

Parallel Programming with Microsoft® .NET

Design Patterns for Decomposition and Coordination on Multicore Architectures

PRELIMINARY

July 8, 2010

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release, and is the confidential and proprietary information of Microsoft Corporation. It is disclosed pursuant to a non-disclosure agreement between the recipient and Microsoft. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft .NET Framework and Visual Studio are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

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. This includes programmers who write in Microsoft Visual C#®, Microsoft Visual Basic®, and Microsoft Visual F#®. No prior knowledge of parallel programming techniques is assumed. However, readers need to be familiar with features of C# such as delegates, 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: 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 and with other languages.

Complete code solutions are posted on CodePlex. See <http://parallelpatterns.codeplex.com/>. There is a C# for every example. In addition to the C# example code, there are also versions of the examples in Visual Basic and F#.

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, processor underutilization, thread contention, cross-core thread migration, synchronization delays, areas of overlapped I/O, and other information.

See Appendix C, "Technology Overview" for a complete overview of the parallel technologies delivered by Microsoft.

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, Microsoft Windows Server® 2008, or Windows XP SP3 (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

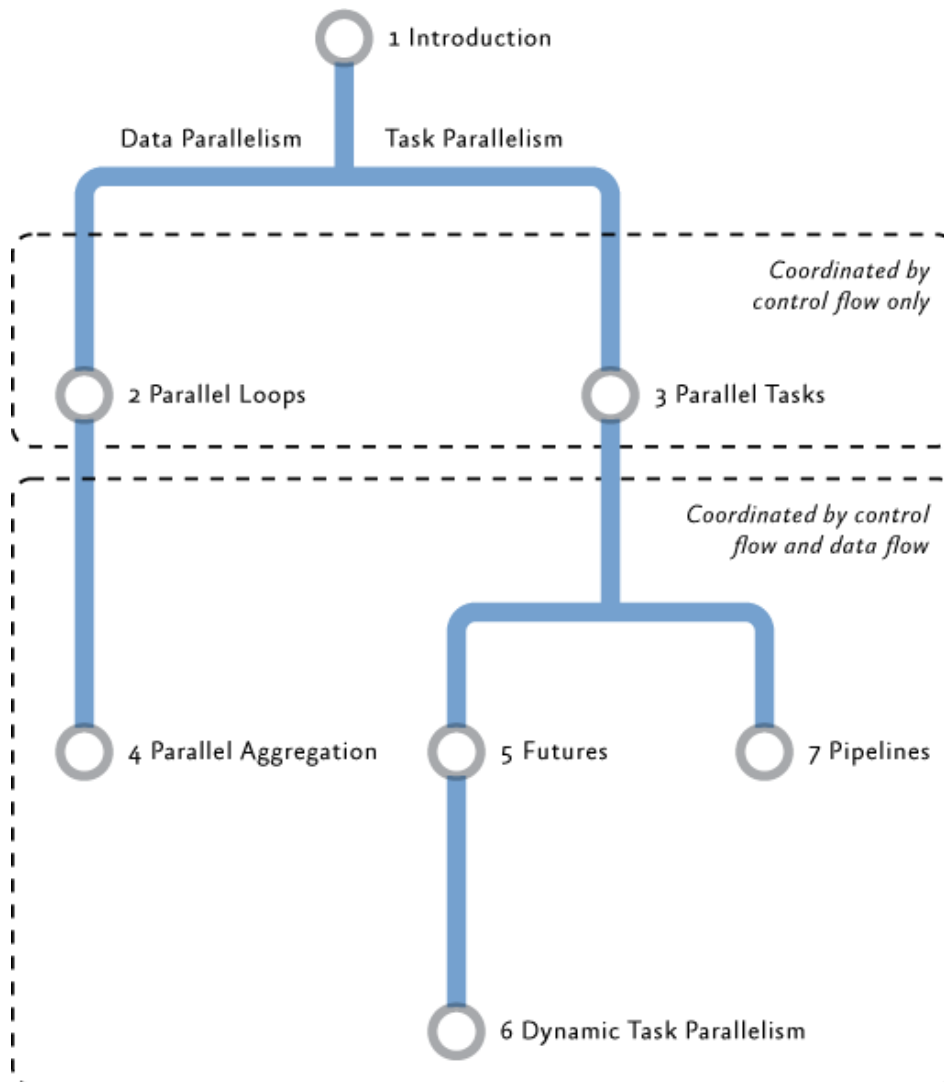


Figure 1

After the introduction, the book has one branch that discusses data parallelism and another that 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. Control flow refers to the steps of an algorithm. Data flow refers to the availability of inputs and outputs.

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. There is a table in the "Design Approaches" section of chapter 1 that can help you select the right patterns for your application.

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, "Adapting Object-Oriented Patterns."** This appendix gives tips for adapting some of the common object-oriented patterns such as facades, decorators, and repositories to multicore architectures.

- **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."** This appendix helps you place the various Microsoft technologies and frameworks for parallel programming in context.
- **Glossary.** The glossary contains definitions of the terms used in this book.
- **References.** These are the references for the works cited 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 in a distinctive style, such as the one shown in the margin, alert you to things you should watch out for.

It's very tempting to take a new tool or technology and try and use it to solve whatever problem is confronting you, regardless of the tool's applicability. As the saying goes, "when all you have is a hammer, everything looks like a nail." The "everything's a nail" mentality can lead to very unfortunate results, which one hopes the bunny in Figure 2 will be able to avoid.

Don't apply the patterns in this book blindly to your applications.

You also want to avoid unfortunate results in your parallel programs. Adding parallelism to your application costs time and adds complexity. For good results you should only parallelize the parts of your application where the benefits outweigh the costs.

When all you have is a hammer, *everything* looks like a nail



Figure 2

What Is Not Covered

This book focuses more on processor-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 processor 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.

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.

Goals

After reading this book you should be able to:

- Answer the questions at the end of each chapter.
 - Figure out if your application fits one of the book's patterns and, if it does, know if there's a good chance of implementing a straightforward parallel implementation.
 - Understand when your application doesn't fit one of these patterns. At that point, you either have to do more reading and research, or enlist the help of an expert.
 - Have an idea of the likely causes, such as conflicting dependencies or erroneously sharing data between tasks, if your implementation of a pattern doesn't work.
 - Use the "Further Reading" sections to find more material.
-

Acknowledgments

Writing a technical book is a communal effort. The patterns & practices group always involves both experts and the broader community in its projects. While this makes the writing process lengthier and more complex, the end result is always more relevant. The authors drove this book's direction and developed its content, but they want to acknowledge the other people who contributed in various ways.

The following subject matter experts were key contributors: Nicholas Chen, Daniel Dig, Munawar Hafiz, Fredrik Berg Kjolstad and Samira Tasharofi, (University of Illinois at Urbana Champaign), Reed Copsey, Jr. (C Tech Development Corporation), and Daan Leijen (Microsoft Research). Judith Bishop (Microsoft Research) reviewed the text and also gave us her valuable perspective as an author. Our schedule was aggressive, but the reviewers worked extra hard to help us meet it. Thank you.

Jon Jacky (Modeled Computation LLC) created many of the programming samples and contributed to the text. Rick Carr (DCB Software Testing, Inc) tested the samples and content.

Many other people reviewed sections of the book or gave us feedback on early outlines and drafts. They include Chris Tavares, Niklas Gustafson, Dana Groff, Wenming Ye and David Callahan (Microsoft), Justin Bozonier (MG-ALFA / Milliman, Inc.), Tim Mattson (Intel), Kurt Keutzer (UC Berkeley), Joe Hummel, Ian Griffiths and Mike Woodring (Pluralsight, LLC).

There were a great many people who spoke to us about the book and provided feedback. They include the attendees at the ParaPLOP 2010 workshop and TechEd 2010 conference, as well as contributors to discussions on the book's CodePlex site.

Tiberiu Covaci (Many-core.se) also deserves special mention for generating interest in the book during his numerous speaking engagements on "Patterns for Parallel Programming" in the US and Europe.

A team of technical writers and editors worked to make the prose readable and interesting. They include Roberta Leibovitz (Modeled Computation LLC), Tina Burden (TinaTech Inc.), and RoAnn Corbisier (Microsoft).

The innovative visual design concept used for this guide was developed by Roberta Leibovitz and Colin Campbell (Modeled Computation LLC) who worked with a group of talented designers and illustrators. The book design was created by John Hubbard (eson). The cartoons that face the chapters were drawn by the award winning Seattle-based cartoonist Ellen Forney. The technical illustrations were done by Katie Niemer (TinaTech Inc.).

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 uses multiple cores at the same time to improve your 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. The .NET Framework 4 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. In addition, Microsoft® Visual Studio® 2010 includes debugging and analysis tools to support the new parallel programming model.

Writing parallel programs has the reputation of being hard, but help has arrived.

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 six patterns presented in the following chapters. If it does, then delve more deeply into the relevant pattern or patterns and study the sample code.

Most parallel programs do conform to these patterns, and it's very likely you'll be successful in finding a match and jumping ahead with the solution to your problem. If you can't use these patterns, you have

probably encountered one of the more difficult cases and 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 parallel execution in every situation. Expressing potential parallelism is a central organizing principle behind the programming model of .NET. 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 also may be unable to predict what other software could be running at the same time as your application.

Don't hard code the degree of parallelism in an application. You can't always predict how many cores will be available at run time.

Even if you do initially know your application's environment, it can change over time. In the past, programmers assumed that their applications would automatically run faster on later generations of hardware. You could rely on this assumption because processor clock speeds kept increasing. With multicore processors, clock speeds are not increasing with newer hardware as much as in the past. Instead, the trend in processor design is toward more cores. If you want your application to benefit from hardware advances in the multicore world, you need to adapt your programming model. You should expect that the programs you write today will run on machines with many more cores within a few years. Focusing on potential parallelism helps to "future proof" your program.

Hardware trends predict more cores instead of faster clock speeds.

Finally, you must plan for 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.

A well-written parallel program runs at approximately the same speed as a sequential program when there is only one core available.

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. 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'll see that the process of designing and implementing a parallel application involves three aspects: methods for *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.

The patterns described in this guide are design patterns. You can apply them when you design and implement your algorithms and when you think about the overall structure of your application. Although the example applications are small, the principles they demonstrate have been shown to 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 hardware resources. If the chosen granularity is too fine, the overhead of managing tasks will dominate. If it's 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. You may also need to take into consideration the heuristics that will be used for task scheduling. Meeting all 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. Tasks should be large, independent and numerous enough 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. If your tasks vary greatly in size, you generally want more of them in order to fill in the gaps.

Another advantage to grouping work into larger and fewer tasks is that such tasks are often more independent of each other than smaller but 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.

The overall goal is to decompose the problem into independent tasks that do not share data while providing sufficient tasks to occupy the number of cores available. When considering the number of cores you should take into account that future generations of hardware will have more cores.

Keep in mind that tasks are not threads. Tasks and threads take very different approaches to scheduling. Tasks are much more compatible with the concept of potential parallelism than threads are. While a new thread immediately introduces additional concurrency to your application, a new task introduces only the *potential* for additional concurrency. A task's potential for additional concurrency will be realized only when there are enough available cores.

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. The problem is that when a program is running in parallel different parts of the program may be racing against each other to perform updates on the same location of memory. The result of such unintended data races can be catastrophic. The solution to the problem of data races includes techniques for synchronizing threads.

You may already be familiar with techniques that synchronize concurrent threads by blocking their execution in certain circumstances. Examples include locks, atomic compare-and-swap operations, and semaphores. All of these techniques have the effect of serializing access to shared resources. Although your first impulse for data sharing might be to add locks or other kinds of synchronization, adding

synchronization reduces the parallelism of your application. Every form of synchronization is a form of serialization. Your tasks can end up contending over the locks instead of doing the work you want them to do. Programming with locks is also error prone.

Fortunately, 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, limiting your program's reliance on shared variables, and introducing new steps in your algorithm that merge local versions of mutable state at appropriate checkpoints. Techniques for scalable sharing may involve changes to an existing algorithm.

See the "Immutable Types" section in Appendix A for more about the importance of immutable types in parallel programs.

[Scalable sharing may involve changes to your algorithm.](#)

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. Unfortunately, this is not a scalable approach to sharing. Locks can often 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.

[Adding synchronization \(locks\) can reduce the scalability of your application.](#)

In addition to performance problems, programs that rely on complex synchronization are prone to a variety of problems, including deadlock. This occurs when two or more tasks are all waiting for each other to release a lock. Most of the horror stories about parallel programming are actually about the incorrect use of shared mutable state or locking protocols.

Nonetheless, synchronizing elements in an object graph plays a legitimate if limited role in scalable parallel programs. This book uses synchronization sparingly. You should, too. Locks can be thought of as the **goto** statements of parallel programming: they are error prone but necessary in certain situations, and they are best left, when possible, to compilers and libraries.

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 incorporate design principles into the design process that limit the need for synchronization. Don't add synchronization to your application as an afterthought.

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 sequential 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 as a whole. What you discover 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. Instead, *prepare* your program for parallel execution by making structural changes.

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?

Use patterns.

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, you will also develop your intuition about how the patterns and their variations can be applied to your own applications. The following section gives more details about how to select the right pattern.

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 distinct operations with well-defined control dependencies? Are these operations largely free of serializing 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? Does the order of processing matter?	The Pipelines pattern (Chapter 7) Pipelines consist of components that are connected by queues, in the style of producers and consumers. All the components run in parallel even though the order of inputs is respected.

One way to become familiar with the possibilities of the six patterns 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'll 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 input-output (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 is necessary in order for a program to react to external stimuli such as user input, devices and sensors. Operating systems and games, by their very nature, are concurrent, even on one core.

With *parallelism*, concurrent threads execute at the same time on multiple cores. Parallel programming focuses on improving the performance of applications that use a lot of processor power and are not constantly interrupted 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 update its display area on a single-core computer. Parallelism, on the other hand, is only about throughput. It's an optimization, not a functional requirement. Its goal is to maximize processor 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 $(1 / \text{percent of time spent in sequential processing})$. Figure 1 illustrates this.

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

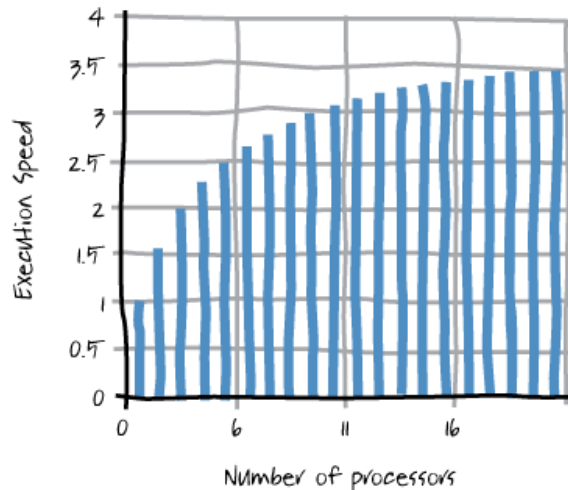


Figure 1

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

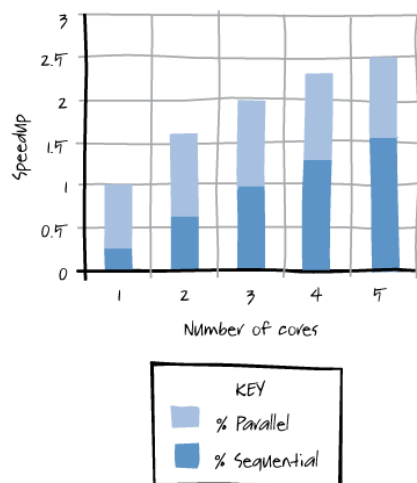


Figure 2

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. Scalability 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 create additional features in the parts of an application that are amenable to parallel execution. For example, a developer 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, because an application consists of parts that must run sequentially as well as parts that can run in parallel, the application overall will rarely see a linear increase in performance with a linear increase in the number of cores, even if certain parts of the application see a near linear speedup. Understanding 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 when analyzing performance.

A Few Tips

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

- Whenever possible, stay at as high a level of abstraction as possible and use constructs or 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). These libraries were written by experts and have been thoroughly tested; 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 10 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 the book by Mattson, et al. and at the Our Pattern Language (OPL) web site. 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.

For a detailed discussion of parallelism on the Windows platform see the book by Duffy. An overview of threading and synchronization in .NET can be found in Albahari.

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

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

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

OPL, Our Pattern Language for Parallel Programming Ver 2.0, 2010.

<http://parlab.eecs.berkeley.edu/wiki/patterns>

2

Parallel Loops

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

The syntax of a parallel loop is very similar to the **for** and **foreach** loops you already know, but the parallel loop runs faster on a computer that has available cores. Another difference is that, unlike a sequential loop, the order of execution isn't defined for a parallel loop. Steps often take place at the same time, in parallel. Sometimes, two steps take place in the opposite order than they would if the loop were sequential. The only guarantee is that all of the loop's iterations will have run by the time the loop finishes.

It's easy to change a sequential loop into a parallel loop. However, it's also easy to use a parallel loop when you shouldn't. This is because it can be hard to tell if the steps are actually independent of each other. It takes practice to learn how to recognize when one step is dependent on another step.

Sometimes, using this pattern on a loop with dependent steps causes the program to behave in a completely unexpected way, and perhaps to stop responding. Other times, it introduces a subtle bug that only appears once in a million runs. In other words, the word "independent" is a key part of the definition of this pattern, and one that this chapter explains in detail.

[The Parallel Loop pattern independently applies an operation to multiple data elements. It is an example of data parallelism.](#)

For parallel loops, the degree of parallelism doesn't need to be specified by your code. Instead, the run-time environment executes the steps of the loop at the same time on as many cores as it can. 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 improves; in many cases, performance improves proportionately with the number of cores.

The Basics

The .NET Framework includes both parallel **For** and parallel **ForEach** loops and is also implemented in the Parallel LINQ (PLINQ) query language. Use the **Parallel.For** method to iterate over a range of integer indices and the **Parallel.ForEach** method to iterate over user-provided values. Use PLINQ if you prefer a high-level, declarative style for describing loops or if you want to take advantage of PLINQ's convenience and flexibility.

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

Parallel For Loops

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

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

To take advantage of multiple cores, replace the **for** keyword with a call to the **Parallel.For** method and convert the body of the loop into a lambda expression.

Don't forget that the steps of the loop body must be independent of one another if you want to use a parallel loop. The steps must not communicate by writing to shared variables.

```
int n = ...
Parallel.For(0, n, i =>
{
    // ...
});
```

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

Parallel.For is a static method with overloaded versions. Here is the signature of the version of **Parallel.For** that is used in the example.

```
Parallel.For(int fromInclusive,
             int toExclusive,
             Action<int> body);
```

In the 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, or the largest index plus one. The third argument is an action that is invoked once per iteration. The action takes the iteration's index as its argument and executes the loop body once for each index.

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

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

The example includes a lambda expression in the form *args => body* as the third argument to the **Parallel.For** invocation. Lambda expressions are unnamed methods that can capture variables from their enclosing scope. Of course, the **body** parameter could also be an instance of a delegate type, an anonymous method (using the **delegate** keyword) or an ordinary named method. In other words, you don't have to use lambda expressions if you don't want to. Examples in this book use lambda expressions because they keep the code within the body of the loop, and are easier to read when the number of lines of code is small.

If you're unfamiliar with the syntax for lambda expressions, see "Further Reading" at the end of this chapter. Once you use lambda expressions, you'll wonder how you ever lived without them.

Parallel.ForEach

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

```
IEnumerable<MyObject> myEnumerable = ...

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

To take advantage of multiple cores, replace the **foreach** keyword with a call to the **Parallel.ForEach** method.

```
IEnumerable<MyObject> myEnumerable = ...

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

Parallel.ForEach runs the loop body for each element in a collection.

Parallel.ForEach is a static method with overloaded versions. Here is the signature of the version of **Parallel.ForEach** that was used in the example.

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

In the example, the first argument is an object that implements the **IEnumerable<MyObject>** interface. The second argument is a 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 **ForEach** loop, the incoming values are not always processed in order.

Don't forget that iterations need to be independent. The loop body must only make updates to fields of the particular instance that is passed to it.

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

Parallel LINQ (PLINQ)

The Language Integrated Query (LINQ) feature of the .NET Framework includes a parallel version named PLINQ (Parallel LINQ). There are many options and variations for expressing PLINQ queries but almost all LINQ-to-Objects expressions can easily be converted to their parallel counterpart by adding a call to the **AsParallel** extension method. Here is an example that shows both the LINQ and PLINQ versions.

You can convert LINQ expressions to parallel code with the **AsParallel** extension method.

```
IEnumerable<MyObject> source = ...

// LINQ
var query1 = from i in source select Normalize(i);

// PLINQ
var query2 = from i in source.AsParallel()
              select Normalize(i);
```

This code example creates two queries that transform values of the enumerable object **source**. The PLINQ version uses multiple cores if they are available.

You can also use PLINQ's **ForAll** extension method in cases where you want to iterate over the input values but you do not want to select output values to return. This is shown in the following code.

```
IEnumerable<MyObject> myEnumerable = ...

myEnumerable.AsParallel().ForAll(obj => DoWork(obj));
```

The **ForAll** extension method is the PLINQ equivalent of **Parallel.ForEach**.

It's important to use PLINQ's **ForAll** extension method instead of giving a PLINQ query as an argument to the **Parallel.ForEach** method. See "Mixing the Parallel Class and PLINQ" later in this chapter.

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, until you reach the point of diminishing returns that Amdahl's Law predicts. How much faster depends on the kind of work your loop does.

Adding cores makes your loop run faster; however, there's always an upper limit.

You must choose the right granularity. Too many small parallel loops can reach a point of over-decomposition where the multicore speedup is more than offset by the parallel loop's overhead.

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 as well as for PLINQ, 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 "Variations" section later in this chapter.

[Robust exception handling is an important aspect of parallel loop processing.](#)

Parallel loops have many variations. There are 12 overloaded methods for **Parallel.For** and 20 overloaded methods for **Parallel.ForEach**. PLINQ has close to 200 extension methods. Although there are many overloaded versions of **For** and **ForEach**, you can think of the overloads as providing optional configuration options. Two examples are a maximum degree of parallelism and hooks for external cancellation. These options allow the loop body to monitor the progress of other steps (for example, to see if exceptions are pending) and to manage task-local state. They are sometimes needed in advanced scenarios. To learn about the most important cases, see the "Variations" section in this chapter.

The Parallel Loop pattern is the first one in this book because it is probably the easiest parallel pattern to apply to existing code.

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

Check carefully for dependencies between loop iterations! Not noticing dependencies between steps is *by far* the most common mistake you'll make with parallel loops.

- **Writing to shared variables.** If the body of a 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, where **total** is shared across iterations.

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

If you encounter this situation, see Chapter 4, "Parallel Aggregation."

Shared variables come in many flavors. Any variable that is declared outside of the scope of the loop body is a shared variable. Shared references to types such as classes or arrays will implicitly allow all fields or array elements to be shared. Parameters that are declared using the keyword **ref** result in shared variables. Even reading and writing files can have the same effect as shared variables.

- **Using properties of an object model.** If the object being processed by a loop body exposes properties, you need to know 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's likely that the loop iterations are not independent. For all values of **i**, **SomeObject[i].Parent** is a reference to a single shared object.

You must be extremely cautious when getting data from properties and methods. Large object models are known for sharing mutable state in unbelievably devious ways.

- **Referencing data types that are not thread safe.** If the body of the parallel loop uses a data type that is not thread safe, the loop body is not independent (there is an implicit dependency on the thread context). An example of this case, along with a solution, is shown in "Using Task-Local State in a Loop Body" in the "Variations" section later in this chapter.
- **Loop-carried dependence.** If the body of a parallel for loop performs arithmetic on the loop index, there is likely to be a dependency that is known as *loop-carried dependence*. This is shown in the following code example. The loop body references **data[i]** and **data[i - 1]**. If **Parallel.For** is used here then there is no guarantee that the loop body that updates **data[i - 1]** has executed before the loop for **data[i]**.

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

It's sometimes possible to use a parallel algorithm in cases of loop-carried dependence, but this is outside of the scope of this book. Your best bet is to look elsewhere in your program for opportunities for parallelism or to 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.

Arithmetic on loop index variables, especially addition or subtraction, usually indicates loop-carried dependence.

When you look 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.

Don't expect miracles from profiling—it can't analyze your algorithms for you. Only you can do that.

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.

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

Sequential Credit Review Example

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;
    }
}
```

The **UpdatePredictionsSequential** method processes each account from the application's account repository. The **Fit** method is a utility function that uses the statistical least squares method to create a trend line from an array of numbers. The **Fit** method is a pure function. This means that it doesn't modify any state.

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.

Credit Review Example 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.

Credit Review Example with PLINQ

You can also use PLINQ to express a parallel loop. Here's 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. 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.AllAccounts** 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 every 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 on a quad-core computer shows that the **Parallel.ForEach** and PLINQ versions run slightly less than four times as fast as the sequential version. Timing numbers vary; you may want to run the online samples on your own computer.

Variations

The credit analysis example shows a typical way to use parallel loops, but there can be variations. This section introduces some of the most important ones. You won't always need to use these variations, but you should be aware that they are available.

Breaking Out of Loops Early

Breaking out of loops is a familiar part of sequential iteration. It's less common in parallel loops, but you'll sometimes need to do it. Here's an example of the sequential case.

```
int n = ...
for (int i = 0; i < n; i++)
{
    // ...
    if (/* stopping condition is true */)
        break;
}
```

The situation is more complicated with parallel loops because more than one step may be active at the same time, and steps of a parallel loop are not necessarily executed in any predetermined order. Consequently, parallel loops have two ways to break or stop a loop instead of just one. Parallel *break* allows all steps with indices lower than the break index to run before terminating the loop. Parallel *stop* terminates the loop without allowing any new steps to begin.

Parallel Break

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's an example.

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

```
int n = ...
Parallel.For(0, n, (i, loopState) =>
{
    // ...
    if (/* stopping condition is true */)
    {
        loopState.Break();
        return;
    }
})
```

```
});
```

This example uses an overloaded version of **Parallel.For** that passes a "loop state" object to each step. Here's the signature of the version of the **Parallel.For** method that was used in the example.

```
Parallel.For(int fromInclusive,  
             int toExclusive,  
             Action<int, ParallelLoopState> body);
```

The object that's passed to the **loopState** argument is an instance of the **ParallelLoopState** class that was created by the parallel loop for use within the loop body.

Calling the **Break** method of the **ParallelLoopState** object begins an orderly shutdown of the loop processing. Any steps that are running as of the call to **Break** will run to completion.

Calling **Break** doesn't stop other steps that might have already started running.

You may want to check for a break condition in long-running loop bodies and exit that step immediately if a break was requested. If you don't do this, the step will continue to run until it finishes. To see if another step running in parallel has requested a break, retrieve the value of the parallel loop state's **LowestBreakIteration** property. If this returns a nullable long integer whose **HasValue** property is **true**, you know that a break has been requested. You can also read the **ShouldExitCurrentIteration** property of the loop state object, which checks for breaks as well as other stopping conditions.

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.

Don't forget that all steps with an index value less than the step that invoked the **Break** method will be allowed to run normally, even *after* you've called **Break**.

Because of parallel execution, it's possible that more than one step may call **Break**. In that case, the lowest index will be used to determine which steps 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 (/* break condition is true */)   
    {  
        loopState.Break();  
    }  
});
```

```

        return;
    }
    result[i] = DoWork(i);
});

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 steps with a higher index value than the one that called the **Break** method may have been started before the call to **Break** occurs.

Be aware that some steps with index values *higher* than the step that called the **Break** method might be run. There's no way of predicting when or if this might happen.

The **Parallel.ForEach** method 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.

The **Parallel.ForEach** method also supports the loop state **Break** method.

Parallel Stop

There are also situations 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 parallel stop.

Use **Stop** to exit a loop early when you don't need all lower-indexed iterations to run before terminating the loop.

