To address these scenarios, the **Parallel.For** method has an overload that provides a **ParallelLoopState** object as a second argument to the loop body. You can ask the loop to break by calling the **Break** method of the **ParallelLoopState** object. Here is an example.

Use **Break** to exit a loop early while ensuring that lower-indexed iterations complete.

```
Parallel.For(0, N, (i, loopState) =>
{
```

25

```
  // ... do some work ...
  if (/* stopping condition is true */)
  {
    loopState.Break();
    return;
  }
}
```

Calling the **Break** method of the **ParallelLoopState** object begins an orderly shutdown of the loop processing. Any iterations that are running as of the call to **Break** will run to completion. However, you may want to check for a break condition in long-running loop bodies and exit that iteration immediately if a break was requested. If you don't do this, the iteration will continue to run until it finishes.

To see if another iteration running in parallel has requested a break, retrieve the value of the parallel loop state's **LowestBreakIteration** property. If this returns an object whose **HasValue** property is **true**, you know that a break has been requested. You can also read the **ShouldExitCurrentIteration** property, which checks for breaks as well as other stopping conditions.

> **Note: Parallel.ForEach** also supports the loop state **Break** method. The parallel loop assigns items a sequence number, starting from zero, as it pulls them from the enumerable input. This sequence number is used as the iteration index for the **LowestBreakIteration** property.

During the processing of a call to the **Break** method, iterations with an index value less than the current index will be allowed to start (if they have not already started), but iterations with an index value greater than the current index will not be started. This ensures that all iterations below the break point will complete.

Because of parallel execution, it's possible that more than one iteration may call **Break**. In that case, the lowest index will be used to determine which iterations will be allowed to start after the break occurred.

The **Parallel.For** and **Parallel.ForEach** methods return an object of type **ParallelLoopResult**. You can find out if a loop terminated with a break by examining the values of two of the loop result properties. If the **IsCompleted** property is **false** and the **LowestBreakIteration** property returns an object whose **HasValue** property is **true**, you know that the loop terminated by a call to the **Break** method. You can query for the specific index with the loop result's **LowestBreakIteration** property. Here is an example.

```
int N = ...
var result = new double[N];

var loopResult = Parallel.For(0, N, (i, loopState) =>
{
   if (/* stopping condition is true */)
   {
      loopState.Break();
      return;
   }
   result[i] = DoWork(i);
});
```

26

```
if (!loopResult.IsCompleted &&
        loopResult.LowestBreakIteration.HasValue)
{
    Console.WriteLine(“Loop encountered a break at {0}”,
                        loopResult.LowestBreakIteration.Value;
}
```

The **Break** method ensures that data up to a particular iteration index value will be processed. Depending on how the iterations are scheduled, it may be possible that some iterations with a higher index value may have been started before the call to **Break** occurs.

There are also cases such as unordered searches where you want the loop to stop as quickly as possible after the stopping condition is met. The difference between "break" and "stop" is that, with stop, no attempt is made to execute loop iterations less than the stopping index if they have not already run. To stop a loop in this way, call the **ParallelLoopState** class's **Stop** method instead of the **Break** method. Here is an example of how to test to see if a stop occurred.

> Use **Stop** to exit a loop early when you do not care if lower-indexed iterations are allowed to run.

```
if (!loopResult.IsCompleted &&
     !loopResult.LowestBreakIteration.HasValue)
{
    Console.WriteLine(“Loop was stopped”);
}
```

The index value of the iteration that caused the stop is not available.

You cannot call both **Break** and **Stop** during the same parallel loop. You have to choose which of the two loop exit behaviors you want to use. If you do call both **Break** and **Stop** in the same parallel loop, an exception will be thrown.

External Loop Cancellation

In some scenarios, you may want to cancel a parallel loop because of an external request. For example, you may need to respond to a request from a user interface. In this variation, use a cancellation token derived from a **CancellationTokenSource** instance. Here is an example.

```
void DoLoop(CancellationTokenSource cts)
{
  CancellationToken token = cts.Token;

  var options = new ParallelOptions
                        { CancellationToken = token };

  try
  {
    Parallel.For(0, N, options, (i) =>
    {
```

```
      // ... do some work ...

      // ... optionally check to see if cancellation happened
      if (token.IsCancellationRequested)
      {
         // ... optionally exit this iteration early
         return;
      }
   });
}
catch (OperationCanceledException ex)
{
   // ... handle the loop cancellation
}
}
```

When the caller of the **DoLoop** method is ready to cancel, it invokes the **Cancel** method of the **CancellationTokenSource** that was provided as an argument to the **DoLoop** method. The parallel loop will allow currently running iterations to complete and then throw an **OperationCanceledException**. No new iterations will be started after cancellation begins.

<p style="text-align:center">External cancellation requires a cancellation token source object.</p>

If external cancellation has been signaled *and* your loop has called either the **Break** or the **Stop** method of the **ParallelLoopState** object, a race occurs to see which will be recognized first. The parallel loop will either throw an **OperationCanceledException** or it will terminate using the mechanism described in the previous section for **Break** and **Stop**.

Exception Handling

If an iteration of the loop throws an unhandled exception, a parallel loop no longer begins any new iterations. By default, iterations that are executing at the time of the exception, other than the iteration that threw the exception, will complete. After they finish, the parallel loop throws an exception in the context of the thread that invoked it.

<p style="text-align:center">Throwing an unhandled exception prevents new iterations from being started.</p>

Long-running iterations may want to test to see whether an exception is pending in another iteration. They can do this with the **ParallelLoopState** class's **IsExceptional** property. This property returns **true** if an exception is pending.

Because more than one exception may occur during parallel execution, exceptions are grouped using an exception type known as an **AggregateException**. The **AggregateException** class has an **InnerExceptions** property that contains all the exceptions that occurred during the execution of the parallel loop.

Exceptions take priority over external cancellations and terminations of a loop initiated by calling the **Break** or **Stop** methods of the **ParallelLoopState** object.

Special Handling of Small Loop Bodies

If the body of the loop performs only a small amount of work, you may find that you achieve better performance by partitioning the iterations into larger units of work.

Consider using a **Partitioner** object when you have many iterations that each perform a small amount of work.

The reason for this is that there are two types of overhead that are introduced when processing a loop: the cost of synchronizing between worker threads and the cost of invoking a delegate method. In most situations, these are negligible, but with very small loop bodies, they can be significant.

Partitioning divides data into sets of non-overlapping regions named partitions; partitions are allocated to available processors. The parallel extensions of .NET Framework 4 include support for partitioning.

The partition-based syntax is more complicated than other parallel loops in .NET, and when the amount of work in each iteration is large (or of uneven size across iterations), it may not result in better performance. Generally, you would only use the more complicated syntax after profiling or in the case where loop bodies are extremely small and the number of iterations large. Here is an example.

```
int[] result = new int[N];
Parallel.ForEach(Partitioner.Create(0, N),
    (range) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
          // very small, equally sized blocks of work
          result[i] = i * i;
        }
    });
```

The **Partitioner** class has several static methods that create objects in order to control how parallel loops subdivide their work. In this example, you can think of the result of the **Create** method as an object that acts like an instance of **IEnumerable<Tuple<int, int>>**. In other words, **Create** returns a collection of tuples (unnamed records) with two integer field values. Each tuple represents a range of values that should be processed in a single iteration of the parallel loop. By grouping iterations into ranges, you can avoid some of the overhead of a normal parallel loop. The number of ranges that will be created depends on the number of cores in your computer. The default number of ranges is approximately four times the number of cores.

If you know how big you want your ranges to be, you can use a special overload of the **Partitioner.Create** method that allows you to specify the size of each range. Here is an example.

```
int[] result = new int[1000000];
Parallel.ForEach(Partitioner.Create(0, 1000000, 50000),
    (range) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
```

```
        // small, equally sized blocks of work
        result[i] = i * i;
      }
    });
```

In this example, each range will span 50,000 index values. In other words, for a million iterations, the system will use twenty parallel iterations (1,000,000 / 50,000). These iterations will be spread out among all the available cores.

Choosing a suitable partitioning strategy may make it possible to see performance improvements even with extremely small loop bodies, as long as the number of iterations is high enough. Very small loop bodies with a small number of iterations may not benefit from parallel execution.

Custom partitioning is an extension point in the API for parallel loops. You can implement your own partitioning strategies. For more information about this topic, see the section, "Further Reading," at the end of this chapter.

Controlling the Degree of Parallelism

Although it is usually recommended to let the system manage how iterations of a parallel loop are mapped to your computer's cores, in some cases, you may want to specify how many threads should be used to process a particular parallel loop. Reducing the degree of parallelism is often used in performance testing to simulate less capable hardware. Increasing the degree of parallelism to a number larger than the number of cores can be appropriate when iterations of your loop spend a lot of time waiting for I/O operations to complete.

> You can control the maximum number of threads used concurrently by a parallel loop.

You can control the maximum number of threads used concurrently by specifying the **MaxDegreeOfParallelism** property of a **ParallelOptions** object. Here is an example.

```
var options = new ParallelOptions()
                    { MaxDegreeOfParallelism = 2};
Parallel.For(0, N, options, i =>
{
  // ... do some work ...
}
```

This loop will run using at most two threads.

You can also do this for PLINQ queries by setting the **WithDegreeOfParallelism** property of a **ParallelQuery<T>** object. Here is an example.

```
IEnumerable<T> myCollection = // ...
myCollection.AsParallel()
          .WithDegreeOfParallelism(8)
          .ForAll(obj => /* do work */);
```

This query will run with a maximum of eight threads.

30

If you specify a larger degree of parallelism, you may also want to use the **ThreadPool** class's **SetMinThreads** method so that these threads are created without delay. If you don't do this, the thread pool's thread injection algorithm may limit how quickly threads can be added to the pool of worker threads that is used by the parallel loop. It may take more time than you want to create the required number of threads.

Using Thread-Local State in a Loop Body

Occasionally, you will need to maintain thread-local state during the execution of a parallel loop. For example, you might want to use a parallel loop to initialize each element of a large array with random values. The .NET Framework's **Random** class does not support multi-threaded access. Therefore, you need a separate instance of the random number generator for each thread. Here is an example of how to do this using one of the overloads of the **Parallel.ForEach** method. The example uses a **Partitioner** object to decompose the work into relatively large pieces, because the amount of work performed by each step is small, and there are a large number of steps.

<p align="center">Thread-local state is available if your loop body needs it.</p>

```
int numberOfSteps = 10000000;
double[] result = new double[numberOfSteps];

// ForEach<TSource, TLocal>(
//      OrderablePartitioner<TSource> source,
//   ParallelOptions parallelOptions,
//   Func<TLocal> localInit,
//   Func<TSource, ParallelLoopState, long, TLocal, TLocal> body,
//   Action<TLocal> localFinally);
Parallel.ForEach(
    // source
    Partitioner.Create(0, numberOfSteps),

    // parallelOptions
    new ParallelOptions(),

    // localInit
    () => { return new Random(MakeRandomSeed()); },

    // body
    (range, loopState, _, random) =>
    {
      for (int i = range.Item1; i < range.Item2; i++)
        result[i] = random.NextDouble();
      return random;
    },

    // localFinally
    _ => {});
```

The **Parallel.ForEach** loop will create a new instance of the **Random** class for each of its worker threads. This instance will be passed as an argument to each partitioned iteration. Each partitioned iteration is responsible for returning the next value of the thread-local state. In this example, the value is always the same object.

Using a Custom Task Scheduler for a Parallel Loop

In some cases, you may want to substitute custom task scheduling logic for the default task scheduler, which uses **ThreadPool** worker threads. This variation can occur, for example, when you want to enable a maximum degree of parallelism that operates across multiple loops instead of over only a single loop. It might also occur if you want to use a dedicated set of threads to process the loop instead of pulling them from the shared thread pool. These are just a few of the advanced scenarios where a custom task scheduler might be needed.

> You can use a custom task scheduler if necessary. This is one of the parallel loop's extension mechanisms.

To specify a custom task scheduler, set the **TaskScheduler** property of a **ParallelOptions** object. Here is an example.

```
TaskScheduler myScheduler = ...
var options = new ParallelOptions()
                    { TaskScheduler = myScheduler};
Parallel.For(0, N, options, i =>
{
  // ... do some work using tasks managed by myScheduler ...
}
```

For more information about tasks and task schedulers, see the section, "Implementation Notes," in Chapter 3, "Parallel Tasks."

It isn't possible to specify a custom task scheduler for PLINQ queries.

Mixing the Parallel Class and PLINQ

PLINQ queries are represented by instances of the **ParallelQuery<T>** class. This class implements the **IEnumerable<T>** interface, so it's possible to use a PLINQ query as the source collection for a **Parallel.ForEach** loop, but it's not recommended. Instead, use PLINQ's **ForAll** extension method for the **ParallelQuery<T>** instance. PLINQ's **ForAll** extension method performs the same kind of iteration as **Parallel.ForEach**, but it avoids unnecessary merge and repartition steps that would otherwise occur.

> Use **ForAll** if you need to iterate over a PLINQ result. Don't use **ForEach** in this case.

Here is an example of how to use the **ForAll** extension method.

```
var q = (from d in data.AsParallel() ... select d => F(d));

q.ForAll(item =>
{
```

```
  // ... process item
});
```

Parallel Loops with Custom Iteration

Sometimes you want to apply a parallel loop to data structures that don't have standard iterators. For example, consider a binary tree.

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

You can implement a custom iterator for the **Tree<T>** class. This iterator can be used by a parallel loop to access the tree's nodes.

Custom iterators can be used with parallel loops.

```
public static IEnumerable<Tree<T>> Iterate<T>(Tree<T> root)
{
    var queue = new Queue<Tree<T>>();
    queue.Enqueue(root);
    while (queue.Count > 0)
    {
        var node = queue.Dequeue();
        yield return node;
        if (node.Left != null) queue.Enqueue(node.Left);
        if (node.Right != null) queue.Enqueue(node.Right);
    }
}
```

The custom iterator is a powerful addition to the **Parallel.ForEach** method.

```
Tree<T> myTree = ...

Parallel.ForEach(Iterate(myTree), node =>
{
   // ... process node in parallel
});
```

> **Note:** A more advanced variation would create partitions based on subtrees. For example, you could implement a **Partitioner** object that divided the tree into subtrees whose roots corresponded to nodes of a certain depth in the original tree. This would be especially efficient if, for example, the locations in memory of the subtrees improved memory cache performance.

Overriding the Default Behavior of IList<T>

In certain rare cases, the parallel loop's default handling of the **IList<T>** type may not be what you want. This can occur when the **IList<T>** implementation has unfavorable random-access performance

characteristics or when there are race conditions that result from multiple threads attempting to perform lazy loading at the same time.

This variation shows you how to override **Parallel.ForEach**'s default handling of a source that provides the **IList<T>** interface.

You can control whether a parallel loop uses **IList** or **IEnumerable** in cases where both are available. This is needed only in certain rare cases.

The **Parallel.ForEach** method requires its sources to provide the **IEnumerable<T>** interface; however, it also checks to see whether its source provides the **IList<T>** interface. In most cases, using **IList<T>** to access elements of the collection results in more efficient partitioning strategies because it provides random (that is, indexed) access to the items in the collection. In contrast, **IEnumerable<T>** only supports access that walks the collection using the **MoveNext** method to retrieve successive elements. In almost all cases, the default behavior results in better performance.

