Microsoft

PARALLEL PROGRAMMING WITH MICROSOFT VISUAL C++®

Design Patterns for Decomposition and Coordination on Multicore Architectures

Colin Campbell
Ade Miller

Forewords by
Tony Hey
Herb Sutter

patterns & practices

PARALLEL PROGRAMMING WITH MICROSOFT VISUAL C++®

Parallel Programming with Microsoft Visual C++®

Design Patterns for Decomposition and Coordination on Multicore Architectures

---

Colin Campbell
Ade Miller

ISBN 978-0-7356-5175-3

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet website references, may change without notice. You bear the risk of using it. Unless otherwise noted, the companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, email 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.

© 2011 Microsoft Corporation. All rights reserved.

Microsoft, MSDN, Visual Basic, Visual C++, Visual C#, Visual Studio, Windows, Windows Live, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

Contents

FOREWORD xi
Tony Hey

FOREWORD xiii
Herb Sutter

PREFACE xv
- Who This Book Is For xv
- Why This Book Is Pertinent Now xvi
- What You Need to Use the Code xvi
- How to Use This Book xvii
  - Introduction xviii
  - Parallelism with Control Dependencies Only xviii
  - Parallelism with Control and Data Dependencies xviii
  - Dynamic Task Parallelism and Pipelines xviii
  - Supporting Material xix
- What Is Not Covered xx
- Goals xx

ACKNOWLEDGMENTS xxi

1 **Introduction** 1
- The Importance of Potential Parallelism 2
- Decomposition, Coordination, and Scalable Sharing 3
  - Understanding Tasks 3
  - Coordinating Tasks 4
  - Scalable Sharing of Data 5
  - Design Approaches 6
- Selecting the Right Pattern 7
- A Word about Terminology 8
- The Limits of Parallelism 8
- A Few Tips 10
- Exercises 11
- For More Information 11

2 **Parallel Loops** 13
The Basics 14
    Parallel for Loops 14
    parallel_for_each 15
    What to Expect 16
An Example 17
    Sequential Credit Review Example 18
    Credit Review Example Using parallel_for_each 18
    Performance Comparison 19
Variations 19
    Breaking out of Loops Early 19
    Exception Handling 20
    Special Handling of Small Loop Bodies 21
    Controlling the Degree of Parallelism 22
Anti-Patterns 23
    Hidden Loop Body Dependencies 23
    Small Loop Bodies with Few Iterations 23
    Duplicates in the Input Enumeration 23
    Scheduling Interactions with Cooperative Blocking 24
Related Patterns 24
Exercises 24
Further Reading 25

3 **Parallel Tasks** 27
The Basics 28
An Example 29
Variations 31
    Coordinating Tasks with Cooperative Blocking 31
    Canceling a Task Group 33
    Handling Exceptions 35
    Speculative Execution 36
Anti-Patterns 37
    Variables Captured by Closures 37
    Unintended Propagation of Cancellation Requests 38
    The Cost of Synchronization 39
Design Notes 39
    Task Group Calling Conventions 39
    Tasks and Threads 40
    How Tasks Are Scheduled 40
    Structured Task Groups and Task Handles 41
    Lightweight Tasks 41
Exercises 42
Further Reading 42

4 **Parallel Aggregation** 45
The Basics 46
An Example 49
Variations 55
  Considerations for Small Loop Bodies 55
  Other Uses for Combinable Objects 55
Design Notes 55
Related Patterns 57
Exercises 58
Further Reading 58

5 **Futures** 61
The Basics 62
  Futures 63
Example: The Adatum Financial Dashboard 65
  The Business Objects 66
  The Analysis Engine 67
Variations 70
  Canceling Futures 70
  Removing Bottlenecks 70
  Modifying the Graph at Run Time 71
Design Notes 72
  Decomposition into Futures 72
  Functional Style 72
Related Patterns 72
  Pipeline Pattern 73
  Master/Worker Pattern 73
  Dynamic Task Parallelism Pattern 73
  Discrete Event Pattern 73
Exercises 73

6 **Dynamic Task Parallelism** 75
The Basics 75
An Example 77
Variations 80
  Parallel While-Not-Empty 80
  Adding Tasks to a Pending Wait Context 81
Exercises 83
Further Reading 83

7 **Pipelines** 85
Types of Messaging Blocks 86
The Basics 86

An Example 92
- Sequential Image Processing 92
- The Image Pipeline 94
- Performance Characteristics 96

Variations 97
- Asynchronous Pipelines 97
- Canceling a Pipeline 101
- Handling Pipeline Exceptions 102
- Load Balancing Using Multiple Producers 104
- Pipelines and Streams 106

Anti-Patterns 107
- Copying Large Amounts of Data between Pipeline Stages 107
- Pipeline Stages that Are Too Small 107
- Forgetting to Use Message Passing for Isolation 107
- Infinite Waits 107
- Unbounded Queue Growth 107
- More Information 107

Design Notes 108

Related Patterns 109

Exercises 109

Further Reading 109

APPENDICES

A THE TASK SCHEDULER AND RESOURCE MANAGER 111

Resource Manager 113
- Why It's Needed 113
- How Resource Management Works 113
- Dynamic Resource Management 115
- Oversubscribing Cores 116
- Querying the Environment 116

Kinds of Tasks 116
- Lightweight Tasks 117
- Tasks Created Using PPL 117

Task Schedulers 118
- Managing Task Schedulers 118
  - Creating and Attaching a Task Scheduler 119
  - Detaching a Task Scheduler 120
  - Destroying a Task Scheduler 120
  - Scenarios for Using Multiple Task Schedulers 120
  - Implementing a Custom Scheduling Component 121

The Scheduling Algorithm 121
- Schedule Groups 121
- Adding Tasks 122
- Running Tasks 123
- Enhanced Locality Mode 124
- Forward Progress Mode 125
- Task Execution Order 125
- Tasks That Are Run Inline 125

Using Contexts to Communicate with the Scheduler 126
- Debugging Information 126
- Querying for Cancellation 126
- Interface to Cooperative Blocking 127
- Waiting 127
- The Caching Suballocator 127
- Long-Running I/O Tasks 128

Setting Scheduler Policy 128

Anti-Patterns 129
- Multiple Resource Managers 129
- Resource Management Overhead 129
- Unintentional Oversubscription from Inlined Tasks 130
- Deadlock from Thread Starvation 131
- Ignored Process Affinity Mask 131

References 132

B DEBUGGING AND PROFILING PARALLEL APPLICATIONS 133

The Parallel Tasks and Parallel Stacks Windows 133

Breakpoints and Memory Allocation 136

The Concurrency Visualizer 137
- Scenario Markers 141

Visual Patterns 142
- Oversubscription 142
- Lock Contention and Serialization 143
- Load Imbalance 145

Further Reading 147

C TECHNOLOGY OVERVIEW 149

Further Reading 151

GLOSSARY 153

INDEX

Foreword

At its inception some 40 or so years ago, parallel computing was the province of experts who applied it to exotic fields, such as high energy physics, and to engineering applications, such as computational fluid dynamics. We've come a long way since those early days.

This change is being driven by hardware trends. The days of perpetually increasing processor clock speeds are now at an end. Instead, the increased chip densities that Moore's Law predicts are being used to create multicore processors, or single chips with multiple processor cores. Quad-core processors are now common, and this trend will continue, with 10's of cores available on the hardware in the not-too-distant future.

In the last five years, Microsoft has taken advantage of this technological shift to create a variety of parallel implementations. These include the Microsoft® Windows® High Performance Cluster (HPC) technology for message-passing interface (MPI) programs, Dryad, which offers a Map-Reduce style of parallel data processing, the Windows Azure™ technology platform, which can supply compute cores on demand, the Parallel Patterns Library (PPL) and Asynchronous Agents Library for native code, and the parallel extensions of the Microsoft .NET Framework 4.

Multicore computation affects the whole spectrum of applications, from complex scientific and design problems to consumer applications and new human/computer interfaces. We used to joke that "parallel computing is the future, and always will be," but the pessimists have been proven wrong. Parallel computing has at last moved from being a niche technology to being center stage for both application developers and the IT industry.

But, there is a catch. To obtain any speed-up of an application, programmers now have to divide the computational work to make efficient use of the power of multicore processors, a skill that still belongs to experts. Parallel programming presents a massive challenge for the majority of developers, many of whom are encountering it for the first time. There is an urgent need to educate them in practical ways so that they can incorporate parallelism into their applications.

Two possible approaches are popular with some of my computer science colleagues: either design a new parallel programming language, or develop a “heroic” parallelizing compiler. While both are certainly interesting academically, neither has had much success in popularizing and simplifying the task of parallel programming for non-experts. In contrast, a more pragmatic approach is to provide programmers with a library that hides much of parallel programming’s complexity and teach programmers how to use it.

To that end, the Microsoft Visual C++® Parallel Patterns Library and Asynchronous Agents Library present a higher-level programming model than earlier APIs. Programmers can, for example, think in terms of tasks rather than threads, and avoid the complexities of thread management. *Parallel Programming with Microsoft Visual C++* teaches programmers how to use these libraries by putting them in the context of design patterns. As a result, developers can quickly learn to write parallel programs and gain immediate performance benefits.

I believe that this book, with its emphasis on parallel design patterns and an up-to-date programming model, represents an important first step in moving parallel programming into the mainstream.

Tony Hey
Corporate Vice President, Microsoft Research

Foreword

This timely book comes as we navigate a major turning point in our industry: parallel hardware + mobile devices = the pocket supercomputer as the mainstream platform for the next 20 years.

Parallel applications are increasingly needed to exploit all kinds of target hardware. As I write this, getting full computational performance out of most machines—nearly all desktops and laptops, most game consoles, and the newest smartphones—already means harnessing local parallel hardware, mainly in the form of multicore CPU processing; this is the commoditization of the supercomputer. Increasingly in the coming years, getting that full performance will also mean using gradually ever-more-heterogeneous processing, from local general-purpose computation on graphics processing units (GPGPU) flavors to harnessing "often-on" remote parallel computing power in the form of elastic compute clouds; this is the generalization of the heterogeneous cluster in all its NUMA glory, with instantiations ranging from on-die to on-machine to on-cloud, with early examples of each kind already available in the wild.

Starting now and for the foreseeable future, for compute-bound applications, "fast" will be synonymous not just with "parallel," but with "*scalably* parallel." Only scalably parallel applications that can be shipped with lots of latent concurrency beyond what can be exploited in this year's mainstream machines will be able to enjoy the new Free Lunch of getting substantially faster when today's binaries can be installed and blossom on tomorrow's hardware that will have more parallelism.