```

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

if (!loopResult.IsCompleted &&
    !loopResult.LowestBreakIteration.HasValue)

```

```
{
    Console.WriteLine("Loop was stopped");
}
```

When the **Stop** method is called 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.

Parallel programs use **Stop** more often than **Break**. Processing all iterations with indices less than the stopping iteration is usually not necessary when the loop bodies are independent of each other. It's also true that **Stop** shuts down a loop more quickly than **Break**.

You'll probably use **Stop** more often than **Break**.

There's no **Stop** method for a PLINQ query, but you can use the **WithCancellation** extension method and then use cancellation as a way to stop PLINQ execution. See the "External Loop Cancellation" section.

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 to stop what you're doing.

In .NET, you use the **CancellationTokenSource** class to signal cancellation and the **CancellationToken** structure to detect and respond to a cancellation request. The structure allows you to find out if there is a pending cancellation request. The class lets you signal that cancellation should occur.

The **Parallel.For** and **Parallel.ForEach** methods include overloaded versions that accept parallel loop options as one of the arguments. You can specify a cancellation token as one of these options. If you provide a cancellation token as an option to a parallel loop, the loop will use that token to look for a cancellation request. Here is an example.

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

    var options = new ParallelOptions
        { CancellationToken = token };

    try
    {
        Parallel.For(0, n, options, (i) =>
        {
            // ...

            // ... optionally check to see if cancellation happened
            if (token.IsCancellationRequested)
            {
```

```

        // ... optionally exit this iteration early
        return;
    }
});
}
catch (OperationCanceledException ex)
{
    // ... handle the loop cancellation
}
}

```

Here is the signature of the **Parallel.For** method that was used in the example.

```

Parallel.For(int fromInclusive,
             int toExclusive,
             ParallelOptions parallelOptions,
             Action<int> body);

```

When the caller of the **DoLoop** method is ready to cancel, it invokes the **Cancel** method of the **CancellationTokenSource** class 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.

[External cancellation requires a cancellation token source object.](#)

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 for **Break** and **Stop** that is described in the "Breaking Out of Loops Early" section.

You can use the **WithCancellation** extension method to add external cancellation capabilities to a PLINQ query.

Exception Handling

If the body of a parallel loop throws an unhandled exception, the parallel loop no longer begins any new steps. 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 will throw an exception in the context of the thread that invoked it. 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.

[Throwing an unhandled exception prevents new iterations from starting.](#)

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

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

See the "Exception Handling" section in chapter 3 for a code example and more information about handling aggregate exceptions.

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

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

The parallel extensions of .NET Framework 4 include support for custom partitioning. A **Partitioner** object divides the indices into non-overlapping intervals named ranges. With partitioners, each parallel loop step handles a range of indices instead of individual indices. By grouping iterations into ranges, you can avoid some of the overhead of a normal parallel loop. Here is an example.

```
int n = ...
double[] result = new double[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] = (double)(i * i);
        }
    });
```

Here is the signature of the **Parallel.For** method that was used in the example.

```
Parallel.ForEach<TSource>(
    Partitioner<TSource> source,
    Action<TSource> body);
```

In this example, you can think of the result of the **Partitioner.Create** method as an object that acts like an instance of **IEnumerable<Tuple<int, int>>** (the actual type is **Partitioner<Tuple<int, int>>**). In other words, **Create** gives you access to a collection of tuples (unnamed records) with two integer field values. Each tuple represents a range of index values that should be processed in a single iteration of the parallel loop. Each iteration of the parallel loop contains a nested sequential **for** loop that processes each index in the range.

The partition-based syntax for parallel loops is more complicated than the syntax for 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.

The number of ranges that will be created by a **Partitioner** object depends on the number of cores in your computer. The default number of ranges is approximately three times the number of those cores.

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

```
double[] result = new double[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] = (double)(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.

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 "Further Reading" section at the end of this chapter.

Controlling the Degree of Parallelism

Although you usually let the system manage how iterations of a parallel loop are mapped to your computer's cores, in some cases, you may want additional control.

You'll see this variation of the Parallel Loop pattern in a variety of circumstances. 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.](#)

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 tasks that can be used simultaneously by the parallel loop. For example, the **MaxDegreeOfParallelism** property of the **ParallelOptions** object refers to the maximum number of worker tasks that will be scheduled at any one time by a parallel loop.

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

managed by the underlying components of the system; the implementation of the **Parallel** class, the default task scheduler, and the .NET thread pool all play a role in optimizing throughput under a wide range of conditions.

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

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

In this code example, the parallel loop will run using at most two tasks at any one time. Here is the signature of the **Parallel.For** method that was used in the example.

```
Parallel.For(int fromInclusive,
            int toExclusive,
            ParallelOptions parallelOptions,
            Action<int> body);
```

You can also configure the maximum number of worker threads 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 => /* ... */);
```

The query in the code example will run with a maximum of eight tasks at any one time.

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 Task-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.

You must use task-local state for loop bodies that make calls to methods that are not thread safe.

Here is an example that uses 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.

Task-local state is available if your loop body needs it.

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

Parallel.ForEach(
    Partitioner.Create(0, numberOfSteps),
    new ParallelOptions(),
    () => { return new Random(MakeRandomSeed()); },
    (range, loopState, random) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
            result[i] = random.NextDouble();
        return random;
    },
    _ => {});
```

Here is the signature of the overload of version of the **Parallel.ForEach** method that was used in the example.

```
ForEach<TSource, TLocal>(
    OrderablePartitioner<TSource> source,
    ParallelOptions parallelOptions,
    Func<TLocal> localInit,
    Func<TSource, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally)
```

The **Parallel.ForEach** loop will create a new instance of the **Random** class for each of its worker tasks. 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 returned value is always the same object that was passed in.

The default constructor of the **Random** class uses the system's clock to generate a random seed. It's possible that the default constructor will use the same seed if the constructor is called in twice in short succession. Therefore, the code makes sure that a distinct seed value is used for each new **Random** object by passing a unique seed value as an argument to the constructor.

Calling **Random** twice in short succession may use the same random seed. Provide your own random seed to prevent duplicate random sequences.

This example is designed to show the use of task-local state. In the **Parallel** class, thread-local state is provided by task-local state. A task is guaranteed to execute on just one thread throughout its entire run.

Generating pseudorandom numbers using the **Random** class in parallel can result in overlapping sequences of numbers on different threads. If your application really requires statistically robust pseudorandom values you should consider using the **RNGCryptoServiceProvider** class, or a third-party library. For more information about generating random numbers see the "Further Reading" section in this chapter.

The **Random** class isn't the right random generator for all simulation scenarios. It's used here only as an example of a class that isn't thread safe.

Using a Custom Task Scheduler for a Parallel Loop

You can substitute custom task scheduling logic for the default task scheduler that uses **ThreadPool** worker threads. For example, you can do this to enable a maximum degree of parallelism that operates across multiple loops instead of over only a single loop, which is the default. Additionally, you can use a custom task scheduler if you want to use a dedicated set of threads to process a parallel loop instead of pulling worker threads from the shared 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 the built-in task schedulers don't meet your needs. 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.

```
int n = ...
TaskScheduler myScheduler = ...
var options = new ParallelOptions()
                { TaskScheduler = myScheduler };
Parallel.For(0, n, options, i =>
{
    // ...
});
```

Here is the signature of the **Parallel.For** that was used in the code example.

```
Parallel.For(int fromInclusive,
            int toExclusive,
            ParallelOptions parallelOptions,
            Action<int> body);
```

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

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

Anti-Patterns

Anti-patterns are cautionary tales. They highlight issues that need to be carefully considered as well as 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 foreach with the values you want. Also, be aware that a step size other than one often indicates a data dependency. Carefully analyze such a computation before you convert it into a parallel loop.

In sequential programs you rarely iterate from high to low values unless the order matters. If your loop has a negative step size, the ordering of the loop is probably significant, and the iterations may not be independent.

Beware! A step size other than one can indicate a loop dependency.

Here's a helpful technique. Before you convert a sequential loop to a parallel loop, you can temporarily reverse the order of the indices (for example, so that they go from high to low). If your sequential code still runs correctly, then that's a bit of evidence that the steps of the loop are independent. (The test isn't foolproof; you still need to understand the algorithm.)

Hidden Loop Body Dependencies

Incorrect analysis of loop dependencies is a frequent source of software defects. Be careful that all parallel loop bodies do 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** or **DbConnection**, which are not thread safe, across parallel iterations is an example of a subtle dependency.

When loop bodies are not fully independent of each other, it may still be possible to use parallel loops. However, in these cases, you must make sure that all shared variables are protected and synchronized, and you must understand the performance characteristics of any synchronization you add. Adding synchronization can *greatly* reduce the performance of a parallel program, but forgetting to add necessary synchronization can result in a program with bugs that are catastrophic and difficult to reproduce.

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."

Small Loop Bodies with Few Iterations

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.

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 core utilization per task. For example, with four cores and 50 percent average core 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. One situation where you may want to restrict the degree of parallelism is 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.

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 F(d);

q.ForAll(item =>
{
    // ... process item
});
```

Duplicates in the Input Enumeration

If you are using the **Parallel.ForEach** method, duplicate objects in the enumeration often indicate an unsafe race condition in your code. If an object (that is, an instance of a class) appears more than once in the input to the loop, then it's possible that two parallel threads could try to update that object at the same time.

Don't allow duplicate instances in parallel loops. If an object appears more than once in the input to a loop, then it's possible that two parallel threads could update the object at the same time.

Design Notes

Here are some fine points about using parallel loops.

Adaptive Partitioning

Parallel loops in .NET use an adaptive partitioning technique where the size of units of work increases over time. Adaptive partitioning is able to meet the needs of both small and large input ranges.

The **Parallel** class adapts to the number of iterations, from only a few to many millions.

Adaptive Concurrency

The **Parallel** class and PLINQ work on slightly different threading models in the .NET 4 Framework. PLINQ uses a fixed number of tasks to execute a query; by default, it creates the same number of tasks as there are 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 tasks. That's why, for example, the **ParallelOptions** class has a **MaxDegreeOfParallelism** property instead of a "MinDegreeOfParallelism" property. The idea is that the system can use fewer threads than requested to process a loop.

The .NET thread pool adapts dynamically to changing workloads by allowing the number of worker threads for parallel tasks to change over time. At run time, the system observes whether increasing the number of threads improves or degrades overall throughput and adjusts the number of worker threads accordingly.

Be careful if you use parallel loops with individual steps that take several seconds or more. This can occur with I/O-bound workloads as well as lengthy calculations. 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.

If the individual steps of your loop take a long time, you may see more worker threads than you intend.

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. 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.

Support for Nested Loops and Server Applications

The .NET Framework parallel extensions allow you to nest parallel loops. In this situation, the run-time environment coordinates the use of processor resources. You will find that with nested parallel loops each loop uses fewer threads than in a nonnested loop.

A related problem is the handling of parallel loops in server applications. The **Parallel** class attempts to deal with multiple AppDomains in server applications in exactly the same manner that nested loops are handled. If the server application is already using all of the available thread pool threads to process other ASP.NET requests, a parallel loop will just run on the thread that invoked it. If the workload decreases and other threads become available and there's no other ASP.NET work to be processed, then the loop will start using additional threads.

Related Patterns

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

Exercises

1. Which of the following problems could be solved using the parallel loop techniques taught in this chapter?
 - a. Sorting an in-memory array of numbers with a million elements.
 - b. Putting the words in each line read from a text file in alphabetical 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**? What is the type 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 <http://parallelpatterns.codeplex.com>. 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 that you can set the **MaxDegreeOfParallelism** property. Observe the effect of different values of this property on the execution time when running on computers with different numbers of cores.
-

Further Reading

The examples in this book use features and libraries of recent C# versions. Some helpful books are by Bishop, and Albahari and Albahari. These sources contain information about lambda expressions.

The book by Mattson, et al. describes software design patterns for parallel programming that are not specialized for a particular language or library.

Many of the cases in the "Variations" section of this chapter are from the report by Toub, which contains additional examples and variations.

For more information about partitioners, refer to the in-depth article on this subject by Toub.

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.

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

T. G. Mattson, "Use and Abuse of Random Numbers" (video), Feb 14, 2008,
<http://software.intel.com/en-us/videos/tim-mattson-use-and-abuse-of-random-numbers/>

S. Toub. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#, 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>

S. Toub. Custom Parallel Partitioning With .NET 4, April 26, 2010,
<http://www.drdoobs.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. The Parallel Tasks pattern is sometimes called the fork/join pattern or the master/worker pattern.

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

Data parallelism and task parallelism are two ends of a spectrum. Data parallelism occurs when a *single operation* is applied to many inputs. Task parallelism uses *multiple operations*, each with its own input.

In the .NET Framework, tasks are implemented by the **Task** class in the **System.Threading.Tasks** namespace. The static **Task.Factory** property contains an instance of the **TaskFactory** class, which is used to create and schedule new tasks. You can wait for a task to complete by invoking the task's **Wait** method. It's also possible to wait for multiple tasks to complete. If you think in terms of fork/join, the **StartNew** method is the fork operation and the **Wait** method is the join operation.

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

Scheduling is an important aspect of parallel tasks. Unlike threads, new tasks do not necessarily begin executing immediately. Instead, they are placed in a work queue. Tasks run when their associated task scheduler removes them from the queue, usually as cores become available. The task scheduler attempts to optimize overall throughput by controlling the system's degree of concurrency. As long as there are enough tasks and the tasks are sufficiently free of serializing dependencies, the program's performance scales with the number of available cores. In this way tasks embody the concept of potential parallelism that was introduced in Chapter 1.

Another important aspect of task-based applications is how they handle exceptions. In .NET, an unhandled exception that occurs during the execution of a task is deferred for later observation. For example, the deferred exception is automatically observed at a later time when you call one of the task class's wait methods. At that time, the exception is rethrown in the calling context of the wait method.

This allows you to use the same exception handling approach in parallel programs that you use in sequential programs.

Tasks in .NET defer exceptions and rethrow them when one of the task's wait methods 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 a list of parallel tasks and waits for them all 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 "The Default Task Scheduler" 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 the "Exception Handling" section of 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 t1 = Task.Factory.StartNew(DoLeft);  
Task t2 = Task.Factory.StartNew(DoRight);  
  
Task.WaitAll(t1, t2);
```

Use the **StartNew** factory method to create a task and schedule it to run. Use **Task.WaitAll** to wait for more than one task to complete before proceeding.

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 can wait for parallel tasks to complete by calling the **Task.WaitAll** method. (Use the **Wait** method if you want to wait on a single task.)

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 run until its task scheduler takes it out of the work queue, which can happen immediately or can occur at some point in the future.

Two tasks, **t1** and **t2**, are needed in this example for correct exception handling. Without the second task, **t2**, the code would not have guaranteed that exceptions thrown during the execution of task **t1** would be observed. See the "Exception Handling" section in this chapter.

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 with 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> in the Chapter3\ImageBlender folder.

```
static int SequentialImageProcessing(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.StartNew** 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 the **Parallel.Invoke** method to achieve parallelism. The **Parallel.Invoke** method has very convenient syntax. This is shown in the following code.

```
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.

Variations

This section describes variations of the .NET implementation of the Parallel Task pattern.

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 running task must poll for the existence of a cancellation request at appropriate intervals and then shut itself down by calling back into the library.

Tasks in .NET use a cooperative cancellation model.

The .NET framework uses separate types. One allows a program request cancellation and the other checks for cancellation requests. 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();

        // Body of for loop.
    }
}, token);

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

The **CancellationTokenSource** object contains a cancellation token. This token is used in two places. It's 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 **OperationCanceledException** should not be handled by user code within the task (it is often handled, however, outside of the task that was canceled). 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**.

To ensure that a running task transitions to status **Canceled**, a cancellation token must have been given as an argument to the **StartNew** method and an unhandled **OperationCanceledException** exception that contains that same cancellation token must be thrown by the task.

There are several requirements that must be met before a task will transition to status **Canceled**.

Checking for cancellation within a loop with many iterations that each 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. For example, in an interactive GUI-based application, checking for cancellation more than once per second is probably a good idea. An application that runs in the background could poll for cancellation much less frequently, perhaps every two to ten seconds. Profiling your application can give you performance data that you can use when determining the best places to test for cancellation requests in your code. You need to understand which code locations will allow you to poll for cancellation at evenly spaced intervals of your choosing, and profiling can help you do this.

If a cancellation token is passed to the **StartNew** method and cancellation is signaled on that token before the new task begins to execute, then the new task's status will transition to **Canceled** without ever running.

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

Whether a task gets the status **Canceled** or not can be important. If you don't do this correctly, the process might be unexpectedly terminated when the improperly canceled task is garbage collected. A typical mistake is forgetting to pass the cancellation token to the **StartNew** method. Another typical mistake is forgetting to throw the **OperationCanceledException** that contains the required cancellation token.

Handling Exceptions

If the execution of a **Task** object's 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 (TPL) 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. The exception is also thrown in the calling context of the **Wait** method. The **Task** class's static **WaitAll** method allows you to observe the unhandled exceptions of more than one task with a single method invocation. The **Parallel.Invoke** method includes an implicit call to **WaitAll**. Exceptions from all of the tasks are grouped together in an **AggregateException** object and thrown in the calling context of the **WaitAll** or **Wait** method.

Getting the **Exception** property of a faulted task causes the task's unhandled exception to be observed. The property returns the aggregate exception object. Getting the value does not automatically cause the exception to be thrown; however, the exception is considered to have been observed when you get the value of the **Exception** property. Use the **Exception** property instead of the **Wait** method or **WaitAll** method when you want to examine the unhandled exception but do not want it to be rethrown in the current context.

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.

You must take care to observe the unhandled exceptions of each task. If you don't do this, .NET's exception escalation policy can terminate your process when the task is garbage collected.

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. TPL 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**. Throwing an aggregate exception preserves the inner exception's stack trace, which would otherwise be overwritten if the exception were rethrown in the current context.

The Task Parallel Library uses **AggregateException** 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 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 should return **true** if it has handled the inner exception and **false** if it has not handled the inner exception. Here's an example.

Use the **Handle** method to iterate through inner exceptions.

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

```

}
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 inner 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 the unhandled ones as inner exceptions.

If the **Handle** method doesn't meet your needs, you can write your own code that iterates through the **InnerExceptions** property of an **AggregateException** object.

The Flatten Method

The **Flatten** method of the **AggregateException** class is useful when tasks are nested within other tasks. In this case, it's 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 flat sequence of inner exceptions. It's 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 includes a method named **Task.WaitAll** that allows you to wait for a set of tasks to complete. However, you can also wait for the first task to complete by calling the **Task.WaitAny** method.

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

Here is an example.

```

var taskIndex = -1;

Task[] tasks = new Task[]
{
    Task.Factory.StartNew(DoLeft),
    Task.Factory.StartNew(DoRight),
    Task.Factory.StartNew(DoCenter)
};
Task[] allTasks = tasks;

// Print completion notices one by one as tasks finish.
while (tasks.Length > 0)
{
    taskIndex = Task.WaitAny(tasks);
    Console.WriteLine("Finished task {0}.", taskIndex + 1);
    tasks = tasks.Where((t) => t != tasks[taskIndex]).ToArray();
}

// Observe any exceptions that might have occurred.
try
{

```

```

    Task.WaitAll(allTasks);
}
catch (AggregateException ae)
{
    ...
}

```

This example shows how you can use the **WaitAny** method to be notified when the first of a group of tasks completes. The code prints a line to the console output after each task is finished. The **while** loop exits when all of the tasks have finished.

You need to observe any exceptions that may have occurred, but note that exceptions are never observed by the **WaitAny** method. Instead, you should add a step that checks for exceptions whenever you use the **WaitAny** method. While this example uses the **WaitAll** method to observe exceptions you could also use the **Exception** property or the **Wait** method for this purpose. The **IsFaulted** property of the **Task** class can be used to check to see whether an unhandled exception occurred within a task.

Exceptions are never observed by the **WaitAny** method. However, you must make sure that all exceptions are eventually observed using one of the techniques that were described in the “Exception Handling” section of this chapter.

Speculative Execution

Speculative execution is a variation that occurs where you perform an operation in anticipation of a particular result. For example, you might predict that the current computation being run will output the value 42, and you start the next computation that depends on this computation's result by using 42 as the input. If the first computation ends with 42, you've now gained parallelism by successfully starting the dependent operation well in advance of when you otherwise could have. If the first computation results in something other than 42, then you can restart the second operation using the correct value.

Another example of speculative execution 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.

Here's an example.

```

SpeculativeInvoke(SearchLeft, SearchRight, SearchCenter);

```

This example executes three delegate methods 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 **SpeculativeInvoke** method executes each of the delegates in the **params** array argument. Its implementation is shown in the following code.

```

public static void SpeculativeInvoke(

```

```

        params Action<CancellationToken>[] actions)
{
    var cts = new CancellationTokenSource();
    var token = cts.Token;
    var tasks =
        (from a in actions
         select Task.Factory.StartNew(() => a(token), token))
        .ToArray();

    // Wait for fastest task to complete.
    Task.WaitAny(tasks);

    // Cancel all of the slower tasks.
    cts.Cancel();

    // Wait for cancellation to finish and observe exceptions.
    try
    {
        Task.WaitAll(tasks);
    }
    catch (AggregateException ae)
    {
        // Filter out the exception caused by cancellation itself.
        ae.Flatten().Handle(e => e is OperationCanceledException);
    }
    finally
    {
        if (cts != null) cts.Dispose();
    }
}

```

The **WaitAny** method does not observe unhandled task exceptions. This is because you need the returned index from **WaitAny** to know which task completed (potentially by throwing an exception). 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.

Creating Tasks with Custom Scheduling

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 **TaskFactory.StartNew** method.

[You can customize the way tasks are scheduled.](#)

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. Other cases occur when the load-balancing heuristics of the default task scheduler don't work well for your application. See the "Thread Injection" section later in this chapter for more information.

Unless you specify otherwise, any new tasks will use the current task scheduler, which is the value of the static property **TaskScheduler.Current**. In other words, subtasks inherit the task scheduler of the enclosing task context. Unless otherwise specified, new top-level tasks (and consequently their subtasks) 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 handles a wide variety of operating conditions. It's a good choice for most applications. See "The Default Task Scheduler" section in this chapter for more information. The **TaskScheduler** class's **FromCurrentSynchronizationContext** static method returns a task scheduler object that schedules the task to run wherever the current thread's **SynchronizationContext** property indicates. This is sometimes the current thread but not necessarily. This task scheduler is useful in cases of thread affinity. Not every thread has a current synchronization context, and there is no API to get a task scheduler that runs in the synchronization context of a thread other than the current thread. For example, you can't use the **FromCurrentSynchronizationContext** method if you're trying to schedule a task in a Windows service's main thread.

You can implement your own task scheduler class. See the "Writing a Custom Task Scheduler" section in this chapter for more information.

Anti-Patterns

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

Variables Captured by Closures

In C#, a closure can be 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 and it's easy to make mistakes. Unless you understand the semantics, you may find that captured variables do not behave as you expect, especially in parallel programs.

Problems occur when you reference a variable without considering its scope. Here's an example:

```
for (int i = 0; i < 4; i++)
{
    // WARNING: BUGGY CODE, i has unexpected value
}
```

```
Task.Factory.StartNew(() => Console.WriteLine(i));  
}
```

Capturing a loop index variable in a closure is usually a bug. Be on the lookout for this very common coding error.

You might think that this code sample would print the numbers 1, 2, 3, 4 in some arbitrary order, but it can print other values, depending on how the threads happen to run. For example, you might see 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 the single, shared variable `i` will probably be different from the value of `i` when the task was created. All of the tasks are sharing the same variable.

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 the numbers 1, 2, 3, 4 in an arbitrary order, but each number will be printed. 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's 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 the **Dispose** method 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. Here is an example.

Be careful not to dispose resources needed by a pending task.

```
Task<string> t;  
using (var file = new StringReader("text"))  
{  
    t = Task<string>.Factory.StartNew(() => file.ReadLine());  
}  
// WARNING: BUGGY CODE, file has been disposed  
Console.WriteLine(t.Result);
```

The **using** keyword introduces an implicit try/finally block that invokes the **Dispose** method on the value of the variable when it exits its scope. The **using** keyword can't be used with captured variables. Instead, you need to call the **Dispose** method when you know that the disposable object is no longer needed. In this example, **Dispose** can only be called after the task's **Result** property is read.

Avoid Thread Abort

Terminating tasks with the **Thread.Abort** method leaves the **AppDomain** in a potentially unusable state. Also, aborting a thread pool worker thread is never recommended. If you need to cancel a task, use the technique described in the "Canceling a Task" section in this chapter. Do not abort the task's thread.

Never attempt to cancel a task by calling the **Abort** method of the thread that is executing the task.

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.

The task 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 **Monitor.Enter** function will be executed in a different thread than the **Monitor.Exit** function.

Task Life Cycle

The **Status** property of a **Task** instance tracks its life cycle. Figure 1 shows the life cycle of the tasks that are described in this chapter:

Task Life Cycle

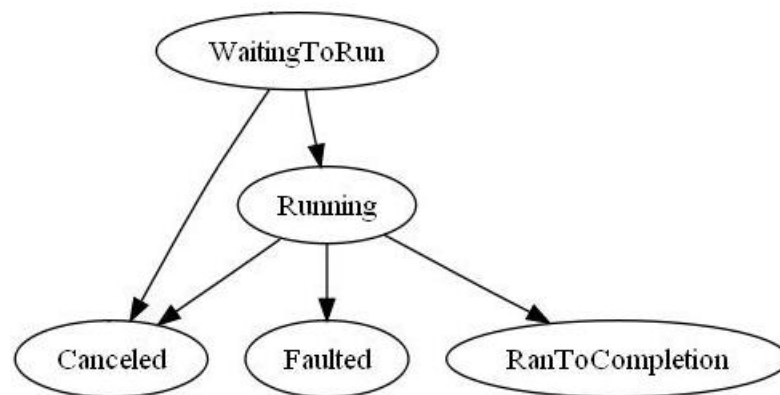


Figure 1

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's status transitions to **TaskStatus.Running**. Once the task has begun to run, it has three possible outcomes. If the

task's user delegate exits normally, the task's status transitions to **TaskStatus.RanToCompletion**. If the task's user delegate throws an unhandled exception, then the task's status becomes **Task.Faulted**.

It's 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 use a factory method to create tasks instead the **new** operator.

Writing a custom task scheduler

The .NET Framework includes two task scheduler implementations: the default task scheduler and a task scheduler that runs tasks on a target synchronization context. 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. Here are a few examples.

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 is that 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 is part of Microsoft's **Parallel Samples** package. See the "Further Reading" section for details of where to get these additional samples. This package includes an implementation of a **QueuedTaskScheduler** class that provides multiple global task queues that use round-robin scheduling. A custom scheduler doesn't need to use its own threads; it can simply limit the number of tasks that are allowed to run concurrently but still use the thread pool. This is the approach taken by the **LimitedConcurrencyLevelTaskScheduler** class in the **ParallelExtensionsExtras**. See the "Further Reading" section in this chapter for more information.

Unobserved Task Exceptions

If you don't 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 the finalizer thread context. The finalizer thread is the system thread that invokes the **Finalize** 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 very 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 as they propagate into the finalizer context.

This last 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.

During finalization, tasks that have a **Status** property of **Faulted** are treated differently from tasks with status **Canceled**. The task's status determines how unobserved task exceptions that arise from task cancellation are treated during finalization. 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.

Relationship between Data Parallelism and Task Parallelism

Parallel.Invoke was described as creating tasks for each of the delegates passed in its **params** array. As a mental model, this is fine. In practice, this isn't always how it's actually executed. For example, with enough delegates in the **params** list, TPL may, for performance reasons, use a parallel loop to invoke all of the delegates.

This highlights how task parallelism and data parallelism are related. If you represent each operation to be performed as a delegate, you can then use data parallelism to perform the same operation (invoking the delegate) on each piece of data (the delegate). Conversely, if you have a data parallel problem, you can approach it in a task-parallel manner by spinning up a task to process each individual operation.

The Default Task Scheduler

All parallel libraries rely on a scheduler to organize and order the execution of tasks and threads, and TPL is no exception. This section is a behind-the-scenes look at the .NET 4 implementation of task

scheduling. The material in this section can help you understand the performance characteristics that you'll 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, and future versions of .NET might optimize task execution differently.

The behind-the-scenes behavior described in this section applies to .NET 4. There's no guarantee that future releases of the runtime or the framework won't behave 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 to the work items 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 Figure 2.

A Thread Pool

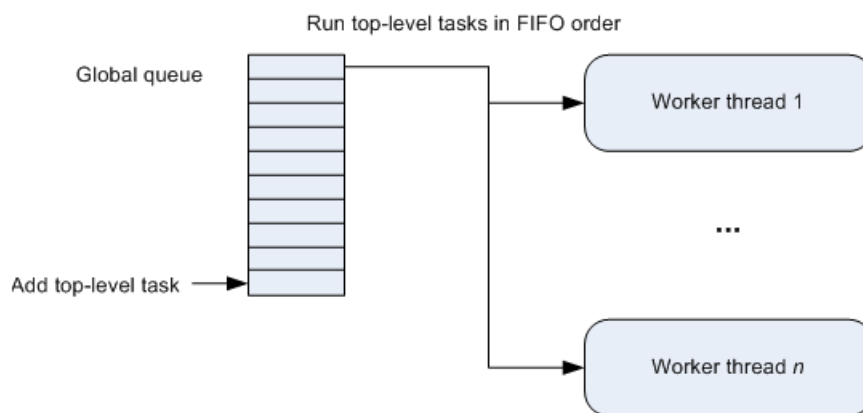


Figure 2

Thread pools have problems scaling to large numbers of cores. The main reason is that the thread pool has a single, global work queue. Each end of 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 processor 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. 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 millions of tasks. 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 .NET Framework provides local task queues for each worker thread in the 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. Giving different parts of the application their own work queues helps avoid a central bottleneck. This is shown Figure 3.