A few types that provide **IList<T>** do so in a way that makes concurrent indexing an expensive or even incorrect operation. For these types, **MoveNext** is a better accessor. You can use a **Partitioner** object to force **Parallel.ForEach** to use the **IEnumerable<T>** interface, even if **IList<T>** is available. Here is an example.

```
IEnumerable<T> source = ...;

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

> **Note:** The **System.Data.Linq.EntitySet<TEntity>** class is an example of a type that should use **IEnumerable<T>** for parallel iteration. This is because of the type's lazy loading semantics. (If two threads attempt to access the indexer concurrently, the **EntitySet<TEntity>** instance may become corrupted.)

Design Notes

The Parallel Loop pattern emphasizes decomposition by data, not tasks. It is designed to scale to any number of cores.

The implementations of the Parallel Loop pattern in the .NET Framework's **Parallel** class and in PLINQ contain sophisticated algorithms for dynamic work distribution. These loops automatically adapt to the workload and to a particular computer.

Mechanisms, such as locks, are still needed to protect concurrent modifications for programs that use shared memory. In general, avoid writing to variables that are shared by iterations of the loop. When this is unavoidable, overloads of the **Parallel.For** and **Parallel.ForEach** methods offer abstractions, such as thread-local state and a thread finalizing phase, that help with synchronization.

Some parallel loops may be more compute-bound than others. The .NET Framework adapts dynamically to this situation by allowing the number of worker threads for a parallel loop to change over time. The load- balancing algorithm also uses a partitioning technique where the size of units of work increases over time. This approach processes both small and large ranges with minimal overhead.

Implementation Notes

When loop bodies are not fully independent of each other, it may still be possible to use parallel loops. However, in these cases, it is up to the developer to ensure that shared variables are properly protected and synchronized. The performance will not scale proportionately with the number of cores. For examples of common types of loop body dependencies, see Chapter 4, "Parallel Aggregation with Map-Reduce."

The presence of multiple cores does not guarantee that they will be used by the parallel loop. For example, cores may be unavailable because they are being used for other concurrent operations. The allocation of cores is managed by the .NET Framework.

The term *degree of parallelism* can be used in two senses. In the simplest case, it refers to the number of cores that are used to process iterations simultaneously. However, .NET also uses this term to refer to the number of threads used by the parallel loop. For example, the **MaxDegreeOfParallelism** property of the **ParallelOptions** object refers to the maximum number of threads used concurrently.

For efficient use of hardware resources, the number of threads is often greater than the number of available cores. For example, parallel loops may use additional threads in cases where there are blocking I/O operations that do not require CPU resources to run.

> If you are using the **Parallel.For** method, remember that duplicates in the enumeration are not allowed because this causes an unsafe race condition. Potentially, the parallel threads could try to update the same object.

The **Parallel** class and PLINQ work on slightly different threading models in the .NET 4 Framework. PLINQ uses a fixed number of threads to execute a query; by default, it uses the number of logical cores in the computer, or it uses the value passed to the **WithDegreeOfParallelism** method if one was specified.

Conversely, by default, the **Parallel.ForEach** and **Parallel.For** methods can use a variable number of threads. The number of threads may grow if some iterations take a long time.

The .NET Framework parallel extensions allow you to nest parallel loops. In this situation, the run-time environment coordinates the use of processor resources. Another option is to combine the nested loops into a single loop. Nesting or unrolling loops is appropriate in cases where the amount of work in each top-level iteration is either very large or of uneven size.

Anti-Patterns

Anti-patterns are cautionary tales. They highlight issues that need to be carefully considered and problem areas.

Step Size Other Than One

If you need to use a step size other than one, transform the loop index in the body of the loop or use parallel for each with the values you want. Also, be aware that a step size other than one often indicates a data dependency, such as the calculation of a sum. Carefully analyze such a computation before you convert it into a parallel loop.

If your loop has a negative step size (that is, it iterates from high to low values), the ordering of the loop is probably significant, and the iterations may not be independent. Programmers rarely iterate from high to low values unless the order matters.

Loop Body Dependencies

Incorrect analysis of loop dependencies is a frequent source of software defects. It is important to do this carefully.

Be careful that your loop body does not contain hidden dependencies. This is a mistake that's easy to make. The case of trying to share an instance of a class such as **Random**, which is not thread safe, across parallel iterations is an example of a subtle dependency.

If the loop body is not independent, for example, when you use an iteration to calculate a sum, you may need to apply the variation on a parallel loop that's described in Chapter 4, "Parallel Aggregation with Map-Reduce."

Problems with Small Loop Bodies

You will probably not get performance improvements if you use a parallel loop for very small loop bodies with only a limited number of data elements to process. In this case, the overhead required by the parallel loop itself will dominate the calculation. Simply changing every sequential **for** loop to **Parallel.For** will not necessarily produce good results.

Instead, the best place to replace a sequential loop with a parallel loop is at the top level of your application, as in the credit review sample. In other words, it's better to introduce parallel steps at a relatively coarse level of granularity if you have the choice. The only reason that large steps would be inappropriate is if there are too few of them, or if the steps are of such uneven size that processing the last step leaves many cores idle for a long time.

Parallel loops may be appropriate for small loop bodies. It depends on the number of iterations. If you have very small loop bodies but a large number of data elements, a parallel loop may still improve performance. For more information, see "Special Handling for Small Loop Bodies" in the section, "Variations," later in this chapter.

After you modify your program, performance testing is the only way to guarantee that you are achieving the expected speedup. For more information, see Appendix B, "Profiling and Debugging."

Using Long-Running Iterations

Be careful if you use parallel loops with steps that take a long time. This can occur with I/O-bound workloads as well as lengthy computations. If the loops take a long time, you may experience an unbounded growth of worker threads due to a heuristic for preventing thread starvation that is used by the .NET **ThreadPool** class's thread injection logic. This heuristic steadily increases the number of worker threads when work items of the current pool run for long periods of time. The motivation is to add more threads in cases where everything in the thread pool is blocked. Unfortunately, if work is actually proceeding, more threads may not necessarily be what you want. The .NET Framework can't distinguish between these two situations.

You can limit the number of threads with the **ParallelOptions** class if you observe, through profiling, that the number of threads chosen by the system is too large. This will only be a potential issue in calculations that last a few minutes or more. You can also limit the number of threads with the **SetMaxThreads** method of the **ThreadPool** class. Generally, using **ParallelOptions** is preferred because **SetMaxThreads** has a global, process-wide effect that you may not want.

Processor Oversubscription and Undersubscription

Arbitrarily increasing the degree of parallelism puts you at risk of *processor oversubscription*, a situation that occurs when there are many more compute-intensive worker threads than there are cores. Tests have shown that, in general, the optimum number of worker threads for a parallel task equals the number of available cores divided by the average fraction of CPU utilization per task. In other words, with four cores and 50 percent average CPU utilization per task, you need eight worker threads for maximum throughput. Increasing the number of worker threads above this number begins to incur extra cost from additional context switching without any improvement in processor utilization.

On the other hand, arbitrarily restricting the degree of parallelism can result in *processor undersubscription*. Having too few tasks misses an opportunity to effectively use the available processor cores. You might restrict the degree of parallelism if you know that other tasks in your application are also running in parallel.

In most cases, the built-in load balancing algorithms in the .NET Framework are the most effective way to manage these tradeoffs. They coordinate resources among parallel loops and other tasks that are running concurrently.

Parallelism and Server Applications

Be extremely careful about specifying an increased degree of parallelism in server applications, such as those running on ASP.NET. In the server applications, multiplying the number of threads needed by the thousands of users sending incoming requests may overload even the most powerful server.

37

Related Patterns

The Parallel Loop pattern is sometimes known as the Map pattern, especially when the operation returns a value. In this case, the elements of the input collection are "mapped" or transformed into other values. The use of the term "map" is especially common in functional languages, such as F#.

You will sometimes see the Parallel Loop pattern referred to as SPMD which stands for Single Program Multiple Data. This is a synonym of the term "data parallelism."

You can contrast data parallelism with task parallelism, which is described in Chapter 3, "Parallel Tasks." Data parallelism applies a single operation to a range of inputs, while task parallelism invokes multiple, distinct operations.

The Parallel Loop pattern is the basis of the parallel aggregation pattern, which is the subject of Chapter 4, "Parallel Aggregation with Map-Reduce."

Exercises

1. Which of the following problems could be solved using the parallel loop techniques taught in this chapter?

    a. Sorting an array of numbers.

    b. Putting the words in each line in a text file in alphabetic order.

    c. Adding together all the numbers in one collection, to obtain a single sum.

    d. Adding numbers from two collections pair-wise, to obtain a collection of sums.

    e. Counting the total number of occurrences of each word in a collection of text files.

    f. Finding the word that occurs most frequently in each file in a collection of text files.

2. Choose a suitable problem from Exercise 1. Code three solutions, using a sequential loop, a parallel loop, and PLINQ.

3. In the credit review example, what is the type of account? What is the type of **account.AllAccounts**? Of **account.AllAccounts.AsParallel()**?

4. Is it possible to use a PLINQ query as the source for a **Parallel.ForEach** loop? Is this recommended? Explain your answers.

5. Do a performance analysis of the credit review example code on the CodePlex site. Use command line options to independently vary the number of iterations (the number of accounts) and the amount of work done in the body of each iteration (the number of months in the credit history). Record the execution times reported by the program for all three versions, using several different combinations of numbers of accounts and months. Repeat the tests on different computers with different numbers of cores and with different execution loads (from other applications).

6. Modify the credit review example from CodePlex so you can set **MaxDegreeOfParallelism**. Observe the effect of different values for this option on execution time, when running on computers with different numbers of cores.

7. When is it appropriate to write a custom partitioner? (Hint: review the binary tree example.)

Further Reading

The examples in this book use features and libraries of recent C# versions. Some helpful books are Bishop (2008) and Albahari and Albahari (2010).

The book by Mattson, et al. (2004) is a survey of software design patterns for parallel programming that is not specialized for a particular language or library.

Many of the cases in the Variations section of this chapter are from Toub (2009).

For more information about partitioners, refer to the in-depth article on this subject in Toub (2010). Toub, Stephen. "Custom Parallel Partitioning With .NET 4" Dr. Dobbs online content 4/26/2010. http://www.drdobbs.com/visualstudio/224600406.

3 Parallel Tasks

Chapter 2 shows how you can use a parallel loop to apply a single operation to many data elements. This is data parallelism. This chapter explains what happens when there are distinct asynchronous operations that can run simultaneously. In this situation, you can temporarily *fork* a program's flow of control with tasks that can potentially execute in parallel. This is task parallelism.

Parallel tasks are asynchronous operations that can run at the same time. This approach is also called task parallelism.

> **Note:** The parallel task pattern is related to what is sometimes called the fork/join pattern.

In other words, data parallelism occurs when a *single operation* is applied to many inputs. Task parallelism uses *multiple operations*, each with its own input.

With task parallelism, new tasks are queued and only start running as cores become available. As long as there are enough tasks, the program's performance scales with the number of available cores. This is how tasks embody the concept of potential parallelism that was introduced in chapter 1.

In the .NET Framework tasks are implemented by the **Task** class in the **System.Threading.Tasks** namespace. You use the **Task.Factory** object to create a task. You can wait for a task to complete by invoking the task's **Wait** method. It is also possible to wait for multiple tasks to complete.

*Parallel tasks in .NET are implemented by the **Task** class.*

An important aspect of task-based applications s is how they handle exceptions. An exception that occurs during the execution of a task is deferred until the task's **Wait** method is called or its **Exception** property is read. At that time the exception is rethrown in the calling context. This allows you to use the same exception handling approach in parallel programs as you use in sequential programs.

*Tasks in .NET defer exceptions and rethrow them when the **Wait** method is invoked.*

The Basics

Each task is a sequential operation; however, tasks can often run in parallel. In .NET, a task is also an object with properties and methods of its own. Here is some sequential code:

```
DoLeft();
DoRight();
```

Let's assume that the methods **DoLeft** and **DoRight** are independent. This means that neither method writes to memory locations or files that the other method might read. Because the methods are independent, you can use the **Invoke** method of the **Parallel** class to call them in parallel. This is shown in the following code.

```
Parallel.Invoke(DoLeft, DoRight);
```

Parallel.Invoke is the simplest expression of the parallel task pattern. It creates new parallel tasks for each delegate method that is in its **params** array argument list. The **Invoke** method returns when all the tasks are finished.

> **Parallel.Invoke** is a method in .NET that creates parallel tasks and waits for them to complete.

You can't assume that all parallel tasks will immediately run. Depending on the current work load and system configuration, tasks might be scheduled to run one after another, or they might run at the same time. See the Design Notes section of this chapter for more information about how tasks are scheduled.

The delegate methods of **Parallel.Invoke** can either complete normally or finish by throwing an exception. Any exceptions that occur during the execution of **Parallel.Invoke** are deferred and rethrown when all tasks have finished. All exceptions are rethrown as inner exceptions of an **AggregateException** instance. See Exception Handling in this chapter for more information and a code example.

Internally, **Parallel.Invoke** creates new tasks and waits for them. It uses methods of the **Task** class to do this.Here's an example.

```
Task t = Task.Factory.StartNew(DoLeft);
DoRight();

t.Wait();
```

The **StartNew** method of the **TaskFactory** class creates and schedules a new task that executes the delegate method that is given as its argument. You wait for a parallel task to complete by calling its **Wait** method.

When you use **StartNew** to create a task, the new task is added to a work queue for eventual execution, but it does not start to execute until its task scheduler takes it out of the work queue, which could be at some point in the future.

> Use the **StartNew** factory method to create a task and schedule it for execution. Call the task's **Wait** method if you want to wait for the task to complete before proceeding.

The previous example only needed one task to express all of its potential parallelism. In other cases, more than one task might be needed. For example, suppose that there is also a **DoCenter** method.

```
Task t1 = Task.Factory.StartNew(DoLeft);
Task t2 = Task.Factory.StartNew(DoRight);
DoCenter();

Task.WaitAll(t1, t2);
```

There are now three operations that can potentially occur at the same time. Therefore, you start two tasks, perform the third operation inline and when it finishes, wait for the parallel tasks to complete. The static method **WaitAll** of the **Task** class allows you to wait for more than one task to complete.

> Use **Task.WaitAll** to wait for more than one task to complete before proceeding.

41

The examples you've seen so far are simple, but they're powerful enough to handle many scenarios. See the Variations section in this chapter for more ways to use tasks.

An Example

An example of task parallelism is an image processing application where images are created with layers. Separate images from different sources are processed independently, and then combined using a process called *alpha blending*. This process superimposes semitransparent layers to form a single image.

The source images that are combined are different, and different image processing operations are performed on each of them. This means that the image processing operations must be performed separately on each source image and must be complete before the images can be blended. In the example, there are only two source images, and the operations are simple: conversion to gray scale and rotation. In a more realistic example, there might be more source images and more complicated operations.

Here is the sequential code. The source code for the complete example is located at http://parallelpatterns.codeplex.com.

```
static int SeqentialImageProcessing(Bitmap source1, Bitmap source2,
                                    Bitmap layer1, Bitmap layer2,
                                    Graphics blender)
{
  SetToGray(source1, layer1);
  Rotate(source2, layer2);
  Blend(layer1, layer2, blender);

  return source1.Width;
}
```

In this example, **source1** and **source2** are bitmaps that are the original source images, **layer1** and **layer2** are bitmaps that have been prepared with additional information needed to blend the images, and **blender** is a **Graphics** instance that performs the blending, and references the bitmap with the final blended image. Internally, **SetToGray**, **Rotate**, and **Blend** use methods from the .NET **System.Drawing** namespace to perform the image processing. The last statement returns an integer that the caller uses to print a progress message.

The **SetToGray** and **Rotate** methods are entirely independent of each other. This means that you can execute them in separate tasks. If two or more cores are available, the tasks might run in parallel and the image processing operations might complete in less elapsed time than a sequential version would.

The parallel code uses tasks explicitly.

```
static int ParallelTaskImageProcessing(Bitmap source1,
   Bitmap source2, Bitmap layer1, Bitmap layer2,
   Graphics blender)
```

```
{
  Task toGray = Task.Factory.StartNew(() =>
     SetToGray(source1, layer1));
  Task rotate = Task.Factory.StartNew(() =>
     Rotate(source2, layer2));
  Task.WaitAll(toGray, rotate);
  Blend(layer1, layer2, blender);
  return source1.Width;
}
```

This code calls **Task.Factory.Create** to create and run two tasks that execute **SetToGray** and **Rotate**, and calls **Task.WaitAll** to wait for both tasks to complete before blending the processed images.

You can also use **Parallel.Invoke** to achieve parallelism.