Visual C++ 2010 with its Parallel Patterns Library (PPL), described in this book, helps enable applications to take the first steps down this new path as it continues to unfold. During the design of PPL, many people did a lot of heavy lifting. For my part, I was glad to be able to contribute the heavy emphasis on lambda functions as the key central language extension that enabled the rest of PPL to be built as Standard Template Library (STL)-like algorithms implemented as a normal library. We could instead have built a half-dozen new kinds of special-purpose parallel loops into the language itself (and almost did), but that would have been terribly invasive and non-general. Adding a single general-purpose language feature like lambdas that can be used everywhere, including with PPL but not limited to only that, is vastly superior to baking special cases into the language.

The good news is that, in large parts of the world, we have as an industry already achieved pervasive computing: the vision of putting a computer on every desk, in every living room, and in everyone's pocket. But now we are in the process of delivering pervasive and even elastic supercomputing: putting a supercomputer on every desk, in every living room, and in everyone's pocket, with both local and non-local resources. In 1984, when I was just finishing high school, the world's fastest computer was a Cray X-MP with four processors, 128MB of RAM, and peak performance of 942MFLOPS—or, put another way, a fraction of the parallelism, memory, and computational power of a 2005 vintage Xbox, never mind modern "phones" and Kinect. We've come a long way, and the pace of change is not only still strong, but still accelerating.

The industry turn to parallelism that has begun with multicore CPUs (for the reasons I outlined a few years ago in my essay "The Free Lunch Is Over") will continue to be accelerated by GPGPU computing, elastic cloud computing, and other new and fundamentally parallel trends that deliver vast amounts of new computational power in forms that will become increasingly available to us through our mainstream programming languages. At Microsoft, we're very happy to be able to be part of delivering this and future generations of tools for mainstream parallel computing across the industry. With PPL in particular, I'm very pleased to see how well the final product has turned out and look forward to seeing its capabilities continue to grow as we re-enable the new Free Lunch applications—scalable parallel applications ready for our next 20 years.

Herb Sutter
Principal Architect, Microsoft
Bellevue, WA, USA
February 2011

Preface

This book describes patterns for parallel programming, with code examples, that use the new parallel programming support in the Microsoft® Visual C++® development system. This support is commonly referred to as the Parallel Patterns Library (PPL). There is also an example of how to use the Asynchronous Agents Library in conjunction with the PPL. You can use the patterns described in this book to improve your application's performance on multicore computers. Adopting the patterns in your code can make your application run faster today and also help 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 native code for the Microsoft Windows® operating system, but the portability of PPL makes this book useful for platforms other than Windows. No prior knowledge of parallel programming techniques is assumed. However, readers need to be familiar with features of the C++ environment such as templates, the Standard Template Library (STL) and lambda expressions (which are new to Visual C++ in the Microsoft Visual Studio® 2010 development system). 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 Parallel Patterns Library (PPL).*
>
> *Complete code solutions are posted on CodePlex. See http://parallelpatternscpp.codeplex.com/.*
>
> *There is also a companion volume to this guide,* ***Parallel Programming with Microsoft .NET****, which presents the same patterns in the context of managed code.*

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 Parallel Patterns Library and Asynchronous Agents Library are for C++ programmers who want to write parallel programs. They simplify the process of adding parallelism and concurrency to applications. PPL dynamically scales the degree of parallelism to most efficiently use all the processors that are available. In addition, PPL and agents assist in the partitioning of work and the scheduling of tasks in threads. The library provides cancellation support, state management, and other services. These libraries make use of the Concurrency Runtime, which is part of the Visual C++ platform.

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.

For a complete overview of the parallel technologies available from Microsoft, see Appendix C, "Technology Overview."

What You Need to Use the Code

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

- Microsoft Windows Vista® SP1, Windows 7, Windows Server® 2008, or Windows XP SP3 (32-bit or 64-bit) operating system.
- Microsoft Visual Studio 2010 SP1 (Ultimate or Premium edition is required for the Concurrency Visualizer, which allows you to analyze the performance of your application); this includes the PPL, which is required to run the samples and the Asynchronous Agents Library.

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.

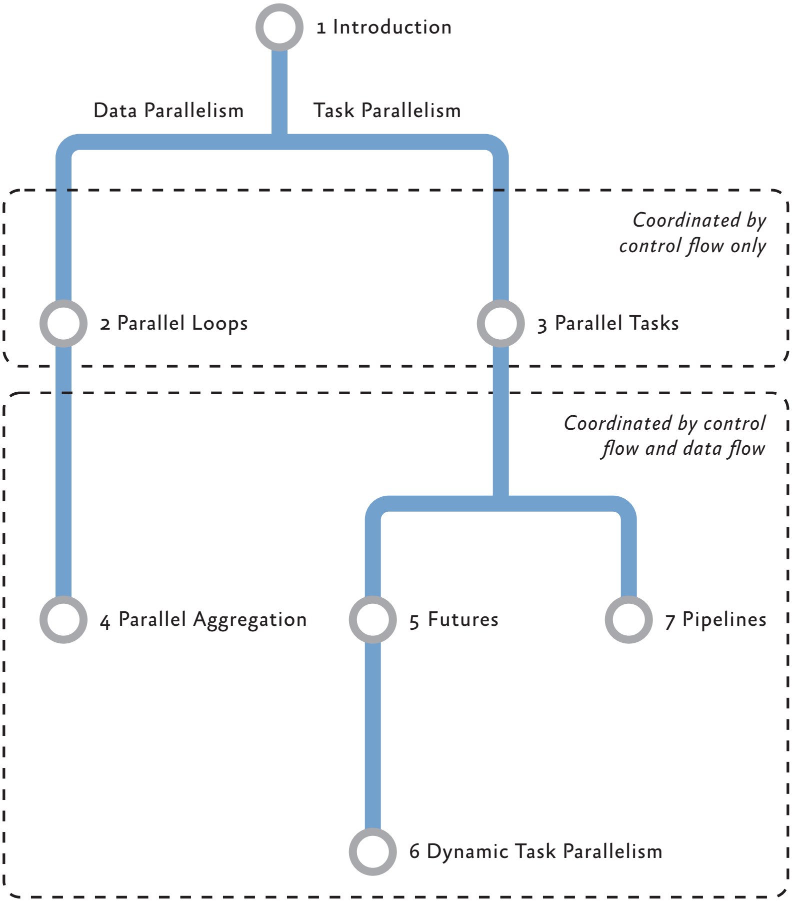

1 Introduction
Data Parallelism
Task Parallelism
Coordinated by control flow only
2 Parallel Loops
3 Parallel Tasks
Coordinated by control flow and data flow
4 Parallel Aggregation
5 Futures
7 Pipelines
6 Dynamic Task Parallelism

FIGURE 1
Parallel programming patterns

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, "Introduction," 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 a pipeline 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, "The Task Scheduler and Resource Manager."** This appendix gives an overview of how the Concurrency Runtime's task scheduler and resource manager function.
- **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 describes the various Microsoft technologies and frameworks for parallel programming.
- **Glossary**. The glossary contains definitions of the terms used in this book.

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

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

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.

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.

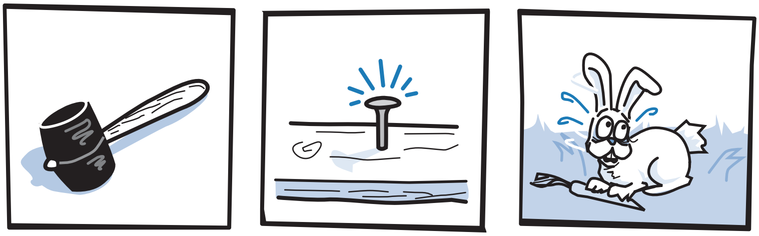


FIGURE 2
When all you have is a hammer, *everything* looks like a nail.

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").

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. Although 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.

This book depends heavily on the work we did in *Parallel Programming with Microsoft .NET*. While much of the text in the current book has changed, it discusses the same fundamental patterns. Because of this shared history, we'd like to again thank the co-authors of the first book: Ralph Johnson (University of Illinois at Urbana Champaign) Stephen Toub (Microsoft), and the following reviewers who provided feedback on the entire text: Nicholas Chen, DannyDig, 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. Their contributions shaped the .NET book and their influence is still apparent in *Parallel Programming with Microsoft Visual C++*.

Once we understood how to implement the patterns in C++, our biggest challenge was to ensure technical accuracy. We relied on members of the Parallel Computing Platform (PCP) team at Microsoft to provide information about the Parallel Patterns Library and the Asynchronous Agents Library, and to review both the text and the accompanying samples. Dana Groff, Niklas Gustafsson and Rick Molloy (Microsoft) devoted many hours to the initial interviews we conducted, as well as to the reviews. Several other members of the PCP team also gave us a great deal of their time. They are: Genevieve Fernandes, Bill Messmer, Artur Laksberg, and Ayman Shoukry (Microsoft).

In addition to the content about the two libraries, the book and samples also contain material on related topics. We were fortunate to have access to members of the Visual Studio teams responsible for these areas. Drake Campbell, Sasha Dadiomov, and Daniel Moth (Microsoft) provided feedback on the debugger and profiler described in Appendix B. Pat Brenner and Stephan T. Lavavej (Microsoft) reviewed the code samples and our use of the Microsoft Foundation Classes and the Standard Template Library.

We would also like to thank, once again, Reed Copsey, Jr. (C Tech Development Corporation), Samira Tasharofi (University of Illinois at Urbana Champaign), and Paul Petersen (Intel) for their reviews of individual chapters. As with the first book, our schedule was aggressive, but the reviewers worked extra hard to help us meet it. Thank you, everyone.

There were a great many people who spoke to us about the book and provided feedback. They include the attendees at the Intel and Microsoft Parallelism Techdays (Bellevue), as well as contributors to discussions on the book's CodePlex site.

A team of technical writers and editors worked to make the prose readable and interesting. They include Roberta Leibovitz (Modeled Computation LLC), Nancy Michell (Content Masters LTD), and RoAnn Corbisier (Microsoft).

Rick Carr (DCB Software Testing, Inc) tested the samples and content.

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 (Modeled Computation LLC).

1

Introduction

Your CPU meter shows a 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. Is there a way to get better performance?

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.

Of course, 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 will face in writing your own parallel programs. Fortunately, help has arrived. The Parallel Patterns Library (PPL) and the Asynchronous Agents Library introduce a new programming model for parallelism that significantly simplifies the job. Behind the scenes are sophisticated algorithms that dynamically distribute computations on multicore architectures. In addition, Microsoft® Visual Studio® 2010 development system includes debugging and analysis tools to support the new parallel programming model.

Attention impatient readers: you can skip ahead to the table of patterns and when they can be used. See "Selecting the Right Pattern" later in this chapter.

Proven design patterns are another source of help. This guide introduces you to the most important and frequently used patterns of parallel programming and provides executable code samples for them, using PPL. 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, delve more deeply into the relevant pattern or patterns and study the sample code.

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

Most parallel programs conform to these patterns, and it's very likely you'll be successful in finding a match to your particular problem. If you can't use these patterns, you've 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://parallelpatternscpp.codeplex.com/.