Per-Thread Local Task Queues

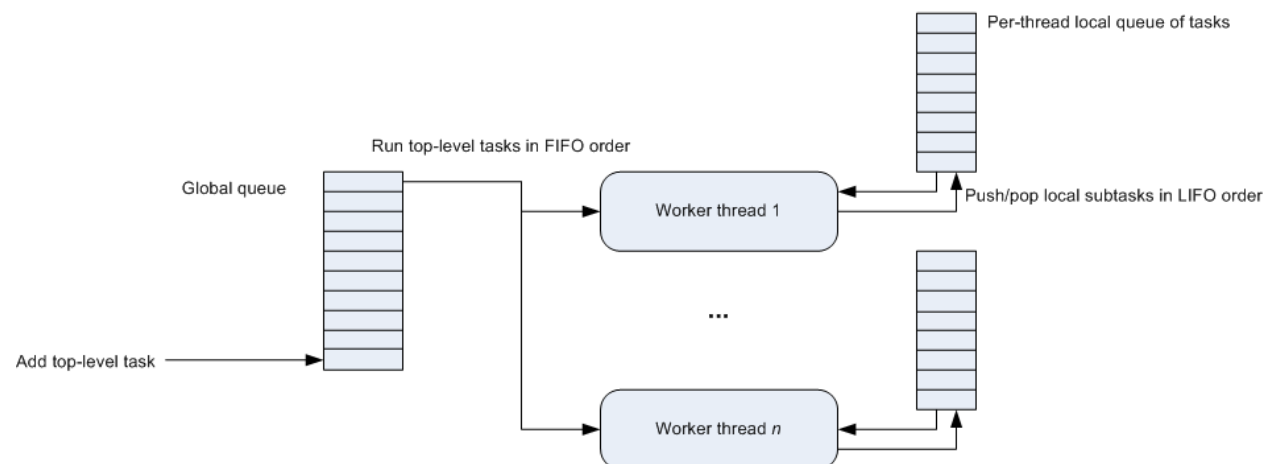


Figure 3

You can see that there are as many task queues as there are worker threads, plus one global queue. 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 heavier synchronization costs than adding to a local queue.

In the typical case, accessing the local queue needs only very limited synchronization. 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 is a double-ended queue that has a private end and a public end. The queue allows lock-free pushes and pops from the private end but requires costlier 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 Figure 4.

Work Stealing

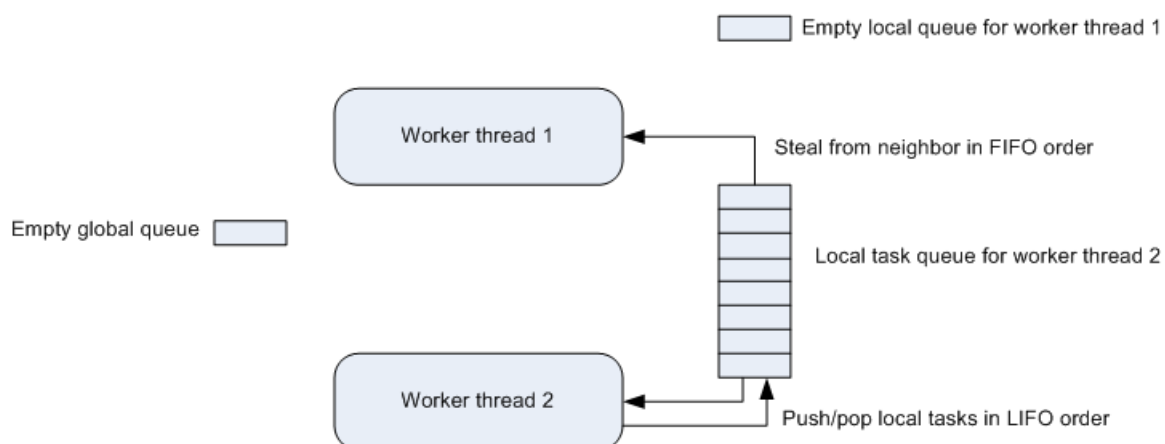


Figure 4

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 with minimal 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 reduces the likelihood of a cache miss. 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, as you will see later in Chapter 6, "Dynamic Task Parallelism." The largest chunks of work tend to get pushed onto the queue before smaller subtasks. With FIFO ordering, these larger chunks are the first to be removed by 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.

In this book, tasks in the global queue are called as top-level tasks. Top-level tasks have approximately 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 faster operation than placing it in the global queue.

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.

In this book, 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's 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 TPL 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** or **Task.WaitAll** is called from within the worker thread whose local queue contains that subtask. Inlining also applies to tasks created by methods of the **Parallel** class if these methods are called from within a worker thread. 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.

Thread Injection

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 .NET thread pool has two main mechanisms for injecting threads: a starvation-avoidance mechanism that adds worker threads if it sees no progress being made on queued items and a hill-climbing heuristic that tries to maximize throughput while using as few threads as possible.

The goal of starvation avoidance is to prevent deadlock. 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 thread pool's global or local queues. 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 goal of the hill-climbing heuristic is to improve the utilization of cores when threads are blocked by I/O or other wait conditions that stall the processor. 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's blocked and a thread that's performing a lengthy, processor-

intensive operation. Therefore, whenever the thread pool's global or local queues contain 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.

The .NET thread pool has an opportunity to inject threads every time a work item completes or at 500 millisecond intervals, whichever is shorter. The thread pool uses this opportunity to try adding threads (or taking them away), guided by feedback from previous changes in the thread count. If adding threads seems to be helping throughput, the thread pool adds more; otherwise, it reduces the number of worker threads. This technique is called the hill-climbing heuristic.

Therefore, one reason to keep individual tasks short is to avoid "starvation detection" but another is to give the thread pool more opportunities to improve throughput by adjusting the thread count. The shorter the duration of individual tasks, the more often the thread pool can measure throughput and adjust the thread count accordingly.

To make this concrete, consider an extreme example. Suppose that you have a complex financial simulation with 500 processor-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 processor-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 processor 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.

As long as your tasks don't each take minutes, the thread pool's hill-climbing algorithm will eventually realize it has too many threads and cut back on its own accord. However, if you do have tasks that occupy a worker thread for many seconds or minutes or hours, that will throw off the thread pool's heuristics, and at that point you should consider an alternative.

The first option is to decompose your application into shorter tasks that complete fast enough for the thread pool to successfully control the number of threads for optimal throughput.

A second possibility is to implement your own task scheduler object that does not perform thread injection. If your tasks are of long duration, you don't need a highly optimized task scheduler because the cost of scheduling will be negligible compared to the execution time of the task. MSDN has an

example of a simple task scheduler implementation that limits the maximum degree of concurrency. See the "Further Reading" section for more information.

As a last resort you can use the **SetMaxThreads** method to configure the **ThreadPool** class with an upper limit for the number of worker threads, usually equal to the number of cores (this is the **Environment.ProcessorCount** property). This upper limit applies for the entire process, including all AppDomains.

The **SetMaxThreads** method can cause deadlock if thread pool worker threads are waiting for scheduled work items to run. Use it with extreme caution.

Bypassing the Thread Pool

If you don't want a task to use a worker thread from 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.

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 don't they 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? Answer the same question for the sample that uses **Parallel.Invoke**.

Further Reading

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

Hoag provides a detailed discussion of task creation options.

Microsoft's Parallel Samples package is ParExtSamples at MSDN.

The How to article at MSDN is a good resource for writing custom task schedulers.

The Task Scheduler Events page at MSDN provides more information about unobserved task exceptions.

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 Applications, pages 227 - 242. ACM, 2009.

J. E. Hoag. A Tour of Various TPL Options. April 2009.

<http://blogs.msdn.com/b/pfxteam/archive/2010/04/19/9997552.aspx>

ParExtSamples software, Samples for Parallel Programming with the .NET Framework 4.

<http://code.msdn.microsoft.com/ParExtSamples>

How to: Create a Task Scheduler That Limits the Degree of Concurrency <http://msdn.microsoft.com/en-us/library/ee789351.aspx>

TaskScheduler Events, .NET Framework Class Library. http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler_events.aspx

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. 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. Using unshared, local variables for partial, locally calculated results 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. For more information about the algorithmic aspects of this pattern, see the "Design Notes" section in this chapter.

The Basics

The most familiar application of aggregation is calculating a sum. Here's a 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 .NET Framework's Language Integrated Query (LINQ) provides a very simple way to express this kind of aggregation and languages such as C#, F# and Visual Basic provide special syntax for LINQ. The following LINQ expression calculates a sum.

```
double[] sequence = ...
```

```
return (from x in sequence select Normalize(x)).Sum();
```

The LINQ expression 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 the compiler to bind to PLINQ rather than to LINQ. The program will use the parallel versions of all further query operations within the expression. The **Sum** extension method executes the query and (behind the scenes and in parallel) calculates the sum of the selected, transformed values. For an introduction to PLINQ, see Chapter 2, "Parallel Loops."

This example 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). These 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's an example.

```
double[] sequence = ...  
return (from x in sequence.AsParallel() select Normalize(x))  
        .Aggregate(1.0d, (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 adding locks in your code. All of the synchronization occurs internally, within PLINQ.

If PLINQ doesn't meet your needs or if you prefer a less declarative style of coding, you can also use **Parallel.For** or **Parallel.ForEach** to implement the parallel aggregation pattern. The **Parallel.For** and **Parallel.ForEach** methods require more complex code than PLINQ. For example, the **Parallel.ForEach** method requires your code to include synchronization primitives to implement parallel aggregation. For

examples and more information, see "Using Parallel Loops for Aggregation" in the "Variations" section of this chapter.

An Example

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

The example is of a social network service, where subscribers can designate other subscribers as friends. The site 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 *multiset*. In a multiset, 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 multiset is a collection where each element only occurs once, yet it can represent duplicates (or larger multiplicities). The order of elements in a multiset doesn't matter.

The recommendation service uses the Map/Reduce 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 multiset 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.

Figure 1

An important feature of the map/reduce 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 multisets; therefore, the map stage does not just produce a list of candidate IDs; instead, it produces a collection of multisets, where each multiset contains just one candidate with a multiplicity of one. In this example, the output of the map stage is a collection of two multisets. The subscribers are the numbers 3 and 4.

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

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

Another important feature of map/reduce 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 *multiset union*, which combines two multisets by collecting the elements and adding their multiplicities. The result of applying the multiset union operation to the two multisets in the preceding collection is:

`{ 3(2), 4(1) }`

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

Here is the code for the sequential version:

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

```
// Reduce
IDMultiset candidates = new IDMultiset();
foreach (IDMultiset foafs in foafsList)
{
    candidates = Multiset.Union(foafs, candidates);
}

// Postprocess
return Multiset.MostNumerous(candidates, maxCandidates);
}
```

In the map phase, this code loops sequentially over the subscriber's friends, and builds a collection of multisets of candidates (a *foaf* is a friend of a friend). In the reduce phase, the code loops sequentially over those multisets, and aggregates them with the multiset 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) }.

Multiset union is commutative: the result does not depend on the order of its arguments. Multiset union is also associative: if you aggregate several multisets into one by successively forming unions pairwise, the final result does not depend on the order of the pairwise operations. If the aggregation function is not associative, it can't be done in parallel without potentially getting different results. If it's not commutative, the potential for parallelism is greatly reduced.

Floating-point arithmetic is neither commutative nor associative. From run to run, parallel computations over floats or doubles may end up with slightly different results.

Here's how to use PLINQ to apply the map/reduce pattern to the social networking example.

```
public IDMultisetItemList 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 IDMultisetItem(foafGroup.Key,
                                              foafGroup.Count()));

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

Recall that in map/reduce, 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 multiset 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 map/reduce is implemented with PLINQ, it need not be a line-by-line translation of the **foreach** version. In this PLINQ example, the output of the map stage is not a collection of multisets, but a collection with duplicates. The multiset 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 Loops 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;

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 Normalize(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. For an illustration of these steps, see Figure 2 in the "Design Notes" section of this chapter.

The overloaded version of **Parallel.ForEach** that is used in this example takes four arguments. The first argument is the data values that will 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 TPL implementation of **Parallel.ForEach** can create more than one task. In this case there will be one partial result value for each task.

The number of worker tasks is determined by the parallel loop implementation, based on heuristics. Generally, there will be at least one worker task per available core, but there could be more. As the loop progresses, TPL will sometimes retire tasks and create new ones.

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 locks the shared result variable and combines 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 value type such as **int** or **double**. When you use **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 in which they acquire the lock is

unpredictable; it is not necessarily in FIFO order. After the thread successfully acquires 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. For more information about locking, see the "Design Notes" section in this chapter.

The **localFinally** delegate (argument four) is executed once at the end of each task. The number of locks taken will therefore be equal to the number of tasks that are used to execute the parallel loop. You can't predict how many tasks will be used. For example, you shouldn't assume that the number of tasks will be less than or equal to the number that you provided with 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 shouldn't consider the state of the accumulator to be strictly "thread local." It's actually "task local." (This is a minor distinction that shouldn't normally affect a program's logic because a task is guaranteed to execute from start to finish on only one thread.)

For comparison, here's 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's possible that the overhead of **Parallel.ForEach** is large compared to the cost of executing the loop body.

In this case it's sometimes more efficient to use a **Partitioner** object for the loop. A **Partitioner** object allows you to embed a sequential **for** loop inside of your **Parallel.ForEach** loop and reduce the number of iterations of the **Parallel.ForEach** loop. Generally, you should profile the application in order to decide whether to use a **Partitioner** object.

Here's an example.

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

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 section, "Using Parallel Loops for Aggregation." The key difference here is that the **for each** loop uses a sequence of index intervals, which are generated by the **Partitioner** object, as its input rather than individual values. This can avoid some of the overhead involved in invoking delegate methods. You'll only notice a benefit when the amount of work in each step is small and there are many steps.

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's 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. They are 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();

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's 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 accumulate the local partial results of the simulation in this task. The second field is an instance of the **Random** class. It's 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. Note that this implementation doesn't require locks.

Design Notes

If you compare the sequential and parallel versions of the aggregation pattern, you see that the design of the parallel version includes an additional step in the algorithm that merges partial results. Figure 1, which illustrates the **Parallel.ForEach** and **Parallel.For** methods, shows this.

Aggregation using Parallel ForEach

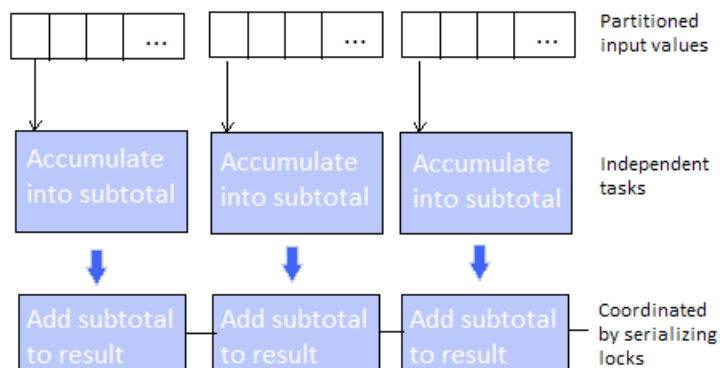


Figure 1

Figure 1 shows that instead of placing the accumulation in a single, shared result, the parallel loop uses unshared local storage for partial results (these are named *subtotals* in Figure 2). Each task processes a single partition of the input values. The number of partitions depends on the degree of parallelism that's 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. Locks are 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 require the developer to use locks. Instead, the final merge step is expressed as a binary operator that combines any two partial 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 requires less synchronization, so it is more scalable. The binary combinations do not require the system to lock the final result; they can be performed in parallel. (This is possible because the binary combination operation is associative, so the order of these operations doesn't affect the result.)

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.For** 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 produces the expected sum. However, it fails completely as an optimization. This code is many times slower than the sequential version it attempted to optimize!

In contrast, 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 group of patterns related to summarizing data in a collection. They include: Reduce, 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 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 patterns 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 multiset 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 map/reduce (see the "Further Reading" section), 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 as: `map (k1,v1) -> list(k2,v2)` and `reduce (k2,list(v2)) -> list(v2)`. In the social network example, what are the types of k1, v1, k2, and v2? What are the map and reduce functions? Provide answers for both the PLINQ and **Parallel.ForEach** versions.
-

Further Reading

MSDN describes the standard query operators of LINQ and PLINQ.

A thorough treatment of synchronization techniques appears in the book by Duffy.

The related patterns of Stencil, Scan, and Pack are discussed by McCool.

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

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

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.

M. McCool, Structured Patterns for Parallel Computation, December 2009, <http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=223101515>

S. Toub. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#, 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>

Standard Query Operators Overview, <http://msdn.microsoft.com/en-us/library/bb397896.aspx>

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 the Futures pattern.

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 computations. Futures is 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.)

Futures are asynchronous functions.

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 hasn't started yet, the task will be executed in the current thread context, if possible.

A future in .NET is a task that returns a value. Futures are implemented by the **Task<TResult>** class.

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 by the continuation task. An antecedent may have more than one continuation task.

A continuation task automatically starts when specified antecedent tasks finish.

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.

The Futures pattern discussed 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.

Be careful not to confuse futures with pipelines. As you will see in Chapter 7, "Pipelines," pipeline tasks are also nodes of a directed graph, but the arcs that connect stages of the pipeline 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.

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

The following example is 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*

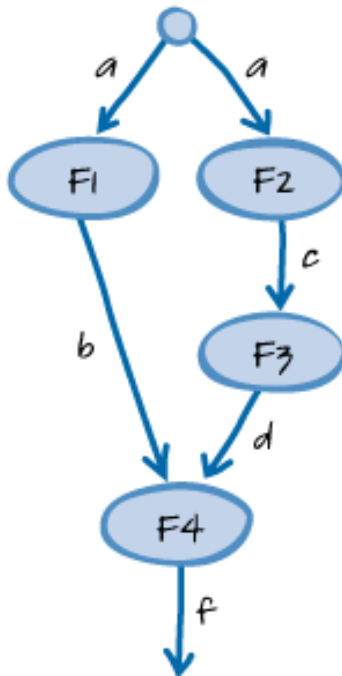


Figure 1

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.

```
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 are 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, as implemented by the **Task<TResult>** class, 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 **Result** property getter executes within a **try** block, which means that the exception can be handled in the corresponding **catch** block.

Continuation Tasks

It's 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 **futureB**. The second task calculates the value of **futureD**. These two tasks can run in parallel. The third task calculates the value of **futureF**. 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 historical market data from the Federal Reserve. 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

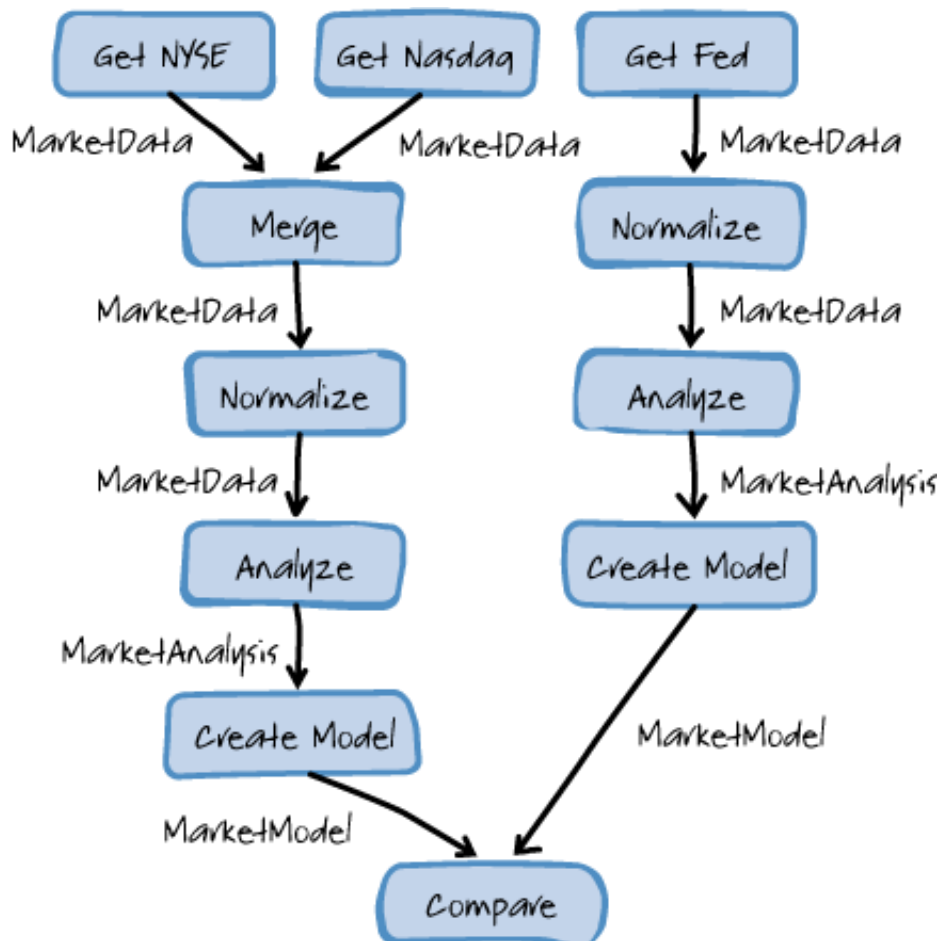


Figure 2

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> in the Chapter5\A-Dash project. 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.

The **StockData** 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 **StockData** values as long as the stock symbols don't overlap. The result of the merge operation is a new **StockData** value that contains the time series of the inputs.

The **StockAnalysis** 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, "Adapting Object-Oriented Patterns."

The Analysis Engine

The Adatum dashboard's **AnalysisEngine** class produces a market recommendation from the market data it receives.

The sequential process is shown in the following code.

```
public MarketRecommendation DoAnalysisSequential()
{
    StockDataCollection nyseData =
        LoadNyseData();
    StockDataCollection nasdaqData =
        LoadNasdaqData();
```

```

StockDataCollection mergedMarketData =
    MergeMarketData(new[]{nyseData, nasdaqData});
StockDataCollection normalizedMarketData =
    NormalizeData(mergedMarketData);
StockDataCollection fedHistoricalData =
    LoadFedHistoricalData();
StockDataCollection normalizedHistoricalData =
    NormalizeData(fedHistoricalData);
StockAnalysisCollection analyzedStockData =
    AnalyzeData(normalizedMarketData);
MarketModel modeledMarketData =
    RunModel(analyzedStockData);
StockAnalysisCollection analyzedHistoricalData =
    AnalyzeData(normalizedHistoricalData);
MarketModel modeledHistoricalData =
    RunModel(analyzedHistoricalData);
MarketRecommendation recommendation =
    CompareModels(new[] { modeledMarketData,
                           modeledHistoricalData });
return recommendation;
}

```

You can see how the binding of local variables corresponds to the data dependencies among the analysis phases shown in Figure 2. The final 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 **DoAnalysisSequential** method only returns when all of the dependent operations are complete.

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

```

public AnalysisTasks DoAnalysisParallel()
{
    TaskFactory factory = Task.Factory;
    // ...

    Task<StockDataCollection> loadNyseData =
        Task<StockDataCollection>.Factory.StartNew(
            () => LoadNyseData(),
            TaskCreationOptions.LongRunning);

    Task<StockDataCollection> loadNasdaqData =
        Task<StockDataCollection>.Factory.StartNew(
            () => LoadNasdaqData(),
            TaskCreationOptions.LongRunning);

    Task<StockDataCollection> mergeMarketData =
        factory.ContinueWhenAll<StockDataCollection,
StockDataCollection>(

```



```

        new[] { loadNyseData, loadNasdaqData },
        (tasks) => MergeMarketData(from t in tasks select
t.Result));

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

Task<StockDataCollection> loadFedHistoricalData =
    Task<StockDataCollection>.Factory.StartNew(
        () => LoadFedHistoricalData(),
        TaskCreationOptions.LongRunning);

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

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

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

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

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

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

Task errorHandler = CreateErrorHandler(loadNyseData,
    loadNasdaqData, loadFedHistoricalData,
    mergeMarketData, normalizeHistoricalData,
    normalizeMarketData, analyzeHistoricalData,
    analyzeMarketData, modelHistoricalData,
    modelMarketData, compareModels);

return new AnalysisTasks()
{
    LoadNyseData = loadNyseData,
    LoadNasdaqData = loadNasdaqData,

```

```

MergeMarketData = mergeMarketData,
NormalizeMarketData = normalizeMarketData,
LoadFedHistoricalData = loadFedHistoricalData,
NormalizeHistoricalData = normalizeHistoricalData,
AnalyzeMarketData = analyzeMarketData,
AnalyzeHistoricalData = analyzeHistoricalData,
ModelMarketData = modelMarketData,
ModelHistoricalData = modelHistoricalData,
CompareModels = compareModels,
ErrorHandler = errorHandler
};
}

```

The parallel version, provided by the **DoAnalysisParallel** method, is similar to the sequential version, except that the synchronous method calls have been replaced with futures and continuation tasks. The method returns an **AnalysisTasks** object that contains the tasks associated with each step of the calculation. The **DoAnalysisParallel** method returns immediately, leaving the tasks running. 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<StockDataCollection> loadNyseData =
    Task<StockDataCollection>.Factory.StartNew(
        () => LoadNyseData(),
        TaskCreationOptions.LongRunning);

Task<StockDataCollection> loadNasdaqData =
    Task<StockDataCollection>.Factory.StartNew(
        () => LoadNasdaqData(),
        TaskCreationOptions.LongRunning);

```

Note that the factory creates futures that return values of type **StockDataCollection**. The **TaskCreationOptions.LongRunning** enumerated value tells the task library that you want more concurrency. To prevent underutilization of processor 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's a continuation task that depends on two antecedent tasks, as shown in the following code.

```
Task<StockDataCollection> mergeMarketData =  
    factory.ContinueWhenAll<StockDataCollection,  
StockDataCollection>(  
        new[] { loadNyseData, loadNasdaqData },  
        (tasks) => 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 **MergeMarketData** method 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 **StockDataCollection** objects as its input. The LINQ expression **from t in tasks select t.Result** maps the input array of futures into a collection of **StockDataCollection** objects by getting the **Result** property of each future.

Normalizing

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

```
Task<StockDataCollection> 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 **StockAnalysisCollection** as input and returns an object of type **MarketAnalysis**, as shown in the following code.