```
static int ParallelInvokeImageProcessing(Bitmap source1, Bitmap source2,
                                         Bitmap layer1, Bitmap layer2,
                                         Graphics blender)
{
  Parallel.Invoke(
    () => SetToGray(source1, layer1),
    () => Rotate(source2, layer2));
  Blend(layer1, layer2, blender);
  return source1.Width;
}
```

Here the tasks are identified implicitly by the arguments to **Parallel.Invoke**. This call does not return until all of the tasks complete.

The performance of the parallel versions is significantly better than the sequential version.

Variations

The parallel task pattern has several variations. This section describes some of these variations as they apply to .NET.

Canceling a Task

In .NET, a cancellation request does not forcibly end a task. Instead, tasks use a cooperative cancellation model. This means that a task must poll for the existence of a cancellation request at appropriate intervals and then shut itself down by calling back into the library.

<div align="center">Tasks in .NET use a cooperative cancellation model.</div>

The .NET Framework provides data types that separate the ability to check for cancellation requests from the ability to request cancellation. Instances of the **CancellationTokenSource** class are used to request cancellation, while **CancellationToken** values indicate whether cancellation has been requested. Here's an example:

```
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken token = cts.Token;

Task myTask = Task.Factory.StartNew(() =>
{
    for (...)
    {
        token.ThrowIfCancellationRequested();
        // ...
    }
}, token);

// ... elsewhere ...
cts.Cancel();
```

The **CancellationTokenSource** object contains a cancellation token. This token is used in two places. It is passed as an argument to the **StartNew** factory method, and its **ThrowIfCancellationRequested** method is invoked to detect and process a cancellation request. Although the example does not show it, you can also read the cancellation token's **IsCancellationRequested** property to test for a cancellation request. For example, you would do this if you need to perform local cleanup actions for a task that is in the process of being canceled.

If the cancellation token indicates that a cancellation has been requested, the **ThrowIfCancellationRequested** method creates an **OperationCanceledException** instance and passes in the cancellation token. It then throws the exception. This exception is the signal that notifies the .NET Framework that the task has been canceled; therefore, the operation canceled exception should not be handled by user code within the task. If you follow the steps that have been described in this section, the task will be stopped and its **Status** property will be set to the enumerated value **TaskStatus.Canceled**.

Checking for cancellation within a loop with many iterations that perform a small amount of work can negatively affect your application's performance, but checking only infrequently for cancellation can introduce unacceptable latency in your application's response to cancellation requests. Checking for cancelation more than once per second and less than ten times per second is recommended. Profiling your application can help you determine where it is best to test for cancellation requests in your code.

For a little fine print about the behavior of cancellation, see the Task Life Cycle and the Unhandled Task Exceptions sections later in this chapter.

Handling Exceptions

If the execution of a **Task** object's user delegate throws an unhandled exception, the task terminates and its **Status** property is set to the enumerated value **TaskStatus.Faulted**. The unhandled exception is temporarily unobserved. This means that the Task Parallel Library catches the exception and records its details in internal data structures associated with the task object. Recovering a deferred exception and

embedding it in a new exception is known as "observing an unhandled task exception."In many cases, unhandled task exceptions will be observed in a different thread than the one that executed the task.

Ways to Observe an Unhandled Task Exception

There are several ways to observe an unhandled task exception.

Invoking the faulted task's **Wait** method causes the task's unhandled exception to be observed.

Static methods of the **Task** class such as **WaitAll** allow you to observe the unhandled exceptions of more than one task with a single method invocation.

Getting the **Exception** property of a faulted task causes the task's unhandled exception to be observed.

Special handling occurs if a faulted task's unhandled exceptions are not observed by the time the task object is garbage-collected. See the Unobserved Task Exceptions section in this chapter for more information.

Aggregate Exceptions

In some cases, such as when you are waiting for more than one task, there can be multiple unhandled exceptions that need to be observed. For this reason the run time collects the unhandled task exceptions and wraps them in an **AggregateException** instance. The framework throws the aggregate exception in the context of the observer (that is, in the waiting thread). You can catch the aggregate exception and examine its **InnerExceptions** property to handle the original exceptions individually. Even if the unhandled exception from just one task is observed, it is still wrapped in an **AggregateException**.

The Task Parallel Library uses **AggegrateException** objects to propagate exceptions.

Like all exceptions that are raised inside of a task, an **OperationCanceledException** instance that is created by canceling a task will become an inner exception of an **AggregateException** thrown by the task in the context of the thread that calls the task's **Wait** method. However, if you follow all of the cancellation rules that are described in the Canceling a Task section of this chapter, the task's final status will be **TaskStatus.Canceled** instead of **TaskStatus.Faulted**.

The Handle Method

The **AggregateException** class has several helper methods to make handling inner exceptions easier. The **Handle** method of the **AggregateException** class invokes a user-provided delegate method for each of the inner exceptions of an aggregate exception. The delegate returns **true** if it has handled inner exception and **false** if it has not handled the inner exception. Here is an example:

Use the Handle method to iterate through inner exceptions.

```
try
{
  Task t = Task.Factory.StartNew( ... );
  // ...
  t.Wait();
}
```

45

```
catch (AggregateException ae)
{
  ae.Handle(e =>
  {
    if (e is MyException)
    {
      // ... handle exception ...
      return true;
    }
    else
    {
      return false;
    }
  });
}
```

When the **Handle** method invokes the user-provided delegate for each inner exception, it keeps track of the results of the invocation. After it has processed all exceptions, it checks to see if the results for one or more of the inner exceptions were **false**, which indicates that they have not been handled. If there are any unhandled exceptions, the **Handle** method creates and throws a new aggregate exception that contains them as inner exceptions.

The Flatten Method

The **Flatten** method of the **AggregateException** class is useful when tasks are nested within other tasks. In this case, it is possible that an aggregate exception can contain other aggregate exceptions as inner exceptions. The **Flatten** method produces a new exception object that merges the inner exceptions of all nested aggregate exceptions into the inner exceptions of the top-level aggregate exception. In other words, it converts a tree of inner exceptions into a sequence of inner exceptions. It is typical to use the **Flatten** and **Handle** methods together. This is shown in the following example.

The **Flatten** method transforms a tree of exceptions into a sequence.

```
try
{
  Task t1 = Task.Factory.StartNew(() =>
  {
    Task t2 = Task.Factory.StartNew(() =>
    {
      // ...
      throw new MyException();
    });
    // ...
    t2.Wait();
  });
  // ...
  t1.Wait();
}
catch (AggregateException ae)
```

```
{
  ae.Flatten().Handle(e =>
  {
    if (e is MyException)
    {
      // ... handle exception ...
      return true;
    }
    else
    {
      return false;
    }
  });
}
```

Waiting for the First Task to Complete

The **Task** class allows you to wait for a set of tasks to complete by invoking the **Task.WaitAll** method. However, you can also wait for the first task to complete by calling the **Task.WaitAny** method. This variation occurs when you execute more than one asynchronous operation in parallel but need just one of the operations to complete before proceeding. Imagine, for example, that you use three different search tasks to search for an item. Once the fastest task finds the item, you don't need to wait for the other searches to complete. In cases like this you wait for the first task to complete and usually cancel the remaining tasks. However, you should always observe any exceptions that might have occurred in any of the tasks.

> Use **Task.WaitAny** as a way to wait for at least one task in a set of tasks to complete.

Here's an example.

```
// create cancellation hooks
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken token = cts.Token;

// create a factory that uses custom options for new tasks
TaskFactory factory = new TaskFactory(token,
    TaskCreationOptions.PreferFairness,
    TaskContinuationOptions.None,
    TaskScheduler.Default);

// start three parallel tasks
Task[] tasks = new Task[]{
    factory.StartNew(() => SearchLeft(token)),
    factory.StartNew(() => SearchRight(token)),
    factory.StartNew(() => SearchCenter(token))};
try
{
    // wait for the quickest task to complete
    Task.WaitAny(tasks);
```