The Importance of Potential Parallelism

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

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 PPL's programming model. It deserves some explanation.

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 process control. The life cycle of such programs matches the life cycle of the specific hardware they were designed to use.

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.

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.

Even if you 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 on newer hardware are not increasing as much as they did 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 computers 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 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 because 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 consider the heuristics that will be used for task scheduling.

Tasks are sequential units of work. Tasks should be large, independent, and numerous enough to keep all cores busy.

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.

An example of these guidelines at work can be seen in 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 duration, you generally want more of them in order to fill in the gaps.

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.

Another advantage to grouping work into larger and fewer tasks is that larger tasks are often more independent of each other than are smaller tasks. Larger tasks are less likely than smaller tasks to share local variables or fields. 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 a sufficient number of 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.

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. The way 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 messages 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 memory location. 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, sending messages instead of updating 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.

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. Deadlock occurs when two or more tasks are 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

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

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, such as those described in the previous section. As a result, traditional profile-and-optimize techniques 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.

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 Pipeline 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 will give 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 speed-up you can ever achieve is (1 / fraction of time spent in sequential processing). Figure 1 illustrates this.

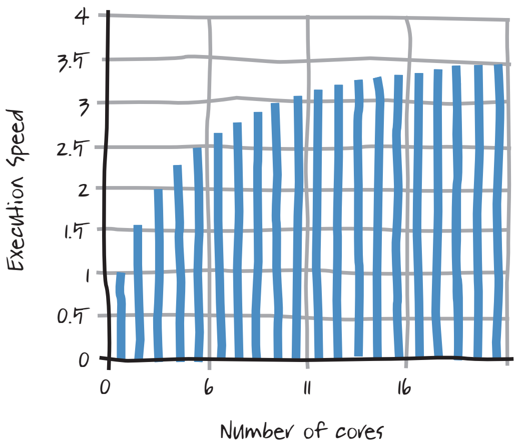


FIGURE 1
Amdahl's law for an application with 25 percent sequential processing

For example, with 11 cores, 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 speed-up is not linear. Figure 2 illustrates this.

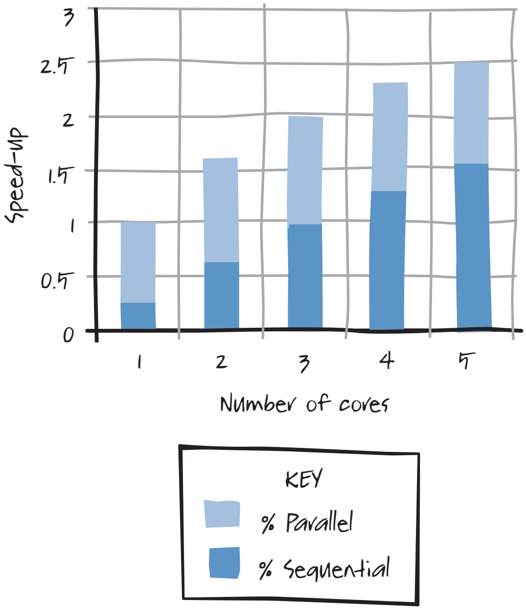


FIGURE 2
Per-core performance improvement for a 25 percent sequential application

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 illustration also shows why you might be satisfied with a 2x speed-up 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 speed-up 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 speed-up. 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 the highest possible level of abstraction 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 the Parallel Patterns Library. These libraries were written by experts and have been thoroughly tested; they help you 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 produce some improvement, 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 and decomposing it into larger tasks?
2. What is the maximum potential speed-up 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 this book.

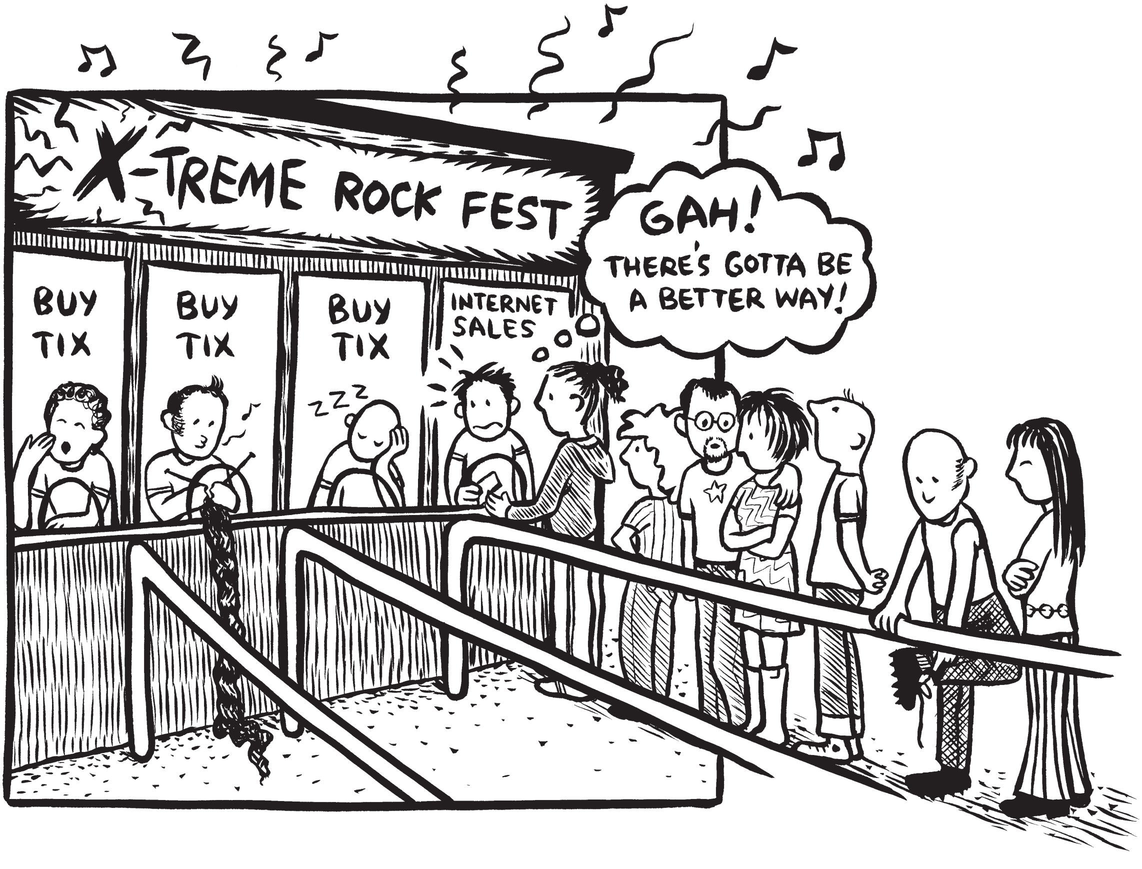
The design patterns presented in this book are consistent with classifications of parallel patterns developed by groups in both industry and academia. In the terminology of these groups, the patterns in this book would be considered to be algorithm or implementation patterns. Classification approaches 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 Microsoft Windows® platform, see the book by Duffy.

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

Mattson, Timothy G., Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

OPL, Our Pattern Language for Parallel Programming ver2.0, 2010. http://parlab.eecs.berkeley.edu/wiki/patterns.


X-TREME ROCK FEST
BUY TIX
BUY TIX
BUY TIX
INTERNET SALES
ZZZ
GAH! THERE'S GOTTA BE A BETTER WAY!

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 **for_each** loops you already know, but the parallel loop completes 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 the Parallel Loop pattern, and one that this chapter explains in detail.

The Parallel Loop pattern independently applies an operation to multiple data elements. It's 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 and assuming the work performed by each iteration is not too small, then 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 Parallel Patterns Library (PPL) includes both parallel **for** and parallel **for_each** loops. Use the **parallel_for** function to iterate over a range of integer indices and the **parallel_for_each** function to iterate over user-provided values.

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

Parallel for Loops

Here's an example of a sequential **for** loop in C++.

```
vector<double> results = ...
int workload = ...
size_t n = results.size();
for (size_t i = 0; i < n; ++i)
{
  results[i] = DoWork(i, workLoad);
}
```

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.

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

```
vector<double> results = ...
int workload = ...
size_t n = results.size();

parallel_for(0u, n,
  [&results, workLoad](size_t i)
  {
    results[i] = DoWork(i, workLoad);
  });
```

The **parallel_for** function uses multiple cores if they're available to operate over the index range.

The example includes a lambda expression in the form **[***captured variables***]** **(***args***)** **{***body***}**. You may need to brush up on the syntax of lambda expressions in C++ before reading further.

The **parallel_for** function has overloaded versions. Here's the signature of the version of **parallel_for** that's used in the example.

```
template <typename _Index_type, typename _Function>
void parallel_for(_Index_type _First,
                  _Index_type _Last,
                  const _Function& _Func);
```

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 a function that's invoked once per iteration. The function takes the iteration's index as its argument and executes the loop body once for each index.

The **parallel_for** method has an additional overloaded version. It is covered in the "Variations" section later in this chapter.

The example includes a lambda expression in the form [*captured variables*] *(args) {body}* as the third argument to the **parallel_for** invocation. Lambda expressions denote function objects that can capture variables from their enclosing scope. Of course, the **_Func** parameter could also be a pointer to a function declared elsewhere. You don't have to use lambda expressions.

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.

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_FOR_EACH

Here's an example of a sequential **for_each** loop in C++ that uses the conventions of the Standard Template Library (STL).

```
vector<size_t> inputs = ...
int workload = ...

for_each(inputs.cbegin(), inputs.cend(),
  [workLoad](size_t i)
  {
    DoWork(i, workLoad);
  });
```

To take advantage of multiple cores, replace the **for_each** keyword with a call to the **parallel_for_each** method.

```
vector<size_t> inputs = ...
int workload = ...

parallel_for_each(inputs.cbegin(), inputs.cend(),
  [workLoad](size_t i)
  {
    DoWork(i, workLoad);
  });
```

parallel_for_each *runs the loop body for each element in a collection.*

The **parallel_for_each** function is very similar in syntax to the **std::for_each** function. The first argument is an iterator that references the position of the first element in the range to be operated on. The second argument is an iterator that references the position one past the final element in the range. The third argument is a function object that's invoked for each element of the input range.

The **parallel_for_each** method does not guarantee the order of execution. Unlike a sequential **for_each** loop, the incoming values aren't 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's passed to it.

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. (See Chapter 1 for a discussion of Amdahl's Law.)

Adding cores makes your loop complete 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.

If an exception is thrown during the execution of one of the iterations of a **parallel_for** or **parallel_for_each** function, that exception will be rethrown in the context of the calling thread. 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.

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.

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

- **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 passed by reference or by pointer result in shared variables, as do variables captured by reference in a lambda expression.