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

```
(t) => RunModel(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 =  
    factory.ContinueWhenAll<MarketModel, MarketRecommendation>(  
        new[] { modelMarketData, modelHistoricalData },  
        (tasks) => CompareModels(  
            from t in tasks select t.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 also 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, "Adapting Object-Oriented 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 a Task," in Chapter 3, "Parallel Tasks."

Continue When "At Least One" Antecedent Completes

It's 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's 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

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 **IAsyncResult** into a task. To do this, use the **Task.Factory** object's **FromAsync** method.

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

Futures in .NET implement the **IAsyncResult** interface.

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 can 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 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

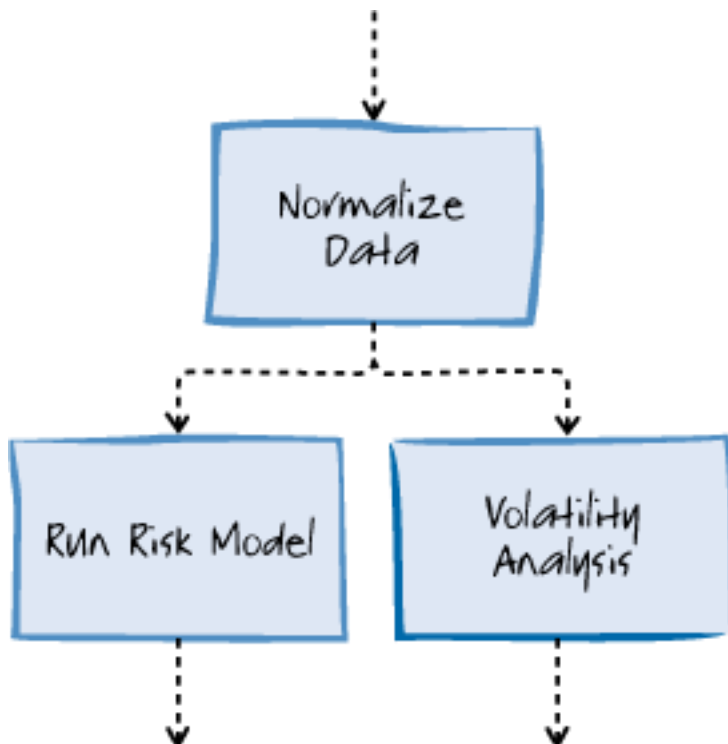


Figure 5

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

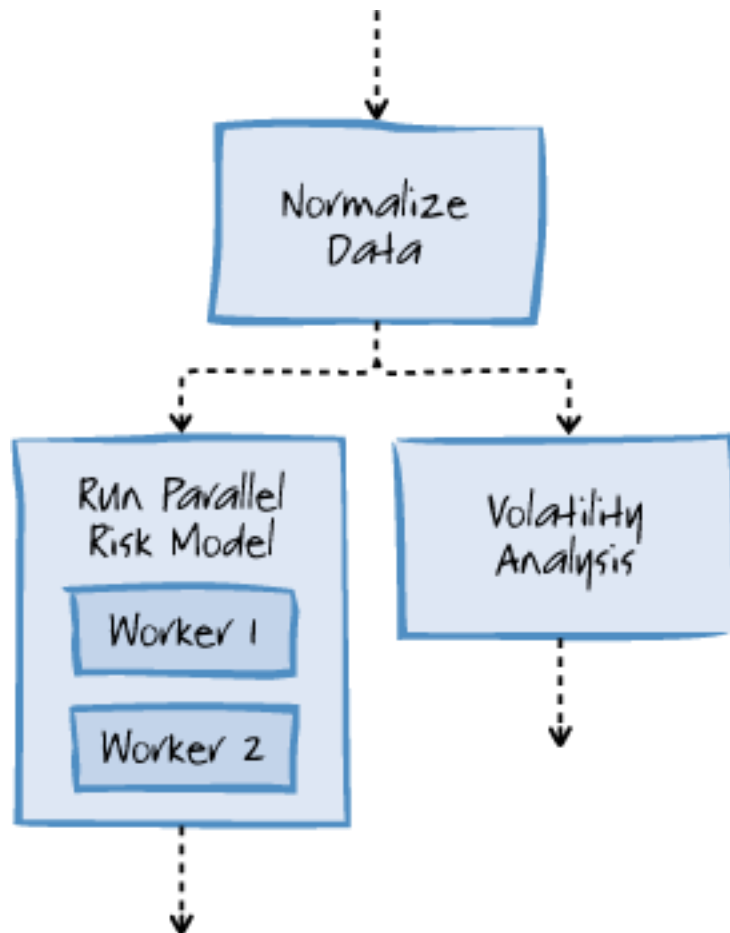


Figure 6

Modifying the Graph at Runtime

The code in the financial program's analysis engine creates a static task graph. In other words, the graph of task dependencies is reflected directly in the code. By reading the implementation of the analysis engine you can determine that there are a fixed number of tasks with a fixed set of dependencies among them.

The extension of the analysis tasks in the UI layer is an example of dynamic task creation. The UI augments the graph of tasks by adding continuation tasks programmatically, outside of the context where these tasks were originally created.

Dynamically created tasks are also a way to structure algorithms used for sorting, searching and graph traversal. See chapter 6, "Dynamic Task Parallelism" for examples.

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 3, "Parallel Tasks," it's 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 than implied data flow.

You can see this by an analogy to functions and subroutines. There's 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 modifying 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 (functions without side effects) 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's 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 that 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 can 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 greatest degree of 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 describes the motivation for including futures in TPL and has references to other work, especially in functional languages.

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

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

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

6 Dynamic Task Parallelism

This chapter covers situations where tasks are dynamically added to the work queue as the computation proceeds. This is called dynamic task parallelism. A simple example of dynamic task parallelism occurs in cases where the sequential version of an algorithm includes recursion.

Dynamic task parallelism is also known as recursive decomposition or "divide and conquer."

Dynamic task parallelism applies to problems that are solved by first solving smaller problems. For example, when you count the number of nodes in a data structure that represents a binary tree, you can count the nodes in the left and right subtrees and then add the results.

[Dynamic task parallelism is similar to recursion. Tasks create sub tasks on the fly to solve subproblems as needed.](#)

Applications that use data structures such as trees and graphs are typical examples of dynamic task parallelism. It's also used for applications that have geographic or geometric aspects, where the problem can be partitioned spatially.

The Basics

The following code shows a binary tree.

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

If you want to perform an action on each data value in the tree, you need to visit each node. This is known as walking the tree, which is a naturally recursive operation. Here's an example that uses sequential code.

```
static void SequentialWalk<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    action(tree.Data);
    SequentialWalk(tree.Left, action);
    SequentialWalk(tree.Right, action);
}
```

You can also use parallel tasks to walk the tree.

```
static void ParallelWalk<T>(Tree<T> tree, Action<T> action)
{
```

```

if (tree == null) return;
var t1 = Task.Factory.StartNew(
    () => action(tree.Data));
var t2 = Task.Factory.StartNew(
    () => ParallelWalk(tree.Left, action));
var t3 = Task.Factory.StartNew(
    () => ParallelWalk(tree.Right, action));

Task.WaitAll(t1, t2, t3);
}

```

You don't need to create a new task to walk the right side of the tree. If you want, you can use the current task; however, using tasks for both the left and right walks makes sure that exceptions that are thrown by either will be observed. For more information about how to observe unhandled task exceptions, see "Handling Exceptions" in chapter 3.

When you use dynamic task parallelism to perform a tree walk, you no longer visit nodes in a fully predictable order. If you need to visit nodes in a sequential order, such as with a preorder, inorder, or postorder traversal, then you may want to consider the pipeline pattern that is described in chapter 7.

Dynamic task parallelism results in a less predictable order of execution than sequential code.

In this example, the number of tasks is three times the number of nodes in the tree, which could be a large number. The Task Parallel Library is designed to handle this kind of situation, but you may want to read the Design Notes sections of this chapter for some performance tips.

An Example

An example of dynamic task parallelism is when you sort a list with an algorithm such as QuickSort. This algorithm first divides an unsorted array of values, then orders and recombines the pieces. Here's a sequential implementation.

Sorting is a typical application that can benefit from dynamic task parallelism.

```

static void SequentialQuickSort(int[] array, int from, int to)
{
    if (to - from <= Threshold)
    {
        InsertionSort(array, from, to);
    }
    else
    {
        int pivot = from + (to - from) / 2;
        pivot = Partition(array, from, to, pivot);
        SequentialQuickSort(array, from, pivot);
        SequentialQuickSort(array, pivot + 1, to);
    }
}

```

This method sorts **array** in place, rather than returning a sorted array. The **from** and **to** arguments identify the segment that is to be sorted. The code includes an optimization. It's not efficient to use the recursive algorithm on short segments, so the method calls the non-recursive **InsertionSort** method on segments that are not longer than **Threshold**, which is set in a global variable. This optimization applies equally to the sequential and parallel versions.

If the segment is longer than **Threshold**, the recursive algorithm is used. The element in the middle of the segment (at **pivot**) is chosen. The **Partition** method moves all the array elements not greater than the element at **pivot** to the segment that precedes **pivot**, and leaves the greater elements in the segment that follows **pivot** (**pivot** itself may be moved). Then the method recursively calls **QuickSort** on both segments.

The following code shows a parallel implementation.

```
static void ParallelQuickSort(int[] array, int from,
                             int to, int depthRemaining)
{
    if (to - from <= Threshold)
    {
        InsertionSort(array, from, to);
    }
    else
    {
        int pivot = from + (to - from) / 2;
        pivot = Partition(array, from, to, pivot);
        if (depthRemaining > 0)
        {
            Parallel.Invoke(
                () => ParallelQuickSort(array, from, pivot,
                                         depthRemaining - 1),
                () => ParallelQuickSort(array, pivot + 1, to,
                                         depthRemaining - 1));
        }
    }
    else
    {
        {
            ParallelQuickSort(array, from, pivot, 0);
            ParallelQuickSort(array, pivot + 1, to, 0);
        }
    }
}
```

This version uses **Parallel.Invoke** to execute the recursive calls in tasks that can run in parallel. Tasks are created dynamically with each recursive call; if the array is large, many tasks might be created.

The parallel version includes another optimization. It's generally not useful to create many more tasks than there are processors to run them. So, the **ParallelQuickSort** method includes an additional argument to limit task creation. The **depthRemaining** argument is decremented on each recursive call,

and tasks are only created when this argument exceeds zero. The following code shows how to calculate an appropriate depth (the **depthRemaining** argument) from the number of processors.

```
public static void ParallelQuickSort(int[] array)
{
    ParallelQuickSort(array, 0, array.Length,
        (int) Math.Log(Environment.ProcessorCount, 2) + 4);
}
```

One relevant factor in selecting the number of tasks is how similar the predicted run times of the tasks will be. In the case of QuickSort, the duration of the tasks may vary a great deal because the pivot points depend on the unsorted data. They do not necessarily result in segments of equal size (in fact, the sizes could vary widely). To compensate for the uneven sizes of the tasks, the formula in this example that calculates the **depthRemaining** argument produces more tasks than cores. The formula limits the number of tasks to approximately sixteen times the number of cores. This is because the number of tasks can be no larger than $2^{\text{depthRemaining}}$. If you substitute $\text{depthRemaining} = \log_2(\text{NCores}) + 4$ and simplify the expression, you see that the number of tasks is $16 \times \text{NCores}$.

Limiting the number of subtasks by measuring the recursion depth is an extremely important technique.

Variations

Dynamic task parallelism has several variations.

Parallel While-Not-Empty

The examples shown so far in this chapter use techniques that are the parallel analogs of sequential depth-first traversal. There are parallel algorithms for other types of traversals, as well. These techniques rely on concurrent collections to keep track of the remaining work to be done. Here's an example.

```
public static void ParallelWhileNotEmpty<T>(
    IEnumerable<T> initialValues,
    Action<T, Action<T>> body)
{
    var from = new ConcurrentQueue<T>(initialValues);
    while (!from.IsEmpty)
    {
        var to = new ConcurrentQueue<T>();
        Action<T> addMethod = to.Enqueue;
        Parallel.ForEach(from, v => body(v, addMethod));
        from = to;
    }
}
```

This method shows how you can use **Parallel.ForEach** to process an initial collection of values. While processing the values, additional values to process may be discovered. The additional values are placed in the **to** queue. When the first batch of values has been processed, the method starts processing the additional values, which may again result in more values to process. This process repeats until no additional values can be produced.

A method that walks a binary tree can use the **ParallelWhileNotEmpty** method.

```
static void ParallelWalk4<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    ParallelWhileNotEmpty(new[] { tree }, (item, adder) =>
    {
        if (item.Left != null) adder(item.Left);
        if (item.Right != null) adder(item.Right);
        action(item.Data);
    }));
}
```

An example of a problem that would be an appropriate use of while-not-empty is a web site link checking tool. The walk task loads the initial page and searches it for links. Each link is checked and removed from the list, and additional links to unchecked pages from the same site are added to the list. Eventually there are no more unchecked links and the application stops.

Task Chaining with Parent/Child Tasks

The Task Parallel Library includes a task creation option named **AttachedToParent**. This option is used most frequently in code that uses the dynamic task parallelism pattern. The purpose of the **AttachedToParent** option is to link a subtask to the task that created it. In this case, the subtask is called a child task and the task that created the child task is called a parent task.

You use the **AttachedToParent** option in two situations. The first is when you want to link the lifetimes of a parent task to the status of its child tasks. The second is when you want to use the Visual Studio debugger to see the parent/child relationships. The order of execution is not changed by using the **AttachedToParent** option.

The following code shows an example.

```
static void ParallelWalk2<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    var t1 = Task.Factory.StartNew(
        () => action(tree.Data),
        TaskCreationOptions.AttachedToParent);
    var t2 = Task.Factory.StartNew(
        () => ParallelWalk2(tree.Left, action),
        TaskCreationOptions.AttachedToParent);
    var t3 = Task.Factory.StartNew(
        () => ParallelWalk2(tree.Right, action),
```

```

        TaskCreationOptions.AttachedToParent);
    Task.WaitAll(t1, t2, t3);
}

```

The **AttachedToParent** option affects the behavior of the parent task. If a parent task with at least one running child task finishes running for any reason, its **Status** property becomes **WaitingForChildrenToComplete**. Only when all of its attached children are no longer running will the parent task's **Status** property transition to one of the three final states. Figure 1 illustrates this.

Life Cycle of Task with Attached Children

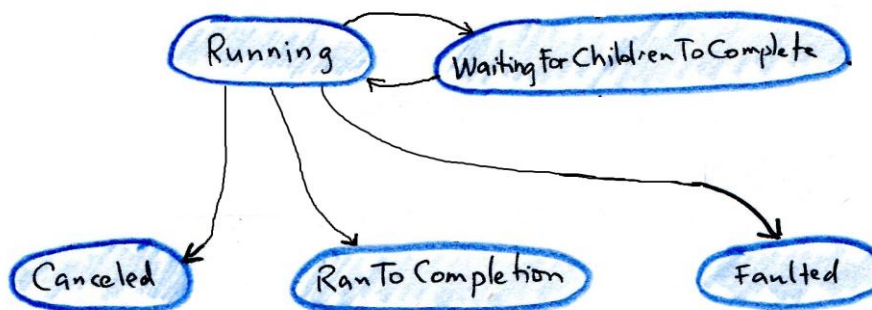


Figure 1

Exceptions from attached children are observed in the parent task.

There is special support for visualizing parent/child task relationships in the Visual Studio debugger if you use the **AttachedToParent** option to create a subtask. To access the parent/child view in the Visual Studio debugger, right-click on the row headings in the **Parallel Tasks** window and select **Parent Child View**.

Design Notes

This section discusses how the default task scheduler handles dynamic task parallelism.

Work Stealing and Dynamic Task Parallelism

The behavior of .NET's default task scheduler was described in Chapter 3, "Parallel Tasks." The default task scheduler uses fast local task queues for each of its worker threads as well as a work stealing algorithm for load balancing. These features are of particular importance to the performance of applications that use dynamic task parallelism.

The default task scheduler's thread injection heuristics are not appropriate for every case of dynamic task parallelism. In particular, if you have long-running, processor-intensive tasks, profiling your application may show processor oversubscription (too many worker threads). If this happens, consider decomposing your problem into more tasks with shorter durations. For example, in the parallel quick

sort example, you can adjust the initial value of the **depthRemaining** parameter to create more or fewer tasks.

When you use the default task scheduler, watch out for long-running, processor-intensive tasks.

You can also write a custom thread scheduler. See Chapter 3 for more information about task scheduling.

Exercises

1. The sample code on Codeplex assigns a particular default value for the **Threshold** segment length. At this point, the QuickSort methods switch to the non-recursive **InsertionSort** algorithm. Use the command line argument to assign different values for the **Threshold** value, and observe the execution times for the sequential version to sort different array sizes. What do you expect to see? What is the best value for **Threshold** on your system?
2. Use the command line argument to vary the array size, and observe the execution time as a function of array size for the sequential and parallel versions. What do you expect? Can you explain your observations?
3. Suggest other measures to limit the number of tasks, besides the number of cores.

Further Reading

Toub discusses additional variations on parallel QuickSort.

S. Toub. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#, 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>

7 Pipelines

The Pipeline pattern uses parallel tasks and concurrent queues to process a sequence of input values. Each task implements a stage of the pipeline, and the queues act as buffers that allow the stages of the pipeline to execute concurrently, even though the values are processed in order. You can think of software pipelines as analogous to assembly lines in a factory, where each item in the assembly line is constructed in stages. The partially assembled item is passed from one assembly stage to another. The outputs of the assembly line occur in the same order as that of the inputs.

A pipeline is a series of parallel tasks that are connected by buffers.

A pipeline is composed of a series of producers/consumers, each one depending on the output of its predecessor. Pipelines allow you to use parallelism in cases where there are too many dependencies to use a parallel loop.

There are many ways to use pipelines. Pipelines are often useful when the data elements are received from a real-time event stream, such as values on stock ticker tapes, user-generated mouse click events, and packets that arrive over the network. Pipelines are also used to process elements from a data stream, as is done with compression and encryption or processing streams of video frames. In all of these cases, it's important that the data elements are processed in sequential order.

Don't confuse pipelines and parallel loops. Pipelines are used in cases where a parallel loop can't be used. With the Pipeline pattern, the data is process in sequential order. The first input is transformed into the first output, the second input into the second output, and so on.

Pipelines are also useful for processor-intensive applications and for applications where the cost of I/O is significant.

The Basics

In .NET, the buffers that connect stages of a software pipeline are usually based on the **BlockingCollection<T>** class.

Here is an example of a pipeline that has four stages. It reads words and sentence fragments from a data source, it corrects the punctuation and capitalization, it groups them into complete sentences, and it writes the sentences to a disk file. Figure 1 illustrates this.

Sample Pipeline

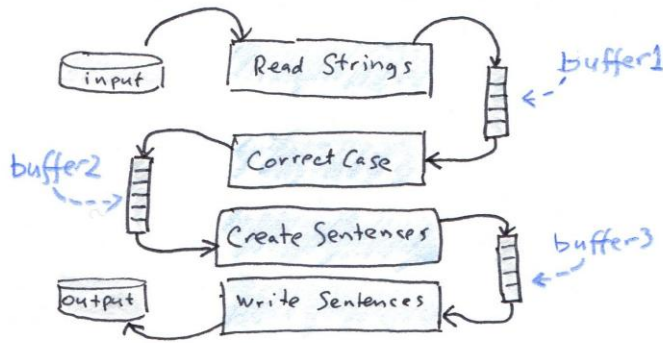


Figure 1

Each stage of the pipeline reads from a dedicated input and writes to a particular output. For example, the "Read Strings" task reads from a source and writes to buffer 1. All the stages of the pipeline can execute at the same time because concurrent queues buffer any shared inputs and outputs. If there are four available cores, the stages can run in parallel. As long as there is room in its output buffer, a stage of the pipeline can add the value it produces to its output queue. If the output buffer is full, then the producer of the new value waits until space becomes available. Stages can also wait (that is, block) on inputs. An input wait is familiar from other programming contexts—if an enumeration or a stream does not have a value, the consumer of that enumeration or stream waits until a value is available or an "end of file" condition occurs. Blocking a collection works the same way. Using buffers that hold more than one value at a time compensates for variability in the time it takes to process each value.

If the queue is full, the producer blocks. If the queue is empty, the consumer blocks.

The **BlockingCollection<T>** class lets you signal the "end of file" condition with the **CompleteAdding** method. This method tells the consumer that it can end its processing loop when all of the data previously added to the collection has been removed or processed.

The following code demonstrates how to implement a pipeline that uses the **BlockingCollection** class for the buffers and tasks for the stages of the pipeline.

```

int seed = ...
int BufferSize = ...
var buffer1 = new BlockingCollection<string>(BufferSize);
var buffer2 = new BlockingCollection<string>(BufferSize);
var buffer3 = new BlockingCollection<string>(BufferSize);

var f = new TaskFactory(TaskCreationOptions.LongRunning,
                        TaskContinuationOptions.None);

var stage1 = f.StartNew(() => ReadStrings(buffer1, ...));
var stage2 = f.StartNew(() => CorrectCase(buffer1, buffer2));
var stage3 = f.StartNew(() => CreateSentences(buffer2, buffer3));
var stage4 = f.StartNew(() => WriteSentences(buffer3));
  
```

```
Task.WaitAll(stage1, stage2, stage3, stage4);
```

The first stage produces the input strings and places them in the first buffer. The second stage transforms the strings. The third stage combines the strings and produces sentences. The final stage writes the corrected sentences to a file.

The buffers are instances of the **BlockingCollection<string>** class. The argument to the constructor specifies the maximum number of values that can be buffered at any one time. In this case, the value is 32 for each buffer. See the "Design Notes" section in this chapter for more about how to choose the buffer size of a blocking collection.

As this example shows, tasks in a pipeline are usually created with the **LongRunning** option. See the "Anti-Patterns" section in this chapter for more information.

Don't worry if you forget the **LongRunning** option. If you use the default task scheduler, the .NET thread pool will automatically add additional worker threads as needed at the rate of about 2 per second.

The first stage of the pipeline includes a sequential loop that writes to an output buffer.

```
static void ReadStrings(BlockingCollection<string> output,
                        int seed)
{
    try
    {
        foreach (var phrase in PhraseSource(seed))
        {
            Stage1AdditionalWork();
            output.Add(phrase);
        }
    }
    finally
    {
        output.CompleteAdding();
    }
}
```

The sequential loop populates the output buffer with values. The values come from an external data source that is accessed by the **PhraseSource** method, which returns an ordinary single-threaded instance of **IEnumerable<string>**. The producer places a value in the blocking collection with the blocking collection's **Add** method. This method can potentially block if the queue is full. This is a way to limit stages of the pipeline that are executing faster than other stages.

Use a sequential loop to process steps in a pipeline stage. Call the **CompleteAdding** method once a pipeline stage will produce no more values.

The call to the **CompleteAdding** method is usually inside of a **finally** block so that it will execute even if an exception occurs.

Stages in the middle of the pipeline consume values from an input buffer and also produce values and place them into an output buffer. The following code shows how the stages are structured.

```
void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output)
{
    try
    {
        foreach (var item in input.GetConsumingEnumerable())
        {
            var result = ...
            output.Add(result);
        }
    }
    finally
    {
        output.CompleteAdding();
    }
}
```

You can look at the online source code to see the implementations of the **CorrectCase** and **CreateSentences** methods that make up stages 2 and 3 of this pipeline. They are structured in a very similar way to this example. The important point of this code is that the input blocking collection's **GetConsumingEnumerable** method returns an enumeration that a consumer can use to take the values.

Although this example doesn't show it, a blocking collection's **GetConsumingEnumerable** method can be called by more than one consumer. This allows values from the producer to be divided among multiple recipients. If a recipient gets a value from the blocking collection, no other consumer will also get that value.

[A blocking collection can have more than one consumer.](#)

The last stage of the pipeline consumes values from a blocking collection but doesn't produce values. Instead, it writes values to a stream. Here's the code.

```
static void WriteSentences(BlockingCollection<string> input)
{
    using (StreamWriter outfile =
           new StreamWriter(PathForPipelineResults))
    {
        // ...
        foreach (var sentence in input.GetConsumingEnumerable())
        {
            var printSentence = ...
            outfile.WriteLine(printSentence);
        }
    }
}
```

One of the things that makes a pipeline easy to write in .NET is that you can rely on familiar sequential techniques such as iteration by using the **IEnumerable<T>** class. There is some synchronization, but it's hidden inside the implementation of the **BlockingCollection<T>** class.

(Some details of error handling, cancellation and the collection of performance data have been omitted from this example for clarity. Review the full ImagePipeline sample that's mentioned later in this chapter to see error handling and cancellation code.)

An Example

The online samples include an application named ImagePipeline. This application takes a directory of JPG images and generates thumbnail versions, which are also post-processed with several image-enhancing filters. The resulting processed images are displayed as a slideshow. Each image in the slideshow is displayed in alphabetical file name order.

You can't use a parallel loop for this example because the application has a requirement that images must be processed in sequence. Parallel loops do not guarantee any particular processing order.

Sequential Image Processing

Each image is processed in four stages: the large color image is loaded from a file, a small thumbnail with a picture frame is generated from it, then noise is added to the image to create a speckling effect, and finally the processed image is displayed as the next picture in the slideshow. Figure 2 illustrates this sequence.

Sequential Image Processing

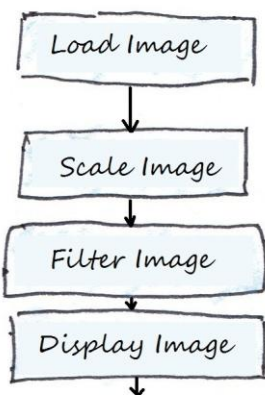


Figure 2

Here's the code for the sequential version.

```
string sourceDir = ...  
IEnumerable<string> fileNames = ...
```

```

int count = 0;
ImageInfo info = null;

foreach (var fileName in fileNames)
{
    /// ...

    info = LoadImage(fileName, sourceDir, count, ...);
    ScaleImage(info);
    FilterImage(info);
    // ...
    DisplayImage(info, count + 1, displayFn, ...);
    // ...

    count += 1;
    info = null;
}

```

The four steps are performed by the **LoadImage**, **ScaleImage**, **FilterImage** and **DisplayImage** methods. This example is slightly abridged for clarity. The code that deals with cancellation, error handling (the disposal of handles to unmanaged objects) and the capture of performance measurements is omitted. You can refer to the online samples to see these details. See the "Variations" section of this chapter for more information about cancellation and error handling.

The Image Pipeline

The sequential loop can only process one image at a time; each image must complete all four stages before work can begin on the next image, and the stages themselves are sequentially linked. In fact, this example seems intractably sequential—the top-level loop has the restriction that images must be displayed in order and within each step are substeps that require inputs from previous substeps. You can't display an image until the filter has been applied to it. You can't apply the filter until after the image has been scaled to thumbnail size. You can't do the scaling until the original image has been loaded.

However, the pipeline pattern can introduce parallelism into this example. Each image still passes through all four stages in sequence, but the stages can execute in parallel on different images. Figure 3 illustrates the image pipeline.

Image Pipeline

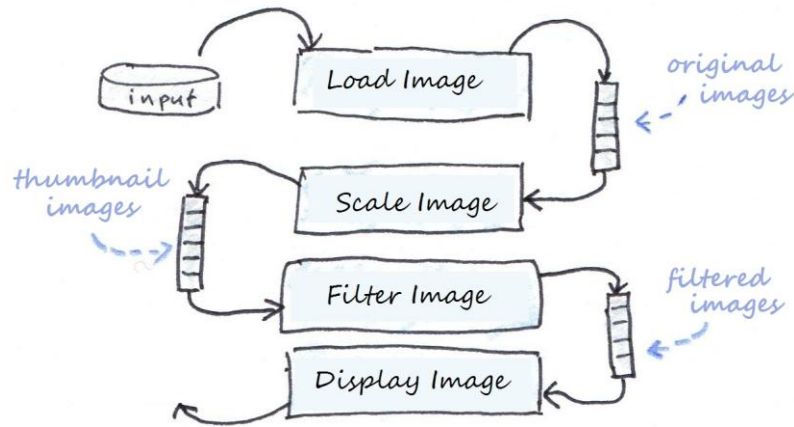


Figure 3

The following code shows the the parallel version.