```
  }
  finally
  {
    // signal cancellation for tasks that did not win the race
    cts.Cancel();
    try
    {
      // wait for cancellation to take effect and
      // observe all unhandled exceptions
      Task.WaitAll(tasks);
    }
    catch (AggregateException ae)
    {
      // filter out exceptions caused by cancelation
      // (including those from any nested tasks)
      // and rethrow if any unhandled exceptions remain;
      // otherwise, proceed without exception
      ae.Flatten().Handle(e =>
                            (e is OperationCanceledException));
    }
  }
```

This example executes three user delegates in parallel. Only one of the delegates needs to complete for the operation to succeed, so the tasks that did not finish first are cancelled.

The **WaitAny** method is not guaranteed to observe unhandled task exceptions in all of the input tasks. Therefore, this example also makes a call to the **WaitAll** method after canceling the tasks that did not finish first. This causes all unhandled task exceptions to be observed in the current thread. The code catches the aggregate exception that is thrown by **WaitAll** and removes the **OperationCanceledException** instances. If there are no remaining unhandled exceptions, the code proceeds normally. Otherwise, the remaining unhandled exceptions are rethrown as the inner exceptions of a new aggregate exception.

If you use the **WaitAny** method, you often also create tasks with the **TaskCreationOptions**.**PreferFairness** option. This option is a tells the task scheduler that you prefer first-in first-out (FIFO) execution of tasks, which is more appropriate when lower worst-case latency is preferred over maximum throughput. See Design Notes in this chapter for information about task scheduling.

This example also shows how to create a task factory object with all of the desired task creation options specified just once. You can use the new task factory to create as many tasks as you want without having to specify the task creation options again.

48

Creating Tasks with a Custom Task Scheduler

You can customize the details of how tasks in .NET are scheduled and run by overriding the default task scheduler that is used by the task factory methods. For example, you can provide a custom task scheduler as an argument to one of the overloaded versions of the **Task.StartNew** method.

Unless you specify otherwise, any new tasks will use the default task scheduler, which is the value of the static property **TaskScheduler.Default**. The default task scheduler is tightly integrated with the .NET thread pool, and it uses sophisticated algorithms to optimize task throughput in a wide variety of operating conditions. It is a good choice for most applications. See The Default Task Scheduler in this chapter for more information.

There are some cases where you might want to override the default. The most common case occurs when you want your task to run in a particular thread context. This can happen when you use objects provided by libraries, such as Windows Presentation Foundation (WPF), that impose thread affinity constraints.

The static method **TaskScheduler** class's **FromCurrentSynchronizationContext** method returns a task scheduler object that always schedules tasks in the thread that created the scheduler object. If you provide this task scheduler as an argument to a task creation method, the new task will run on the desired thread.

You can implement your own task scheduler class. See Task Schedulers in this chapter for more information.

Design Notes

This section describes some of the design considerations that were used to create the Task Parallel Library.

Tasks and Threads

When a task begins to run, the applicable task scheduler invokes the task's user delegate in a thread of its choosing.

There is a guarantee that the action will not migrate among threads at run time. This is a useful guarantee because it lets you use thread-affine abstractions such as Windows critical sections without having to worry, for example, that the **LeaveCriticalSection** functionwill be executed in a different thread than the **EnterCriticalSection** function.

Task Life Cycle

The **Status** property of a **Task** instance tracks its life cycle. The following diagram shows the life cycle of the tasks that are described in this chapter:
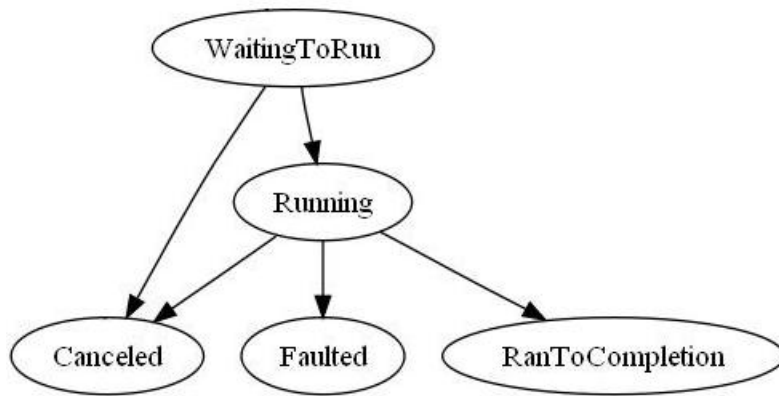
Figure 4

The **TaskFactory.StartNew** method creates and schedules a new task, which results in the status **TaskStatus.WaitingToRun**. Eventually the **TaskScheduler** instance that is responsible for managing the task begins to execute the task's user delegate on a thread of its choosing. At this point the task transitions to **TaskStatus.Running**. Once the task has begun to run, it has three possible dispositionsoutcomes. If the task's user delegate exits normally, the task transitions to **TaskStatus.RanToCompletion**. If the task's user delegate throws an unhandled exception, then the task's status becomes **Task.Faulted**.

It' is also possible for a task to end in **TaskStatus.Canceled**. For this to occur, you must pass a **CancellationToken** as an argument to the factory method that created the task. If that token signals a request for cancellation *before* the task begins to execute, then the task will not be allowed to run. The task's **Status** property will transition directly to **TaskStatus.Canceled** without ever invoking the task's user delegate. If the token signals a cancellation request *after* the task begins to execute, the task's **Status** property will only transition to **TaskStatus.Canceled** if the user delegate throws an **OperationCanceledException** and that exception's **CancellationToken** property contains the token that was given when the task was created.

Later in this book you will see two variations of the task life cycle. Chapter 5, "Futures" introduces continuation tasks whose life cycle also depends on antecedent tasks. Chapter 6, "Dynamic Task Parallelism" describes tasks whose life cycle depends on subtasks that have been created with the **AttachedToParent** task creation option.

There is one more task status, **TaskStatus.Created**. This is the status of a task immediately after it is created by the **Task** class's constructor; however, it is recommended that you always use a factory method to create tasks and instead of using the **new** operator. Tasks with **TaskStatus.Created** are outside of the scope of this book.

The Default Task Scheduler

This section is a behind-the-scenes look at the .NET 4 implementation of task scheduling. Although the material in this section may be helpful for understanding the performance characteristics that you will

observe when using TPL and PLINQ, be aware that the scheduling algorithms described here represent an implementation choice. They are not constraints imposed by TPL itself. Also, future versions of .NET might optimize task execution differently.

In .NET 4 the default task scheduler is tightly integrated with the thread pool. If you use the default task scheduler, the worker threads that execute parallel tasks are managed by the .NET **ThreadPool** class. Generally, there are at least as many worker threads as cores on your computer. When there are more tasks than available worker threads, some tasks will be queued and wait until the thread pool provides an available worker thread.

An example of this approach is the thread pool's **QueueUserWorkItem** method. In fact, you can think of the default task scheduler as an improved thread pool where work items return a handle that a thread can use as a wait condition, and where unhandled task exceptions are forwarded to the wait context. The handles are called tasks, and the wait condition occurs when you call the task's **Wait** method. In addition to these enhancements, the default thread scheduler is capable of better performance than the thread pool alone as the number of cores increases. Here's how this works.

The Thread Pool

In its simplest form a thread pool consists of a global queue of pending work items and a set of threads that process the work items, usually on a first-in first-out (FIFO) basis. This is shown in the following diagram.

A Thread Pool
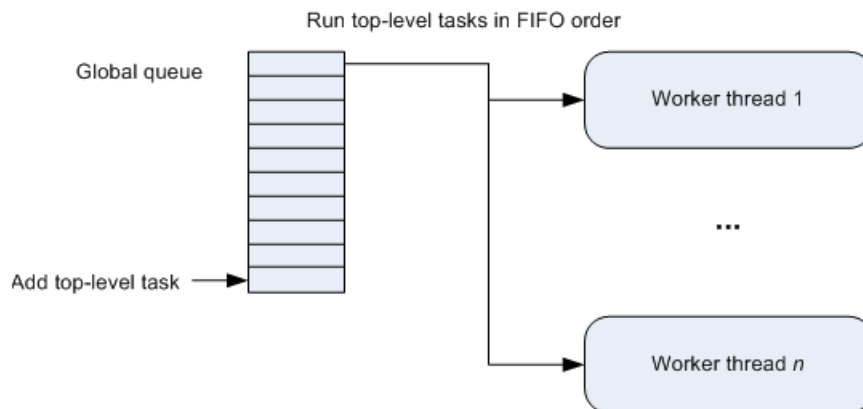
Figure 1

Thread pools have problems scaling to large numbers of cores. The main reason is that the thread pool has a single, global work queue. The global queue can be accessed by only one thread at a time, and this can become a bottleneck. When there are only a few, coarse-grained work items and a limited number of cores, the synchronization overhead of a global queue (that is, the cost of ensuring that only one thread at a time has access) is small. For example, the overhead is negligible when the number of cores is four or fewer and when each work item takes many thousands of CPU cycles. However, as the number of cores increases and the amount of work that you want to do in each work item decreases (due to the

finer-grained parallelism that is needed to exploit more of the cores), the synchronization cost of the traditional thread pool design begins to dominate.

Synchronization is an umbrella term that includes many techniques for coordinating the activities of multi-threaded applications. Locks are a familiar example of a synchronization technique. Threads use locks to make sure that they don't try to modify a location of memory at the same time as another thread. All types of synchronization have the potential of causing a thread to block (that is, to do no work) until a condition is met.

Tasks in .NET are designed to scale to large numbers of cores, perhaps as many as 256. They are also lightweight enough to perform very small units of work, in the hundreds or low thousands of CPU cycles. In .NET it's possible for an application to run efficiently with 100,000 tasks on tens or hundreds of cores. To handle this kind of scale, a more decentralized approach to scheduling than one that uses a single global queue is needed.

Decentralized Scheduling Techniques

The Task Parallel Library provides local task queues for each worker thread in the .NET thread pool. Local task queues distribute the responsibility for queue management and avoid much of the serial access required by a global queue of work items. By giving different parts of the application their own work queues, the system eliminates the central bottleneck. This is shown in the following figure.

Per-Thread Local Task Queues
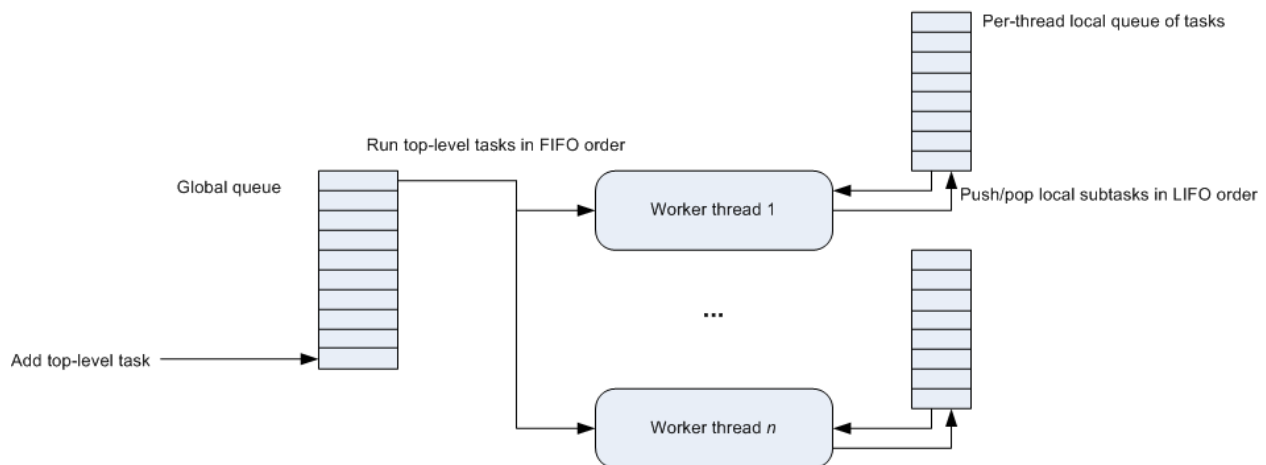
Figure 2

You can see that there are as many task queues as there are worker threads, plus one global queue shared by all threads. All of these queues operate concurrently. The basic idea is that when a new task needs to be added it can sometimes be added to a thread-local queue instead of to the global queue, and when a thread is ready for a new task to run it can sometimes find one waiting in its local queue rather than having to go to the global queue. Of course, any work that comes from a thread that is not

one of the thread pool's worker threads still has to be placed on the global queue, which always incurs synchronization costs as it ensures that only one thread at a time can modify it.

In the typical case, accessing the local queue needs no synchronization at all. Items can be locally added and removed very quickly. The reason for this efficiency is that the local queues are implemented using a special concurrent data structure called a work-stealing queue. A work-stealing queue has a private end and a public end. The queue allows lock-free pushes and pops from the private end but requires synchronization for operations at the public end. When the length of the queue is small, synchronization is required from both ends due to the locking strategy used by the implementation.

Moving to a distributed scheduling approach makes the order of task execution less predictable than with a single global queue. Although the global queue executes work in FIFO order, the local work-stealing queues use LIFO order to avoid synchronization costs. However, the overall throughput is likely to be better because a thread only uses the relatively more expensive global queue when it runs out of work from its local queue.

Work stealing

What happens when a thread's local work queue is empty and the global queue is also empty? There still may be work on the local queues of other worker threads. This is where work stealing comes into play. This is shown in the following figure.

Work Stealing
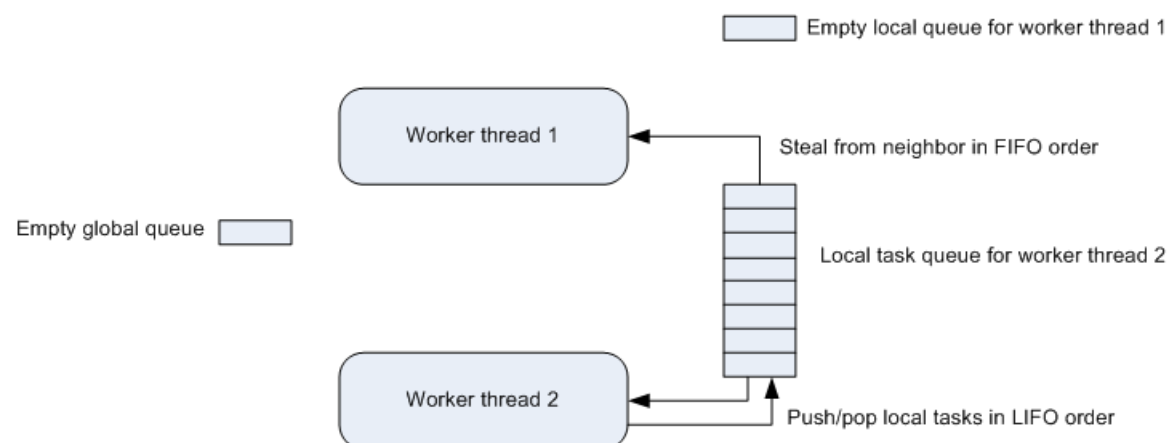
Figure 3

The diagram shows that when a thread has no items in its local queue, and there are also no items in the global queue, the system "steals" work from the local queue of one of the other worker threads. To minimize the amount of synchronization, the system takes the task from the public end of the second thread's work-stealing queue. This means that unless the queue is very short the second thread can continue to push and pop from the private end of its local queue without any overhead for synchronization.

This mix of LIFO and FIFO ordering has other interesting benefits that arise from the work distribution patterns of typical applications. It turns out that LIFO order makes sense for local queues because it
53

improves the likelihood of cache coherence. Cache coherence refers to the consistency of data stored in local caches of a shared resource. Something that has just been placed in a work queue has a good chance of referencing objects that are still present in the system's memory caches. One way to take advantage of the cache is by prioritizing recently added tasks.

Many parallel algorithms have a divide-and-conquer approach similar to recursion. The largest chunks of work tend to get pushed onto the queue before smaller subtasks. With FIFO ordering, these larger chunks are removed first and given to other threads. Transferring a larger task to another thread reduces the need for stealing additional tasks in the future. As one of these larger tasks executes, it pushes and pops its subtasks in the new thread's local queue. This is a very efficient way to schedule these kinds of tasks.

Top-level Tasks in the Global Queue

Tasks are placed in the global queue whenever a task factory method is invoked from a thread that is not one of the thread pool worker threads. (Of course, the factory method must be allowed to use the default task scheduler for this to be true. The information in this section applies only to tasks managed by the default task scheduler.)

You can also force the default task scheduler to place a task in the global queue by passing the task creation option **PreferFairness** to the factory method.

Informally, tasks in the global queue are known as top-level tasks. Top-level tasks have the same performance characteristics as work items that have been created with the thread pool's **QueueUserWorkItem** method.

Subtasks in a Local Queue

When one of the task factory methods is called from within a thread pool worker thread, the default task scheduler places the new task in that thread's local task queue. This is a very fast operation.

The default task scheduler assumes that minimizing the worst-case latency for subtasks isn't important. Instead, its goal is to optimize overall system throughput. This makes sense if you believe that any time you create a task from within another task or from within a thread pool work item, you are performing an operation that is part of some larger computation (such as a top-level task). In this case the only latency that matters is that of the top-level task. Therefore, the default task scheduler doesn't care about FIFO ordering of subtasks. While these assumptions don't hold in all cases, understanding them will help you use the features of the default task scheduler in the most efficient way for your application.

Informally, tasks in a local queue are called subtasks. The motivation for this term is that most tasks that end up in a local queue are created while executing the user delegate of some other task.

Inlined Execution of Subtasks

It is often the case that a task must wait for a second task to complete before the first task can continue. If the second task hasn't begun to execute, you might imagine that the thread that is executing the first

task blocks until the second task is eventually allowed to run and complete. An unlucky queue position for the second task can make this an arbitrarily long wait, and in extreme cases can even result in deadlock if all other worker threads are busy.

Fortunately, the Task Parallel Library can detect whether the second task has begun to execute. If the second task is not yet running, the default task scheduler can sometimes execute it immediately in the first task's thread context. This technique, known as inlined execution, enables the reuse of a thread that would otherwise be blocked. It also eliminates the possibility of deadlock due to thread starvation. A nice side effect is that inline execution can reduce overall latency by acting as a scheduling short cut for urgent tasks.

The default task scheduler in .NET 4 inlines a pending subtask if **Task.Wait**, **Task.WaitAll** or **Parallel.Invoke** is called from within the worker thread whose local queue contains that subtask. In other words, a thread pool worker thread can perform inline execution of tasks that it created. Top-level tasks in the global queue are never eligible to be inlined, and tasks created with the **LongRunning** task creation option do not inline other tasks.

The default task scheduler's policy for inline execution was motivated by the synchronization requirements of the work-stealing queues. Removing or marking a task in another local queue as processed would require additional, expensive cross-thread synchronization. Also, it turns out that typical applications almost never need cross-thread inlining. The most common coding patterns result in subtasks that reside in the local queue of the thread that executes the parent task.

The Interaction of Threads and Tasks

The .NET thread pool automatically manages the number of worker threads in the pool. It adds and removes threads according to built-in heuristics. The main goal of injecting new worker threads into the pool is to prevent deadlock from thread starvation. This kind of deadlock can occur when a worker thread waits for a synchronization event that can only be satisfied by a work item that is still pending in the global queue. If there were a fixed number of worker threads and all of those threads were similarly blocked, the system would be unable to ever make further progress. Adding a new worker thread resolves the problem.

A secondary goal of thread injection is to improve the utilization of cores when threads are blocked by I/O or other wait conditions that stall the CPU. By default, the managed thread pool has one worker thread per core. If one of these worker threads becomes blocked, there's a chance that a core might be underutilized, depending on the machine's overall workload. The thread injection logic doesn't distinguish between a thread that is blocked and a thread that is performing a lengthy, CPU-intensive operation. Therefore, whenever the thread pool's global queue contains pending work items, active work items that take a long time to run (more than a half second) can trigger the creation of new thread pool worker threads.

To make this concrete, consider an extreme example. Suppose that you have a complex financial simulation with 500 CPU-intensive operations, each one of which takes ten minutes on average to complete. If you create top-level tasks in the global queue for each of these operations, you will find

that after about five minutes the thread pool will grow to 500 worker threads. The reason is that the thread pool sees all of the tasks as blocked and begins to add new threads at the rate of approximately two threads per second.

What's wrong with 500 worker threads? In principle, nothing if you have 500 cores for them to use and vast amounts of system memory. In fact, this is the long-term vision of parallel computing. However, if you *don't* have that many cores on your machine, you are in a situation where many threads are competing for time slices. This situation is known as processor oversubscription. Allowing many CPU-intensive threads to compete for time on a single core adds context switching overhead that can severely reduce overall system throughput. Even if you don't run out of memory, performance in this situation can be much, much worse than in sequential computation. (Each context switch takes between 6,000 and 8,000 CPU cycles.)

The cost of context switching is not the only source of overhead. A managed thread in .NET consumes roughly a megabyte of stack space, whether or not that space is used for currently executing functions. It takes about 200,000 CPU cycles to create a new thread, and about 100,000 cycles to retire a thread. These are expensive operations. Clearly, the best strategy is to be careful about when threads are created.

Let's continue with the example of 500 independent tasks that take 10 minutes each. If instead of 500 top-level tasks you use the default task scheduler to create a *single* task that starts 500 *subtasks*, you will find that the number of worker threads does not grow at all. The number remains equal, by default, to the number of cores. All of the subtasks reside in the local queue of the worker thread that executes the initial top-level task. These subtasks are allocated to idle worker threads by work stealing. The subtasks are ignored by the thread injection logic of the thread pool. (Recall that when the global queue is empty no new threads are added to the thread pool.)

You might ask why disabling thread injection for local queues doesn't put the program at risk of deadlock. If the number of worker threads can't grow, why doesn't a task in a local queue have the same risk as a task in the global queue of waiting on a condition that can only be satisfied by some other pending task? There are two answers. The first is that thread starvation isn't possible in this example because the tasks are mutually independent.

The second and more general answer is inline execution. Whenever there are dependencies between tasks, you *always* notify the runtime of that dependency by invoking one of the library's wait methods. This allows the run-time system to unblock a waiting thread by immediately executing the needed pending subtask in that thread. No deadlock is possible, and no thread injection is necessary. You get the benefits of thread injection without the risk of uncontrolled growth of the thread pool under extreme load conditions. If the operations in this example did in fact have dependencies, you would structure the computation accordingly using nested subtasks and explicit task-based wait operations. In that way no matter how complex the graph of interdependencies happened to be, the entire computation could be safely executed using any fixed number of threads. The number of threads would be chosen to meet load balancing goals and not as a way to prevent deadlock.

The ability to use subtasks with inlined execution is one of the reasons that you should use tasks for all new application development instead of work items created by the **ThreadPool.QueueUserWorkItem** method.

Bypassing the Thread Pool

If you don't want a task to use a worker thread form the thread pool, you can create a new thread for its dedicated use. The new thread will not be a thread pool worker thread. To do this, include the **LongRunning** task creation option as an argument to one of the task factory methods. The option is mostly used for tasks with long I/O-related wait conditions and for tasks that act as background helpers.

A disadvantage of bypassing the thread pool is that, unlike a worker thread that is created by the thread pool's thread injection logic, a thread created with the **LongRunning** option cannot use inline execution for its subtasks.

Also, note that the **LongRunning** option does not help the example of 500 top-level tasks mentioned previously. You would still end up with 500 threads. Subtasks residing in local queues would be a better solution.

Writing a custom task scheduler

The .NET Framework includes two task scheduler implementations: the default task scheduler and a task scheduler that runs tasks in a single, chosen thread. If these task schedulers do not meet your application's needs, it's possible to implement a custom task scheduler. There are a number of advanced scenarios where creating a custom task scheduler is relevant.

You could implement a custom task scheduler if you wanted to enable a single maximum degree of parallelism across multiple loops and tasks instead of across a single loop.

You could create a custom task scheduler if you wanted to implement alternative scheduling algorithms, for example, to ensure fairness among batches of work.

A typical reason for implementing a custom task scheduler happens when you want to use a specific set of threads, such as Single Thread Apartment (STA) threads, instead of thread pool worker threads.

You can see how to create a custom task scheduler by looking at the ParallelExtensionsExtras project that can be downloaded as part of Microsoft's Parallel Samples package. This package includes an implementation of a **QueuedTaskScheduler** class that provides multiple global task queues with round-robin scheduling among them. You could use one queue for UI work and another for background tasks, for example.

Unobserved Task Exceptions

If you do not give a faulted task the opportunity to propagate its exceptions (for example, by calling the **Wait** method), the runtime will escalate the task's unobserved exceptions according to the current .NET exception policy when the task is garbage-collected. Unobserved task exceptions will eventually be observed in finalizer thread context. The finalizer thread is the system thread that invokes the **Finalize**

57

method of objects that are ready to be garbage collected. If an unhandled exception is thrown during the execution of a **Finalize** method, the runtime will by default terminate the current process, and no active try-finally blocks or additional finalizers will be executed, including finalizers that release handles to unmanaged resources. To prevent this from happening you should be careful that your application never leaks unobserved task exceptions. You can also elect to receive notification of any unobserved task exceptions by subscribing to the **UnobservedTaskException** event of the **TaskScheduler** class and choose to handle them before they propagate into the finalizer context.

This latter technique can be useful in scenarios such as hosting untrusted plug-ins that have benign exceptions that would be cumbersome to observe. For more information, see the documentation on MSDN.

Special handling occurs for unobserved task exceptions that arise from task cancellation. If the cancellation token that was passed as an argument to the **StartNew** method is the same token as the one embedded in the unobserved **OperationCanceledException** instance, then the task does not propagate the operation canceled exception to the **UnobservedTaskException** event or finalizer thread context. In other words, if you follow the cancellation protocol described in this chapter, unobserved cancellation exceptions will not be escalated into the finalizer's thread context.

Anti-Patterns

Here are some things to watch out for when you use tasks.

Variables Captured by Closures

In C#, a closure is created with a lambda expression in the form args **=>** body that represents an unnamed (anonymous) delegate. A unique feature of closures is that they may refer to variables defined outside of their lexical scope, such as local variables that were declared in a scope that contains the closure.

The semantics of closures in C# may not be intuitive to some programmers. Many programmers make mistakes in this area. Unless you understand the semantics, you may find that captured variables do not behave as you expect.

The problem arises when you reference a variable without considering its scope. Here is an example,

```
for (int i = 0; i < 4; i++)
{
    Task.Factory.StartNew(() => Console.WriteLine(i));
}
```

You might think that this code sample would print 1, 2, 3, 4. It actually prints 4, 4, 4, 4. The reason is that the variable **i** is shared by all of the closures created by the steps of the **for** loop. By the time the tasks start, the value of **i** has the value 4.

The solution is to introduce an additional temporary variable in the appropriate scope.