- **Using data accessors of an object model**. If the object being processed by a loop body exposes data accessors, you need to know whether they refer to shared state or state that's local to the object itself. For example, an accessor method named **GetParent** is likely to refer to global state. Here's an example.

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

  In this example, it's likely that the loop iterations are not independent. It's possible that, for all values of **i**, **SomeObject[i].GetParent()** is a reference to a single shared object.

You must be extremely cautious when getting data from accessors. Large object models are known for sharing mutable state in unbelievably convoluted ways.

- **Referencing data types or functions that are not thread safe**. If the body of the parallel loop uses a data type or function that is not thread safe, the loop body is not independent because there's an implicit dependency on the thread context.

- **Loop-carried dependence**. If the body of a **parallel_for** loop performs arithmetic on a loop-indexed variable, 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's no guarantee that the loop body that updates **data[i – 1]** has executed before the loop body 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 the scope of this book. Your best options are 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 **for_each** loop with a parallel loop.

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

Sequential Credit Review Example

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

```
void UpdatePredictionsSequential(AccountRepository& accounts)
{
  for_each(accounts.begin(), accounts.end(),
  [](AccountRepository::value_type& record)
  {
    Account& account = record.second;
    Trend trend = Fit(account.Balances());
    double prediction = PredictIntercept(trend,
         (account.Balances().size() + g_predictionWindow));
    account.SeqPrediction() = prediction;
    account.SeqWarning() = prediction < account.GetOverdraft();
   });
}
```

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_for_each

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

```
void UpdatePredictionsParallel(AccountRepository& accounts)
{
  parallel_for_each(accounts.begin(), accounts.end(),
    []
    (AccountRepository::value_type& record)
    {
      Account& account = record.second;
      Trend trend = Fit(account.Balances());
      double prediction = PredictIntercept(trend,
        (account.Balances().size() + g_predictionWindow));
      account.ParPrediction() = prediction;
      account.ParWarning() = prediction < account.GetOverdraft();
  });
}
```

The **UpdatePredictionsParallel** method is identical to the **UpdatePredictionsSequential** method, except that the **parallel_for_each** function replaces the **for_each** operator.

PERFORMANCE COMPARISON

Running the credit review example on a quad-core computer shows that the **parallel_for_each** version runs 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. However, you can break out of a parallel loop by canceling the task group that contains it. Task groups are described in Chapter 3, "Parallel Tasks."

Use the **task_group::cancel** *method to break out of a parallel loop early.*

Here's an example of how to break out of a parallel loop early.

```
vector<double> results = ...
int workLoad = ...
task_group tg;
size_t fillTo = results.size() - 5 ;
fill(results.begin(), results.end(), -1.0);

task_group_status status = tg.run_and_wait([&]
  {
     parallel_for(0u, results.size(), [&](size_t i)
     {
       if (i > fillTo)
         tg.cancel();
       else
         results[i] = DoWork(i, workLoad);
     });
   });
```

The example code shows that if you want to break out of a parallel loop early, you need to create a task group object and execute the parallel loop within that task group. When you want to break out of the loop, you invoke the task group's **cancel** method.

Don't forget that parallel loops can execute steps out of order. Canceling a parallel loop doesn't ensure that iterations with higher-valued indices won't run.

You should keep in mind that parallel loops may execute steps out of order. Unlike breaking from a sequential loop, canceling the task group of a parallel loop cannot guarantee that higher-indexed iterations won't have had a chance to run before the **cancel** operation takes effect.

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.

Throwing an unhandled exception prevents new iterations from starting.

Because the loop runs in parallel, there may be more than one exception. If more than one exception has occurred, the parallel loop will nondeterministically choose one of the exceptions to throw. The remaining exceptions will not be externally observable.

Here's an example of how to handle exceptions from a parallel loop.

```
vector<double> results = ...

try
{
  size_t n = results.size();
  parallel_for(0u, n, [&results](size_t i)
  {
    results[i] = DoWork(i, 10); // throws exception
  });
}
catch (ParallelForExampleException e)
{
  printf("Exception caught as expected.\n");
}
```

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 the function object. In most situations, these costs are negligible, but with very small loop bodies they can be significant.

Consider using partitioning strategies when you have many iterations that each perform a small amount of work.

An overloaded version of **parallel_for** allows you to specify a step size for the indices. Iterating with step sizes greater than one lets you embed a sequential loop within your parallel loop. Each iteration of the outer (parallel) loop 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's an example.

```
size_t size = results.size();
size_t rangeSize = size / (GetProcessorCount() * 10);
rangeSize = max(1, rangeSize);

parallel_for(0u, size, rangeSize,
  [&results, size, rangeSize, workLoad](size_t i)
  {
    for (size_t j = 0; (j < rangeSize) && (i + j < size); ++j)
        results[i + j] = DoWork(i + j, workLoad);
  });
```

Partitioning your data into ranges results in more complicated application logic than using an ordinary **parallel_for** function without partitioning. When the amount of work in each iteration is large (or of uneven size across iterations), partitioning 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 you use will normally depend on the number of cores in your computer. A good default number of ranges is approximately three to ten times the number of cores.

Another approach for handling smaller loop bodies is to use the **parallel_for_fixed** or **parallel_for_each_fixed** functions that are provided in the Concurrency Runtime sample pack. By default, the **parallel_for** and **parallel_for_each** functions perform dynamic load balancing. When the amount of work for each item is small, the cost of load balancing can become significant. The sample pack's **parallel_for_fixed** and **parallel_for_each_fixed** functions do not perform load balancing so they may outperform **parallel_for** and **parallel_for_each** when the loop bodies are small.

Another difference is that the **parallel_for** and **parallel_for_each** functions check for cancellation with each iteration. In contrast, the **parallel_for_fixed** and **parallel_for_each_fixed** functions do not check for cancellation within their subranges.

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

You'll see this variation of the Parallel Loop pattern in a variety of circumstances. Reducing the degree of parallelism is often done 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.

The term *degree of parallelism* refers to the number of cores that are used to process iterations simultaneously. The degree of parallelism is automatically managed by the underlying components of the system. The implementation of the **parallel_for** and **parallel_for_each** functions, the Concurrency Runtime's task scheduler, and the operating system's thread scheduler all play a role in optimizing throughput under a wide range of conditions. You can't control the degree of parallelism directly, but you can influence it by controlling the number of threads that are simultaneously executed by a parallel loop. To do this, you need to set the **MinConcurrency** and **Max Concurrency** policies of the **SchedulerPolicy** class. For more information about setting these policies, see Appendix A, "The Task Scheduler and Resource Manager."

Anti-Patterns

Anti-patterns are cautionary tales. They highlight issues that need to be carefully considered as well as problem areas. Here are some issues to think about when you implement a parallel loop.

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 which is not thread safe across parallel iterations is an example of a subtle dependency. You should also be careful when you share state by using reference variables from the enclosing lexical scope in a lambda expression.

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 probably won't 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.

DUPLICATES IN THE INPUT ENUMERATION

If you're using the **parallel_for_each** function, duplicate references or pointers to objects in the enumeration often indicate an unsafe race condition in your code. If an object reference (that is, a pointer or reference to 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.

SCHEDULING INTERACTIONS WITH COOPERATIVE BLOCKING

If you perform a cooperative blocking operation in every iteration of a parallel loop, the task scheduler may create more threads than you intend. Cooperative blocking operations should be performed infrequently within the parallel loop.

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 that's 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 two solutions, using a sequential loop and a parallel loop.
3. Do a performance analysis of the credit review example code on the CodePlex site http://parallelpatternscpp.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).

Further Reading

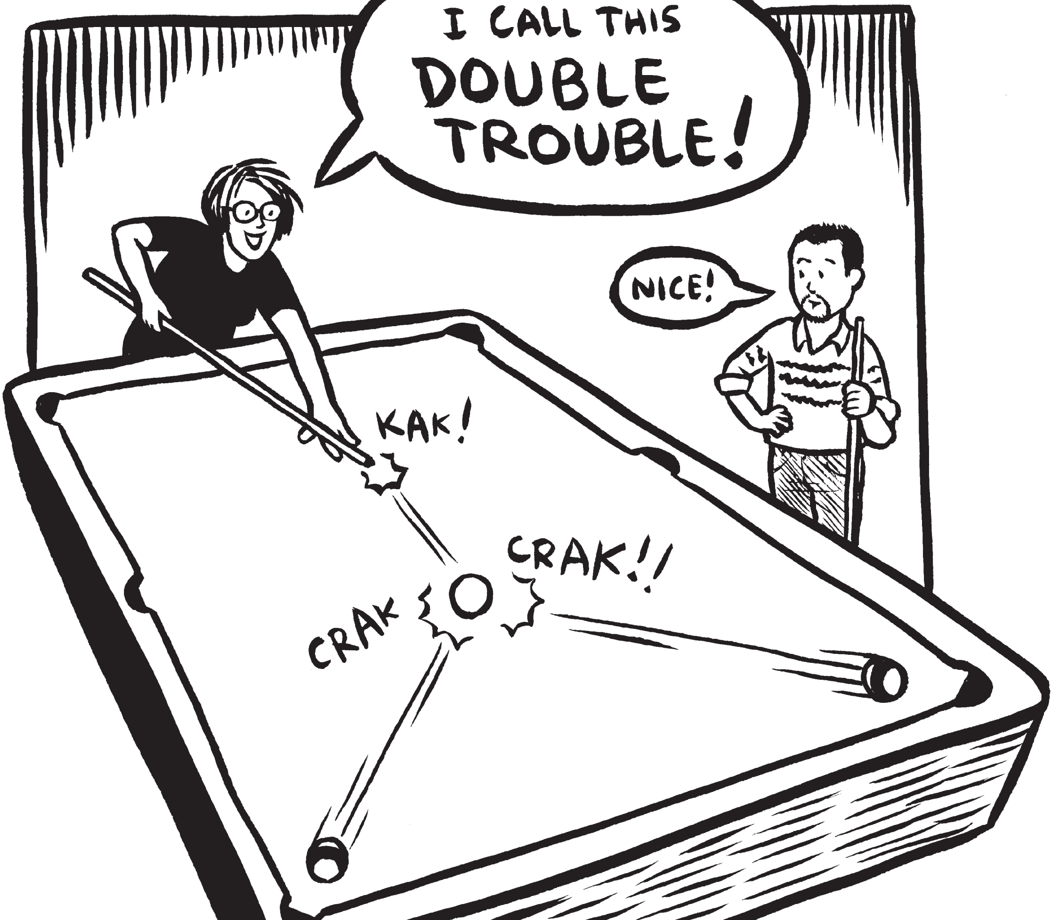
The examples in this book use features and libraries of Microsoft® Visual C++®. MSDN® is the recommended source for reference information about these features and libraries, including lambda expressions. The book by Mattson, et al. describes software design patterns for parallel programming that are not specialized for a particular language or library. Messmer's article gives a number of related patterns and tips for parallel loops in PPL.