```

IEnumerable<string> fileNames = ...
string sourceDir = ...
Action<ImageInfo> displayFn = ...
int limit = ...

var originalImages = new BlockingCollection<ImageInfo>(limit);
var thumbnailImages = new BlockingCollection<ImageInfo>(limit);
var filteredImages = new BlockingCollection<ImageInfo>(limit);
try
{
    var f = new TaskFactory(TaskCreationOptions.LongRunning,
                           TaskContinuationOptions.None);
    // ...

    var loadTask = f.StartNew(() =>
        LoadPipelinedImages(fileNames, sourceDir,
                             originalImages, ...));
    var scaleTask = f.StartNew(() =>
        ScalePipelinedImages(originalImages,
                             thumbnailImages, ...));
    var filterTask = f.StartNew(() =>
        FilterPipelinedImages(thumbnailImages,
                              filteredImages, ...));
    var displayTask = f.StartNew(() =>
        DisplayPipelinedImages(
            filteredImages.GetConsumingEnumerable(),
            displayFn, ...));

    Task.WaitAll(loadTask, scaleTask, filterTask, displayTask);
}
finally

```



```
{
  // ... release handles to unmanaged resources ...
}
```

(Some details of error handling, cancellation and the collection of performance data have been omitted here for clarity. Refer to the online sample for the complete implementation.)

There are three blocking collections that act as buffers between the stages of the pipeline. The four stages are the same as in the sequential version. A task factory **StartNew** call executes each processing stage in its own long-running task.

The code calls **Task.WaitAll** to defer cleanup until all stages have completed processing all images.

Performance Characteristics

To understand the performance characteristics of the sequential and pipelined versions, it's useful to look at a scheduling diagram. Figure 4 illustrates this.

Image Pipeline with Stages of Equal Speed

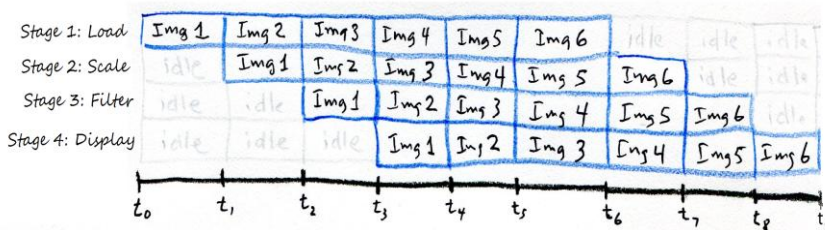


Figure 4

Figure 4 shows how the tasks in the image pipeline example execute over time. For example, the top row shows that stage 1 processes image 1 starting at time t_0 and image 2 starting at time t_1 . Stage 2 begins processing image 1 at time t_1 . Assume for a moment that the pipeline is perfectly balanced; that is, each stage of the pipeline takes exactly the same amount of time to do its work. Call that duration T . Therefore, in the figure, t_1 occurs after T units of time have elapsed, t_2 after $2 \times T$ units of time and so on.

If there are enough available cores to allow the pipeline's tasks to run in parallel, then the figure shows that the expected execution time for six images in a pipeline with four stages is approximately $9 \times T$. In contrast, the sequential version takes approximately $24 \times T$ because each of the 24 steps must be processed one after another.

The average performance improves as more images are processed. The reason, as Figure 4 illustrates, is that some cores are idle as the pipeline fills upon startup and drains upon shutdown. With a large number of images, the startup and shutdown time becomes relatively insignificant. The average time per image would approach T .

If there are enough available cores and if all stages of a pipeline take an equal amount of time, then the execution time for the pipeline as a whole is the same as the time for just one stage.

However, there's one catch: the assumption that all of the pipeline steps take exactly the same amount of time isn't always true. Figure 5 shows the scheduling diagram that occurs when the filter stage takes twice as long as the other stages.

Image Pipeline with Unequal Stages

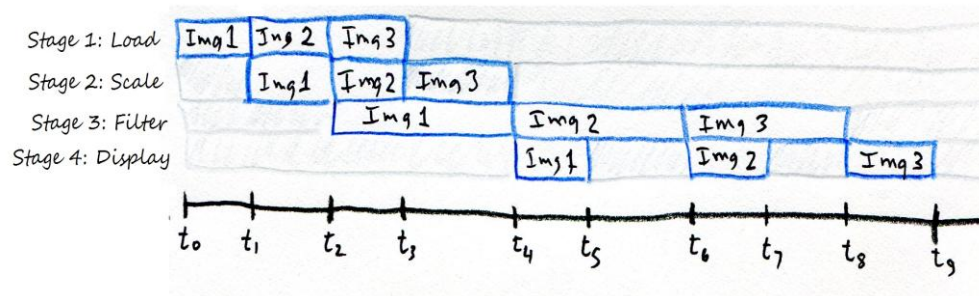


Figure 5

When one of the stages takes $2 \times T$ units of time while the other stages take T units of time, you can see that it's not possible to keep all of the cores completely busy. On average (with a large number of images), the time to process an images is $2 \times T$. In other words, when there are enough cores for each pipeline stage, the speed of a pipeline is approximately equal to the speed of its slowest stage.

When the stages of a pipeline don't take the same amount of time, the speed of a pipeline is approximately equal to the speed of its slowest stage.

If you run the ImagePipeline application in the Visual Studio debugger, you can see this effect for yourself. The ImagePipeline sample has a UI feature that reports the average length of time in milliseconds for each of the stages of the pipeline. It also reports the overall average length of time that is needed to process each image. When you run the sample in sequential mode (by selecting the **Sequential** radio button) you'll notice that the steady-state elapsed time per image equals the sum of all of the stages. When you run in pipeline mode, the average elapsed time per image converges to approximately the same amount of time as slowest stage. The most efficient pipelines have stages of equal speed. You won't always achieve this, but it's a worthy goal.

Variations

There are several variations to the pipeline pattern.

Canceling a Pipeline

Pipeline tasks work together to perform their work; they must also work together when they respond to a cancellation.

In the standard cancellation scenario, which was explained in Chapter 3, your task is passed a **CancellationToken** value from a higher layer of the application, such as the user interface. In the case of a pipeline stage, you need to observe this cancellation token in two places. This is shown in the following code.

```

void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output,
             CancellationToken token)
{
    try
    {
        foreach (var item in input.GetConsumingEnumerable())
        {
            if (token.IsCancellationRequested) break;
            var result = ...
            output.Add(result, token);
        }
    }
    catch (OperationCanceledException) { }
    finally
    {
        output.CompleteAdding();
    }
}

```

A natural place to check for cancellation is at the beginning of the loop that processes items from the blocking collection. At this point, you only need to **break** from the loop, as is shown in the following code.

```
if (token.IsCancellationRequested) break;
```

The second place is less obvious. You use an overloaded version of the **BlockingCollection<T>** class's **Add** method that accepts a cancellation token as an argument. The following code shows how to do this.

```
output.Add(result, token);
```

If you give the **Add** method a cancellation token, then the blocking collection will be able to detect the cancellation request while it's waiting to add a new value.

To understand why this is necessary, recall that it's possible that a pipeline stage that's creating values can be blocked when it calls the **Add** method of its output queue. (If the output queue is full, the call to **Add** waits until space is available before returning.) If a cancellation request happened in this situation and you didn't check for it, it would be possible for your program to experience deadlock. Deadlock would be possible because stages of the pipeline that consume values produced by this stage can also be cancelled, and therefore it's very possible that later stages in the pipeline would terminate without draining the queue. The producer, which is blocked while waiting for space to become available, would have no way to proceed.

Cancelling a pipeline improperly can cause your program to experience deadlock. Follow the guidelines in this section carefully.

Check for cancellation at the beginning of a stage's main loop and also when adding values to a blocking collection.

The solution, as shown in the example, is to use the overloaded version of the blocking collection's **Add** method that accepts a cancellation token as an argument. If a cancellation is requested while the producer waits for space to become available, then the blocking collection will create and throw an **OperationCanceledException** instance.

Whether the loop exits normally, or from the **break** keyword, or from a cancellation exception, the **finally** clause guarantees that the output buffer will be marked as completed. This will unblock any consumers that might be waiting for input, and they can then process the cancellation request.

Although it's also possible to check for a cancellation token while waiting for input from a blocking collection (there is an overloaded version of the **GetConsumingEnumerable** method that accepts a cancellation token), you don't need to do this if you use the techniques described in this section.

Be aware that if the type **T** implements the **IDisposable** interface then, under .NET coding conventions, you also must call the **Dispose** method on cancellation. You need to dispose of the current iteration's object as well as instances of **T** stored in the blocking queues. The online source code of the ImagePipeline example shows how to do this.

Handling Pipeline Exceptions

Exceptions are similar to cancellations. The difference between the two is that when an unhandled exception occurs within one of the pipeline stages, the tasks that execute the other stages don't by default receive notification that an exception has occurred elsewhere. Without such notification, there are several ways for the application to deadlock.

When there is an unhandled exception in one pipeline stage, you should cancel the other stages. If you don't do this, deadlock can occur. Follow the guidelines in this section carefully.

Use a special instantiation of the **CancellationTokenSource** class to allow your application to coordinate the shutdown of all the pipeline stages when an exception occurs in one of them. Here is an example.

```
static void DoPipeline(CancellationToken token)
{
    using (CancellationTokenSource cts =
        CancellationTokenSource.CreateLinkedTokenSource(token))
    {
        var f = new TaskFactory(TaskCreationOptions.LongRunning,
                                TaskContinuationOptions.None);

        var stage1 = f.StartNew(() => DoStage1(..., cts));
        var stage2 = f.StartNew(() => DoStage2(..., cts));
        var stage3 = f.StartNew(() => DoStage3(..., cts));
        var stage4 = f.StartNew(() => DoStage4(..., cts));
    }
    Task.WaitAll(stage1, stage2, stage3, stage4);
}
```

The **CreateLinkedTokenSource** method of the **CancellationTokenSource** class creates a handle that allows you to respond to an external cancellation request and also to initiate (and recognize) an internal cancellation request of your own. You pass the linked cancellation token source as an argument to the methods that execute your pipeline stages, so each stage can use the token source to initiate cancellation.

Here is an example.

```
void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output,
             CancellationToken cts)
{
    try
    {
        var token = cts.Token;
        foreach (var item in input.GetConsumingEnumerable())
        {
            if (token.IsCancellationRequested) break;
            var result = ...
            output.Add(result, token);
        }
    }
    catch (Exception e)
    {
        // if an exception occurs, notify all other pipeline stages
        cts.Cancel();
        if (!(e is OperationCanceledException))
            throw;
    }
    finally
    {
        output.CompleteAdding();
    }
}
```

This code is similar to the cancellation variation described earlier, except that when an unhandled exception occurs, the exception is intercepted by the **catch** block, which also signals cancellation for all of the other pipeline stages. Consequently, each pipeline stage will begin an orderly shutdown.

When all pipeline tasks have stopped, the original exception, wrapped as an inner exception of an **AggregateException** instance will be thrown by the **Task.WaitAll** method. You should be sure to include a **catch** or **finally** block to do any cleanup, such as releasing handles to unmanaged resources.

You might ask why you can't just pass a **CancellationTokenSource** object to the **DoPipeline** method instead of a **CancellationToken** value. By convention, only the cancellation token is passed as an argument. Passing a **CancellationToken** value allows lower-level library code to respond to external cancellation but prevents lower-level libraries from initiating the cancellation of higher-level components. In other words, responding to a cancellation request requires less privilege than initiating such a request.

Load Balancing using Multiple Producers

The **BlockingCollection<T>** class allows you to read values from more than one producer. This feature is provided by the **TakeFromAny** static method and its variants. You can use **TakeFromAny** to implement load balancing strategies for some pipeline scenarios (but not all). This variation is sometimes called a nonlinear pipeline.

The image pipeline example described earlier in this chapter requires that the slideshow of thumbnail images be performed in the same order as the input files. This is a constraint that is common to many pipeline scenarios, such as processing a series of video frames. However, in the case of the image pipeline example, the filter operations on successive images are independent of each other. In this case, you can insert an additional pipeline task. This is shown in Figure 6.

It is sometimes possible to implement load balancing by increasing the number of tasks used for a particular pipeline stage.

Consuming Values from Multiple Producers

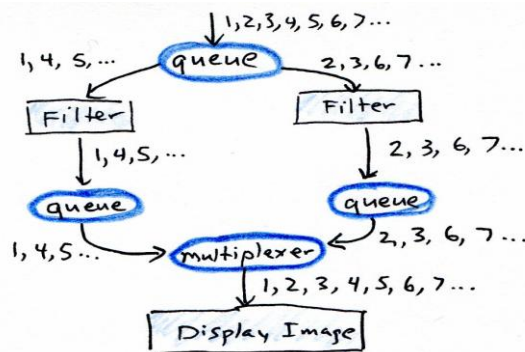


Figure 6

The figure shows what happens when you add an additional filter task. Both of the filter tasks take images produced by the previous stage of the pipeline. The order in which they consume these images is not fully determined, although from a filter's local point of view, no input image ever arrives out of order.

Each of the filter stages has its own blocking collection for enqueueing the elements that it produces. The consumer of these queues is a component called a multiplexer, which combines the inputs from all of the producers. The multiplexer allows its consumer, which in this case is the display stage of the pipeline, to receive the images in the correct sequential order. The images do not need to be sorted or reordered. Instead, the fact that each producer queue is *locally ordered* allows the multiplexer to look for the next value in the sequence by simultaneously monitoring the heads of all of the producer queues. This is where the blocking collection's **TakeFromAny** method comes into play. The method allows the multiplexer to block until any of the producer queues has a value to read.

To make this more concrete, let's look at an example. Suppose that each image has a unique sequence number that is available as a property. The image numbers start with 1 and increase sequentially. As the

figure shows, the first filter might process images that are numbered 1, 4 and 5 while the second filter processes images with sequence numbers 2, 3, 6 and 7. Each load-balanced filter stage collects its output images into two queues. The two output queues are correctly ordered (that is, no higher numbered image comes before a lower numbered image), but there are gaps in the sequence of numbers. For example, if you take values from the first filter's output queue you get image 1, followed by image 4, followed by image 5. Images 2 and 3 are missing because they are found in the second filter's output queue.

The gaps are a problem. The next stage of the pipeline, the Display Image stage, needs to show images in order and without gaps in the sequence. This is where the multiplexer comes in. Using the **TakeFromAny** method, the multiplexer waits for input from both of the filter stage producer queues. When an image arrives, the multiplexer looks to see if the image's sequence number is the next in the expected sequence. If it is, the multiplexer passes it to the Display Image stage. If the image is not the next in the sequence, the multiplexer holds the value in an internal look-ahead buffer and repeats the take operation for the input queue that does not have a look-ahead value. This algorithm allows the multiplexer to sew together the inputs from the incoming producer queues in a way that ensures sequential order without sorting the values.

Figure 7 shows the performance benefit of doubling the number of filter stages when the filter operation is twice as expensive as the other pipeline stages.

Image Pipeline with Load Balancing

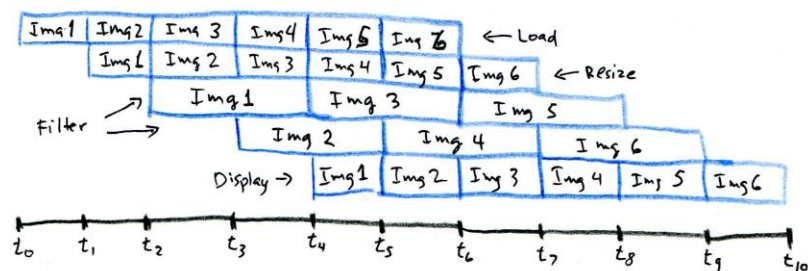


Figure 7

If all pipeline stages except the filter stage take T units of time to process an image, and the filter stage takes $2 \times T$ units of time, then using two filter stages and two producer queues to load balance the pipeline results in an overall speed of approximately T units of time per image as the number of images grows. If you run the ImagePipeline and select the **Load Balanced** radio button, you will see this effect. The speed of the pipeline (after a suitable number of images have been processed) will converge on the average time of the slowest single-instance stage or on one-half of the average filter time, whichever is greater.

The queue wait time of Queue 3, which is shown on the ImagePipeline's user interface, indicates the overhead that is introduced by waiting on multiple producer queues. This is an example of how adding

overhead to a parallel computation can actually increase the overall speed if the change also allows more efficient use of the available cores.

Pipelines and Streams

You may have noticed that blocking collections and streams have some similarities. It's sometimes useful to treat a blocking collection as a stream and vice versa. For example, you may want to use a pipeline pattern with library methods that read and write to streams. For example, suppose that you wanted to compress a file and then encrypt it. Both compression and encryption are supported by the .NET Framework, but the methods expect streams, not blocking collections, as input.

It is possible to create a stream whose underlying implementation relies on tasks and a blocking collection. See the Further Reading section in this chapter for more information.

Asynchronous Pipelines

The pipelines that have been described so far are synchronous. Producers and consumers are long-running tasks that block on inputs and outputs. You could also have an asynchronous pipeline, where tasks aren't created until data becomes available. For that, you could use something like the **AsyncCall** class that is found in the `ParallelExtensionsExtras` sample on MSDN. The **AsyncCall** class is a queue that a producer puts data into; if there's currently no task processing the queue when data arrives, a new task is spun up to process the queue, and it stays alive processing data as long as there's incoming data to process. If it ever finds there is no more data, the task goes away. If more data arrives, a new task will be started.

Anti-Patterns

There are a few things to watch out for when implementing a pipeline.

Thread starvation

A pipeline requires all of its tasks to be executing concurrently. If there are not enough threads to run all pipeline tasks, then the blocking collections can fill and block indefinitely. Task inlining, which was described in chapter 3, doesn't help. To guarantee that a thread will be available to run each pipeline task, you can use the task creation option **LongRunning** of the default task scheduler. If you forget this option and are using the default task scheduler, you are still safe. The thread pool will eventually notice that early stages of the pipeline are not making progress and inject the additional threads that are needed for the later pipeline stages.

Infinite Blocking Collection Waits

If a pipeline task throws an exception, it will no longer take values from its input blocking collection. If that blocking collection happens to be full, then the task that writes values to that collection will be blocked indefinitely. You can avoid this situation by using the technique that was described in the `Canceling a Pipeline` section of this chapter.

Forgetting GetConsumingEnumerable()

Blocking collections implement **IEnumerable<T>**, so it's easy to forget to call the **GetConsumingEnumerable** method. If you make this mistake, the enumeration will be a snapshot of the blocking collection's state, and enumerating the results will not consume from the collection or modify it in any way, which means that multiple consumers could get the same items.

It's easy to forget to call the **GetConsumingEnumerable** because the **BlockingCollection** class implements **IEnumerable<T>**. Enumerating over the blocking collection instance won't consume values. Watch out for this!

Using Other Producer/Consumer Collections

The **BlockingCollection<T>** class uses a concurrent queue as its default storage mechanism. However, you can also specify your own storage mechanism. The only requirement is that the underlying storage must implement the **IProducerConsumerCollection** interface. In general, using the blocking collections provided by the .NET Framework is easier and safer than writing your own implementation.

The .NET Framework provides several implementations of the **IProducerConsumerCollection** interface. These include the **ConcurrentBag** and the **ConcurrentStack** classes. Therefore, in principle, you could use bag (unordered) or stack (LIFO) semantics for the buffers between your pipeline's stages.

Generally, only the default FIFO ordering is recommended. If you use a concurrent bag, then the outputs of your pipeline stages do not depend on order. In this case, a parallel loop could be used instead of a pipeline. Parallel loops are faster and easier to code. The fact that you are using unordered buffers is evidence that you are using the wrong pattern.

Design Notes

When you use the pipeline pattern to decompose a problem, you need to consider how many pipeline stages to use. This depends on the number of cores you expect to have available at runtime. Unlike most of the other patterns in this book, the pipeline pattern does not automatically scale with the number of cores. This is one of its limitations, unless additional parallelism is introduced within a pipeline stage itself.

More stages work well unless the overhead of adding and removing elements from the buffers becomes significant. This is usually only a problem for stages that perform very small amounts of work.

However, to achieve a high degree of parallelism you need to be careful that all the stages in the pipeline take approximately the same amount of time to perform their work. If they don't, the pipeline will be gated by the slowest component and may experience processor undersubscription.

Buffer size is also important for overall performance. Very small buffers will block the pipeline, especially if their buffer elements take a variable amount of processing time. Allocating larger buffers accommodates the variability in processing time. The size of the buffer can also depend on the size of

the objects it contains. You would probably want to use a buffer with fewer entries if each element contained an object such as a large bitmapped image that required a lot of memory.

In general, buffers should be large enough to absorb variability in the pipeline flow but no larger. Use the profiler to understand the throughput characteristics of the pipeline and modify the buffer sizes to minimize the amount of time each stage is blocked by I/O waits.

Related Patterns

Pipelines have much in common with operating system concepts of pipes and filters. Pipelines are also related to streaming concepts.

Pipelines are expressions of a general technique known as producer/consumer. The pipeline is composed of a series of producer/consumers, each one depending on the output of its predecessor.

Exercises

4. Write your own pipeline by modifying the example shown in the first section of this chapter.
5. Execute the code with the Concurrency Visualizer. View and interpret the results.

Further Reading

The use of streams that are implemented on top of tasks and blocking collections is discussed by Toub. This reference uses the term transfer stream for this concept.

Multiplexing inputs from multiple producer queues is covered by Campbell.

A description of the pipes and filters pattern used by command shells for operating systems is described by Buschmann.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

C. Campbell, M. Veanes, J. Huo, and A. Petrenko, Multiplexing of Partially Ordered Events, in *TestCom 2005*, Springer Verlag, June 2005. Available online at <http://research.microsoft.com/apps/pubs/default.aspx?id=77808>.

S. Toub. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#, 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>

Appendix A Adapting Object-Oriented Patterns

Facades, decorators and repositories are examples of well established object-oriented patterns that may already be part of your development toolkit. In this appendix, you'll learn how to adapt these patterns and others to work with parallel programs. Simply applying some of these patterns can reduce the complexity of your parallel code. You'll also learn about some things to watch out for. This appendix isn't a complete list of object-oriented patterns, it simply highlights some of the more common patterns and the considerations for using them in a parallel program.

If you're not already familiar with object-oriented patterns, see the "Further Reading" section of this appendix for more information.

Structural Patterns

Façades, adapters and decorators are all structural patterns that can hide the complexities of parallelism behind a simpler interface. Of course, any time you add another layer of abstraction there is a tradeoff between simplicity and performance. Consider this tradeoff then applying these patterns.

When you use any of these patterns to hide the underlying parallelism, methods and properties should copy values or expose immutable types rather than provide references to shared objects with mutable state. Using immutable types or copied objects removes dependencies and prevents the possibility of sharing the same object (reference) across parallel tasks. This is especially important if you are hiding parallelism from the author of the calling code, they may be unaware of the problems with passing shared references or mutable objects to code that then executes in parallel. Your implementation may require additional type mapping code to protect the caller from these problems. See the section on "Immutable Types" for a further discussion of this approach.

Façade

The Façade pattern presents a simplified view of a larger system. You may want to use this pattern, or libraries that use this pattern, to hide the complexities of parallelism from other parts of an application. You can use a façade to hide your parallel functionality behind a serial API that protects the developer from any side effects caused by parallelism.

A façade adds to the complexity of an application's architecture; it only makes things simpler if the system behind the façade is already very complex. If you're adding a façade to hide parallelism, you probably needed one in any case!

Example

In the application sample shown in chapter 5, the view model acts as a façade to the underlying application model. The view model abstracts the inner workings of the user interface layer of the application. The view has no knowledge of the inner workings of the view model. Instead, it passively depends on the view model to set its properties. See "Model-View-ViewModel" for more information.

Guidelines

- The façade reduces complexity by hiding the details of a parallel implementation behind a simpler interface.
 - A façade can also be easily refactored to allow different (parallel) implementations of particular sub-systems to be used, depending on the available hardware or problem size, without altering the calling code.
 - The previous two points are general properties of the façade pattern. They are particularly valuable when dealing with the complexity of parallelism.
-

Decorators

The Decorator pattern overrides the behavior of an underlying class. Decorators use a "contains" relationship and inheritance. You can decorate serial implementations to create parallel ones. Like the Façade pattern, the Decorator pattern can hide the underlying complexity of parallelism from the calling code.

One obvious application of decorators is to write a locking decorator for a type that is not thread safe. There are two reasons to avoid this approach. A locking decorator is unlikely to have good parallel performance unless the locks are used infrequently. While adding locks to individual methods may appear to add thread safety, combinations of those methods may still require additional locking to ensure correctness.

Example

The following example shows a decorated **IImageEditor** interface. The underlying **SerialEditor** implementation is decorated by the **ParallelEditor** class, a parallel implementation that calls the decorated type.

```
public interface IImageEditor
{
    void Rotate(RotateFlipType rotation,
               IEnumerable<Bitmap> images);
}

public class SerialEditor : IImageEditor
{
    public void Rotate(RotateFlipType rotation,
                     IEnumerable<Bitmap> images)
    {
```

```

        foreach (Bitmap b in images)
            b.RotateFlip(rotation);
    }
}

public class ParallelEditor : IImageEditor
{
    private IImageEditor decorated;

    public ParallelEditor(IImageEditor decorated)
    {
        this.decorated = decorated;
    }

    // ... Modified behavior
    public void Rotate(RotateFlipType rotation,
        IEnumerable<Bitmap> images)
    {
        if (decorated == null)
            return;
        Parallel.ForEach(images, b =>
        {
            b.RotateFlip(rotation);
        });
    }

    // Additional behavior...
}

```

The example creates a decorated **ParallelEditor** and passes an instance of **SerialEditor** to the constructor. **ParallelEditor** modifies the behavior of the **Rotate** method to run using **Parallel.ForEach** but may rely completely on the **SerialEditor** for other, unmodified, method implementations.

```

IList<Bitmap> images = new List<Bitmap>();
// Load images...

IImageEditor parallel = new ParallelEditor(new SerialEditor());
parallel.Rotate(RotateFlipType.RotateNoneFlipX, images);

```

It is important that the decorated **Calculate** method does not access mutable shared state. Each step of the **Parallel.ForEach** loop must be independent.

Guidelines

- The principle advantage of the decorator is that it encapsulates parallelism while preserving the existing interface. The calling code remains largely unchanged.
- A decorator allows different (parallel) implementations to be used, depending on the available hardware or problem size, without altering the calling code.

- The decorator separates concerns. The decorated type implements the work being done while the decorator is only responsible for parallelizing the work.
 - The first two points are general properties of the decorator pattern. They are particularly valuable when dealing with the complexity of parallelism.
-

Adapters

The Adapter pattern translates from one interface to another. This pattern is applicable to parallelism because, as with the Façade pattern, it allows you to hide complexity and to expose an interface that is easier for developers to use.

Example

The following code exposes an interface named **IWithEvents** that is based on events, and an interface named **IWithFutures** that uses futures. The goal is to use interfaces that are more familiar to developers who are not parallel programming experts or who prefer an event-based model.

```
public interface IWithFutures
{
    Task<int> Start();
}

public class FuturesBased : IWithFutures
{
    public Task<int> Start()
    {
        return Task<int>.Factory.StartNew(() =>
        {
            // ...
        });
    }
}

public interface IWithEvents
{
    void Start();
    event EventHandler<CompletedEventArgs> Completed;
}

public class EventBased : IWithEvents
{
    readonly IWithFutures instance = new FuturesBased();

    public void Start()
    {
        Task<int> task = instance.Start();
        task.ContinueWith((t) =>
```

```

        {
            var evt = Completed;
            if (evt != null)
                evt(this, new CompletedEventArgs(t.Result));
        });
    }

    public event EventHandler<CompletedEventArgs> Completed;
}

```

The event based implementation adapts **IWithFutures** interface and allows results to be handled by an event handler.

```

IWithEvents model = new EventBased();
bool completed = false;

model.Completed += (s, e) =>
{
    Console.WriteLine("Completed Event: Result = {0}", e.Result);
    completed = true;
};

// Start model and wait for Completed event...
model.Start();

```

Guidelines

- Use the adapter pattern to modify an existing interface to make it easier for developers who are not familiar with parallelism to use.

Repositories and Parallel Data Access

A repository mediates between the application logic and the data access layers. You can think of a repository as an application of the Façade pattern to one or more data sources. Repositories in .NET commonly use ADO.NET to access a database. This section deals with the considerations arising from calling an ADO.NET based repository from a parallel application, rather than implementing the Repository pattern itself.

ADO.NET is optimized for throughput, and scalability. ADO.NET objects do not lock resources and must only be used on a single thread. Similarly, the ADO.NET Entity Framework (EF) is built on top of ADO.NET and has the same limitation. Given these constraints, how can your application access a repository from multiple tasks and when would it be appropriate to do this?

Accessing data from a set of parallel tasks is very similar to accessing it from the ASP.NET thread pool. Much of the same ASP.NET and data access guidance applies. Specifically, when connecting to a database from multiple tasks, do not share connections between tasks and keep connections open for as short a time as possible so that they can be reused.

Connecting from many parallel tasks to a single database may seem like a good way to improve performance. For example, you may wonder why there's no such thing as "PLINQ-to-SQL." In fact, using multiple tasks for data access can reduce the time taken by individual clients to access a database, but it negatively affects the overall throughput of the database. In other words, individual clients will each use more database resources, which may make them individually faster, but it's at the expense of the number of clients that can get access.

Parallelism is appropriate for applications that merge the results of queries from several different data sources. Applications that run on Windows Azure are good examples. Windows Azure applications often store data in a mixture of table storage and blob storage and may break data up between databases for reasons of scalability and performance.

Example

For examples of how to connect to several different databases in parallel see the "Further Reading" section.

Guidelines

- Don't share ADO.NET connections between tasks. ADO.NET is not thread safe. Each task should use its own connection.
- Keep connections to the data source for as short a time as possible. Explicitly close connections to ensure that unused connections are not left open. Don't rely on garbage collection to do this for you.
- Use tasks to parallelize connections to different databases. Using tasks to open multiple database connections to the same database may have significant performance implications.

Singletons and Service Locators

Singletons are classes with only one instance. Service locators are singletons that control object resolution, or the mapping of interfaces to specific implementation types at run time. Service locators support unit testing and composite or composable applications. Querying a service locator for an instance that provides a particular interface can return a single shared object, a new instance that is not shared, or a cached instance from a pool. This is often referred to as "resolving" an instance.

Normally instances of an object are created using the class constructor as in this example.

```
MyClass variable = new MyClass();
```

Rather, you want to be able to obtain the single instance, as shown here.

```
MyClass variable = MyClass.Instance;
```

Where **Instance** is a static read-only property of the class, and each time it is called, it will refer to the same single instance of **MyClass**. By adding this special **Instance** property, **MyClass** is transformed into a singleton class.

The instance returned by the singleton is also evaluated lazily. Lazy evaluation means that the instance is only created when needed. So your program can avoid creating an object that either uses lots of resources or is time consuming to create, until it is actually needed.

The use of singletons is sometimes criticized because it can lead to shared state. As you've seen throughout this book, mutable shared state prevents parallel execution. However, singletons are a good way to easily identify global state. Remember that when you see singletons and service locators in an application, it's likely that there's shared state. Shared state will likely impact scalability of your application so it's worth considering if this can be removed before simply applying the singleton or service locator pattern.

The presence of service locators and singletons suggests shared state.

See the “Further Reading” section for various approaches to implementing thread safe singletons.

Implementing a Singleton with the Lazy<T> Class

The .NET 4 Framework makes it easy to implement singletons. It provides a class named **Lazy<T>**. Here is an example.