```
for (int i = 0; i < 4; i++)
{
    var tmp = i;
    Task.Factory.StartNew(() => Console.WriteLine(tmp));
}
```

This version prints 1, 2, 3, 4. The reason is that the variable **tmp** is declared within the block scope of the **for** loop's body. This causes a new variable named **tmp** to be instantiated with each iteration of the **for** loop. (In contrast, all iterations of the **for** loop share a single instance of the variable **i**.)

This bug is one reason why you should use **Parallel.For** instead of coding a loop yourself. It is also one of the most common mistakes made by programmers who are new to tasks.

Disposing a Resource Needed by a Task

When you create a task, don't forget that you can't call dispose on the objects that the task needs to do its work. Careless use of the C# **using** keyword is a common way to make this mistake.

```
Task<string> t;
using (var file = FileSystem.OpenTextFileReader(…))
{
    t = Task<string>.Factory.StartNew(() => file.ReadToEnd());
}
// WARNING: BUGGY CODE, file has been disposed
Console.WriteLine(t.Result); // exception thrown!
```

Avoid Thread Abort

If you terminate tasks with the **Thread.Abort** this leaves the AppDomain in a potentially unusable state. Also, aborting a thread pool worker thread is not ever recommended. If you need to cancel a task, use the technique described in Canceling a Task in the chapter. Do not abort the task's thread.

Related Patterns

SPMD – Distributed systems (MPI, Batch & SOA)

Master/Worker – Task centric (TPL or PPL)

Loop Parallelism – Data centric (OpenMP)

Task Parallelism – Algorithm strategy pattern

Speculation

Exercises

1. The image blender example in this chapter uses task parallelism: a different task processes each image layer.   A typical strategy in image processing uses data parallelism: the same

computation processes different portions of an image, or different images. Is there a way to use data parallelism in the image blender example? If so, what are the advantages and disadvantages, compared to the task parallelism discussed here?

2. In the image blender sample, the image processing methods **SetToGray** and **Rotate** are void methods that do not return results, but save their results by updating their second argument. Why do they not return their results?

3. In the image blender sample that uses **Task.Factory.StartNew**, what happens if one of the parallel tasks throws an exception? In the sample that uses **Parallel.Invoke**?

Further Reading

Leijen et. al (2009) discuss design considerations including scheduling and work stealing.

Hoag (2009) provides a detailed discussion of task creation options.

Microsoft's Parallel Samples package is ParExtSamples (2008).

MSDN (2010) provides more information about unobserved task exceptions.

4 Parallel Aggregation

Chapter 2 shows how to use parallel techniques that apply the same independent operation to many input values. However, not all parallel loops have loop bodies that execute independently. For example, a loop that calculates a sum does *not* have independent steps. All of the steps accumulate their results in a single variable that represents the sum calculated up to that point. This accumulated value is an aggregation.

Nonetheless, there is a way for an aggregation operation to use a parallel loop. This is the parallel aggregation pattern.

> The parallel aggregation pattern lets you use multiple cores to calculate sums and other types of accumulations.

While calculating a sum is an example of aggregation, the pattern is more general than that.

> **Note:** The parallel aggregation pattern is also known as parallel reduction.

The parallel aggregation pattern uses unshared, local variables that are merged at the end of the computation to give the final result. This is how the steps of a loop can become independent of each other. Parallel aggregation demonstrates the principle that it is usually better to make changes to your algorithm than to add synchronization primitives to an existing algorithm. See Design Notes in this chapter for more information about the algorithmic aspects of this pattern.

The Basics

The most familiar application of aggregation is calculating a sum. Here is the sequential version.

```
double[] sequence = ...
double sum = 0.0d;
for (int i = 0; i < sequence.Length; i++)
{
  sum += Normalize(sequence[i]);
}
return sum;
```

This is a typical sequential **for** loop. In this example and the ones that follow, **Normalize** is a user-provided method that transforms the input values in some way, such as converting them to an appropriate scale. The result is the sum of the transformed values.

The Language Integrated Query (LINQ) feature of C# provides a very simple way to express this kind of aggregation.

```
double[] sequence = ...
return (from x in sequence select Normalize(x)).Sum();
```

The LINQ expression calculates the sum. This is a sequential operation whose performance is comparable to the sequential **for** loop previously shown.

To convert a LINQ expression into a parallel query is extremely easy. The following code gives an example.

```
double[] sequence = ...
return (from x in sequence.AsParallel()
        select Normalize(x)).Sum();
```

If you invoke the **AsParallel** extension method, you are instructing PLINQ to use the parallel versions of all further query operations. The **Sum** extension method executes the query and (behind the scenes and in parallel) calculates the sum of the selected, transformed values. See chapter 2 for an introduction to PLINQ.

This code uses addition as the underlying aggregation operator, but there are many others. For example, LINQ and PLINQ have built-in standard query operators that count the number of elements, and calculate the average, maximum, or minimum. They also have operators that create and combine sets (duplicate elimination, union, intersection, and difference), transform sequences (concatenation, filtering, and partitioning) and group (projection). The standard query operators are sufficient for many types of aggregation tasks, and with PLINQ they all can efficiently use the hardware resources of a multicore computer.

If PLINQ's standard query operators aren't what you need, you can also use the **Aggregate** extension method to define your own aggregation operators. Here is an example.

```
double[] sequence = ...
return (from x in sequence.AsParallel() select Normalize(x))
        .Aggregate((y1, y2) => y1 * y2);
```

This code shows one of the overloaded versions of the **Aggregate** extension method. It applies a user-provided transformation to each element of the input sequence and then returns the mathematical product of the transformed values.

PLINQ is usually the recommended approach whenever you need to apply the parallel aggregation pattern to .NET applications. Its declarative nature makes it less prone to error than other approaches, and its performance on multicore computers is competitive. Implementing parallel aggregation with PLINQ doesn't require locks in your code.

 If PLINQ doesn't meet your needs or if you prefer a less declarative style of coding, you can also use **Parallel.ForEach** to implement the parallel aggregation pattern. The **Parallel.ForEach** method requires more complex code than PLINQ. For example, the **Parallel.ForEach** method requires your code to include locking primitives to implement parallel aggregation. See Using Parallel For Each for Aggregation in the Variations section of this chapter for examples and more information.

An Example

Aggregation doesn't just apply to numeric values. It's a more general pattern that arises in many application contexts. The following application-oriented example shows how a variation of parallel aggregation known as map/reduce can be used to aggregate nonscalar data types.

Consider a social network service, where subscribers can designate other subscribers as friends. The site occasionally recommends new friends to each subscriber by identifying other subscribers who are friends of friends. To limit the number of recommendations, the service only recommends the candidates who have the largest number of mutual friends. Candidates can be identified in independent parallel operations; then candidates are ranked and selected in an aggregation operation.

Here's how the data structures and algorithms used by the recommendation service work. Subscribers are identified by integer ID numbers. A subscriber's friends are represented by the collection of their IDs. The collection is a *set* because each element (a friend's ID number) occurs only once and the order of the elements doesn't matter. For example, the subscriber whose ID is 0 has two friends whose IDs are 1 and 2. This can be written as:

0 -> { 1, 2 }

The social network repository stores an entry like this for every subscriber. In order to recommend friends to a subscriber, the recommendation service must consider that subscriber's entry, as well as the entries for all of that subscriber's friends. For example, to recommend friends for subscriber 0, the pertinent entries in the repository are:

 0 -> { 1, 2 }

1 -> { 0, 2, 3 }

2 -> { 0, 1, 3, 4 }

You can see that the service should recommend subscribers 3 and 4 to subscriber 0 because they appear among the friends of subscribers 1 and 2, who are already friends of 0. In addition, the recommendation service should rank subscriber 3 higher than 4, because 3 is a friend of both of 0's friends, while 4 is a friend of just one of them. You can write the results like this:

{ 3(2), 4(1) }

This means that subscriber 3 shares two mutual friends with subscriber 0, and subscriber 4 shares one. This is an example of a kind of collection called a *bag*. In a bag, each element (3 and 4 in this example) is associated with a *multiplicity*, which is the number of times it occurs in the collection (2 and 1, respectively). So a bag is a collection where each element only occurs once, yet it can represent duplicates (or larger multiplicities). The order of elements in a bag doesn't matter.

The recommendation service uses the MapReduce pattern, which has several phases. In the first phase, which is the *map phase*, the service creates a collection of candidates that can contain duplicates — the same candidate's ID can occur several times in the list (once for each mutual friend). In the second phase, which is the *reduce phase*, the service aggregates this collection to create a bag where each

candidate's ID occurs only once, but is associated with its multiplicity in the first collection (the number of mutual friends). There is also a postprocessing phase where the service ranks candidates by sorting them according to their multiplicity, and selects only the candidates with the largest multiplicities.

An important feature of the MapReduce pattern is that the results of the map stage is a collection of items that is compatible with the reduce stage. The reduce stage uses bags; therefore, the map stage does not just produce a list of candidate IDs; instead, it produces a collection of bags, where each bag contains just one candidate with a multiplicity of one. In this example, the output of the map stage is a collection of two bags:

{ 3(1) }, { 3(1) , 4(1) }

Here the first bag contains friends of subscriber 1, and the second bag contains friends of subscriber 2.

Another important feature of MapReduce is that the aggregation in the reduce phase is performed by applying a binary operation to pairs of elements from the collection that is produced by the map phase. Here that operation is *bag union*, which combines two bags by collecting the elements and adding their multiplicities. The result of applying the bag union operation to the two bags in the preceding collection is:

{ 3(2), 4(1) }

Now that there is just one bag, the reduce phase is complete. By repeatedly applying bag union, the reduce phase can aggregate any collection of bags, no matter how large, into one.

Here is the code for the sequential version:

```
public IDBagItemList PotentialFriendsSequential(SubscriberID id,
                                                 int
maxCandidates)
{
  // Map
  var foafsList = new List<IDBag>();
  foreach (SubscriberID friend in _subscribers[id].Friends)
  {
    var foafs = _subscribers[friend].Friends;
    foafs.RemoveWhere(foaf => foaf == id ||
                      _subscribers[id].Friends.Contains(foaf));
    foafsList.Add(Bag.Create(foafs));
  }

  // Reduce
  IDBag candidates = new IDBag();
  foreach (IDBag foafs in foafsList)
  {
    candidates = Bag.Union(foafs, candidates);
  }

  // Postprocess
```

```
    return Bag.MostNumerous(candidates, maxCandidates);
}
```

In the map phase, this code loops sequentially over the subscriber's friends, and builds a collection of bags of candidates. In the reduce phase, the code loops sequentially over those bags, and aggregates them with the bag union operation. If this code executes with the few subscribers in the example, the **id** argument is 0 and **_subscribers[id].Friends** is { 1, 2}. When the map phase completes, **foafsList** is { 3(1) }, { 3(1) , 4(1) }, and when the reduce phase completes, **candidates** is { 3(2), 4(1) }.

Here is how to implement MapReduce in PLINQ.

```
public IDBagItemList PotentialFriendsPLinq(SubscriberID id,
                                           int maxCandidates)
{
  var candidates =
    _subscribers[id].Friends.AsParallel()
    .SelectMany(friend => _subscribers[friend].Friends)
    .Where(foaf => foaf != id &&
           !(_subscribers[id].Friends.Contains(foaf)))
    .GroupBy(foaf => foaf)
    .Select(foafGroup => new IDBagItem(foafGroup.Key,
                                       foafGroup.Count()));
  return Bag.MostNumerous(candidates, maxCandidates);
}
```

Recall that in MapReduce, independent parallel operations (the map phase) are followed by aggregation (the reduce phase). In the map phase, the parallel operations iterate over all of the friends of subscriber 0. The map phase is performed by the **SelectMany method**, which finds all the friends of each friend of the subscriber, and the **Where method**, which prevents redundant recommendations by removing the subscriber and the subscriber's own friends. The output of the map phase is a collection of candidate IDs, including duplicates. The reduce phase is performed by the **GroupBy method**, which collects duplicate candidate IDs into groups, and the **Select method**, which converts each group into a bag item that associates the candidate ID with a multiplicity (or **Count**). The **return** statement performs the final postprocessing step that selects the candidates with the highest multiplicities.

When MapReduce is implemented with PLINQ**,** the parallel version is not quite a line-by-line translation of the sequential version. The output of the map stage is not a collection of bags, but a collection with duplicates. The bag is not formed until the reduce stage.

The online source code for this example also includes map/reduce implemented with the **Parallel.ForEach** method.

Variations

This section contains some common variations of the parallel aggregation pattern.

Using Parallel For Each for Aggregation

The **Parallel.ForEach** and **Parallel.For** methods have overloaded versions that can implement the parallel aggregation pattern. Here is an example.