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

Mattson, T.G., "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/.

Messmer, B., Parallel Patterns Library, Asynchronous Agents Library, & Concurrency Runtime: Patterns and Practices, 2010. http://www.microsoft.com/downloads/en/confirmation.aspx?displaylang=en&FamilyID=0e70b21e-3f10-4635-9af2-e2f7bdd-ba4ae.

MSDN, Lambda Expressions in C++, http://msdn.microsoft.com/en-us/library/dd293608.aspx.


I CALL THIS DOUBLE TROUBLE!
NICE!
KAK!
CRAK!!
CRAK

3

Parallel Tasks

Chapter 2, "Parallel Loops," shows how you can use a parallel loop to apply a single operation to many data elements. This is data parallelism. Chapter 3 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 known as 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 known as 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 Parallel Patterns Library (PPL), tasks are started and managed by methods of the **task_group** class, which is declared in the ppl.h header file. The task group class's **run** method creates and schedules new tasks. You can wait for all tasks created by the task group to complete by invoking the task group's **wait** method. If you think in terms of fork/join, the **run** method is the fork operation and the **wait** method is the join operation.

Parallel tasks in PPL are managed by the **task_group** *class.*

Scheduling is an important aspect of parallel tasks. Unlike threads, new tasks don't necessarily begin to execute immediately. Instead, they are placed in a work queue. Tasks run when their associated task scheduler removes them from the queue, usually as processor resources become available. 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 PPL, 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 the **task_group**::**wait** method. At that time, the exception is

Tasks in PPL defer exceptions and rethrow them when the task group's **wait** *method is invoked.*

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.

The Basics

Each task is a sequential operation; however, tasks can often run in parallel. Here's 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 **parallel_invoke** function of the **Concurrency** namespace to run them in parallel. This is shown in the following code.

```
parallel_invoke(
  []() { DoLeft(); },
  []() { DoRight(); }
);
```

The **parallel_invoke** *function in the* **Concurrency** *namespace creates a group of parallel tasks and waits for them all to complete.*

The **parallel_invoke** function is the simplest expression of the Parallel Tasks pattern. The function creates a new task group with new parallel tasks for each lambda expression in its argument list. The **parallel_invoke** function returns when all the tasks are finished. The arguments to **parallel_invoke** are known as work functions.

There are overloaded versions of the **parallel_invoke** function that accept up to nine work functions.

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. For more information about how tasks are scheduled see "How Tasks Are Scheduled," later in this chapter.

The functions run by **parallel_invoke** can either complete normally or finish by throwing an exception. If an exception is thrown by one of the work functions during the execution of **parallel_invoke**, it will be deferred and rethrown when all tasks finish. If more than one of the work functions throws an exception, the runtime chooses one of the exceptions to be rethrown. The remaining exceptions will not be externally observed. For more information and a code example, see the section, "Handling Exceptions," later in this chapter.

Internally, **parallel_invoke** creates new tasks and waits for them. You can reproduce this functionality by creating a task group object and calling its **run** and **wait** methods. Here's an example.

```
task_group tg;

tg.run([](){ DoLeft(); });
tg.run([](){ DoRight(); });
tg.wait();
```

The **run** method of the **task_group** class creates and schedules a new task. The run method's argument is a lambda expression, a pointer to function, or a function object that will be invoked when the task eventually executes. In other words, the argument can be any object that supports the function call operator with the signature **void operator()()**. When you use the **task_group::run** method to create a task, the new task is added to a work queue for eventual execution, but it does not start to execute until its task scheduler takes it out of the work queue, which can happen immediately or can occur at some point in the future.

Use the task group's **run** *method to create a task and schedule its execution. Use the* **wait** *method to block the current context until all of the tasks in a task group have completed.*

If you pass a reference to an instance of a class that supports **operator()** as an argument to the **task_group::run** method, you must make sure to manage the memory of the function object. The function object can safely be destroyed only after the task group object's **wait** method returns. Lambda expressions and pointers to static functions do not require explicit deletion and are therefore easier to use than class-type functors.

You can wait for the tasks of the task group to complete by calling the task group's **wait** method.

It is also possible to combine the run and wait steps into a single operation. This is shown in the following code.

```
task_group tg;

tg.run([](){ DoLeft(); });
tg.run_and_wait([](){ DoRight(); });
```

Calling the task group's **run_and_wait** method instead of the **run** method followed by a call to the **wait** method can result in slightly more efficient use of threads. The **run_and_wait** method acts as a hint to the task scheduler that it can reuse the current context to execute the new task.

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

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 known as *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's the sequential code. The source code for the complete example is located at http://parallelpatternscpp.codeplex.com in the Chapter3\ImageBlender folder.

```
static void SequentialImageProcessing(
     Bitmap* const source1, Bitmap* const source2,
     Bitmap* const layer1, Bitmap* const layer2,
     Graphics* const blender)
{
  SetToGray(source1, layer1);
  Rotate(source2, layer2);
  Blend(layer1, layer2, blender);
}
```

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 platform's **Gdiplus** namespace to perform the image processing.

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_invoke** function creates tasks and waits for them to complete before proceeding. This is shown in the following code.

```
static void ParallelInvokeImageProcessing(
     Bitmap* const source1, Bitmap* const source2,
     Bitmap* layer1, Bitmap* layer2, Graphics* blender)
{
  parallel_invoke(
    [&source1, &layer1](){ SetToGray(source1, layer1); },
    [&source2, &layer2](){ Rotate(source2, layer2);}
  );
  Blend(layer1, layer2, blender);
}
```

Use the **parallel_invoke** *function whenever tasks can be defined in a single lexical scope.*

In this example, the tasks are identified implicitly by the arguments to **parallel_invoke**. This call does not return until all of the tasks complete.

You can also create parallel tasks explicitly. This is shown in the following code.

```
static void ParallelTaskGroupImageProcessing(
    Bitmap* const source1, Bitmap* const source2,
    Bitmap* layer1, Bitmap* layer2, Graphics* blender)
{
  task_group tasks;
  tasks.run(
     [&source1, &layer1](){ SetToGray(source1, layer1);}
  );
  tasks.run_and_wait(
     [&source2, &layer2](){ Rotate(source2, layer2); }
  );
  Blend(layer1, layer2, blender);
}
```

This code allocates a task group on the stack. It then calls task group run methods to create and run two tasks that execute **SetToGray** and **Rotate**. The example uses the **run_and_wait** method to create the second task and to wait for all tasks to finish before blending the processed images.

Variations

This section describes variations of PPL's implementation of the Parallel Task pattern.

Coordinating Tasks with Cooperative Blocking

The classes and functions in the **Concurrency** namespace implement a task-coordination feature known as cooperative blocking. With cooperative blocking, your task can suspend its execution and relinquish control to the task scheduler until a specific condition is met. This usually occurs when another task performs an action that the first task needs. A typical example of a cooperative blocking operation is the **task_group::wait** method. If **wait** is invoked from within a running task, the task scheduler knows that the current task can't continue until all the tasks of the specified task group run to completion. Cooperative blocking provides a robust and highly programmable way to coordinate tasks.

Cooperative blocking provides a robust and highly programmable way to coordinate tasks.

Cooperative blocking can improve the performance of a parallel application by enabling fuller use of processor resources. A cooperatively blocked task represents an opportunity for the task scheduler to apply processor resources to other tasks. If you are going to use PPL effectively, it is important to understand the interaction of coop-

Invoking cooperative blocking acts as a hint to the scheduler that other work may be started or resumed.

Cooperative blocking can improve the performance of a parallel application by enabling fuller use of processor resources.

erative blocking with the task scheduler. For more information, see the "Task Scheduler" section of Appendix A, "The Task Scheduler and Resource Manager."

You can also invoke any of the synchronization features of the operating system from within tasks. Using lower-level blocking operations of the operating system is sometimes called noncooperative blocking. If you do any type of blocking, you will need to consider whether cooperative or noncooperative blocking is most appropriate. In general, cooperative blocking has the advantage of better coordination with the task scheduler.

The following table lists the most important operations that the runtime considers to be cooperative blocking operations. Note that these operations are not guaranteed to block every time they are called; instead, they have the *potential* to block if certain conditions exist. All of the classes are in the **Concurrency** namespace.

Cooperative blocking operation	Description
task_group::wait method	This method blocks the current task until the tasks of another task group have completed their work.
critical_section::lock method	A critical section provides mutual exclusion. Acquiring the lock may block the current task if the lock is in use by another task. Only one task at a time can possess the lock.
critical_section:: scoped_lock class constructor	An exception-safe way to acquire and release a critical section within a block of code is to define a **critical_ section ::scoped_lock** at the beginning of that block.
reader_writer_lock:: lock method	This method acquires a reader/writer lock for concurrency-safe modification of shared data. Calling the lock method will block the current task if a reader or reader/writer lock is already held by another thread.
reader_writer_lock:: scoped_lock class constructor	An exception-safe way to acquire and release a reader/writer lock within a block of code is to define a **reader_writer_lock:: scoped_lock** at the beginning of that block.
reader_writer_lock:: lock_read method	This method acquires a reader lock for concurrency-safe reading of shared data. The **lock_read** method may block the current thread if another thread holds the reader/writer lock. Unlike the reader/writer lock, more than one task may hold a reader lock at the ame time.
reader_writer_lock:: scoped_lock_read class constructor	An exception-safe way to acquire and release a reader lock within a block of code is to define a **reader_ writer_lock:: scoped_lock_read** at the beginning of that block.

event::wait method	This method is a cooperatively blocking equivalent of a manual reset event. Invoking the **event::wait** method may block if the event has not yet been set.
agent::wait method **agent::wait_for_*** methods	These methods block the current task until the agent instance completes its work.
wait(...) function	The **Concurrency::wait** function cooperatively blocks execution for a specified time interval.
Context::Block method	This method suspends the current context until it is cooperatively reenabled by another task's invocation of the **Context::Unblock** method. This operation is used by libraries to implement new task-coordination control structures. It is not normally used by application code.
Context::Yield method	This method suspends the current thread so that another worker thread may be given the opportunity to resume execution. Although **Yield** potentially suspends execution of the current thread, it never results in a blocked context. Therefore, **Yield** can only loosely be considered a blocking operation.
parallel_for, parallel_for_each and parallel_invoke functions	PPL's functions for parallel algorithms internally invoke blocking operations, such as the **wait** method.
send and **asend** functions	These functions transmit data to messaging blocks. In Microsoft® Visual Studio® 2010 SP1, the implementations of the **send** and **asend** functions sometimes invoke cooperative blocking, depending on certain internal system details. The **asend** function is expected to be nonblocking in future versions of the runtime.
receive function	This function gets a value from a messaging block. It may block if the messaging block does not yet have a value to provide.