```
public sealed class LazySingleton
{
    private readonly static Lazy<LazySingleton> instance =
        new Lazy<LazySingleton>(() => new LazySingleton() );

    private LazySingleton() { }

    public static LazySingleton Instance
    {
        get { return instance.Value; }
    }
}
```

The **Lazy<T>** class allows you to implement the singleton pattern. It not only handles locks around the creation of the instance value, but it also throws any exceptions from the **instance** constructor to the calling code. The class uses the **sealed** keyword to prevent further inheritance and unintended modifications to the class.

The Lazy<T> class is an easier way to implement a singleton.

Notes

The singleton implementation discussed in this section is an example of *lazy evaluation*. This is a technique that defers initialization until first use. In many cases, objects expose data that only requires evaluation when it is needed. It may be that the object itself can be initialized just when it is needed. In other words, the evaluation can be done lazily, or on demand. Lazy evaluation avoids unnecessary work. An example of lazy evaluation is with the **instance** member in the singleton example. The **Lazy<T>** class allows you to introduce lazy initialization in a thread-safe way.

The singleton implementation using **Lazy<T>** provides a thread safe way to resolve an instance of your class but does nothing to make the remainder of your class thread safe. Singletons imply shared state, so further synchronization may still be required.

Implementations of service locators also need to deal with synchronization issues. If you've written your own service locator or dependency injection (DI) container and you want to make it thread safe, then you need to either ensure that multiple resolves do not interfere with each other or implement locking to ensure that only one thread resolves a type at a time. If you're using a third party DI container then you should check that its resolution-related classes are thread-safe. Most DI containers, such as Unity, support thread-safe resolution but not thread-safe configuration.

Guidelines

Here are some guidelines for using singletons and service locators in parallel programs.

- Inclusion of service locators or singletons strongly suggests that your program relies on a shared resource or shared data. Consider modifying your algorithm to remove the need for shared resources or data where possible.
- Shared resources may impact the performance of your parallel application because tasks that access the shared resource have to wait their turn. The degree of impact depends on the amount of time your tasks spend waiting relative to the amount of time spent doing other work.
- Ask yourself if your singleton is necessary and if all the data inside the singleton needs to be there. A singleton should contain only essential data.
- Evaluate how often the data that must be protected is accessed. If the singleton contains different kinds of data that are accessed with different levels of frequency, you will probably need to take too many locks.
- If you're using a service locator, whether from a third party or one you wrote yourself, then ensure that its object-resolution functionality is thread safe. Remember that this may impact performance if the locator serializes access from multiple threads to ensure correctness.
- The examples in this section demonstrate how to create thread-safe singletons. You must also think about parallel access when you design the shared objects that they return.

Model-View-ViewModel

Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM) are both patterns that separate the domain logic from the UI. This separation makes your application easier to understand and test. The model encapsulates the application logic and state. The view model exposes appropriate state to the view in a bindable form. It supports the view by managing state and interaction logic. The view model acts as a façade to the underlying model. The view model may add additional behaviors that are

required by the view. The view model may also implement a Composite pattern, composing itself from more than one model object.

MVVM is particularly applicable to Windows Presentation Foundation (WPF) based applications because it takes advantage of WPF's extensive support of data binding. In WPF, the view can be extremely simple and contain little or no code. It relies on data and command binding to connect it to the view model. The MVVM pattern is a specialization of the Presentation Model pattern that is tailored for WPF.

MVP has similar goals to MVVM in separating the concerns of the domain logic and UI logic, but does not rely on data binding. In MVP, the presenter acts as an intermediary between the model and the view. This section only shows examples of the MVVM pattern and not MVP but much of the MVVM example also applies to MVP. See the "Further Reading" section for more about view models and presenters.

From a parallel programmer's perspective, the biggest challenge is to ensure that your parallel code, which runs in the model, handles the UI thread correctly, updates the UI and exposes any data in a thread-safe way. This section discusses how to use continuation tasks to do this. There are also further issues related to concurrency and how to build multithreaded UIs that are responsive to users, but these are outside the scope of this book.

Example

The Adatum Dashboard application shown in chapter 5 is an example of a parallel MVVM application. To understand this explanation, refer to the source code that is located in the project Chapter5\A-Dash directory of the online samples. Figure 1 illustrates the architecture.

The Model View-View Model pattern

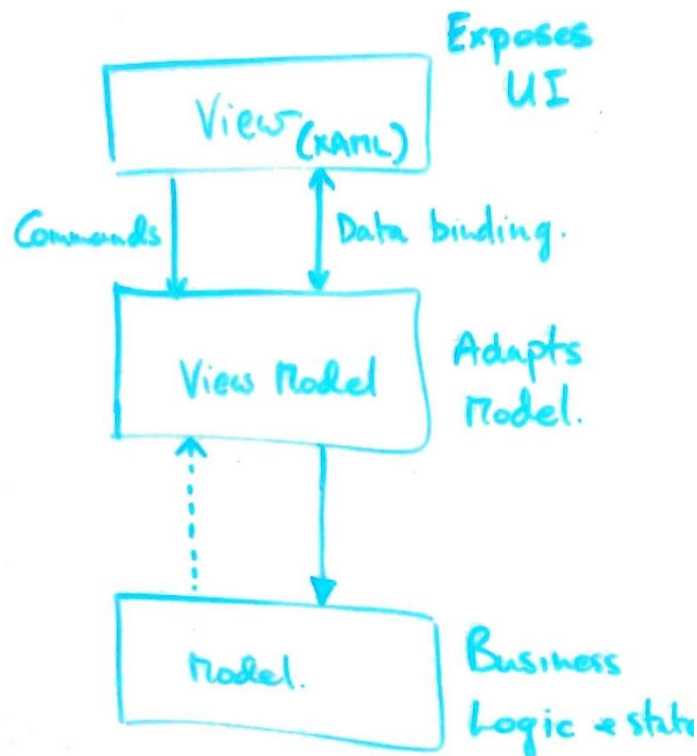


Figure 1

Employees expect the application to be responsive regardless of computational load and I/O latencies. Blocking the UI for more than a fraction of a second is unacceptable. Because of this requirement, even a program that takes advantage of any potential parallelism to perform the computations would still need to use background threads to ensure that the UI remains responsive. The Adatum Dashboard example demonstrates how to execute the model on the .NET thread pool and how to use the view model to expose the model's state to the view.

The **DoAnalysisParallel** method returns an **AnalysisTasks** object that contains futures for each of the operations in the task graph. For example, the **CompareModels** property contains a future that returns a market recommendation value. The future is implemented as a **Task<MarketRecommendation>** object. To following code retrieves the final result of the analysis.

```
AnalysisTasks tasks = engine.DoAnalysisParallel();
```

This code immediately returns an **AnalysisTasks** object that contains a **Task<>** object for each step of the calculation.

You don't need to wait until the final answer is ready to explore some of the early results. If you're interested in examining any partial results that are available before the entire analysis finishes, you can do this. For example, the following code retrieves the information produced by the task that loads the NYSE market data from the network.

```
MarketData nyseData = tasks.LoadNyseData.Result;
```

The Dashboard's User Interface

Futures and continuation tasks are also used in the Adatum Dashboard's UI, which relies on WPF. The Adatum Dashboard UI is designed so that the result of each step in the analysis can be viewed by the user as the computation progresses. There are individual buttons for each step. As each result becomes available, the corresponding button is enabled. It's also possible to cancel the analysis from the UI. The Adatum Dashboard application demonstrates how an application can use continuation tasks to keep the UI up to date.

The application uses continuation tasks instead of event handlers or other callback mechanisms. In fact, you can think of a continuation task as a kind of callback.

Here's a description of how the notification works. The following code is for the main window view model.

```
public class MainWindowViewModel : IMainWindowViewModel
    INotifyPropertyChanged, IDisposable
{
    // ...

    void OnRequestCalculate()
    {
        // ...

        AnalysisTasks tasks = engine.DoAnalysisParallel();
        AddButtonContinuations(tasks);
    }

    // ...
}
```

The user interface uses the MVVM pattern. The main window's view model has a **Calculate** command that invokes the **OnRequestCalculate** method when a user clicks the **Calculate** button.

Rather than starting background tasks with a method such as **QueueUserWorkItem** from the **ThreadPool** class, the view model asks the analysis engine to create an **AnalysisTasks** object that contains tasks that correspond to each of the analysis results.

This architecture demonstrates decoupling. The developer who wrote the application's analysis engine did not need to know how other parts of the application would use the results of the analysis.

Next, the handler uses the **AddButtonContinuations** method to create UI-specific continuation tasks. This is shown in the following code.

```
void AddButtonContinuations(AnalysisTasks tasks)
{
    AddButtonContinuation(tasks.LoadNyseData,
        t => { NyseMarketData = t.Result; });
}
```

```

AddButtonContinuation(tasks.LoadNasdaqData,
    t => { NasdaqMarketData = t.Result; });

AddButtonContinuation(tasks.LoadFedHistoricalData,
    t => { FedHistoricalData = t.Result; });

AddButtonContinuation(tasks.MergeMarketData,
    t => { MergedMarketData = t.Result; });

AddButtonContinuation(tasks.NormalizeHistoricalData,
    t => { NormalizedHistoricalData = t.Result; });

AddButtonContinuation(tasks.NormalizeMarketData,
    t => { NormalizedMarketData = t.Result; });

AddButtonContinuation(tasks.AnalyzeHistoricalData,
    t => { AnalyzedHistoricalData = t.Result; });

AddButtonContinuation(tasks.AnalyzeMarketData,
    t => { AnalyzedStockData = t.Result; });

AddButtonContinuation(tasks.ModelHistoricalData,
    t => { ModeledHistoricalData = t.Result; });

AddButtonContinuation(tasks.ModelMarketData,
    t => { ModeledMarketData = t.Result; });

AddButtonContinuation(tasks.CompareModels,
    t =>
    {
        this.Recommendation = t.Result;
        this.StatusTextBoxText =
            (this.Recommendation == null) ?
                "Canceled" : this.Recommendation.Value;
        this.ModelState = State.Ready;
    });

tasks.ErrorHandler.ContinueWith(
    t =>
    {
        if (t.Status == TaskStatus.Faulted)
            this.StatusTextBoxText = "Error";
        this.ModelState = State.Ready;
    }, TaskScheduler.FromCurrentSynchronizationContext());
}

```

The **AddButtonContinuations** method uses a special scheduler to create continuation tasks that automatically run in the UI thread after each of the tasks finishes and has results that are ready to view. The result of the **FromCurrentSynchronizationContext** method of the **TaskScheduler** class is a

TaskScheduler object that only allows its tasks to run in the current thread (that is, in this example the UI thread).

Each continuation task sets a property on the view model based on the task's result. When the final **CompareModels** task finishes, the UI view model is updated with the final recommendation. Callbacks registered with the view model notify the UI of the changes so that they will be reflected in the UI.

The last continuation task is the error handler. If an application error occurs, the Adatum Dashboard application observes the exception using a continuation task dedicated to error handling. The error handling continuation task updates the UI with a second continuation task that runs in the UI thread.

The helper method **AddButtonContinuation** is a simple application of the **ContinueWith** method to create continuation tasks.

```
void AddButtonContinuation<T>(Task<T> task,
                             Action<Task<T>> action)
{
    task.ContinueWith(
        action,
        CancellationTokens.None,
        TaskContinuationOptions.OnlyOnRanToCompletion,
        TaskScheduler.FromCurrentSynchronizationContext());
}
```

Before incorporating parallelism into the application, Adatum used background worker threads to handle the computationally intensive parts of the dashboard. However, the Adatum Dashboard application has some requirements, such as the use of WPF for the UI, that make continuation tasks more appropriate.

One of the reasons that the futures patterns works for the Adatum Dashboard is because it satisfies the thread affinity requirements imposed by WPF. Some frameworks place this constraint on objects they expose. For example, in WPF you have to run all methods of a UI object on the same thread that you used to create that object. Typically, applications use the **Dispatcher** class to route work that will be processed on the UI thread (See the "Further Reading" section).

The futures pattern makes it easy to deal with thread affinity. Continuation tasks can be configured with a task scheduler that runs the task on a particular thread. Antecedents of the task do not need to run on the same thread as the continuation task. This feature allowed Adatum's developers to distribute computationally intensive work among many cores and to ensure that the calculated values appeared in the UI without violating WPF's thread affinity constraint.

[Tasks make it easy to satisfy thread affinity constraints.](#)

Guidelines

- Use continuation tasks that are attached to the model's tasks to update the view model. This approach has the added benefit of further decoupling the model from the view.

- Use immutable types to pass results from the continuation tasks to the UI thread. This makes it easier to ensure that your code is correct.
- Use continuation tasks instead of background worker threads when you want to execute several different tasks in the background and merge the results before passing them back to the UI thread.
- Use the correct task scheduler to make sure that modifications to UI objects, which have thread affinity, take place on the main UI thread.

Immutable Types

When writing parallel code you can minimize the chances of one task interfering with another by not sharing objects with mutable fields between tasks. Instead, use immutable types, which are also sometimes called *purely functional data structures*. You can use immutable types to implement the (Immutable) Value Object pattern.

Do not overlook the importance of immutable types in parallel programming. They can improve reliability as well as performance.

A familiar example of an immutable data type is the .NET string class. The **System.String** class has many operations for creating strings, but none of these operations allow you to modify a string. Instead, the operations create a new string. Another example of an immutable type is the .NET **System.Uri** class. Instances of this class represent a uniform resource identifier (URI), with methods and properties that let you easily create a URI and query it for its constituent elements. You don't need to worry that another thread can modify a URI while you are using it.

Immutable types are a very useful form of scalable sharing. For example, when you append to a string, the result is a string that includes the addition. The original string is unmodified. This means that you can always use a particular string value in parallel code without worrying that another task or thread is modifying it. The combination of immutable types and the .NET Framework's concurrent collection classes is particularly useful. See the discussion of scalable sharing in chapter 1 for more about why immutable types are a good idea for parallel programs.

To be considered immutable, a type must meet the following conditions:

- The type must contain only fields that are not modified outside of the constructor. This means that an object's state cannot be changed after it has been constructed (provided the other constraints are met). The C# **readonly** keyword can be used to enforce this at compile time.
- The type must contain only fields that are also immutable types.
- The type must only inherit from other immutable types.
- The type cannot be inherited by mutable types. While this might seem unnecessary, it prevents a mutable type from inheriting an immutable type and altering the immutable type's behavior

by overriding a virtual method. You may want to mark your immutable types with the **sealed** keyword to enforce this.

- An instance of the type cannot publish references to itself until after construction is complete. If an object exposes a reference to itself (its **this** reference) during construction it might be possible for the reference to be accessed by another thread before construction is completed. It would then be possible for the second thread to observe an object whose internal state was changing during construction. In other words, the assumption of immutability would be broken.

The conditions above can be weakened if you only require observational immutability, an object that appears immutable publically but contains some private mutable state. If you choose this approach then your implementation must ensure that changes to the private mutable state of the object are done in the thread safe manner. **ReadOnlyCollection<T>** and **Lazy<T>** are both examples of observationally immutable types.

Objects in C# are not immutable by default, but it is possible to implement immutable types. Languages such as F# include built-in support for creating immutable types.

Example

Many of the types used by the Adatum Dashboard sample in chapter 5 are immutable. Values of these types are used to communicate the result of the financial analysis. For example, here is an implementation of the **StockData** class. Instances of this class represent the price history of a financial asset. The Adatum dashboard example passes **StockData** and **StockDataCollection** objects between tasks. No locking is required because the types themselves are immutable and therefore free of side effects.

```
public sealed class StockData
{
    readonly string name;
    readonly ReadOnlyCollection<double> priceHistory;

    public string Name
    {
        get { return name; }
    }

    public ReadOnlyCollection<double> PriceHistory
    {
        get { return priceHistory; }
    }

    public StockData(string name, double[] priceHistory)
    {
        this.name = name;
        this.priceHistory = new
            ReadOnlyCollection<double>(priceHistory);
    }
}
```

```

    }
    // ...
}

```

The **StockData** class uses the **readonly** keyword to ensure that the class does not inadvertently modify its fields. If you try to modify a read-only field, the compiler will issue an error.

Although the **StockData** class does not do this, you could also have defined the **Name** property as an automatic property with a private setter. Here is an example.

```

class StockData2
{
    public string Name { get; private set; }
    // ...
}

```

An advantage of automatic properties is that they are concise and easy to specify. A disadvantage is that you must manually ensure that none of the other operations of the class modify the property. Marking the setter as private only prevents modification of the property by callers that are outside of the class.

Immutable Types as Value Types

Value types use copy semantics for assignment. Two values of such a type are equal if all of their corresponding fields are equal. If two values are equal, their hash codes are also equal. The .NET type **System.Double** is an example of a value type. You can contrast value types with reference types. Classes in C# are reference types. By default, instances of a class are equal only if they are the result of the same invocation of the **new** operator.

You often want immutable types to behave like value types even though they are implemented using reference types. For example, you might want to use instances of your type as keys of a dictionary. The **System.String** class is an example of a reference type that in most ways emulates the behavior of a value type. (You can bypass the "value type" abstraction of the **String** class by invoking the **Object.ReferenceEquals** method.)

You can make an immutable type behave like a value type by implementing an **Equals** method that uses structural equality (field-by-field comparison) instead of reference equality (the test used by the **Object.ReferenceEquals** method). If you implement the **Equals** method in your type, you must also implement the **GetHashCode** method so that values that are equal always have the same hash code. It is also recommended that you implement the equals operator (**==**). Here is an example.

```

public sealed class StockData
{
    // ...

    public static bool operator ==(StockData a, StockData b)
    {
        if (System.Object.ReferenceEquals(a, b))
            return true;
    }
}

```

```

    if (((object)a == null) || ((object)b == null))
        return false;
    if (a.Name != b.Name)
        return false;
    if (a.PriceHistory.Count != b.PriceHistory.Count)
        return false;
    for (int i = 0; i < a.PriceHistory.Count; i++)
    {
        if (a.PriceHistory[i] != b.PriceHistory[i])
            return false;
    }
    return true;
}

public static bool operator !=(StockData a, StockData b)
{
    return !(a == b);
}

public override bool Equals(object obj)
{
    if (obj == null)
        return false;
    return Equals(obj as StockData);
}

public bool Equals(StockData d)
{
    if (d == null)
        return false;
    return (this == d);
}

public override int GetHashCode()
{
    int result = name.GetHashCode() ^ priceHistory.Count;
    for (int i = 0; i < priceHistory.Count; i++)
        result ^= priceHistory[i].GetHashCode();
    return result;
}
}

```

For further discussion of equality implementations see the "Further Reading" section.

Compound Values

Immutable types are not only limited to records with fixed numbers of fields. It's also possible to implement complex structures such as tables, trees and sequences as immutable or purely functional

data types. For example, the "Insert node" operation on an immutable tree data type returns a new tree.

You might at first think that immutable types such as tables and trees would be very inefficient. However, there are implementation techniques that do not require you to copy the entire structure with each insertion or deletion operation. It's possible for an implementation to use a significant amount of structure sharing. See the "Further Reading" section for more information.

Guidelines

- Consider using immutable types when sharing data between threads. You may want to use them in conjunction with the shared data structures discussed earlier in this appendix.
- For more complex types you may want to consider implementing "freezable" or "popsicle" immutability. Such types are mutable when they are created but become immutable when the user calls a "freeze" method. WPF uses this approach and provides a **System.Windows.Freezable** base class. You can also use a Builder pattern to provide a way to incrementally create an immutable value before freezing it.

For more information about creating immutable types in C# see the "Further Reading" section.

Shared Data Classes

Many of the patterns discussed in this book deliberately try to minimize the amount of shared data. .NET 4 includes a **System.Collections.Concurrent** namespace that contains several thread-safe data structures. If you decide that your application needs to share state then first review the collections provided by the .NET 4 before writing your own implementation. The challenge of writing shared data structures that are both correct and perform well shouldn't be underestimated. The following table describes the .NET 4 data structures.

Collections from the namespace **System.Collections.Generic** are not thread safe. You must use the collections in **System.Collections.Concurrent** whenever collections are shared by more than one thread.

Type	Description
BlockingCollection<T>	Implements a bounded and unbounded producer/consumer. Use this for connecting tasks in the style of producers and consumers. The Pipeline pattern in Chapter 7 uses blocking collections to connect pipeline stages.
ConcurrentBag<T>	An unordered collection of objects that is useful for storing data where ordering is not important. The Chapter 6 code samples include examples of storing unordered results in a concurrent bag.

ConcurrentDictionary<T>	A concurrent dictionary for storing and retrieving key/value pairs.
ConcurrentQueue<T>	A concurrent and scalable first in-first out (FIFO) non-blocking queue. Use this for connecting different tasks or for giving work to tasks. Chapter 6 discusses variations that use queues to provide values to tasks.
ConcurrentStack<T>	A concurrent and scalable last in-first out (LIFO) stack. Use this where the last item added is the most important one to process next.

Guidelines

- Think twice before resorting to sharing data. Many of the patterns described in this book either try to limit sharing or avoid it entirely.
- Where possible, use shared data collections in preference to locks.
- Use the shared data classes provided by the .NET 4 Framework in preference to writing your own classes. Writing shared data structures that work correctly and perform well in parallel code is difficult and should not be undertaken lightly.

Iterators

Iterators are a pattern that lets you program, in the style of a coroutine, the control flow of another method. The C# language provides built-in support for writing iterators. Here is an example.

```
public static IEnumerable<string> GetImageFileNames(
    string sourceDir, int maxImages)
{
    var names = GetImageFileNamesList(sourceDir, maxImages);
    while (true)
    {
        foreach (var name in names)
            yield return name;
    }
}
```

This example produces a non-terminating list of file names.

Example

Iterators are very useful when combined with parallel loops. In fact, a custom iterator is a kind of extension method for parallel loops. 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 IEnumerable<Tree<T>> Iterate<T>()
{
    var queue = new Queue<Tree<T>>();
    queue.Enqueue(this);
    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(myTree.Iterator(), node =>
{
    // ... process node in parallel
});
```

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.

Lists and Enumerables

The .NET type **ICollection<T>** defines the functionality of indexed lists. The type **IEnumerator<T>** is used for unindexed iteration.

You should be aware of how these two types interact with parallel loops. In certain rare cases, the parallel loop's default handling of the **ICollection<T>** type may not be what you want. This can occur when the **ICollection<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.

You can override **Parallel.ForEach**'s default handling of a source that provides the **ICollection<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 ... */ });
```

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.)

Further Reading

Structural Patterns

For best practices for using ADO.NET, go to <http://msdn.microsoft.com/en-us/library/ms971481.aspx> on MSDN. ADO.NET performance guidelines can also be found at <http://msdn.microsoft.com/en-us/library/ff647768.aspx> on MSDN. An overview of ADO.NET Entity Framework is also available on MSDN, <http://msdn.microsoft.com/en-us/library/bb399572.aspx>.

Martin Fowler describes the Repository pattern in "Patterns of Enterprise Architecture", Fowler (2003).

The patterns & practices guidance for Azure, which includes a discussion on repositories with parallel database access, is available online at <http://wag.codeplex.com/>.

Singleton

See Gamma (1994) for the original "Gang of Four" patterns and Bishop (2008) for a further discussion of implementing singletons and other design patterns in C#. For readers who want considerably more detail Duffy (2008), chapter 10 contains an excellent in depth description of singletons.

Documentation for the Unity dependency injection container and a discussion of containers can be found on MSDN, <http://msdn.microsoft.com/unity>.

For further discussion of lazy initialization, go to <http://msdn.microsoft.com/en-us/library/dd997286.aspx>

Model-View-ViewModel

For an in-depth discussion of the MVVM pattern, see the article in MSDN Magazine located at <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>. The MVP pattern is explained at <http://msdn.microsoft.com/en-us/magazine/cc188690.aspx>.

For an overview of the WPF threading model and how to update WPF objects with the **Dispatcher** class, go to <http://msdn.microsoft.com/en-us/magazine/cc163328.aspx>. The **BackgroundWorker** class and how to use it is explained, and has an example at <http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx> on MSDN.

Immutable Types

Eric Lippert's blog includes an excellent series of posts on immutability in C#. It's located at <http://blogs.msdn.com/b/ericlippert/archive/tags/immutability/>. Joe Duffy's blog also covers the requirements for immutability from a parallel perspective. It's located at <http://www.bluebytesoftware.com/blog/2007/11/11/ImmutableTypesForC.aspx>

Details of the requirements for implementing equality are covered on MSDN at [http://msdn.microsoft.com/en-us/library/ms173147\(VS.90\).aspx](http://msdn.microsoft.com/en-us/library/ms173147(VS.90).aspx).

For more information on implementing immutable types, including complex structures such as sequences, tables and trees, see Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998. For an example of a C# implementation of immutable types for sequences, sets and maps, see the NModel framework on CodePlex (<http://nmodel.codeplex.com>).

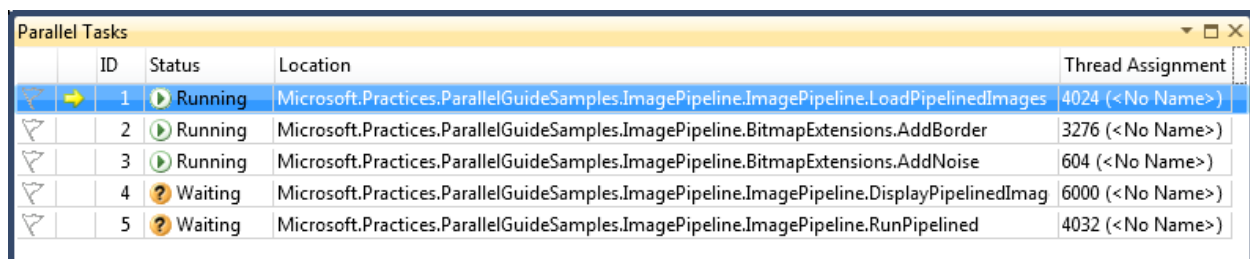
Appendix B: Debugging and Profiling Parallel Applications

The Visual Studio 2010 debugger includes two windows that assist with parallel programming: the Parallel Stacks Window, and the Parallel Tasks Window. The Premium and Ultimate editions of Visual Studio 2010 include a profiling tool. This appendix gives examples of how to use these windows and the profiler to visualize the execution of a parallel program and to confirm that it is working as you expect. After you gain some experience at this, you'll be able to use these tools to help identify and fix problems.

In Visual Studio, open the parallel guide samples solution. Set the **ImagePipeline** project from chapter 7 to be the Startup Project. Open **ImagePipeline.cs** and find the **LoadPipelinedImages** method. This is the method executed by the first task in the pipeline. Insert a breakpoint at the first statement in the body of the **foreach** loop. This is the loop that reads images from disk, and fills the pipeline as it iterates over the images.

Start the debugging process. You can either press F5, or on the **Debug** menu, click **Start Debugging**. The ImagePipeline sample begins to run and opens its graphical user interface window on the desktop. Select the **Pipelined** option, then click **Start**. When execution reaches the breakpoint, all tasks stop and the familiar **Call Stack** window appears. From the **Debug** menu, point to **Windows**, then click **Parallel Tasks**.

When execution first reaches the breakpoint, the Parallel Tasks window shows that the first pipeline task, **LoadPipelinedImages**, is running and all the other tasks are waiting. This is because there are no images in the pipeline yet. Pressing F5 several times (or from the **Debug** menu, click **Continue**) causes several images to be loaded, so the pipeline starts to fill and other tasks can run. This is shown in Figure 1. Recall that each task runs in a thread. This window shows the assignment of tasks to threads. More than one task can run in a thread when task inlining occurs, so it is possible that a thread may be running when one of its tasks is blocked.



	ID	Status	Location	Thread Assignment
▶	1	▶ Running	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.ImagePipeline.LoadPipelinedImages	4024 (<No Name>)
▶	2	▶ Running	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.BitmapExtensions.AddBorder	3276 (<No Name>)
▶	3	▶ Running	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.BitmapExtensions.AddNoise	604 (<No Name>)
▶	4	⚠ Waiting	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.ImagePipeline.DisplayPipelinedImag	6000 (<No Name>)
▶	5	⚠ Waiting	Microsoft.Practices.ParallelGuideSamples.ImagePipeline.ImagePipeline.RunPipelined	4032 (<No Name>)

Figure 1. The Parallel Tasks window

From the **Debug** menu, point to **Windows**, then click **Parallel Stacks**. In the Parallel Stacks window, from the drop-down menu in the upper left corner, click **Tasks**, then right-click on the background of the Parallel Tasks window and click **Show External Code**. This window shows the stack for each of the tasks. In Figure 2, the window contents have been enlarged (point to the zoom control in the left side of the window and use the slider) so only two stacks appear, but all stacks can be accessed.

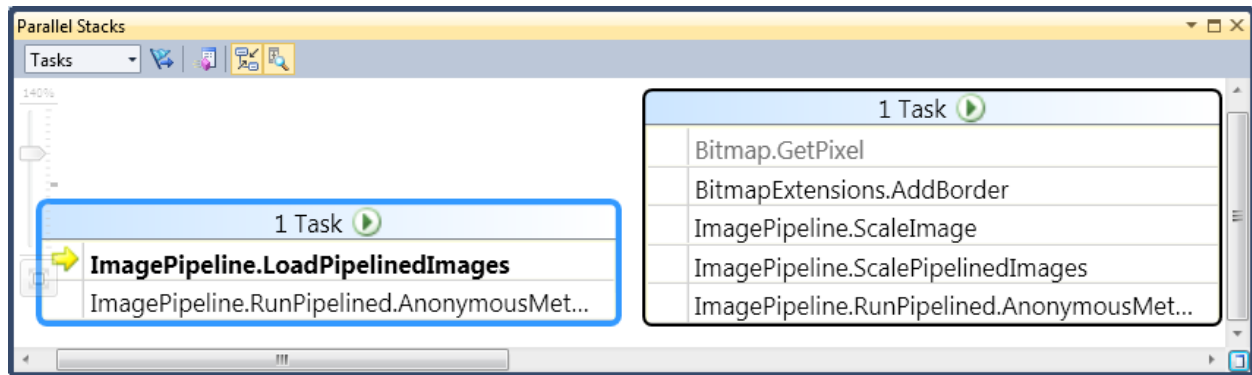


Figure 2. The Parallel Stacks window

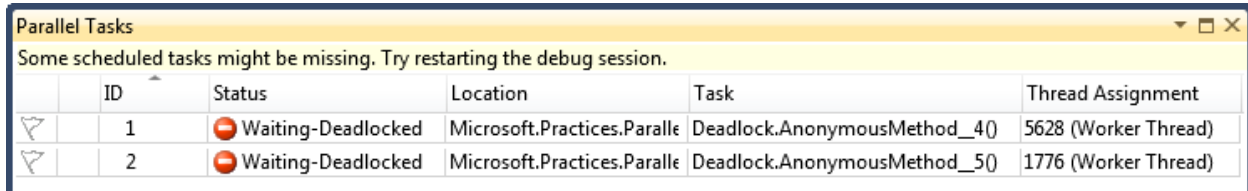
As you continue to press F5, the contents of each window change as the buffers between pipeline stages empty and fill. This is the expected behavior, so no bugs are indicated. These windows can also reveal unexpected behavior that can help you identify and fix performance problems and synchronization errors. For example, the Parallel Tasks and Parallel Stacks window can help to identify common concurrency problems like deadlocks. Consider the following code:

```
static void Deadlock()
{
    object obj1 = new object();
    object obj2 = new object();

    Parallel.Invoke(
        () => {
            for (int i = 0; ; i++) {
                lock (obj1) {
                    Console.WriteLine("Got 1 at {0}", i);
                    lock (obj2) Console.WriteLine("Got 2 at {0}", i);
                }
            }
        },
        () => {
            for (int i = 0; ; i++) {
                lock (obj2) {
                    Console.WriteLine("Got 2 at {0}", i);
                    lock (obj1) Console.WriteLine("Got 1 at {0}", i);
                }
            }
        }
    );
}
```

```
}
```

This code is a classic example of a deadlock. Two tasks each attempt to acquire locks in an order that leads to a cycle, resulting in eventual deadlock. When running this code, at some point the application will stop making forward progress (which will be visible due to lack of new console output). At that point, if you break into the debugger using the **Break All** option on the **Debug** menu, and open the Parallel Tasks window, you'll see something like that in Figure 3.



The screenshot shows the 'Parallel Tasks' window with a yellow header bar. Below the header, a message states: 'Some scheduled tasks might be missing. Try restarting the debug session.' Below this message is a table with the following data:

	ID	Status	Location	Task	Thread Assignment
▼	1	⏻ Waiting-Deadlocked	Microsoft.Practices.Paralle	Deadlock.AnonymousMethod_4()	5628 (Worker Thread)
▼	2	⏻ Waiting-Deadlocked	Microsoft.Practices.Paralle	Deadlock.AnonymousMethod_5()	1776 (Worker Thread)

Figure 3. Parallel Tasks window showing deadlock

The Concurrency Visualizer

The Visual Studio 2010 profiler includes the Concurrency Visualizer. It shows how parallel code uses resources as it runs: how many cores it uses, how threads are distributed among cores, and the activity of each thread. This information helps you to confirm that your parallel code behaves as you intend, and can help you diagnose performance problems.

The Concurrency Visualizer has two stages: data collection and visualization. In the collection stage, you enable data collection and run your application. In the visualization stage, you examine the data you collected. This example uses the Concurrency Visualizer to profile the `ImagePipeline` sample from chapter 7 on a computer with two cores.

You first perform the data collection stage. To do this, you must run Visual Studio as an administrator because data collection uses kernel-level logging. Open the parallel guide samples solution in Visual Studio. There are several ways to start a data collection run. One way is to open the Visual Studio **Debug** menu, and click **Start Performance Analysis**. The Performance Wizard begins. Click **Concurrency**, and select **Visualize the behavior of a multithreaded application**. The next page of the wizard shows the solution that is currently open in Visual Studio. Select the project you want to profile, **ImagePipeline**. Click **Next**. The last page of the wizard asks if you want to begin profiling after the wizard finishes. This checkbox is selected by default. Click **Next**. The Visual Studio profiler window appears and indicates it is **Currently Profiling**. The `ImagePipeline` sample begins to run and opens its graphical user interface window on the desktop. In order to maximize processor utilization, select the **Load Balanced** option, then click **Start**. In order to collect plenty of data to visualize, let the **Images** counter (on the graphical interface) reach at least 20. Then click **Stop Profiling** in the Visual Studio profiler window.

During data collection, the performance analyzer frequently takes a sample of data (called a snapshot) that records the state of your running parallel code. Each data collection run writes several data files, including a .vsp file. A single data collection run can write files that are hundreds of megabytes. Data collected during separate runs of the same program can differ because of uncontrolled factors such as other processes running on the same computer.

You can run the visualization stage whenever the files are available. You do not need to be running Visual Studio as an administrator to do this. There are several ways to begin visualization. You can request the Performance Wizard to start visualization as soon as data collection finishes. Alternatively, you can simply open any .vsp file in Visual Studio. If you select the first option, you'll see a summary report once the data is collected and analyzed. The summary report shows the different views you can see. These include a Threads view, a CPU Utilization view, and a Cores view. Figure 4 shows the Concurrency Visualizer screen displaying the Threads view; more explanation of the views appears below. Recall that each task is executed in a thread. The Concurrency Visualizer shows the thread for each task (recall that there may be more than one task per thread, due to inline tasks).

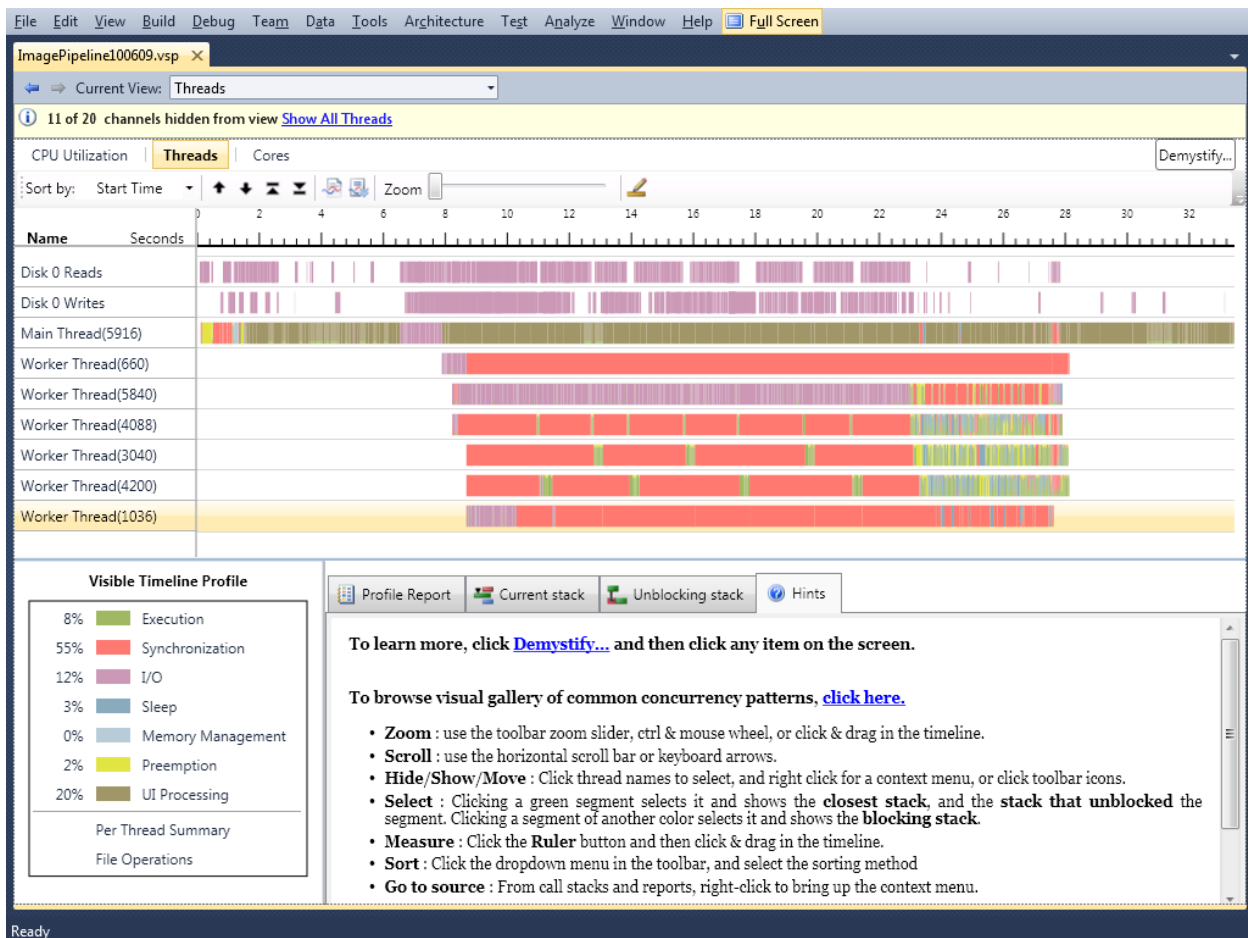


Figure 4 Concurrency Visualizer, Threads view

The Concurrency Visualizer screens contain many details that may not be clear in these figures, which are reduced in size and are not in full color. Files containing the full size, full color screen shots are available on the CodePlex site at <http://parallelpatterns.codeplex.com/>.

The **CPU Utilization** view shows how many processors (logical cores) the entire application (all tasks) uses, as a function of time. On the computer used for this example, there are two logical cores. Other processes not related to the application are also shown. For each process, there is a graph that shows

how many processors it is using at each point in time. To make the processes easier to distinguish, the area under each process's graph appears in a different color (some colors may not be reproduced accurately in this figure). Some data points show a fraction, not 0, 1, or 2, because each point represents an average calculated over the sampling interval.

During this particular data collection run, the graphical user interface showed that the first few images appeared slowly, then the remaining images appeared more rapidly. (This behavior did not occur on every run). The view reflects the behavior. Early in the run (before about 20000 on the time scale) the application runs in bursts, and fills the pipeline as it loads images. When there are several images in the pipeline (after about 20000), pipeline tasks can run in parallel and the application uses two logical cores.

This view also shows intervals between bursts (before 20000) where the application gets no processors. During these intervals the application is blocked, or is preempted by other processes

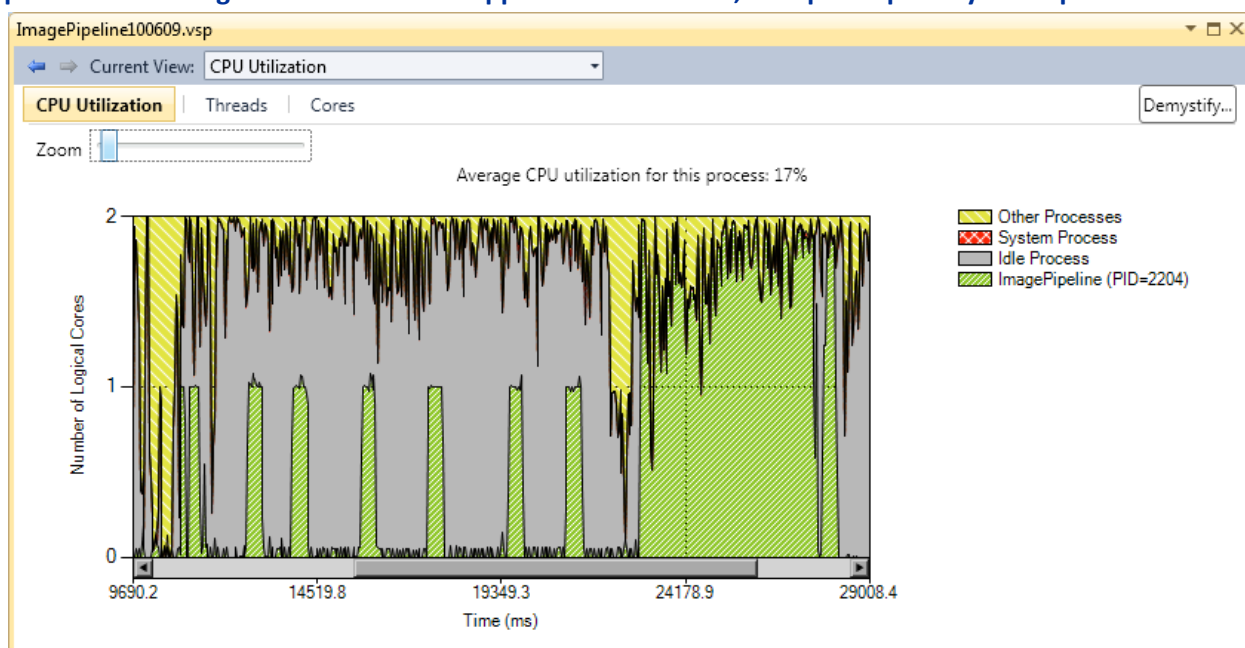


Figure 5. CPU Utilization view, detail

The **Cores** view shows how the application uses the available cores. There is a timeline for each core, with a color-coded band that indicates when each thread is running (a different color indicates each thread, but some colors may not be reproduced accurately in this figure). Between 10 and 22 on the time scale, the application runs in bursts and the empty intervals indicate when there was no work from this process running on either core. Between 22 and 28 the pipeline is filled and more tasks are eligible to run than there are cores. Several threads alternate on each core and the table shows much context switching. This is shown in Figure 6.

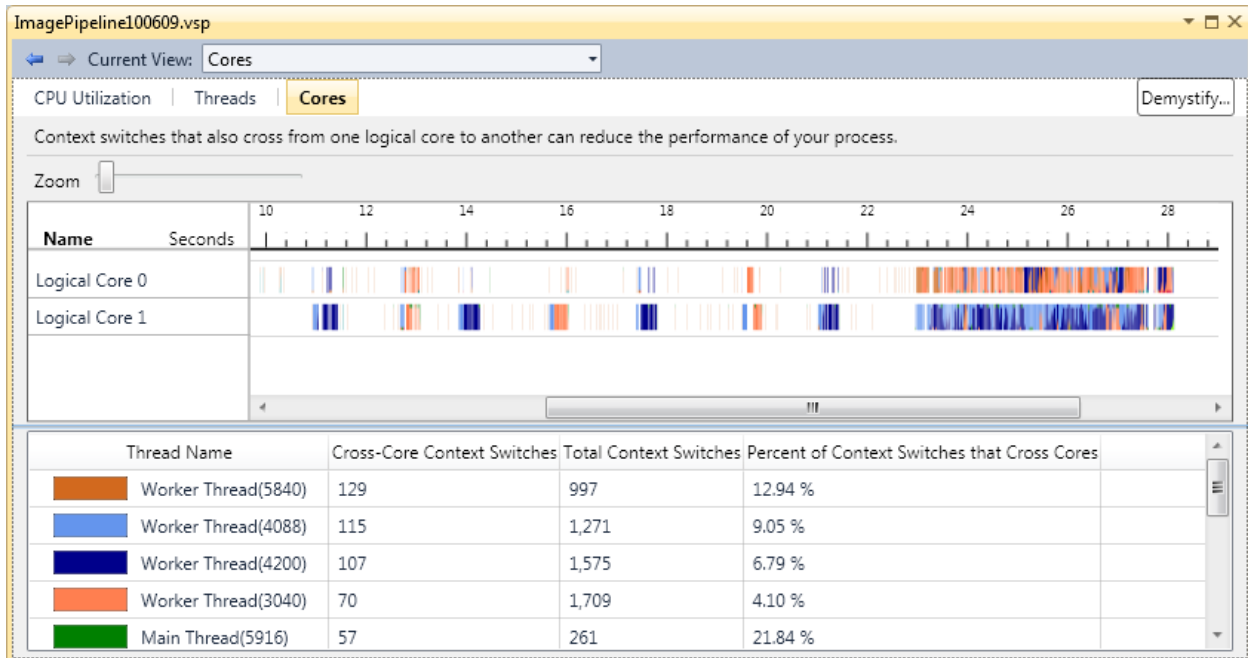


Figure 6. Detail of Cores view

The **Threads** view shows how each thread spends its time. There is a timeline with color-coded bands that indicate different kinds of activity (some colors may not be reproduced accurately in this figure). For example, red indicates when the thread is synchronizing (waiting for something). This view initially shows idle threads in the thread pool. (You can hide them by right clicking and selecting **Hide**). This view shows that the main thread is active throughout; the green color indicates user interface activity.

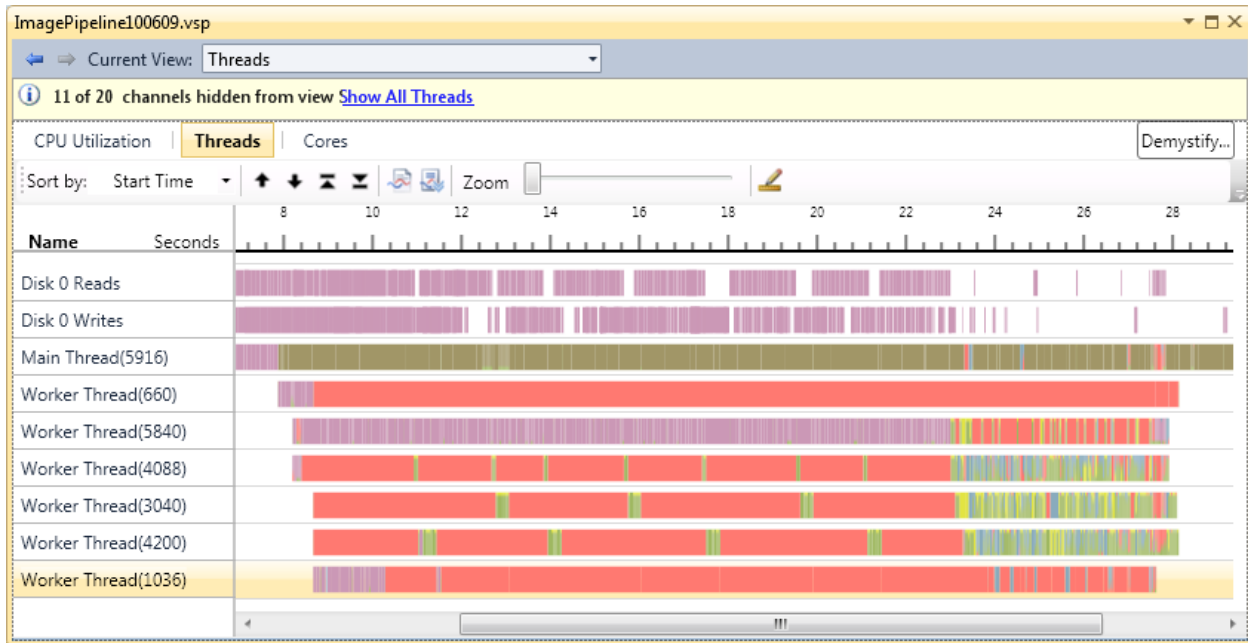


Figure 7. Threads view, detail

The pipeline threads execute in bursts before time 22, alternating between running and synchronizing, as they wait for the pipeline to fill. After 22, some pipeline threads execute frequently and others execute almost continuously. There are more pipeline threads than cores, so some pipeline threads must alternate running and preemption.

For complex applications with different phases you may want to mark different parts of your application. You can use the Scenario library to mark different phases of your application.

```
Scenario.Scenario myScenario = new Scenario.Scenario();
myScenario.Begin(0, "Main Calculation");

// Main calculation phase...

myScenario.End(0, "Main Calculation");
```

These markers will be displayed in the **Threads** view and **CPU Utilization** view. Don't use too many markers as they can easily swamp the visualization making it hard to read. The tool may hide some markers to improve visibility. You can use the zoom feature to increase the magnification and see the hidden markers for a specific section of the view.

Visual Patterns

The patterns discussed in this book are primarily focused on ways to express potential parallelism. However, there are other kinds of patterns useful in parallel development. The human mind is very good at recognizing visual patterns, a fact we can put to good use with the Concurrency Visualizer in order to

recognize and identify common patterns for poorly-behaved multithreaded applications. In this section, we outline a few such common visual patterns that will aid you in finding and fixing such performance issues.

Oversubscription

Oversubscription is the state that occurs when there are more runnable threads than the system has logical processors to run those threads. This can result in poor performance due to a high number of context switches, each of which takes some processing time and which can decrease the benefits provided by memory caches.

The Concurrency Visualizer makes it easy to visualize and recognize patterns of oversubscription, as they lead to large amounts of yellow regions in a profiler trace. Yellow shows that the thread was preempted, the thread was switched out. The following code snippet when traced will yield a quintessential rendering of oversubscription.

```
static void Oversubscription()
{
    for (int i = 0; i < (Environment.ProcessorCount * 4); i++)
    {
        new Thread(() => {
            // Do work
            for (int j = 0; j < 1000000000; j++) ;
        }).Start();
    }
}
```

Here is the Threads view from one run of this function on a quad-core system, producing a very distinguishable pattern:

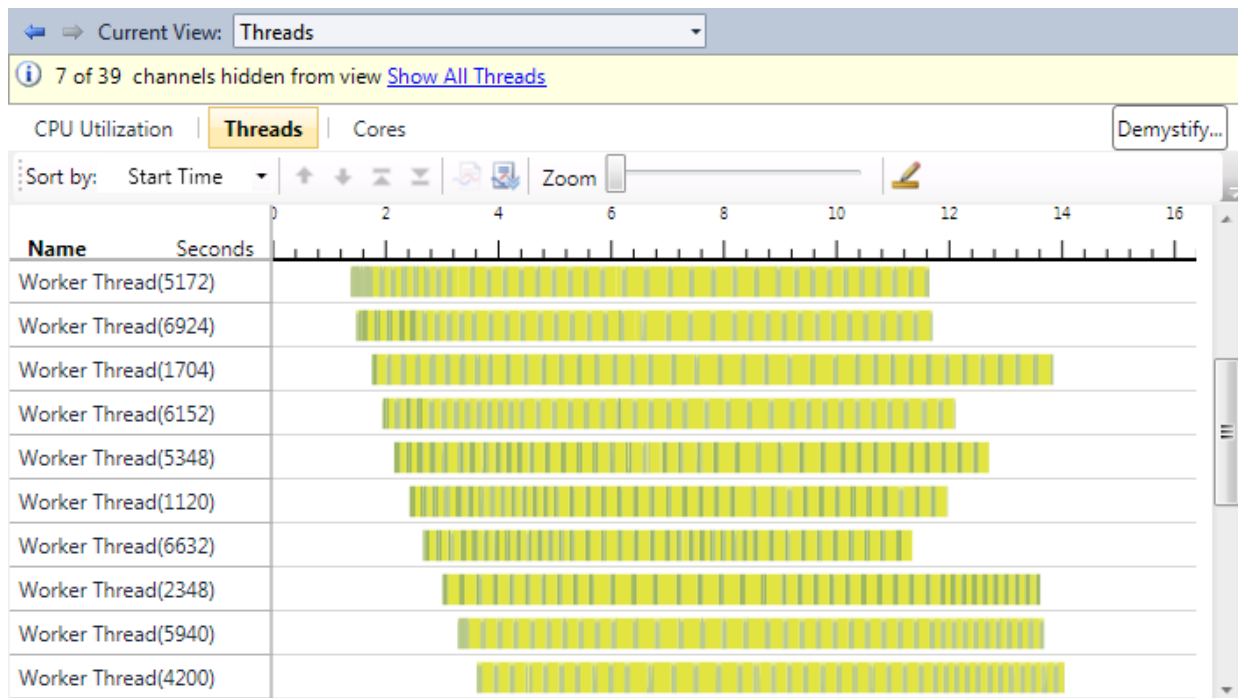


Figure 8. Threads view showing oversubscription

Lock Contention and Serialization

Contention occurs when a thread attempts to acquire a lock held by another thread. In many cases, this results in the first thread blocking until the lock is released. Such blocking shows in red in the Threads view in the Concurrency Visualizer, and is often a sign of decreased performance. Taken to the extreme, some applications find themselves serialized on one or more locks, even though multiple threads are being utilized.

As an example, consider the following method meant to produce a lock convoy, which leads to significant lock contention and serialization of the program even though multiple threads are in use.