```
double[] sequence = ...
object lockObject = new object();
double sum = 0.0d;

// ForEach<TSource, TLocal>(
//    IEnumerable<TSource> source,
//    Func<TLocal> localInit,
//    Func<TSource, ParallelLoopState, TLocal, TLocal> body,
//    Action<TLocal> localFinally);
Parallel.ForEach(
  // 1- The values to be aggregated
  sequence,

  // 2- The local initial partial result
  () => 0.0d,

  // 3- The loop body
  (x, loopState, partialResult) =>
  {
     return x + partialResult;
  },

  // 4- The final step of each local context
  (localPartialSum) =>
  {
     // Enforce serial access to single, shared result
     lock (lockObject)
     {
       sum += localPartialSum;
     }
  });
return sum;
```

Parallel.ForEach partitions the input based on the desired degree of parallelism and creates parallel tasks to perform the work. Each parallel task has local state that is not shared with other tasks. The loop body updates only the task-local state. In other words, the loop body accumulates its value first into a subtotal and not directly into the final total. When each task finishes, it adds its subtotal into the final sum. See Figure 1 in the Design Notes section of this chapter for an illustration of these steps.

The overloaded version of **Parallel.ForEach** that is used in this example takes four arguments. The first argument is the data values that are to be aggregated. The second argument is a delegate that returns the initial value of the aggregation. The third argument is a delegate that combines one of the input data

values with a partial sum from a previous step. The implementation of **Parallel.ForEach** can create more than one task. In this case there will be one partial result value for each task.

When the parallel loop is ready to finish, it must merge the partial results from all of its worker tasks together to produce the final result. The fourth argument to **Parallel.ForEach** is a delegate that performs the merge step. The delegate is invoked once for each of the worker tasks. The argument to the delegate is the partial result that was calculated by the task. This delegate is responsible for locking the shared result variable and combining the partial sum with the result.

This example uses the C# **lock** keyword to enforce interleaved, sequential access to variables that are shared by concurrent threads. There are other synchronization techniques that could be used in this situation, but they are outside the scope of this book. Be aware that locking is cooperative; that is, unless all threads that access the shared variable use locking consistently and correctly, serial access to the variable is not guaranteed.

The syntax for locking in C# is **lock** (*object*) { *body* }. The object uniquely identifies the lock. All cooperating threads must use the same synchronizing object, which must be a reference type such as **Object** and not a structure type such as **int** or **double**. When using **lock** with **Parallel.For** or **Parallel.ForEach** you should create a dummy object and set it as the value of a captured local variable dedicated to this purpose. (A captured variable is a local variable from the enclosing scope that is referenced in the body of a lambda expression.) The lock's body is the region of code that will be protected by the lock. The body should take only a small amount of execution time. Which shared variables are protected by the lock object is a matter of convention and is something that all programmers whose code accesses those variables must be careful not to contradict. In this example, the lock object ensures serial access to the **sum** variable.

When a thread encounters a lock statement, it attempts to acquire the lock associated with the synchronizing object. Only one thread at a time can hold this lock. If another thread has already acquired the lock but has not yet released it, the current thread will block until the lock becomes available. When multiple threads compete for a lock, the order that they will acquire the lock is unpredictable; it is not necessarily FIFO order. After the thread has successfully acquired the lock, it executes the statements inside of the body. When the thread exits the lock body (whether normally or by throwing an exception), it releases the lock. See the Design Notes section in this chapter for more about locking.

The "local finally" delegate (argument four) is executed once at the end of each task. The number of locks will therefore be equal to the number of tasks that are used to execute the parallel loop. You cannot predict how many tasks will be used. For example, you should not assume that the number of tasks will be less than or equal to the number that you provided using the parallel option **MaxDegreeOfParallelism** (or its default value). The reason for this is that the parallel loop sometimes retires tasks during the execution of the loop and continues with new tasks. This cooperative yielding of control can in some cases reduce system latency by allowing other tasks to run. It also helps prevent an unintended increase in the number of thread pool worker threads. You should not consider the state of the accumulator to be strictly "thread local." It is actually "task local." (This is a minor distinction that

should not normally affect a program's logic because a task is guaranteed to execute from start to finish on only one thread.)

For comparison, here is the PLINQ version of the example used in this variation:

```
double[] sequence = ...
return sequence.AsParallel().Sum();
```

Using a Range Partitioner for Aggregation

When you have a loop body with a very small amount of work to do, and there are many iterations to perform, it is possible that the overhead of **Parallel.ForEach** is large compared to the cost of executing the loop body.

In this case it is sometimes more efficient to use a partitioner object for the loop. With a partitioner object, you can embed a sequential **for** loop inside of your **Parallel.ForEach** loop. The number of iterations of the **Parallel.ForEach** loop can therefore be reduced. Generally, you should profile the application in order to decide whether to use a partitioner.

Here is an example.

```
double[] sequence = ...
object lockObject = new object();
double sum = 0.0d;
var rangePartitioner = Partitioner.Create(0, sequence.Length);

// ForEach<TSource, TLocal>(
//    Partitioner<TSource> source,
//    Func<TLocal> localInit,
//    Func<TSource, ParallelLoopState, TLocal, TLocal> body,
//    Action<TLocal> localFinally);
Parallel.ForEach(
  // 1- the input intervals
  rangePartitioner,

  // 2- The local initial partial result
  () => 0.0,

  // 3- The loop body for each interval
  (range, loopState, initialValue) =>
  {
    double partialSum = initialValue;
    for (int i = range.Item1; i < range.Item2; i++)
    {
      partialSum += Normalize(sequence[i]);
    }
    return partialSum;
  },

  // 4- The final step of each local context
```

```
    (localPartialSum) =>
    {
      // Use lock to enforce serial access to shared result
      lock (lockObject)
      {
        sum += localPartialSum;
      }
    });
  return sum;
```

This code is very similar to the example in the previous section, Using Parallel ForEach for Aggregation. The difference is that the **for each** loop uses a sequence of index intervals as its input rather than individual values. This may avoid some of the overhead involved in invoking delegate methods.

Using PLINQ Aggregation with Range Selection

The PLINQ Aggregate extension method includes an overloaded version that allows a very general application of the parallel aggregation pattern. Here is an example from an application that does financial simulation. The method performs repeated simulation trials and aggregates results into a histogram. There are two dependencies that must be handled by this code—the accumulation of partial results into the result histogram and the use of instances of the **Random** class. (An instance of **Random** cannot be shared across multiple threads.)

```
int[] histogram = MakeEmptyHistogram();

// 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);
return ParallelEnumerable.Range(0, count).Aggregate(

  // 1- create an empty local accumulator object
  //    that includes all task-local state
  () => new Tuple<int[], Random>(
          MakeEmptyHistogram(),
          new Random(SampleUtilities.MakeRandomSeed())),

  // 2- run the simulation, adding result to local accumulator
  (localAccumulator, i) =>
  {
    // With each iteration get the next random value
    var sample = localAccumulator.Item2.NextDouble();

    if (sample > 0.0 && sample < 1.0)
```

```
      {
        // Perform a simulation trial for the sample value
        var simulationResult =
                             DoSimulation(sample, mean, stdDev);

        // Add result to the histogram of the local accumulator
        int histogramBucket =
                 (int)Math.Floor(simulationResult / BucketSize);
        if (0 <= histogramBucket && histogramBucket < TableSize)
          localAccumulator.Item1[histogramBucket] += 1;
      }
      return localAccumulator;
    },

    // 3- Combine local results pairwise.
    (localAccumulator1, localAccumulator2) =>
    {
      return new Tuple<int[], Random>(
        CombineHistograms(localAccumulator1.Item1,
                          localAccumulator2.Item1),
        null);
    },

    // 4- Extract answer from final combination
    finalAccumulator => finalAccumulator.Item1
    ); // Aggregate
```

The data source in this example is a parallel query that is created with the **Range** static method of the **ParallelEnumerable** class. Applying the **Aggregate** extension to this object also results in a parallel query. The parallel query partitions the input data into intervals and then creates a worker task for each partition. The overloaded version of **Aggregate** in this example has four arguments. Each of the arguments is a delegate method.

The first argument is a delegate that establishes the local state of each worker task that the query creates. This is called once, at the beginning of the task. In this example, the delegate returns a tuple (unnamed record) instance that contains two fields. The first field is an empty histogram. This will be used to accumulate the local partial results of the simulation in this task. The second field is an instance of the **Random** class. It is part of the local state to ensure that the simulation does not violate the requirements of the **Random** class by sharing instances across threads. Note that you can store virtually any kind of local state in the object you create.

The second argument is the loop body. This delegate is invoked once per data element in the partition. In this example, the loop body creates a new random sample and performs the simulation experiment. Then it classifies the result of the simulation into buckets used by the histogram and increments the appropriate bucket of the task's local histogram.

The third argument is invoked for pairwise combination of local partial results. The delegate merges the input histograms (it adds their corresponding bucket values and returns a new histogram with the sum). It returns a new tuple. The null argument reflects the fact that the random number generator is no longer needed. The combination delegate is invoked as many times as necessary in order to consolidate all of the local partial results.

The fourth argument selects the result from the final, merged local state object.

This variation can be adapted to many situations that use the parallel aggregation pattern. Again, it should be mentioned that this is a lock-free implementation.

Design Notes

If you compare the sequential and parallel versions of the aggregation pattern, you see that the design of parallel aggregation includes an additional step in the algorithm that merges partial results. This is shown in the following diagram of the **Parallel.ForEach** and **Parallel.For** methods:

Aggregation using Parallel ForEach
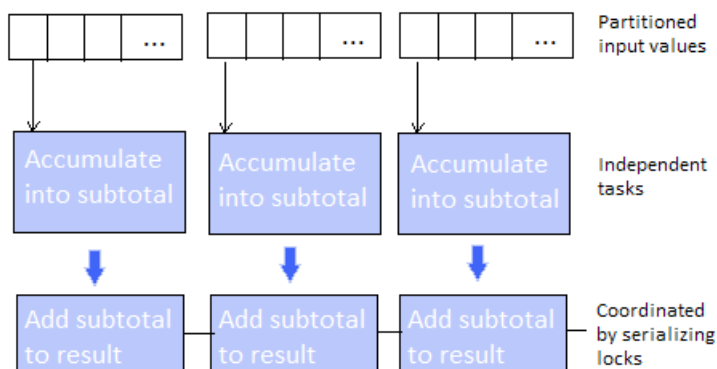
Figure 1

The figure shows that instead of the accumulation residing in a single, shared result, the parallel loop uses unshared local storage for partial results (these are called *subtotals* in the figure). Each task processes a single partition of the input values. The number of partitions depends on the degree of parallelism that is needed to efficiently use the computer's available cores. When a task has finished accumulating the values in its assigned partition, it merges its local result into the final, global result. The final result is shared by all tasks; therefore, locking is required to ensure that updates are consistent.

The reason that this approach is fast is because there are very few locks. In normal cases, the number of elements to be processed is many times larger than the number of tasks and partitions. The cost of serializing locks can be amortized over many individual accumulation operations.

The algorithm used by PLINQ differs slightly from this picture. Aggregation in PLINQ does not use locks in user code. Instead the final merge step is expressed as a binary operator that combines any two partial

71

result values (that is, two of the subtotals) into another partial result. Repeating this process on subsequent pairs of partial results eventually converges on a final result. One of the advantages of the PLINQ approach is that it can scale to an unlimited number of cores. The binary combinations do not require the system to lock the final result; they can be performed in parallel.

This discussion shows that the parallel aggregation pattern is a good example of why changes to your algorithm are often needed when moving from a sequential to parallel approach.

> You can't just add locks and expect to get good performance. You also need to think about the algorithm.

To make this point clear, here's an example of what parallel aggregation would look like if you simply added locks to the existing sequential algorithm. You just need to convert sequential **foreach** to **Parallel.ForEach** and add one lock statement.

```
// Do not copy this code. This version will run much slower
// than the sequential version. It is included here to
// illustrate what not to do.
double[] sequence = ...
object lockObject = new object();
double sum = 0.0d;

// BUG – don't do this
Parallel.For(0, sequence.Length, i =>
{
    // BUG – don't do this
    lock (lockObject)
    {
        sum += sequence[i];
    }
});
return sum;
```

If you forget to add the **lock** statement, this code fails to calculate the correct sum on a multicore computer. Adding the lock statement makes this code example correct with respect to serialization. If you run this code, it is guaranteed to produce the expected sum. However, it fails completely as an optimization. This code executes many times slower than the sequential version it attempted to optimize!

On the other hand, the examples of the parallel aggregation pattern that you have seen elsewhere in this chapter will run much faster on multicore computers than the sequential equivalent, and their performance also improves in approximate proportion to the number of cores.

It might at first seem counterintuitive that adding additional steps can make an algorithm go faster, but it's true. If you introduce extra work and that work has the effect of preventing data dependencies between parallel tasks, you often benefit in terms of performance.

Related Patterns

There is a cluster of patterns related to summarizing data in a collection: reduce, stencil, scan and pack. The Reduce pattern occurs when a result is accumulated into a single location of memory. Reduce is another name for the parallel aggregation pattern.

The Scan pattern occurs when each iteration of a loop depends on data computed in the previous iteration.

The Stencil pattern has parallel array reads that are relative to offset indices. The offset indices are also given as an array. The output is computed by reading a region of memory determined by the offset array (the stencil). This pattern appears in simulation applications and in image processing, for example, blurring or sharpening.

The Pack pattern uses a parallel loop to select elements to retain or discard. The result of a pack operation is a subset of the original input.

These can be combined, as in the Fold and Scan pattern.

Exercises

4. Consider the small social network example (with subscribers 0, 1, 2). What constraints exist in the data?   How are these constraints observed in the sample code?

5. In the social network example, there is a separate postprocessing step where the bag of candidates, which is an unordered collection, is transformed into a sequence that is sorted by the number of mutual friends, and then the top N candidates are selected. Could some or all of this postprocessing be incorporated into the reduction step? Provide answers for both the PLINQ and **Parallel.ForEach** versions.

6. In the standard reference on MapReduce (in Further Readings), the map phase executes a map function that takes an input pair and produces a set of intermediate key/value pairs. All pairs for the same intermediate key are passed to the reduce phase. That reduce phase executes a reduce function that merges all the values for the same intermediate key to a possibly smaller set of values. The signatures of these functions can be expressed: map (k1,v1) -> list(k2,v2) and reduce (k2,list(v2)) -> list(v2).   In our social network example, what are the types of k1, v1, k2, and v2?   What are our map and reduce functions?   Provide answers for both the PLINQ and **Parallel.ForEach** versions.

Further Reading

MSDN (2010) describes the standard query operators of LINQ and PLINQ.

A thorough treatment of synchronization techniques is Duffy (2008).

If you're interested in the related patterns of Stencil, Scan and Pack, see Michael McCool, "Structured Patterns for Parallel Computation" (December 2009) http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=223101515

The standard reference on map/reduce is the paper by Dean and Ghemawat (2004).

Other cases of algorithms that use parallel loops with some dependencies between steps are described in Toub (2009). These include fold-and-scan and dynamic programming.

5

Futures

In Chapter 3, "Parallel Tasks," you saw how the Parallel Task pattern allows you to fork the flow of control in a program. In this chapter, you'll see how control flow and data flow can be integrated with a pattern known as Futures.

A *future* is a stand-in for a computational result that is initially unknown but becomes available at a later time. The process of calculating the result can occur in parallel with other computation. Futures are an important pattern that integrates task parallelism with the familiar world of arguments and return values. If the parallel tasks described in Chapter 3 are asynchronous *actions*, futures are asynchronous *functions*. (Recall that actions don't return values, but functions do.)

<p style="text-align:center;color:#4a7ebb;">Futures are asynchronous functions.</p>

Futures express the concept of potential parallelism that was introduced in Chapter 1, "Introduction." Decomposing a sequential operation with futures can result in faster execution if hardware resources are available for parallel execution. However, if all cores are otherwise occupied, futures will be evaluated without parallelism.

In the .NET Framework, futures are implemented with the **Task<TResult>** class, where the type parameter **TResult** gives the type of the result. In other words, a future in .NET is a task that returns a value. Instead of explicitly waiting for the task to complete using a method such as **Wait**, you simply ask the task for its result when you are ready to use it. If the task has already finished, its result is waiting for you and is immediately returned. If the task is running but has not yet finished, the current thread blocks until the result value becomes available. (While the thread is blocked, the core is available for other work.) If the task has not yet started, the task will be executed in the current thread context, if possible.

<p style="text-align:center;color:#4a7ebb;">A future in .NET is a task that returns a value. Futures are implemented by the Task<TResult> class.</p>

The .NET Framework also implements a variation of the Futures pattern called *continuation tasks.* A .NET continuation task is a task that automatically starts when other tasks, known as its antecedents, complete. In many cases, the antecedents consist of futures whose result values are used as input to the continuation task. An antecedent may have more than one continuation task.

<p style="text-align:center;color:#4a7ebb;">A continuation task automatically starts when specified antecedent tasks finish.</p>

Continuation tasks represent the nested application of asynchronous functions. In some ways, continuation tasks act like callback methods—in both cases, you register an operation that will be automatically invoked at a specified point in the future.

> **Note:** The Futures pattern in this chapter is closely related to what is sometimes known as a *task graph*. When futures provide results that are the inputs to other futures, this can be seen as a directed graph. The nodes are tasks, and the arcs are values that act as inputs and outputs of the tasks.
>
> You should be careful not confuse futures with pipelines. As you will see in Chapter 7, "Pipelines," with pipelines, tasks are also nodes of a directed graph, but the arcs are concurrent queues that convey a series of values, like an assembly line or data stream. In contrast, with futures, nodes of the task graph are connected by singleton values, similar to arguments and return values. Avoiding confusion between futures and pipelines is one of the reasons that this chapter uses the term "futures" instead of "task graph" for this pattern.

The Basics

When you think about the Parallel Task pattern described in Chapter 3, you see that, in many cases, the purpose of a task is to calculate a result. In other words, asynchronous operations often act like functions. Of course, tasks can also do other things, such as reordering values in an array, but calculating new values is common enough to warrant a pattern tailored to it. It's also much easier to reason about pure functions, which don't have side effects and therefore exist purely for their results. This simplicity becomes very useful as the number of cores becomes large.

Futures

Consider the following example, taken from the body of a sequential method:

```
var b = F1(a);
var c = F2(a);
var d = F3(c);
var f = F4(b, d);
return f;
```

Suppose that **F1**, **F2**, **F3**, and **F4** are CPU-intensive functions that communicate with one another using arguments and return values instead of reading and updating shared state variables.

Suppose, also, that you want to distribute the work of these functions across available cores, and you want your code to run correctly no matter how many cores are available. When you look at the inputs and outputs, you can see that **F1** can run in parallel with **F2** and **F3** but that **F3** can't start until after **F2** finishes. How do you know this? The possible orderings become apparent when you visualize the function calls as a graph. Figure 1 illustrates this.

A task graph for calculating *f*
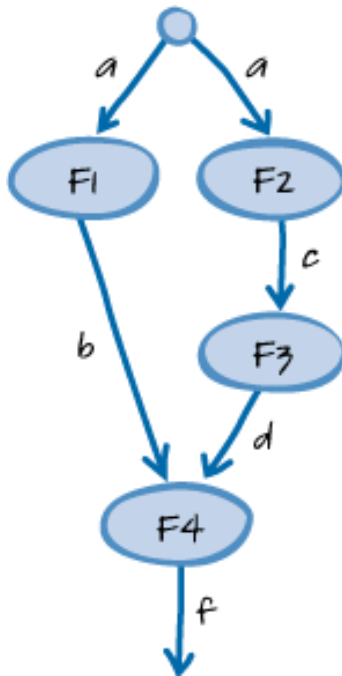
Figure 1

The nodes of the graph are the functions **F1**, **F2**, **F3**, and **F4**. The incoming arrows for each node are the inputs required by the function, and the outgoing arrows are values calculated by each function. It's easy to see that **F1** and **F2** can run at the same time but that **F3** must follow **F2**.

Here's an example that shows how to create futures for this example. The code assumes for simplicity that the values being calculated are integers and that the value of variable *a* has already been supplied, perhaps as an argument to the current method.

```
Task<int> futureB = Task.Factory.StartNew<int>(() => F1(a));
int c = F2(a);
int d = F3(c);
int f = F4(futureB.Result, d);
return f;
```

The **Result** property returns a precalculated value immediately or waits until the value becomes available.

This code creates a future that begins to asynchronously calculate the value of **F1(a)**. On a multicore system, **F1** will be able to run in parallel with the current thread. This means that **F2** can begin executing without waiting for **F1**. The function **F4** will execute as soon as the data it needs becomes available. It doesn't matter whether **F1** or **F3** finishes first, because the results of both functions are required before **F4** can be invoked. (Recall that the **Result** property does not return until the future's value is available.) Note that the calls to **F2**, **F3**, and **F4** do not need to be wrapped inside of a future because a single additional asynchronous operation is all that is needed to take advantage of the parallelism of this example.

Of course, you could equivalently have put **F2** and **F3** inside of a future, as shown here.

77

```
Task<int> futureD = Task.Factory.StartNew<int>(
                                     () => F3(F2(a)));
int b = F1(a);
int f = F4(b, futureD.Result);
return f;
```

It doesn't matter which branch of the task graph shown in the figure runs asynchronously.

An important point of this example is that exceptions that occur during the execution of a future will be thrown by the **Result** property. This makes exception handling easy, even in cases with many futures and complex chains of continuation tasks. You can think of futures as either returning a result or throwing an exception. Conceptually, this is very similar to the way any .NET function works. Here is an example.

Futures defer exceptions until the **Result** property is read.

```
Task<int> futureD = Task.Factory.StartNew<int>(
                                     () => F3(F2(a)));
try
{
   int b = F1(a);
   int f = F4(b, futureD.Result);
   return f;
}
catch (MyException)
{
   Console.WriteLine("Saw MyException exception");
   return -1;
}
```

If an exception of type **MyException** were thrown in **F2** or **F3**, it would be deferred and rethrown when the **Result** property of **futureD** is read. The get-property happens within a **try** block, which means that the exception can be handled in the corresponding **catch** block.

Continuation Tasks

It is very common for one asynchronous operation to invoke a second asynchronous operation and pass data to it. Continuation tasks make the dependencies among futures apparent to the run-time environment that is responsible for scheduling them. This helps to allocate work efficiently among cores.

For example, if you wanted to update the user interface with the result produced by the function **F4** from the previous section, you could use the following code.

```
TextBox myTextBox = ...;

var futureB = Task.Factory.StartNew<int>(() => F1(a));
var futureD = Task.Factory.StartNew<int>(() => F3(F2(a)));

var futureF = Task.Factory.ContinueWhenAll<int, int>(
                new[] { futureB, futureD },
```

```
                     (tasks) => F4(futureB.Result, futureD.Result));

futureF.ContinueWith((t) =>
  myTextBox.Dispatcher.Invoke(
       (Action)(() => { myTextBox.Text = t.Result.ToString(); }))
             );
```

This code structures the computation into four tasks. The system understands the ordering dependencies between continuation tasks and their antecedents. It makes sure that the continuation tasks will start only after their antecedent tasks complete.

The first task calculates the value of **b**. The second task calculates the value of **d**. These two tasks can run in parallel. The third task calculates the value of **f**. It can run only after the first two tasks are complete. Finally, the fourth task takes the value calculated by **F4** and updates a text box on the user interface.

The **ContinueWith** method creates a continuation task with a single antecedent. The **ContinueWhenAll<TAntecedentResult, TResult>** method of the **Task.Factory** object allows you to create a continuation task that depends on more than one antecedent task.

The Adatum Financial Dashboard

Here's an example of how the Futures pattern can be used in an application. The example shows how you can run computationally intensive operations in parallel in an application that uses a graphical user interface (GUI).

Adatum is a financial services firm that provides a financial dashboard application to its employees. The application, known as the Adatum dashboard, allows employees to perform analyses of financial markets. The dashboard application runs on an employee's desktop workstation. The Adatum dashboard analyzes historical data instead of a stream of real-time price data. The analysis it performs is computationally intensive, but there is also some I/O latency because the Adatum dashboard application collects input data from several sources over the network.

After it has the market data, the application merges the datasets together. The application normalizes the merged market data and then performs an analysis step. After the analysis, it creates a market model. It also performs these same steps for Fed historical market data. After the current and historical models are ready, the application compares the two models and makes a market recommendation of "buy," "sell," or "hold." You can visualize these steps as a graph. Figure 2 illustrates this.

Adatum dashboard tasks

79
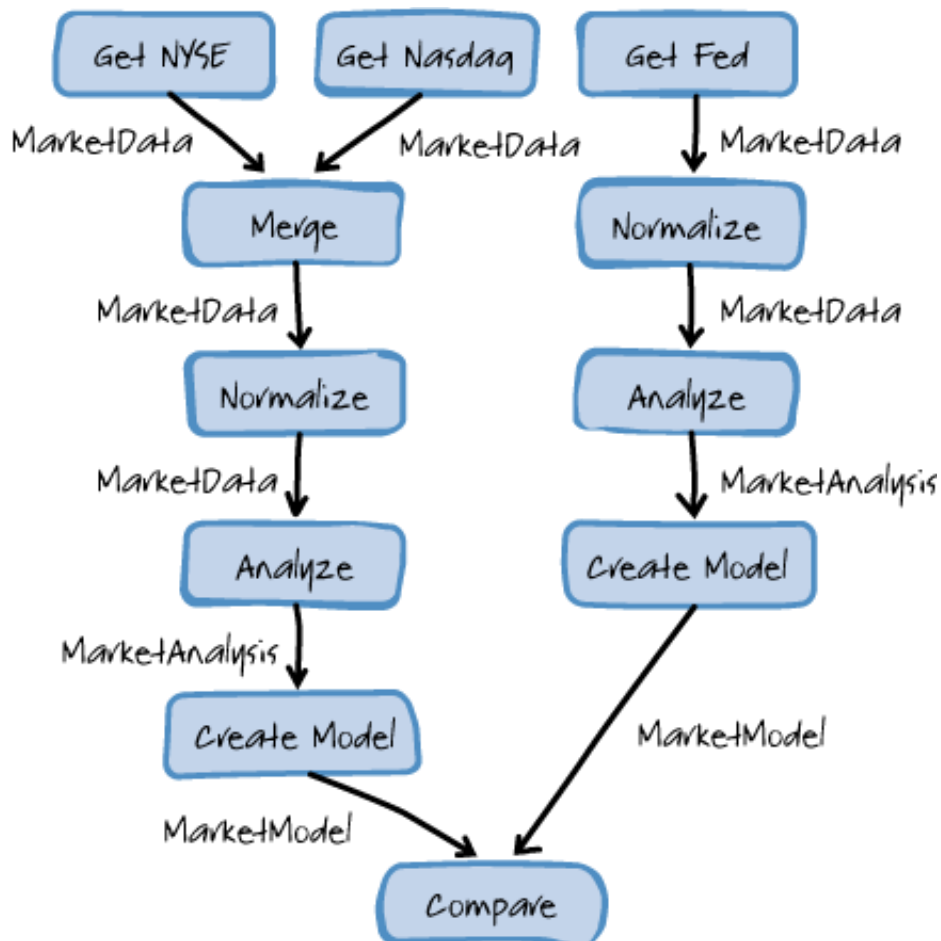
Figure 2
The tasks in this diagram communicate by specific types of business objects. These are implemented as .NET classes in the Adatum dashboard application.

You can download the source code for the Adatum dashboard application from the CodePlex site at http://parallelpatterns.codeplex.com. The application consists of four parts: the business object definitions, an analysis engine, a view model, and the user interface, or the view. Figure 3 illustrates this.

Adatum dashboard application

Figure 3

The Business Objects

The Adatum dashboard uses immutable data types. Objects of these types cannot be modified after they are created, which makes them well suited to parallel applications.

A good example of a familiar immutable data type is the .NET string class. There are many operations for creating strings, but none of them allow you to modify the string. If you want to append to a string, for example, the result is a string that includes the addition. The original string is unmodified.

The **MarketData** class represents a time series of closing prices for a group of securities. You can think of this as a dictionary indexed by a stock symbol. Conceptually, the values are arrays of prices for each security. You can merge **MarketData** values as long as the stock symbols don't overlap. The result of the merge operation is a new **MarketData** value that contains the time series of the inputs.

The **MarketAnalysis** class is the result of the analysis step. Similarly, the **MarketModel** and **MarketRecommendation** classes are the outputs of the modeling and the comparison phases of the application. The **MarketRecommendation** class has a property that contains a "buy, hold, or sell" decision.

For more information about how to implement your own immutable types, see the section, "Immutable Types," in Appendix A, "Supporting Patterns."

The Analysis Engine

The Adatum dashboard's **AnalysisEngine** class produces a market recommendation from the market data it receives. It supports both parallel and sequential modes of execution.

The sequential process is shown in the following code.

```
public MarketRecommendation DoAnalysisSequential()
{
```

```
    MarketData nyseData =
       LoadNyseData();

    MarketData nasdaqData =
       LoadNasdaqData();

    MarketData mergedMarketData =
       MergeMarketData(new[]{nyseData, nyseData});

    MarketData normalizedMarketData =
       NormalizeData(mergedMarketData);

    MarketData fedHistoricalData =
       LoadFedHistoricalData();

    MarketData normalizedHistoricalData =
       NormalizeData(fedHistoricalData);

    MarketAnalysis analyzedMarketData =
       AnalyzeData(normalizedMarketData);

    MarketModel modeledMarketData =
       CreateModel(analyzedMarketData);

    MarketAnalysis analyzedHistoricalData =
       AnalyzeData(normalizedHistoricalData);

    MarketModel modeledHistoricalData =
       CreateModel(analyzedHistoricalData);

    MarketRecommendation recommendation =
       CompareModels(new[] {modeledMarketData,
                            modeledHistoricalData});
    return recommendation;
}
```

You can see how binding of local variables corresponds to the data dependencies among the analysis phases shown in the figure. The result of the computation is a **MarketRecommendation** object. Each of the method calls returns data that becomes the input to the operation that invokes it. When you use method invocations in this way, you are limited to sequential execution.

The parallel version uses futures and continuation tasks for each of the operational steps. Here is the code.

```
public MarketRecommendation DoAnalysisParallel()
{
    TaskFactory f = Task.Factory;

    Task<MarketData> loadNyseData =
```

```csharp
    f.StartNew<MarketData>(
      () => LoadNyseData(),
      TaskCreationOptions.LongRunning);

  Task<MarketData> loadNasdaqData =
    f.StartNew<MarketData>(
      () => LoadNasdaqData(),
      TaskCreationOptions.LongRunning);

  Task<MarketData> mergeMarketData =
    f.ContinueWhenAll<MarketData, MarketData>(
      new[] { loadNyseData, loadNasdaqData },
      (tasks) =>
        {
          Task.WaitAll(tasks);
          return MergeMarketData(
                    from t in tasks select t.Result);
        });

  Task<MarketData> normalizeMarketData =
    mergeMarketData.ContinueWith(
      (t) => NormalizeData(t.Result));

  Task<MarketData> loadFedHistoricalData =
    f.StartNew<MarketData>(
      () => LoadFedHistoricalData(),
      TaskCreationOptions.LongRunning);

  Task<MarketData> normalizeHistoricalData =
    loadFedHistoricalData.ContinueWith(
      (t) => NormalizeData(t.Result));

  Task<MarketAnalysis> analyzeMarketData =
    normalizeMarketData.ContinueWith(
      (t) => AnalyzeData(t.Result));

  Task<MarketModel> modelMarketData =
    analyzeMarketData.ContinueWith(
      (t) => CreateModel(t.Result));

  Task<MarketAnalysis> analyzeHistoricalData =
    normalizeHistoricalData.ContinueWith(
      (t) => AnalyzeData(t.Result));

  Task<MarketModel> modelHistoricalData =
    analyzeHistoricalData.ContinueWith(
      (t) => CreateModel(t.Result));

  Task<MarketRecommendation> compareModels =
```

```
    f.ContinueWhenAll<MarketModel, MarketRecommendation>(
      new[] { modelMarketData, modelHistoricalData },    (tasks)
=>
        {
          Task.WaitAll(tasks);
          return CompareModels(from t in tasks select t.Result);
        });

  return compareModels.Result;
}
```

The parallel version, provided by the **DoAnalysisParallel** method, is not much different from the sequential version, except that the synchronous method calls have been replaced with futures and continuation tasks. The next sections describe how each of the tasks is created.

Loading External Data

The methods that gather the external data from the network are long-running, I/O intensive operations. Unlike the other steps, they are not particularly CPU-intensive, but they may take a relatively long time to complete. Most of their time is spent waiting for I/O operations to finish. You create these tasks with a factory object, and you use an argument to specify that the tasks are of long duration. This temporarily increases the degree of concurrency that is allowed by the system. The following code shows how to load the external data.

```
Task<MarketData> loadNyseData =
    f.StartNew<MarketData>(
      () => LoadNyseData(),
      TaskCreationOptions.LongRunning);

  Task<MarketData> loadNasdaqData =
    f.StartNew<MarketData>(
      () => LoadNasdaqData(),
      TaskCreationOptions.LongRunning);
```

Note that the factory **f** object creates futures that return values of type **MarketData**. The **TaskCreationOptions.LongRunning** enumerated value tells the task library that the operations are expected to run for a long time and that other tasks should be given a chance to run at the same time. To prevent underutilization of CPU resources, the task library may choose to run tasks like these in additional threads.

Use "long running" tasks in cases where you want additional concurrency, such as when there are long-running I/O operations.

Merging

The merge operation takes inputs from both the **loadNyseData** and the **loadNasdaqData** tasks. It is a continuation task that depends on two antecedent tasks, as shown in the following code.