CANCELING A TASK GROUP

You can signal cancellation by invoking the task group's **cancel** method. Here's an example:

Use the **task_group::cancel** *method to signal cancellation of all tasks in a task group.*

```
task_group tg;
tg.run([](){ DoLeft(); });
tg.cancel();                    // could be called from any thread
wcout << L"  Is canceling: " << tg.is_canceling() << endl;

task_group_status status = tg.wait();
wcout << L"  Status:    " << status << endl;
```

Invoking a task group's **cancel** method causes the task group to transition to a state where its **is_canceling** method will return **true**. Tasks in the task group that have not yet started are not allowed to run. New tasks that are added to the task group by the **run** method are ignored after the **cancel** method has been called.

Tasks in the task group that have started before cancellation is signaled continue to run, but their behavior may change. If a task of a task group that is being canceled invokes any function in the **Concurrency** namespace, an exception may be thrown. For example, if a running task of a task group that is being canceled makes a call to another task group's **wait** method, an exception may be thrown by the runtime. The specific set of functions in the **Concurrency** namespace that will throw exceptions when a task group is undergoing cancellation and the specific set of circumstances that will cause such exceptions to be thrown are intentionally not specified. Therefore, your application logic should not depend on whether any particular library function throws or does not throw an exception when a task group is being canceled.

You must ensure that your task work functions are exception safe. Use the Resource Acquisition is Initialization (RAII) pattern for automatic cleanup.

In the current version of PPL, canceling a task group with cooperatively (or noncooperatively) blocked tasks may result in deadlock in certain cases. For example, consider the case where you create two tasks in a task group that share an instance **E** of the cooperatively blocking **event** class. One of the tasks calls the **wait** method of event **E,** and the other task calls the **signal** method of event **E**. If the task group's **cancel** method is called while the first task is blocked waiting for event **E** but *before* the second task has started to execute, there will be no way for the wait condition of the first task to be satisfied.

Use extreme caution if you mix blocking operations other than **task_group::wait** with task group cancellation. In such cases you must ensure that all possible code paths are deadlock free.

Cancellation will automatically propagate across task groups in certain situations. For example, cancellation will be propagated if a task in task group **A** is waiting for the tasks of task group **B** to complete. In this situation, if task group **A**'s **cancel** method is called before the call to task group **B**'s **wait** method completes, then the runtime *also invokes* task group **B**'s **cancel** method. The task in task group **A** that is waiting on task group **B** remains blocked until task group **B** has no more running tasks. At that time, the call to task group **B**'s **wait** method will throw an exception. (Of course, if task group **B** is very fast, its **wait** function might return normally before there is a chance to propagate a cancellation from task group **A**.)

A cancellation request automatically propagates to another task group if a call to that group's ***wait*** *method is blocking any of the tasks of a task group that is being cancelled.*

Note that the runtime only returns the enumerated value **canceled** for the **wait** method of the top-most task group in a tree of dependent task groups. The other stack frames will have an internal exception passed through them.

Long-running tasks can use the **is_canceling** method to poll their task group for its cancellation status and shut themselves down if cancellation has been requested. The **is_canceling** method might also be used if you need to perform local cleanup actions for a task that's in the process of being canceled. When the task group returns from a call to its **wait** method, its state is reset and its **is_canceling** method thereafter returns **false**.

Returning from the **task_group::wait** *method returns a task group object to its initial, default state. The task group has the same behavior as if it were newly created.*

Checking for cancellation within a loop that has many iterations that each performs a small amount of work can negatively affect your application's performance. On the other hand, 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 less frequently, perhaps every two to ten seconds. Profile your application to collect performance data that will help you determine the best places to test for cancellation requests in your code. In particular, look for places in your code where you can poll for cancellation at evenly spaced intervals. For more information about profiling, see "Appendix B, Profiling and Debugging."

HANDLING EXCEPTIONS

If the execution of a task's work function throws an unhandled exception, the unhandled exception is temporarily unobserved by your application. The runtime catches the exception and records its details in internal data structures. Then, the runtime cancels the task group that contains the faulted task. See the previous section, "Canceling a Task Group," for more information about what happens during task group cancellation. Under certain conditions, the cancellation is automatically propagated to other task groups.

When a task of a task group throws an unhandled exception, the runtime cancels that task group. The task group's **is_canceling** *method returns* **true** *during the course of the shutdown.*

Recovering a deferred exception and rethrowing it is known as "observing an unhandled task exception." When all of the tasks in the task group have completed, the **task_group::wait** method rethrows the faulted task's exception in the runtime context of the thread that invoked the **wait** method. If there is more than one exception (that is, if more than one task threw an unhandled exception), the runtime will choose one of the exceptions to rethrow. The remaining exceptions will not be observed.

Be aware that if more than one task throws an exception, only one of the exceptions will be observed by the **task_group::wait** method. You can't control which exception will be rethrown.

If you need to record all of the exceptions, you can implement custom logging or tracing code.

Speculative Execution

Speculative execution occurs when you perform an operation in anticipation of a particular result. For example, you might predict that the current computation will produce the value 42. You start the next computation, which depends on the current computation's result, by using 42 as the input. If the first computation ends with 42, you've 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, you can restart the second operation using the correct value as the input.

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. After the fastest task finds the item, you don't need to wait for the other searches to complete. In cases like this you wait for the first task to complete and usually cancel the remaining tasks. However, you should always observe any exceptions that might have occurred in any of the tasks.

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

You can use the **task_group::cancel** method to implement speculative execution. Here's an example.

```
task_group tg;

tg.run([&tg](){ SearchLeft(tg); });
tg.run([&tg](){ SearchRight(tg); });
tg.run_and_wait([&tg](){ SearchCenter(tg); });
```

In this example, you perform three searches in parallel. You want to continue if any of the three functions completes. You don't need to wait for all of them. To make this possible, the code passes a reference to the task group object to each of the work functions. Inside of the work functions, the code cancels the task group when it completes its work. The following code shows how to accomplish this inside of the **SearchLeft** function.

```
void SearchLeft(task_group& tg)
{
  bool result =
              DoCpuIntensiveOperation(g_TaskMilliseconds/5, &tg);
  wcout << L"  Left search finished, completed = " << result
        << endl;
  tg.cancel();
}
```

The long-running function, **DoCpuIntensiveOperation**, checks for the cancellation status. This is shown in the following code.

```
bool DoCpuIntensiveOperation(DWORD milliseconds,
                             task_group* tg = nullptr)
{
  // ...
  int i = 0;
  DWORD checkInterval = ...
  while (true)
  {
    if ((milliseconds == 0) || (++i % checkInterval == 0))
    {
      if (tg && tg->is_canceling())
        return false;
    }
    // ...
  }
}
```

The body of the **while** loop periodically checks to see if the task group, **tg**, has received a cancellation request. For performance reasons the code only polls at a specified number of loop iterations.

Anti-Patterns

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

VARIABLES CAPTURED BY CLOSURES

In C++, a closure can be created using a lambda expression that represents an unnamed (anonymous) function. Closures can 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. If you code your closure incorrectly, you may find that captured variables don't behave as you expect, especially in parallel programs.

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

```
  task_group tg;
  for (int i = 0; i < 4; i++)
  {
     // WARNING: BUGGY CODE, i has unexpected value
     tg.run([&i]() { wcout << i << endl; } );
  }
  tg.wait();
```

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 captured *by reference* and shared by all 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.

The solution is to capture the variable *by value* in the appropriate scope.

```
task_group tg;
for (int i = 0; i < 4; i++)
{
   tg.run([i]() { wcout << i << endl; } );
}
tg.wait();
```

It's a good idea to make sure that every lambda expression uses explicit capture instead of implicit capture.

This version prints the numbers 1, 2, 3, 4 in an arbitrary order, as was originally intended. The reason is that the value of the variable **i** is passed to the closure. Effectively, a new variable named **i** is instantiated with each iteration of the **for** loop.

UNINTENDED PROPAGATION OF CANCELLATION REQUESTS

Be aware that implicit parent-child relationships can influence the behavior of cancellation and exception handling, especially with libraries.

If you use a library that is implemented with PPL, the API calls into that library may create task groups that are internal to the library. If you call that library's APIs from a task context within your application, you might unintentionally create a situation where a task of your application's task group is waiting on a task group inside of the library's implementation. According to the behavior described in the "Canceling a Task Group" section, if you invoke the **cancel** method of a task group in your application, you may implicitly cause the cancellation of task groups that were created by the library you are calling. Transitive propagation of cancellation into another component's internal task groups may not be the behavior you intend; in some cases, you may prefer that library functions run to completion even though a higher-level component is beginning to shut down.

You avoid cases of unintended propagation of runtime context by using a neutral, non-PPL thread context to call into any library functions that may depend on task group wait operations. For example, you could use a lightweight task to invoke library functions. A lightweight task is a lower-level type of task that does not result in the propagation of cancellation requests. They are described in Appendix A.

THE COST OF SYNCHRONIZATION

Locks and other synchronization operations are sometimes necessary in parallel programs. However, programmers often underestimate how much serializing operations can degrade performance.

You may want to review the "Scalable Sharing of Data" section of Chapter 1, "Introduction" for guidance on how to factor synchronization into the design of your application. Well-designed applications require explicit synchronization operations less often than poorly designed applications.

To avoid performance bottlenecks, review your use of locks and other synchronization operations carefully.

Design Notes

This section describes some of the design considerations that were used to create the Parallel Patterns Library, along with some recommended coding practices.

TASK GROUP CALLING CONVENTIONS

Whenever you call a task group's **run** method, you must subsequently call its **wait** method and allow that call to return before you destroy the task group. Otherwise, the runtime will throw a "missing wait" exception. The missing wait exception only occurs in the normal flow of execution; it will not be thrown if you unwind the stack due to an application exception. Therefore, you do not need an RAII wrapper that calls the **wait** method.

The **task_group** class's methods are all concurrency safe, so there are many ways to invoke these methods. However, no matter which method you choose, you must make sure a return from a call to the task group's **wait** method is the last operation that happens in the normal (no exceptions thrown) execution path before allowing a task group's destructor to run.

If a task group's **wait** method is called from multiple contexts, and you interleave these calls with calls to the task group's **run** method, be aware that the results may vary from run to run. For example, it is possible to call a task group's **wait** method concurrently from two different contexts. If there are pending tasks in the task group, both invocations of **task_group::wait** will return only after the task group has completed all of its pending tasks. However, if the task group is canceled while the tasks are running, only one of the wait functions will return the **canceled** status value. The other invocation will return a normal status, due to interleaving. (Returning from the **wait** method resets the task group's **is_canceling** status as a side effect; whichever invocation returns first will perform the reset.)

TASKS AND THREADS

When a task begins to run, the applicable task scheduler invokes the task's work function 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 critical sections, without having to worry, for example, that the **EnterCriticalSection** function will be executed in a different thread than the **LeaveCriticalSection** function.