```
static void LockContention()
{
    object syncObj = new object();

    for (int p = 0; p < Environment.ProcessorCount; p++)
    {
        new Thread(() => {
            for(int i=0; i<50; i++)
            {
                // Do work
                for (int j = 0; j < 1000; j++);

                // Do protected work
                lock (syncObj)
                    for (int j = 0; j < 100000000; j++);
            }
        });
    }
}
```

```

    }
    }).Start();
}
}

```

This code snippet produced the following pattern visible in the Threads view of the Concurrency Visualizer:

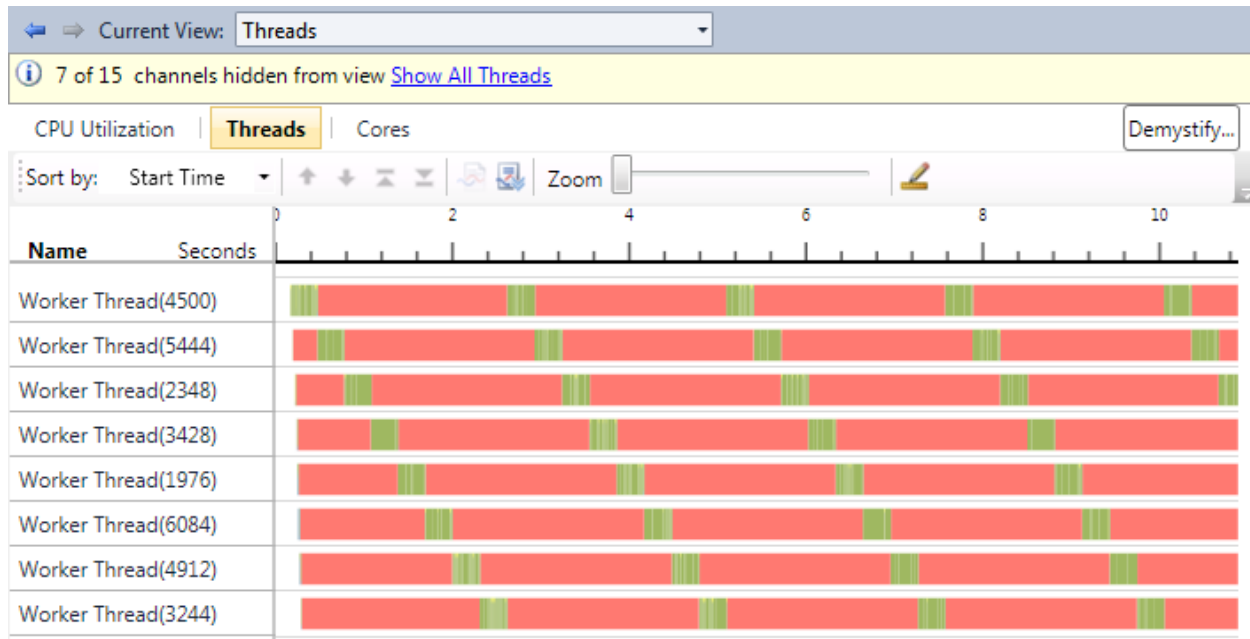


Figure 9. Threads view showing lock convoy

Load Imbalance

A load imbalance occurs when work is unevenly distributed across all of the worker threads participating in a parallel operation. This results in underutilization of the system, as some threads/cores sit idly while others finish processing the operation. The visual pattern produced by a load imbalance is recognizable in several of the views presented by the Concurrency Visualizer.

Consider the following code snippet:

```

static void LoadImbalance()
{
    const int loadFactor = 10;

    ParallelEnumerable.Range(0, 100000).ForAll(i =>
    {
        for (int j = 0; j < (i * loadFactor); j++) ;
    });
}

```

While most of the parallelism support in .NET 4 utilizes dynamic partitioning to apportion work between worker tasks, the **ParallelEnumerable.Range** method from PLINQ uses static partitioning. In this

example, on a system with eight logical cores, this will result in elements [0, 12500) being processed by one task, and elements [12500, 25000) being processed by another task, and so on. The body of the workload simply iterates from 0 to the current index value, which means that the amount of work to be done is proportional to the index. So, workers that process lower ranges will have significantly less work to do than the workers that process the upper ranges. This is evident by looking at the CPU Utilization view in the Concurrency Visualizer.

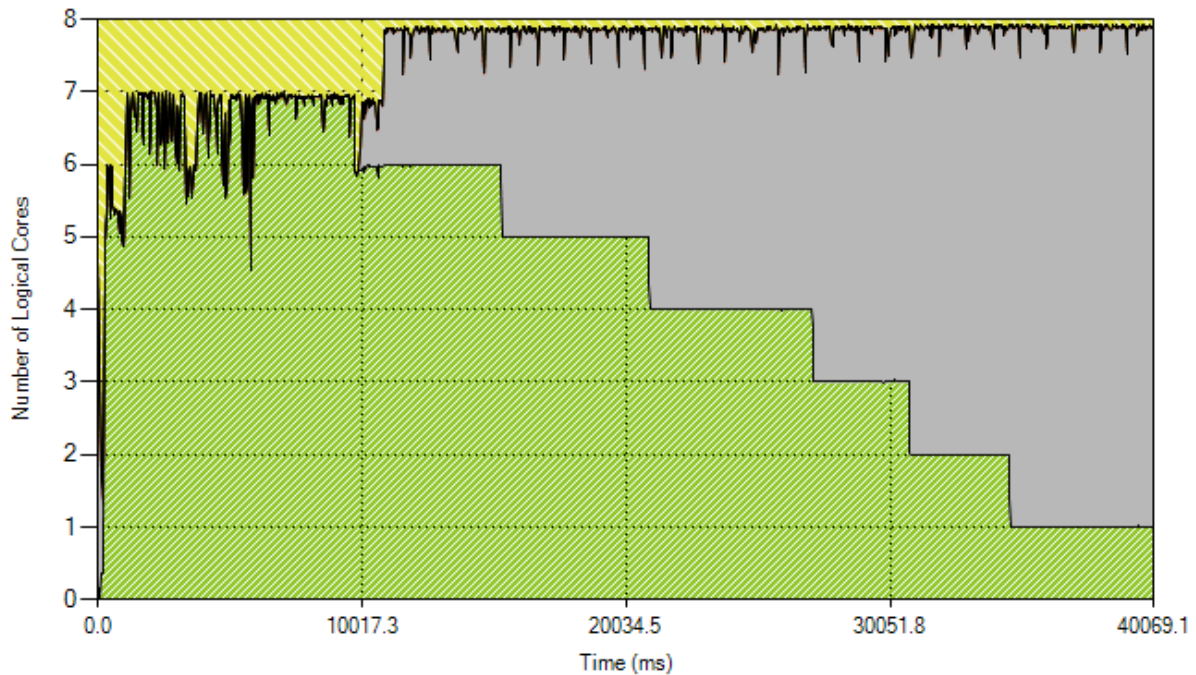


Figure 10. CPU view showing load imbalance

At the beginning of the method's execution, seven logical cores on this system are being utilized. However, after a period of time, usage drops as each core completes its work. This yields a "stair-step pattern," as threads are dropped from participation after they completed their portion of the work. This is also visible in the Threads view.

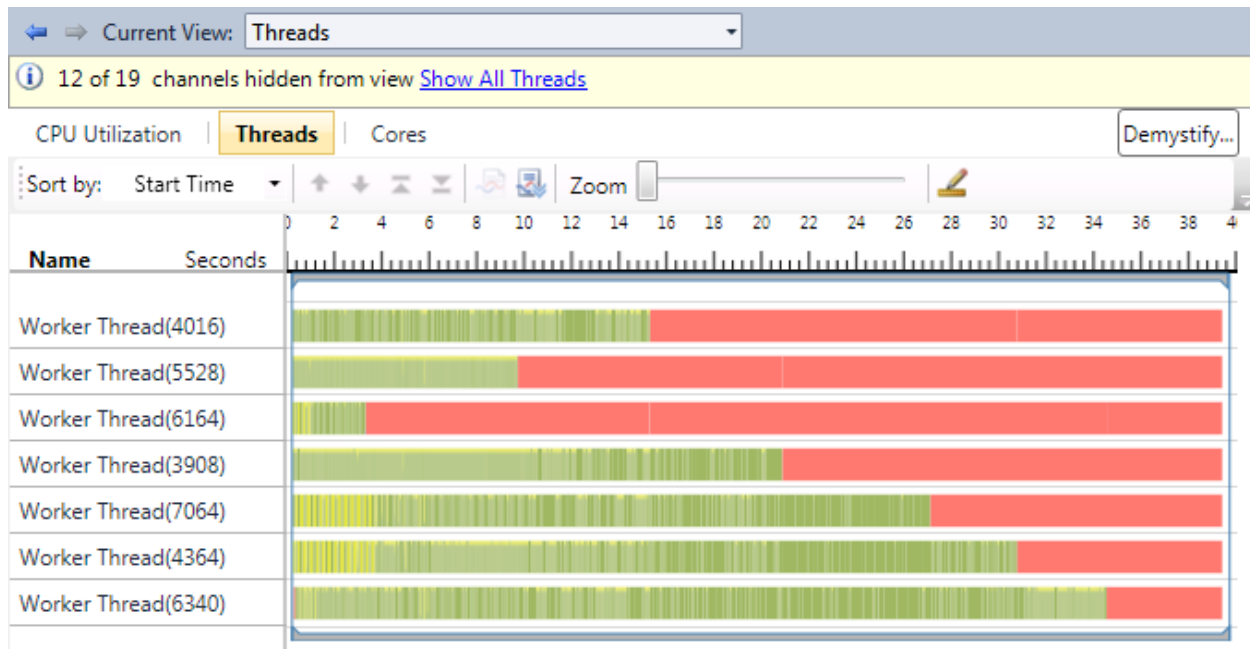


Figure 11. Threads view showing load imbalance

Here, we can see that after completing a portion of the work, the worker threads were idle while they waited for CLR Worker Thread 6340 to complete the remaining work.

Further Reading

The Parallel Performance Analysis blog at MSDN discusses many techniques and examples. MSDN also provides information on the Scenario library.

Parallel Performance Analysis in Visual Studio 2010. <http://blogs.msdn.com/b/visualizeparallel/>

Performance Tuning with the Concurrency Visualizer in Visual Studio 2010

<http://msdn.microsoft.com/en-us/magazine/ee336027.aspx>

Scenario Home Page <http://code.msdn.microsoft.com/scenario>

Appendix C: Technology Overview

Appendix C describes some of the parallel computing resources offered by Microsoft that are not covered in this guide. The "Further Reading" section includes links to web sites that have more information.

Microsoft's parallel programming resources

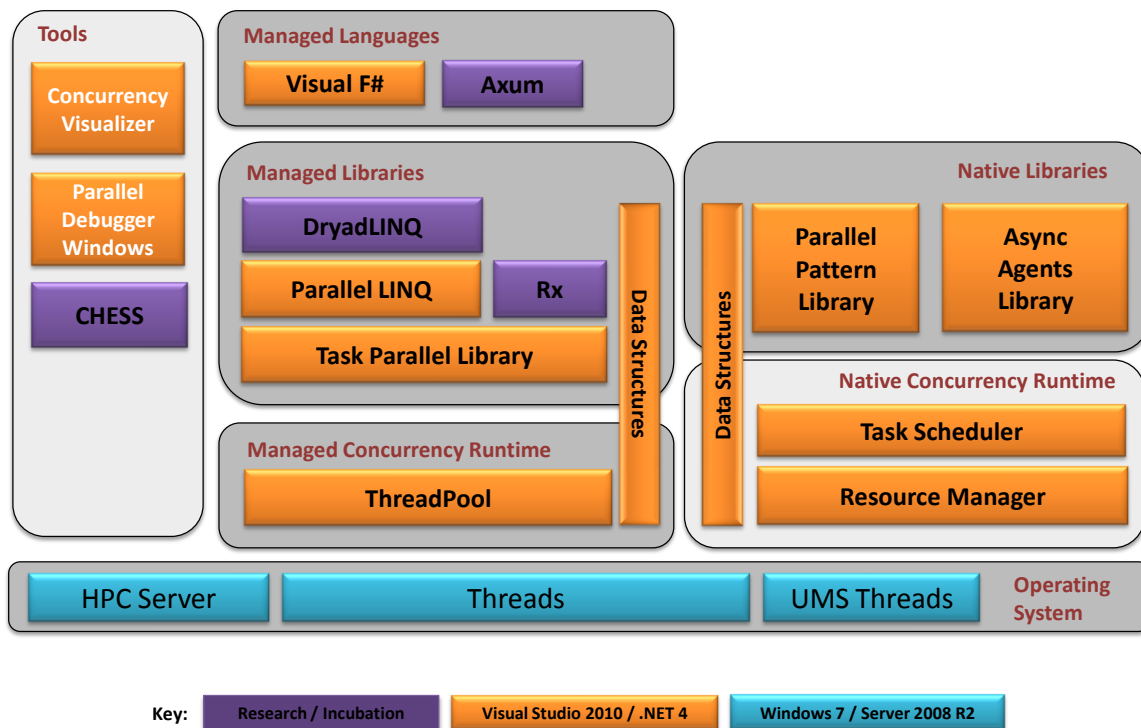


Figure 1

This guide covers the Task Parallel Library (TPL) and Parallel LINQ (PLINQ), which are managed libraries that ship as part of the .NET 4 Framework. They rely on the .NET ThreadPool and on coordination data structures and synchronization primitives.

The F# language also ships with the Visual Studio 2010. It exposes a more functional approach to parallelism than do TPL or PLINQ directly, and it emphasizes immutability. However, F#'s runtime

libraries build on and integrate with TPL, and the F# PowerPack includes parallelization support built on top of PLINQ. The samples in this guide have been ported to F#.

For Visual C++ developers, there are the Parallel Pattern and Asynchronous Agents libraries. These use the native Concurrency Runtime, a powerful scheduler and resource manager that are designed to efficiently execute native parallel workloads on manycore architectures. The Visual C++ version of this guide, *Parallel Programming with Visual C++: Design Patterns for Decomposition, Coordination on Multicore Architectures*, will be available soon.

Visual Studio 2010 contains several tools for debugging and profiling parallel applications. See Appendix B, "Debugging and Profiling Parallel Applications" for examples of how to use them. You can also use the CHES tools from Microsoft Research to detect bugs in your parallel code.

In addition to the technologies that ship with Visual Studio 2010 and the .NET 4 Framework, there are a number of other interesting research projects that focus on parallelism and concurrency. They include the Axum language, the Reactive Extensions for .NET (Rx), and DryadLINQ. Axum is a concurrent programming language built on the principles of isolation, actors, and message-passing to increase application safety, responsiveness, scalability, and developer productivity. Rx is a library for composing asynchronous and event-based programs using observable collections. DryadLINQ is a programming environment for writing large-scale data parallel applications that run on large PC clusters. Accelerator (not shown above) is a library that uses a data parallelism programming model and targets both GPUs and multi-core CPUs.

Axum, Rx, Accelerator, and DryadLINQ are incubation or research projects and Microsoft has made no commitment to ship them. However, they contain many new ideas and approaches that will interest anyone who has read this far in the book. You are encouraged to download them, evaluate them and provide the respective teams with feedback.

All of the above technologies, with the exception of Axum and DryadLINQ, are largely for parallelism on a single computer. Windows HPC Server 2008 targets clusters of servers and supports scale out across many machines. While the technologies are very different for clustered computing, some of the fundamental patterns discussed in this book, such as Parallel Tasks and Parallel Aggregation, are still applicable.

Further Reading

The MSDN Parallel Computing Developer Center covers parallel development on both the managed and native concurrency runtimes, as well as the Visual Studio 2010 tools that support writing parallel programs. For more information, go to <http://msdn.microsoft.com/concurrency>.

For material on F# including the language reference and walkthroughs, see the Microsoft F# Developer Center at <http://msdn.microsoft.com/fsharp>. F# versions of the samples in this guide are on CodePlex at <http://parallelpatterns.codeplex.com/>.

Windows HPC Server 2008 product information and developer resources are available on the Windows HPC Server site at <http://www.microsoft.com/hpc>.

Details of the Microsoft Research projects Accelerator and DryadLINQ are available at <http://research.microsoft.com/accelerator> and <http://research.microsoft.com/en-us/projects/DryadLINQ> respectively.

Descriptions and downloads for Rx, Axum and CHESS, which are all part of the Microsoft DevLabs project, are available at <http://msdn.microsoft.com/devlabs>.

Glossary

aggregate exception. An exception that contains one or more inner exceptions, rather than just one inner exception.

aggregation. To combine multiple data items into a single result.

alpha blending. Merging different images into a single image by superimposing them in semitransparent layers.

antecedent task. A task whose completion starts a continuation task. An antecedent is often a future whose result value is used as input to the continuation task.

asynchronous. An operation that does not block the current thread of control when the operation is started.

background thread. A thread that stops when a process shuts down. A running background thread does not keep a process running. Threads in the thread pool are background threads. Contrast to foreground thread.

barrier. A synchronization point where all participating threads must stop and wait until every thread reaches it. Supported in .NET by the **System.Threading.Barrier** class.

block. To pause execution, waiting for some event or condition.

blocking collection. A collection where removal attempts block until data is available to be removed, and addition attempts block until space is available. Supported in .NET by the **System.Collections.Concurrent.BlockingCollection<T>** class.

captured variable. A variable defined outside a lambda expression that is used in the lambda expression. The lambda expression can update the captured variable.

cluster. A parallel computing system composed of multiple computers connected by a network, not multiple cores in a single physical processor.

closure. A lambda expression that captures variables.

concurrency. Programming with multiple activities at the same time. Concurrency enables programs to respond promptly to external stimuli; its goal is to reduce latency. Concurrency can be implemented with asynchronous operations or with threads, where it is expected that threads will take turns executing on processors. Compare to parallelism.

Concurrency Visualizer. An addition to the Visual Studio profiler that collects and displays information about the execution and performance of parallel programs.

context switch. When one thread stops executing on a processor and a different thread resumes. Excessive context switching can occur when processors are oversubscribed, and can result in poor performance.

continuation task. A task that automatically starts when other tasks, known as its antecedents, complete.

control flow. A basis for coordination where tasks execute according to the steps of algorithm, as in a parallel loop.

coordination. Arranging for tasks to work together to ensure a correct outcome. Coordination may be based on data flow or control flow.

core. The part of a physical processor that executes instructions. Most recent physical processor models have more than one core, so they can execute tasks in parallel.

coroutine. A generalized subroutine (method) that supports multiple entry points and can return more than once, yielding different return values. In C# iterators are often implemented as coroutines using the **yield** keyword.

data flow. A basis for coordination where tasks execute when data becomes available, as in a pipeline or task graph.

data parallelism. A form of parallel processing where the same computation executes in parallel on different data. Data parallelism is supported in .NET by the **Parallel.For** and **Parallel.ForEach** method, and by PLINQ. Compare to task parallelism.

data partitioning. Dividing a collection of data into parts, in order to use data parallelism.

data race. When more than one concurrent thread read and write data without synchronization.

deadlock. When execution stops and cannot resume, waiting for a condition that cannot occur. Threads can deadlock when each holds resources that another needs.

decomposition. To break a problem into smaller parts. For parallel processing, we can decompose by data or by task.

degree of parallelism. The number of parallel tasks that may execute concurrently at any one time. The degree of parallelism can be set in PLINQ by the `WithDegreeOfParallelism` method, and can be limited in parallel loops by the `MaxDegreeOfParallelism` option.

dependency. When one operation uses the results of another. When there is a dependency between operations, they cannot run in parallel. Contrast to independent.

dependency injection (DI) container. A framework or library for building graphs of objects by passing in (injecting) each object's dependencies. Object lifetimes are handled by the container, rather than the consuming object.

double-checked locking. Testing a condition, then, only if the condition is true, acquiring a lock and testing the same condition again, this time to determine whether to update shared data. This maneuver can often avoid the expensive operation of acquiring the lock in conditions where it will not be useful.

dynamic partitioning. Data partitioning where the parts are selected as the parallel tasks execute. Contrast to static partitioning.

foreground thread. A thread that keeps a process running. When all its foreground threads have stopped, the process shuts down. Contrast to background thread.

fork/join. A parallel computing pattern that uses task parallelism. Fork occurs when tasks are started; join occurs when all tasks are finished.

future. A task that returns a value. In the .NET Framework, futures are implemented with the **Task<TResult>** class.

granularity. The quantity of data in a partition, or work in a task. Equivalently, the number of data partitions, or tasks. A coarse level of granularity has a few large partitions or tasks, a fine level of granularity has many small partitions or tasks.

hardware thread. An execution pipeline on a core. Hyperthreading enables more than one hardware thread to execute on a single core. Each hardware thread is considered a separate logical processor.

Immutable. Data that cannot be modified after it is created. For example, .NET strings are immutable. Compare to mutable.

immutable type. A type whose instances are immutable. Its instances are purely functional data structures.

independent. When one operation does not use the results of another. Independent operations can execute in parallel. Contrast to dependency.

inner exception. An exception thrown and wrapped in another exception.

lambda expression. An anonymous function that can contain expressions or statements.

lazy initialization. Where data is not initialized until it is needed. Lazy initialization is supported in .NET by the **Lazy<T>** and **LazyInitializer** classes.

livelock. When execution continues but does not make progress toward its goal, for example when executing an endless loop, waiting for something to happen.

load balancing. When similar amounts of work are assigned to different tasks, so the available processors are used efficiently. Compare to load imbalance.

load imbalance. When different amounts of work are assigned to different tasks, so some tasks do not have enough work to do, and the available processors are not used efficiently. Compare to load balancing.

lock. A synchronization mechanism that ensures only one thread can execute a particular section of code at a time.

lock convoy. When multiple tasks contend repeatedly for the same lock. Frequent failures to acquire the lock can result in poor performance.

logical processor. The processor associated with a single hardware thread. In .NET, **System.Environment.ProcessorCount** returns the number of logical processors. Compare to physical processor.

manycore. Multicore.

map. A parallel computation where multiple tasks independently perform the same transformation on different data. An example of data parallelism.

mapreduce. A parallel programming pattern where a data parallel phase (map) is followed by an aggregation phase (reduce).

memory barrier. A machine instruction that enforces an ordering constraint on memory operations. Memory operations that precede the barrier are guaranteed to occur before operations that follow the barrier.

multicore. Having more than one core, able to execute parallel tasks. Most recent physical processor models are multicore.

multiplicity. The number of times an element occurs in a multiset.

multiset. An unordered collection that may contain duplicates. Each element in the collection is associated with a multiplicity (or count) that indicates how many times it occurs. Contrast to *set*.

multiset union. An operation that combines multisets by merging their elements and adding the multiplicities of each element.

mutable. Data that can be modified after it is created. For example, .NET arrays are mutable. Compare to immutable.

mutable type. A type whose instances are mutable.

nested parallelism. When one parallel programming construct appears within another. In .NET, when you use a **Parallel.For** loop within another **Parallel.For** loop, they coordinate with each other to share threading resources.

node. A computer in a cluster.

nonblocking algorithm. An algorithm that allows multiple tasks to make progress on a problem without ever blocking each other.

object graph. A data structure consisting of objects that reference each other. Object graphs are often shared mutable data that can complicate parallel programming.

overlapped I/O. I/O operations that proceed (or wait) while other tasks are executing.

oversubscription. When there are more threads than processors available to run them. Oversubscription can result in poor performance, because time is spent context switching.

parallelism. Programming with multiple threads, where it is expected that threads will execute at the same time on multiple processors. Its goal is to increase throughput. Compare to concurrency.

partitioning. Dividing data into parts, in order to use data parallelism.

physical processor. A processor chip, also called a package or socket. Most recent physical processor models have more than one core and more than one logical processor per core.

pipeline. A series of producer/consumers, where each one consumes the output produced by its predecessor.

priority inversion. When a lower priority thread runs while a higher priority thread waits. This can occur when the lower priority thread holds a resource that the higher-priority thread requires.

process. A running application. Processes can run in parallel and are isolated from one another (they usually do not share data). A process can include several (or many) threads or tasks. In .NET, processes can be monitored and controlled with the **System.Diagnostics.Process** class. Compare to thread, task.

profiler. A tool that collects and displays information for performance analysis. The Concurrency Visualizer is a profiler for concurrent and parallel programs.

pure function. A function (or method or operation) that has no side effects (does not update any data nor produce any output), but only returns a value.

purely functional data structure. A data structure that can only be accessed by pure functions. An instance of an immutable type.

race. When the outcome of a computation depends on which statement executes first, but the order of execution is not controlled or synchronized.

race condition. A condition where a race occurs. Race conditions are usually errors; good programming practice prevents them.

recursive decomposition. In parallel programming, where the tasks themselves can start more tasks.

reduce. A kind of aggregation where data are combined by an operation which is associative, which often makes it possible to perform much of the reduction in parallel.

round robin. A scheduling algorithm where each thread is given its turn to run in a fixed order in a repeating cycle, so that during each cycle, each thread runs once.

scalable. A parallel computation whose performance increases when more processors are available.

semaphore. A synchronization mechanism that ensures not more than a specified number of threads can execute a particular section of code at a time. Compare to, lock.

serialize. To run in sequence, not in parallel.

set. An unordered collection without duplicates. Compare to multiset.

shared data. Data used by more than one thread. Access to mutable shared data requires synchronization.

single-threaded data type. A type which is not thread-safe. It cannot be accessed by multiple threads, unless there is additional synchronization in user code.

simultaneous multithreading (SMT). A technique for executing multiple threads on a single core.

socket. Physical processor.

speculative execution. To execute tasks even though their results may not be needed.

static partitioning. Data partitioning where the parts are selected before the program executes. Contrast to dynamic partitioning.

synchronization. Coordinating the actions of threads to ensure a correct outcome. A lock is an example of a synchronization mechanism.

task. A parallelizable unit of work. A task executes in a thread but is not the same as a thread; it is at a higher level of abstraction. Tasks are recommended for parallel programming in .NET, using the Task Parallel Library in the **System.Threading.Tasks** namespace. Compare to thread, process.

task graph. When tasks provide results that are the inputs to other tasks, 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.

task inlining. When more than one task executes in a single thread concurrently due to one task requesting that the other task run synchronously at the current point of execution.

task parallelism. A form of parallel processing where different computations execute in parallel on different data. Task parallelism is supported in .NET by the **Parallel.Invoke** and **Task.Factory.StartNew** methods. Compare to data parallelism.

thread. An executing sequence of statements. Several (or many) threads can run within a single process. Threads are not isolated; all the threads in a process share data. A thread can run a task but is not the same as a task; it is at a lower level of abstraction. Threads are supported in .NET by the **System.Threading** namespace. Compare to process, task.

thread affinity. When certain operations must only be performed by a particular thread. For example, a WPF control must be accessed from the same thread that created it.

thread injection. To add threads to the thread pool.

thread pool. A collection of threads managed by .NET, to avoid the overhead of creating and disposing threads. Tasks are usually run by threads in the thread pool.

thread starvation. When tasks are unable to run because there are no threads available to run them in the thread pool.

thread-local state. Variables that are accessed by only one thread. No locking or other synchronization is needed to safely access thread-local state. Thread-local state is supported in .NET by the **ThreadStatic** attribute and the **ThreadLocal** class.

thread-safe. A type that can be used concurrently by multiple threads, without requiring additional synchronization in user code. A thread-safe type ensures that its data can only be accessed by one thread at a time, and its operations are atomic with respect to multiple threads. The .NET **System.Collections.Concurrent** namespace provides several thread-safe collection types. Compare to single-threaded data type.

torn read. When reading a variable requires more than one machine instruction and another task writes in the variable between the read instructions.

torn write. When writing a variable requires more than one machine instruction, and another task reads the variable between the write instructions.

Tuple. An unnamed record containing an ordered list of elements. Tuples are supported in .NET by the **Tuple** classes.

two-step dance. To signal an event while holding a lock when the waking thread needs to acquire that lock. It will wake only to find that it must wait again. This can cause context switching and poorer performance.

undersubscription. When there are fewer tasks than there are processors available to run them, so processors remain idle.

virtual core. Logical processor.

volatile. A keyword that tells the C# compiler that a field can be modified by multiple threads, the operating system, or other hardware.

work stealing. When a thread executes a task queued for another thread in the thread pool, in order to remain busy.

XAML. A declarative markup language used in the .NET Framework programming model to specify UI for .NET applications.

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. Boodhoo. Model View Presenter. <http://msdn.microsoft.com/en-us/magazine/cc188690.aspx>
- C. Campbell, M. Veanes, J. Huo, and A. Petrenko, Multiplexing of Partially Ordered Events, in *TestCom 2005*, Springer Verlag, June 2005. Available online at <http://research.microsoft.com/apps/pubs/default.aspx?id=77808>.
- 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. Duffy. Immutable types for C#. <http://www.bluebytesoftware.com/blog/2007/11/11/ImmutableTypesForC.aspx>
- M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- J. E. Hoag. A Tour of Various TPL Options. April 2009. <http://blogs.msdn.com/b/pfxteam/archive/2010/04/19/9997552.aspx>
- J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#*, Cambridge University Press, 2008.
- D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In S. Arora and G.T. Leavens, editors, *OOPSLA 2009: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227 - 242. ACM, 2009.
- E. Lippert. Fabulous Adventures in Coding. <http://blogs.msdn.com/b/ericlippert/archive/tags/immutability/>
- M. McCool, Structured Patterns for Parallel Computation, December 2009. <http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=223101515>
- T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.
- C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

J. Smith. WPF Apps With the Model-View-ViewModel Design Pattern. <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

S. Toub. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#, 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>

S. Toub . Custom Parallel Partitioning With .NET 4, April 26, 2010. <http://www.drdobbs.com/visualstudio/224600406>

S. Wildermuth. WPF Threads: Build More Responsive Apps With The Dispatcher. <http://msdn.microsoft.com/en-us/magazine/cc163328.aspx>

Other online sources

Accelerator. <http://research.microsoft.com/en-us/projects/accelerator/>

The ADO.NET Entity Framework Overview. [http://msdn.microsoft.com/en-us/library/aa697427\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx)

Best Practices for using ADO.NET. <http://msdn.microsoft.com/en-us/library/ms971481.aspx>

DevLabs Portal. <http://msdn.microsoft.com/en-us/devlabs/default.aspx>

DryadLINQ. <http://research.microsoft.com/en-us/projects/DryadLINQ/>

Guide to Parallel Programming. <http://parallelpatterns.codeplex.com/>

Guidelines for Overriding Equals() and Operator == (C# Programming Guide). [http://msdn.microsoft.com/en-us/library/ms173147\(VS.90\).aspx](http://msdn.microsoft.com/en-us/library/ms173147(VS.90).aspx)

How to: Create a Task Scheduler That Limits the Degree of Concurrency. <http://msdn.microsoft.com/en-us/library/ee789351.aspx>

Improving ADO.NET Performance. <http://msdn.microsoft.com/en-us/library/ff647768.aspx>

Lazy Initialization. <http://msdn.microsoft.com/en-us/library/dd997286.aspx>

Microsoft F# Developer Center. <http://msdn.microsoft.com/en-us/fsharp/default.aspx>

NModel software. <http://nmodel.codeplex.com/>.

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

Parallel Computing. <http://msdn.microsoft.com/en-us/concurrency/default.aspx>

Parallel Performance Analysis in Visual Studio 2010. <http://blogs.msdn.com/b/visualizeparallel/>

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

Windows Azure Guidance. <http://wag.codeplex.com/>

Performance Tuning with the Concurrency Visualizer in Visual Studio 2010.

<http://msdn.microsoft.com/en-us/magazine/ee336027.aspx>

Scenario Home Page. <http://code.msdn.microsoft.com/scenario>

Standard Query Operators Overview. <http://msdn.microsoft.com/en-us/library/bb397896.aspx>

TaskScheduler Events. http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler_events.aspx

Unity Application Block. <http://msdn.microsoft.com/en-us/library/dd203101.aspx>

Windows HPC Server 2008. <http://www.microsoft.com/hpc/en/us/default.aspx>