```
Task<MarketData> mergeMarketData =
```

```
f.ContinueWhenAll<MarketData, MarketData>(
  new[] { loadNyseData, loadNasdaqData },
  (tasks) =>
    {
      Task.WaitAll(tasks);
      return MergeMarketData(
          from t in tasks select t.Result);
    });
```

The **ContinueWhenAll<TAntecedentResult, TResult>** method of the **Task.Factory** object allows you to create a continuation task that gets data from more than one antecedent task. After the **loadNyseData** and **loadNasdaqData** tasks complete, the anonymous delegate given as an argument is invoked. At that point, the **tasks** parameter will be an array of antecedent tasks.

The example includes a call to the **Task.WaitAll** method at the beginning of the continuation. This forces all exceptions that may have occurred during the execution of the tasks to be rethrown. (By this point, all the antecedent tasks will have completed, so the **WaitAll** invocation is purely for the purpose of exception handling.)

The **MergeMarketData** method takes an array of **MarketData** objects as its input. The LINQ expression **from t in tasks select t.Result** maps the input array of futures into a collection of **MarketData** objects by getting the **Result** property of each future.

Normalizing

After the market data is merged, it undergoes a normalization step.

```
Task<MarketData> normalizeMarketData =
    mergeMarketData.ContinueWith(
      (t) => NormalizeData(t.Result));
```

The **ContinueWith** method creates a continuation task with a single antecedent. The continuation task gets the result value from the task referenced by the **mergeMarketData** variable and invokes the **NormalizeData** method.

Analysis and Model Creation

After the market data is normalized, the application performs an analysis step. This takes an object of type **MarketData** as input and returns an object of type **MarketAnalysis**, as shown in the following code.

```
Task<MarketAnalysis> analyzeMarketData =
    normalizeMarketData.ContinueWith(
      (t) => AnalyzeData(t.Result));

Task<MarketModel> modelMarketData =
    analyzeMarketData.ContinueWith(
      (t) => CreateModel(t.Result));
```

Processing Historical Data

The application also creates a model of historical data. The steps that create the tasks are similar to those for the current market data. However, because these steps are performed by tasks, they may be run in parallel if the hardware resources allow it.

Comparing Models

Here is the code that compares the two models.

```
Task<MarketRecommendation> compareModels =
  f.ContinueWhenAll<MarketModel, MarketRecommendation>(
    new[] { modelMarketData, modelHistoricalData },    (tasks) =>
      {
        Task.WaitAll(tasks);
        return CompareModels(from t in tasks select t.Result);
      });

return compareModels.Result;
```

The "compare models" step compares the current and historical market models and produces the final result. The **return** statement blocks until the result is available.

View and View Model

The Adatum dashboard is a GUI-based application that uses the Model-View-ViewModel (MVVM) pattern. It breaks the parallel computation into subtasks whose status can be independently seen from the user interface. For more information about how the dashboard application interacts with the view model and view, see the section, "Model-View-ViewModel," in Appendix A, "Supporting Patterns."

Variations

So far, you've seen some of the most common ways to use futures and continuation tasks to create tasks. This section describes some other ways to use them.

Canceling Futures and Continuation Tasks

There are several ways to cancel futures and continuation tasks. You can handle cancellation entirely from within the task, as Adatum's dashboard does, or you can pass cancellation tokens when the tasks are created.

The Adatum dashboard application supports cancellation from the user interface. It does this by calling the **Cancel** method of the **CancellationTokenSource** class. This sets the **IsCancellationRequested** property of the cancellation token to **true**.

The application checks for this condition at various checkpoints. If a cancellation has been requested, the operation is canceled. For a code example, see the section, "Canceling Tasks," in Chapter 3, "Parallel Tasks."

86

Continue When "At Least One" Antecedent Completes

It is possible to invoke a continuation task when the first of multiple antecedents completes. To do this, use the **Task.Factory** object's **ContinueWhenAny** method. Here is an example.

```
var t1 = Task.Factory.StartNew<int>(F1);
var t2 = Task.Factory.StartNew<int>(F2);
var t3 = Task.Factory.StartNew<int>(F3);
var t4 = Task.Factory.ContinueWhenAny<string>(new[] { t1, t2, t3
},
      (t) => "The answer is" + t.Result.ToString());
Console.WriteLine(t4.Result);
```

This approach is useful when the result of any of the tasks will do. For example, you may have an application where each task queries a web service that gives the local weather. The application returns the first answer it receives to the user.

Using .NET Asynchronous Calls with Futures

Futures and continuation tasks are similar in some ways to asynchronous methods that use the .NET Asynchronous Programming Model (APM) pattern and the **IAsyncResult** interface. In fact, tasks in .NET Framework 4 are **IAsyncResult** objects. They implement this interface. This allows you to use the **Task** class when you implement the APM pattern.

You can convert a pair of begin/end methods that use **IAsynchResult** into a future. To do this, use the **Task.Factory** object's **FromAsync** method.

In general, futures can be easier to use than other implementations of **IAsyncResult** because futures rethrow exceptions when the result is requested.

<div align="center">Futures in .NET implement the IAsynchResult interface.</div>

Removing Bottlenecks

By default, a task graph that uses futures does not inherently load balance the computation (like master/worker pattern). Depending on the layout of the graph, tasks may be of differing sizes, which may cause bottlenecks. The load characteristic of the graph is determined by the critical path through the slowest tasks. The Futures pattern does nothing to resolve this issue; it is up to the programmer who implements the task graph. You can remove bottlenecks either by breaking down the slowest tasks into additional tasks on the graph, which can then execute in parallel, or by modifying the task so that it executes in parallel internally. Figure 4 illustrates the initial graph..

The initial graph

Figure 4

In Figure 4, the Run Risk Model and Volatility Analysis task in the initial graph represents one such bottleneck in a larger graph discussed in the Examples section. One option is to split the Run Risk Model and Volatility Analysis task into smaller tasks and add them as separate tasks on the graph. Figure 5 illustrates this approach.
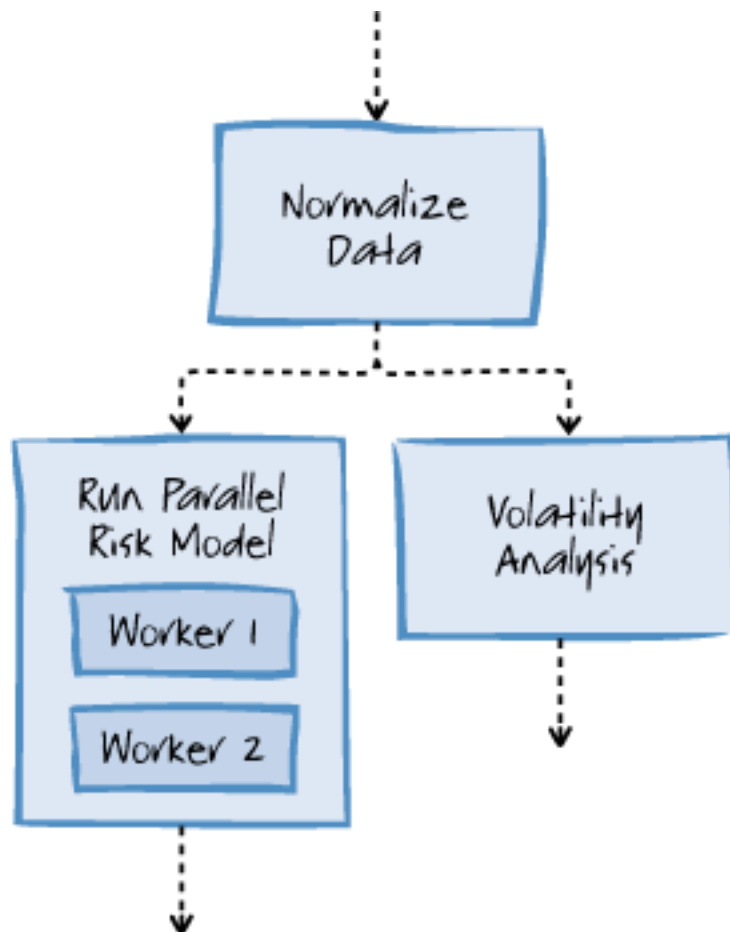
Splitting the graph
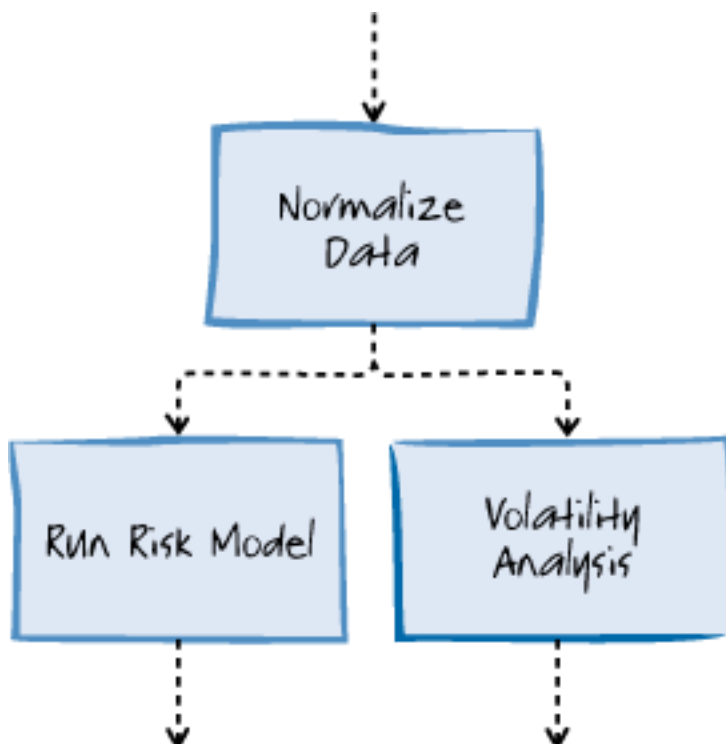
Figure 5

Another option is to parallelize the Run Risk Model internally because it cannot be broken down into atomic tasks or is better suited to parallelization with another pattern such as Master/Worker). Figure 6 illustrates how to parallelize the task internally.

Internal parallelization

Figure 6

Design Notes

There are several ideas behind the design of the Adatum dashboard application.

Decomposition into Futures and Continuation Tasks

The first design decision is the most obvious one: the Adatum dashboard introduces parallelism by means of futures and continuation tasks. This makes sense because the problem space could be decomposed into operations with well-defined inputs and outputs.

Functional Style

There are implicit and explicit approaches to synchronizing data between tasks. In this chapter, the examples used an explicit approach. Data is passed between tasks as parameters, which makes the data dependencies very obvious to the programmer. Alternatively, as you saw in Chapter 2, "Parallel Tasks," it is also possible for tasks to communicate with side effects that modify shared data structures. In this case, you rely on the tasks to use control dependencies that block appropriately. In other words, the control flow is chosen for the (implied) data flow. This approach can be appropriate for legacy applications or in cases where you must use an API that has side effects. However, in general, explicit data flow is less prone to error that implied data flow.

You can see this by an analogy to functions and subroutines. There is no need for a programming language to support methods with return values. Programmers can always use methods without return values and perform updates on shared global variables as a way of communicating the results of a computation to other components of an application. This is how subroutines behave. However, in practice, return values are considered to be much less subject to error, so functions are often a better choice.

Similarly, futures (tasks that return values) can reduce the possibility of error in a parallel program as compared to tasks that communicate results by a modified shared global state. Tasks that return values can often require less synchronization than tasks that globally access state variables and they are much easier to understand. By default, pure functions (and futures without side effects) are isolated: their result is determined only by their arguments, not by the heap state or interaction with other threads. The same is true of immutable data types such as .NET strings.

Using a control flow approach with the dashboard removed the overhead of passing data between tasks, but it makes it much less clear as to what tasks are operating on what data.

The design of the Adatum dashboard uses the functional style of programming, which relies on operations that communicate with input and output values. Functional programming is well suited to parallel programming because of the isolation it provides and functional programs are very easy to adapt to multicore environments. Let's examine a few of the assumptions.

The first is a commitment to scalable sharing of data. This means that futures should generally only communicate with the outside world by means of their return values. Usually, they should be as free as possible of side effects such as writes to mutable shared variables. If you do read and write to shared variables, you will need to either serialize your program with synchronization objects or be extraordinarily careful when you write to shared state.

Communicating among tasks by means of arguments and return values scales well as the number of cores increases.

It is also a good practice to use immutable types for return values. .NET strings are a good example of a complex type implemented as an immutable class.

Related Patterns

There are a number of patterns have some similarities to the Futures pattern but also some important differences. This section provides a brief comparison.

Pipeline Pattern

The Pipeline pattern is described in Chapter 7. It differs in several important respects from a task graph. The pipeline focuses on data flow by means of queues (message buffers), instead of task dependencies. In a pipeline, the same task is executed on multiple data items.

Master/Worker Pattern

Tasks within the Master/Worker pattern have a child/parent relationship instead of the antecedent/dependent relationship that continuation tasks have. The master task creates all tasks, passes data to them, and waits for a result to be returned. Typically, worker tasks all execute the same computation against different data. The implementation of parallel loops in .NET Framework 4 uses the Master/Worker pattern internally.

Dynamic Task Parallelism Pattern

This is also known as the Divide and Conquer pattern and is the subject of Chapter 6, "Dynamic Task Parallelism." Dynamic task parallelism creates trees of tasks on the fly in a manner similar to recursion. If futures are asynchronous functions, dynamic task parallelism produces asynchronous recursive functions.

Discrete Event Pattern

The Discrete Event pattern focuses on sending messages between tasks. There is no limitation on the number of events a task raises or when it raises them. Events may also pass between tasks in either direction; there is no antecedent/dependency relationship. The Discrete Event pattern can be used to implement a task graph by placing additional restrictions on it.

Exercises

1. Suppose you parallelize the following sequential code using futures in the style of the first example in the section, "The Basics."

```
var b = F1(a);  var d  = F2(c);   var e = F3(b,d);   var f =
F4(e);   var g = F5(e); var h = F6(f,g);
```

Draw the task graph. In order to achieve the most possible concurrency, what is the minimum number of futures you must define? What is the largest number of these futures that can be running at the same time?

2. Modify the BasicFutures sample from CodePlex so that one of the futures throws an exception. What should happen? Observe the behavior when you execute the modified sample.

Further Reading

Leijen (2009) describes the motivation for including futures in TPL and has references to other work, especially in functional languages.

The NModel framework (2008) provides a C# library of immutable collection types including set, bag, sequence, and map.

6 Dynamic Task Parallelism

THIS CHAPTER IS TBD

7 Pipelines

THIS CHAPTER IS TBD

Appendix A: Supporting Patterns

THIS APPENDIX IS TBD

Appendix B: Debugging and Profiling Parallel Applications

THIS APPENDIX IS TBD

Appendix C: Technology Roadmap

THIS APPENDIX IS TBD

References

J. Albahari and B. Albahari. C# 4 in a Nutshell. O'Reilly, fourth edition, 2010.

J. Bishop. C# 3.0 Design Patterns. O'Reilly, 2008.

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In OSDI '04: Sixth Symposium on Operating System Design and Implementation, 137 - 150, 2004

J. Duffy. Concurrent Programming on Windows, Addison-Wesley, 2008.

J. E. Hoag. A Tour of Various TPL Options. 2009. Available from MSDN.

D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In S. Arora and G.T. Leavens, editors, OOP-SLA 2009: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applicatons, pages 227 - 242. ACM, 2009.

T. G. Mattson, B. A. Sanders, and B. L. Massingill. Patterns for Parallel Programming. Addison-Wesley, 2004.

MSDN Library,. Standard Query Operators Overview. 2010.

MSDN Library, TaskSchedulerEvents, .NET Framework Class Library, 2010.

NModel software, 2008. http://nmodel.codeplex.com/.

OPL, Our Pattern Language for Parallel Programming Ver 2.0, 2010. http://parlab.eecs.berkeley.edu/wiki/patterns

ParExtSamples software, Samples for Parallel Programming with the .NET Framework 4. 2008. http://code.msdn.microsoft.com/ParExtSamples

S. Toub. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#, 2009. Available from MSDN.