In general, creating a new task is a much less resource-intensive operation than creating a new thread. It is possible for an application to create hundreds of thousands or even millions of tasks and still run efficiently.

You may want to profile your application as you experiment with strategies for using tasks in your application. If your tasks are too fine grained, you will incur overhead for task management that may hurt performance. For example, a task that performs a single arithmetic operation is not going to improve performance. However, if you decompose your application into too few tasks, you will not fully exploit the potential parallelism of the application.

HOW TASKS ARE SCHEDULED

The techniques for scheduling tasks and scheduling threads demonstrate fundamentally different goals. The operating system generally schedules threads in a way that minimizes latency. Preemptive thread scheduling allows users to interact with a busy system with very little perceived delay, even on a system with only one core.

In contrast, the task scheduler tries to maximize overall throughput rather than ensure low latency for any given task. In other words, when you decompose a computationally intensive operation into tasks that can potentially run in parallel, you want to complete the *overall operation* as quickly as possible without concern for the scheduling delays that any given task might experience. For task-based systems such as PPL and the underlying Concurrency Runtime, the measure of success is the speed of the overall result. The goal is to optimize the use of processor resources.

For these reasons you should not expect that tasks in PPL are scheduled "fairly." Instead, a variety of techniques are used to improve throughput. These techniques mainly focus on keeping cores busy and on an effective use of system caches. For more information about scheduling, see Appendix A, "The Task Scheduler and Resource Manager."

There are a number of options available that allow you to control how the scheduler deals with tasks. See "Schedule Policies" in Appendix A for more information.

STRUCTURED TASK GROUPS AND TASK HANDLES

In addition to the **task_group** class, PPL also provides a lower-level interface called the **structured_task_group** class, which is documented on MSDN®.

Structured task groups have lower overhead than the task groups, but there are restrictions on how structured task groups can be used. These restrictions require stack-based work functions and strict nesting of subordinate structured task groups. Although the task groups are recommended for most application programming, structured task groups are important for implementing parallel libraries. For example, the PPL **parallel_invoke** function is implemented with structured task groups. The **parallel_invoke** function is usually enough in cases of strict nested parallelism, and because it is much easier to use than structured task groups, you probably won't ever need to use structured task groups directly.

PPL includes a data type named the **task_handle** class. It encapsulates a work function used by a task. One of the overloaded versions of the **task_group** class's **run** method accepts a task handle as its argument. Task handles are created by means of the **make_task** function. Most applications will never need access to task handles; however, you must use task handles with structured task groups. Unlike lambda expressions, task handles require explicit memory management by your application.

LIGHTWEIGHT TASKS

In addition to the **task_group** objects that were described in this chapter, the Concurrency Runtime provides lower-level APIs that may be useful to some programmers, especially those who are adapting existing applications that create many threads. However, in general it is recommended that programmers start with the Parallel Patterns Library (PPL) or the Asynchronous Agents Library.

See the "Lightweight Tasks" section of Appendix A, "The Task Scheduler and Resource Manager" for more information.

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 there is, 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 they save their results by updating their second argument. Why don't they return their results?
3. In the image blender sample that uses **task_group::run** method, what happens if one of the parallel tasks throws an exception? Answer the same question for the sample that uses the **parallel_invoke** function.

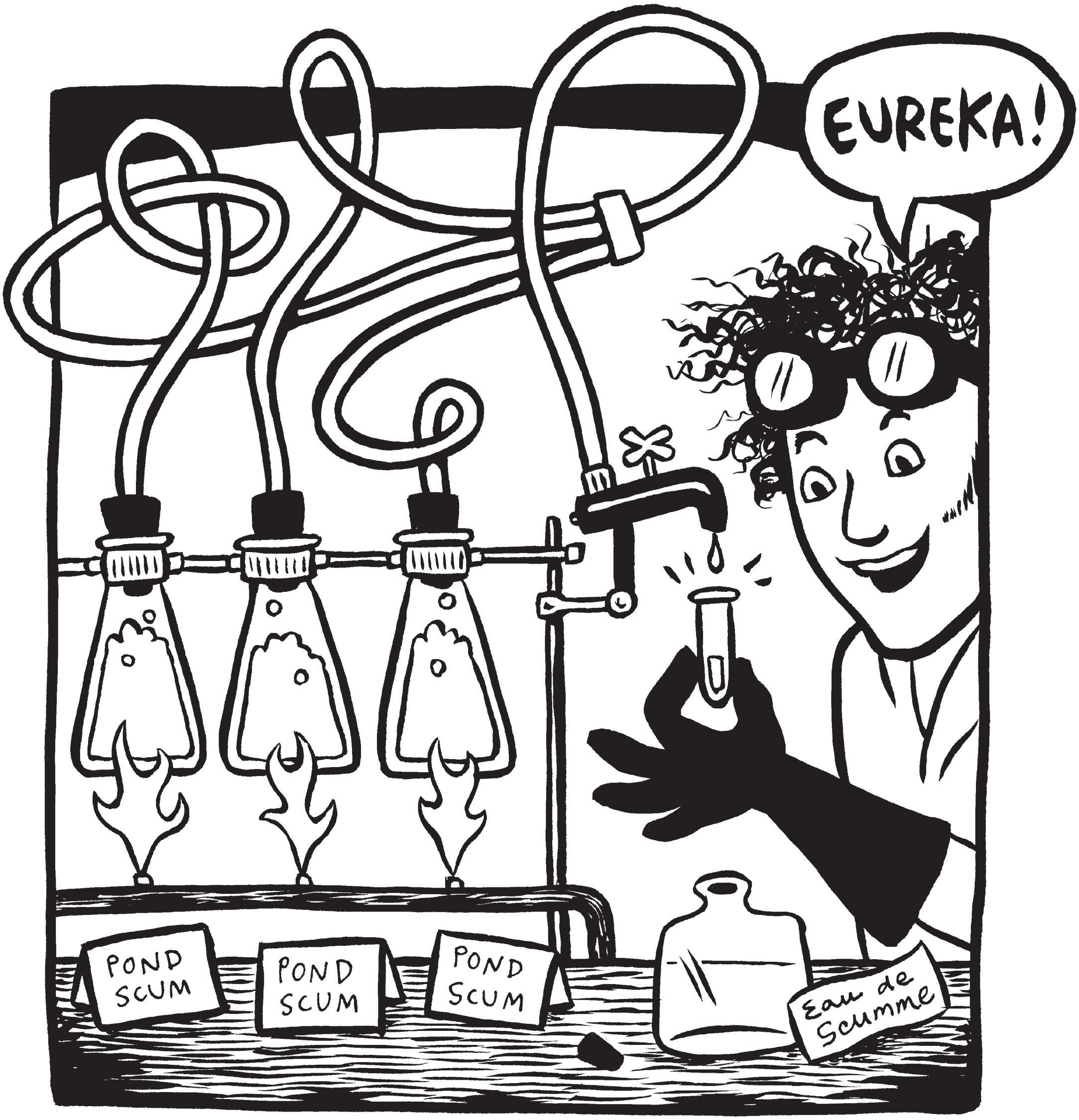
Further Reading

Leijen et al. discusses design considerations, including the concepts of task-based scheduling and work stealing algorithms.

The code samples for the Concurrency Runtime and Parallel Pattern Library package is ConcRTExtras on CodePlex.

Leijen, D., W. Schulte, and S. Burckhardt. "The Design of a Task Parallel Library." 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.

ConcRTExtras software. "Code Samples for the Concurrency Runtime and Parallel Pattern Library in Visual Studio 2010." http://code.msdn.microsoft.com/concrtextras.


EUREKA!
POND SCUM
POND SCUM
POND SCUM
Eau de Scumme

4

Parallel Aggregation

Chapter 2, "Parallel Loops," 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 sequential loop that calculates a sum does *not* have independent steps. All the steps accumulate their results in a single variable that represents the sum calculated up to that point. This accumulated value is an aggregation. If you were to convert the sequential loop to a parallel loop without making any other changes, your code would fail to produce the expected result. Parallel reads and writes of the single variable would corrupt its state.

The Parallel Aggregation pattern lets you use multiple cores to calculate sums and other types of accumulations that are based on associative operators.

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

Although calculating a sum is an example of aggregation, the pattern is more general than that. It works for any binary operation that is associative. However, some implementations of the Parallel Aggregation pattern also expect the operations to be commutative.

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 makes the steps of a loop independent of each other. Parallel aggregation demonstrates the principle that it's 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 later in this chapter.

The Parallel Aggregation pattern is also known as the Parallel Reduction pattern because it combines multiple inputs into a single output.

The Basics

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

```
vector<int> sequence = ...
int count = 0;
for (size_t i = 0; i < sequence.size(); i++)
    count += IsPrime(sequence[i]) ? 1 : 0;
return count;
```

This is a typical sequential **for** loop. In this example and the ones that follow, **IsPrime** is a user-provided function that determines if its argument is a prime number. The result is a count of how many prime numbers are contained in the input sequence. (Of course, you could also have used the Standard Template Library (STL) **count_if** operation in this particular example.)

How can sequential accumulation be adapted for parallel processing? As was explained in Chapter 2, simply swapping the **for** operator with **parallel_for** won't work because the **count** variable is shared by all iterations. You might also be tempted to wrap a critical section around the operation that increments the **count** variable. The critical section would prevent parallel iterations from performing conflicting reads and writes, but the performance of that approach would be much, much worse than the sequential version you are trying to optimize. The cost of synchronization would be prohibitive. (In fact, programmers often underestimate the performance cost of synchronization operations.)

Typical of many situations in parallel programming, the answer is not to apply synchronization operations to the existing sequential algorithm in order to make it "safe" for parallel execution. Instead, redesign the algorithm to use a two-phase approach. First, subdivide the problem into as many tasks as you have cores and calculate partial results locally on a per-core basis. Then, once all of the per-task partial results are ready, sequentially merge the results into one final accumulated value. The process of combining partial reductions is graphically illustrated by the cartoon illustration on this chapter's facing page.

The **combinable** *class makes it easy to create per-task local results in parallel and merge them as a final sequential step.*

PPL provides a special data structure that makes it easy to create per-task local results in parallel and merge them as a final sequential step. This data structure is the **combinable** class. The following code examples show how to use the combinable class to implement the Parallel Aggregation pattern.

```
vector<int> sequence = ...

combinable<int> count([]() { return 0; });
```

```
parallel_for_each(sequence.cbegin(), sequence.cend(),
  [&count](int i)
  {
      count.local() += IsPrime(i) ? 1 : 0;
  });
return count.combine(plus<int>());
```

The **count** variable is a **combinable** object that provides thread-private values. To compute the initial, local values the constructor of the **combinable** class takes a function as an argument.

Next, a **parallel_for_each** loop creates multiple tasks (typically, equal to some multiple of the number of cores on your computer) and runs the loop body function in parallel. The tasks collect the partial, per-core results into per-task variables that are provided by the **combinable** object's **local** method.

The combinable class assumes that the operation provided as an argument to the **combine** method is commutative.

The number of tasks depends on the level of concurrency available in the current context. See Appendix A, "The Task Scheduler and Resource Manager" for more information about runtime policy settings for concurrency. Also, the **parallel_for_each** loop uses dynamic range stealing to equalize the amount of work among its internal worker threads.

After the **parallel_for_each** loop completes, the **combinable** object's **combine** method applies a user-specified binary operation to aggregate the values of each of the per-task partial results. In this example the combination function is integer addition. The return value of the **combine** method is the final aggregated value.

The Concurrency Runtime sample pack provides several STL-style parallel aggregation functions. The easiest way to understand how these functions work is to compare them with their corresponding sequential operations in STL.

STL provides a very simple way to express sequential aggregation with iterators. Here is an example.

```
vector<int> sequence = ...
return accumulate(sequence.cbegin(), sequence.cend(), 0,
                  IncrementIfPrime());
```

The STL **accumulate** function applies a binary function to an internal accumulation variable and to each element of a sequence, updating the accumulation variable with each iteration. The first and second arguments to the **accumulate** function give the iteration bounds. The third argument is the initial value of the accumulation variable, and the fourth argument is a binary reduction function that will be successively applied to the accumulation variable and to each iterated value. The job of the reduction function is to combine two input values. Here is the implementation of the reduction function, **IncrementIfPrime**.

If the conventions of STL algorithms are unfamiliar to you, you should brush up on them before reading this chapter. See the "Further Reading" section for more information.

```
struct IncrementIfPrime
{
    int operator()(int total, int element) const
    {
        return total + (IsPrime(element) ? 1 : 0);
    }
};
```

The STL **accumulate** function is a sequential operation whose performance is comparable to the sequential **for** loop shown in the earlier example. To convert an STL **accumulate** expression into a parallel aggregation you can use the **parallel_reduce** function of the Concurrency Runtime sample pack. The following code gives an example.

```
using namespace ::Concurrency::samples;
vector<int> sequence = ...
return parallel_reduce(sequence.cbegin(), sequence.cend(), 0,
                       CountPrimes(), plus<int>());
```

The **parallel_reduce** function takes five arguments. The first two arguments give the iteration bounds. The third argument gives the value of the reduction's identity element. If the reduction is based on addition, this element will be 0. For multiplicative reduction, the identity element is 1. For reductions such as aggregate set union, the identity element is the empty set.

The fourth argument is a function object that can be applied on a subrange of an iterator to produce a local partial aggregation. This example uses a functor created by instantiating the **CountPrimes** class. The return value of the function object is the local partial result from the first phase of the Parallel Aggregation pattern.

The fifth argument is a reduction function that will combine the partial results that have been calculated for each of the subranges.

Here is the implementation of the **CountPrimes** class-type functor.

```
struct CountPrimes
{
  int operator()(vector<int>::const_iterator begin,
                 vector<int>::const_iterator end,
                 int right) const
  {
    return right + accumulate(begin, end, 0, IncrementIfPrime());
  }
};
```

The **parallel_reduce** function divides the input iterator into ranges. There will be enough ranges to compensate for the effects of uneven workloads, but not so many ranges that the overhead of calculating them dominates the computation. PPL determines how many ranges to create.

In this example, the **CountPrimes** function object will be invoked one time for each of the ranges. It executes a sequential accumulation operation on the subrange and collects the result.

The **parallel_reduce** function is usually the recommended approach whenever you need to apply the Parallel Aggregation pattern within applications that use PPL. Its declarative nature makes it less prone to error than other approaches, and its performance on multicore computers is competitive with them. Implementing parallel aggregation with **parallel_reduce** doesn't require adding locks in your code. Instead, all the synchronization occurs internally. Of course, if **parallel_reduce** doesn't meet your needs or if you prefer a less declarative style of coding, you can also use the **combinable** class with **parallel_for** or **parallel_for_each** to implement the parallel aggregation.

You should be aware that **parallel_for** and **parallel_for_each** add overhead due to their support of features such as cancellation and dynamic range stealing. Also, a call to the **combinable::local()** method inside of a parallel loop adds the cost of a hash table lookup to each iteration of the loop. In general, use parallel aggregation to increase performance when iterations perform complex computations.

An Example

Aggregation doesn't only apply to numeric values. It arises in many other application contexts. The following example shows how to use a variation of parallel aggregation known as map/reduce 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, and then candidates are ranked and selected in an aggregation operation.

Here's how the data structures and algorithms that are 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 a 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 only 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 type of collection known as 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 example in this section uses a multiset implementation that differs from STL.

The recommendation service uses map/reduce and has three phases.

In the first phase, which is the *map* **phase**, the service creates collections of friend candidates. The collections of potential friends are calculated by iterating through the subscriber's friends and searching their friends for people that are not currently friends of the subscriber.

In the second phase, which is the *reduce* **phase**, the service aggregates the sets of potential friends into a multiset where each candidate's ID is associated with its multiplicity (the number of mutual friends). For each set of possible friends, the reduce phase merges the sets of potential friends and maintains a count of the occurrences. It uses a **hash_map<FriendID, int>** instance for this purpose.

The final phase performs postprocessing. The service ranks candidates by sorting them according to their multiplicity and selects only the candidates with the largest multiplicities.

An important feature of map/reduce is that the result 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 produce only a list of candidate IDs; instead, it produces a vector of multisets, where each multiset contains only 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. In this example, the operation is a *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 only 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.

This is the code that defines the main data types that are used in the sample.

```
typedef int SubscriberID;
typedef int FriendID;
typedef set<FriendID> FriendsSet;
typedef shared_ptr<FriendsSet> FriendsSetPtr;
typedef hash_map<SubscriberID, FriendsSetPtr> SubscriberMap;


class FriendMultiSet : public hash_map<FriendID, int>
{
  // Multiset of potential friends.
  // ...
}

typedef shared_ptr<FriendMultiSet> FriendMultiSetPtr;
```

The **FriendsSet** type is implemented by an STL set. The **Friend MultiSet** type has a custom implementation. In addition to these data types, the sample also uses an ordered list of potential friends that is sorted by multiplicity in decreasing order. Here is the code.

```
struct LessMultisetItem
{
  bool operator()(const pair<FriendID, int> value1,
                  const pair<FriendID, int> value2) const
  {
    return (value1.second == value2.second) ?
              (value1.first > value2.first) :
              (value1.second > value2.second);
  }
};

typedef public set<pair<FriendID, int>, LessMultisetItem>
                                            FriendOrderedMultiSet;
```

Note that STL also implements a **std::multiset** type, but it is used to store sets which contain multiple key values with equal values, rather than key/value pairs.

Finally, here is the code for the sequential version of the algorithm that suggests potential friends.

```
FriendOrderedMultiSet
PotentialFriendsSequentialTransform(
  const SubscriberMap& subscribers,
  SubscriberID id,
  int maxCandidates)
{
  // Map:

  FriendsSetPtr friends = subscribers.find(id)->second;
  vector<FriendMultiSetPtr> friendsOfFriends(friends->size());

  transform(friends->cbegin(),friends->cend(),
    friendsOfFriends.begin(),
    [&subscribers,&friends,&id](int friendID)->FriendMultiSetPtr
    {
      FriendsSetPtr theirFriends =
                      subscribers.find(friendID)->second;
      FriendsSetPtr friendsOfFriend = make_shared<FriendsSet>();

      set_difference(theirFriends->cbegin(),
        theirFriends->cend(),
        friends->cbegin(),friends->cend(),
        inserter(*friendsOfFriend, friendsOfFriend->end()));
      friendsOfFriend->erase(id);

      return FriendMultiSetPtr(
                          new FriendMultiSet(friendsOfFriend));
    });

  // Reduce:
```

```
  // The reduction does not use std:accumulate because
  // this results in too much copying of intermediate
  // FriendCountMap
  FriendMultiSet candidates;
  for_each(friendsOfFriends.cbegin(), friendsOfFriends.cend(),
    [&candidates](FriendMultiSetPtr set)
    {
      candidates.Union(set);
    });

  // Postprocess:

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

In the map phase, this code loops sequentially over the subscriber's friends and builds a collection of multisets of candidates. In the reduce phase, the code loops sequentially over those multisets and aggregates them with the multiset union operation, which is implemented by the **Union** method. If this code executes with the few subscribers in the example, the **id** argument is 0 and **subscribers.find(id)->second** returns { 1, 2}. When the map phase completes, the **friendsOfFriend** variable contains { 3(1) }, { 3(1) , 4(1) }. When the reduce phase completes, **candidates** contains { 3(2), 4(1) }.

Multiset union is associative; if you aggregate several multisets into one by successively forming unions in a pair-wise manner, the final result does not depend on the order of the union operations. Multiset union is also commutative; the result does not depend on the order of its arguments. 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.

Strictly speaking, 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 due to rounding errors.

Here's how to use the **parallel_transform** and **parallel_reduce** functions of the Concurrency Runtime sample pack to apply map/reduce to the social networking example.

```
FriendOrderedMultiSet
PotentialFriendsParallel(const SubscriberMap& subscribers,
                         SubscriberID id,
                         int maxCandidates)
{
  // Map:

  FriendsSetPtr friends = subscribers.find(id)->second;
  vector<FriendMultiSetPtr> friendsOfFriends(friends->size());

  parallel_transform(friends->cbegin(),friends->cend(),
    friendsOfFriends.begin(),
    [&subscribers,&friends,&id](int friendID)->FriendMultiSetPtr
    {
      FriendsSetPtr theirFriends =
                             subscribers.find(id)->second;
      FriendsSetPtr friendsOfFriend = make_shared<FriendsSet>();

      set_difference(
        theirFriends->cbegin(), theirFriends->cend(),
        friends->cbegin(), friends->cend(),
        inserter(*friendsOfFriend, friendsOfFriend->end()));
      friendsOfFriend->erase(id);

      return FriendMultiSetPtr(
                          new FriendMultiSet(friendsOfFriend));
    });

  // Reduce:

  FriendMultiSet candidates;
  candidates =
    parallel_reduce(friendsOfFriends.cbegin(),
      friendsOfFriends.cend(),
      FriendMultiSet(),
      [](vector<FriendMultiSetPtr>::const_iterator cbegin,
         vector<FriendMultiSetPtr>::const_iterator cend,
         const FriendMultiSet& right)
      {
        return right.Union(cbegin, cend);
      },
      [](const FriendMultiSet& left, const FriendMultiSet& right)
      {
        return left.Union(right);
      });

  // Postprocess:

  return candidates.MostNumerous(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 the friends of subscriber 0. The map phase is performed by the **parallel_transform** function, which finds all the friends of each friend of the subscriber. The **set_difference** function prevents redundant recommendations by removing the subscriber. The output of the map phase is a vector of multisets for each of the subscriber's friends.

The reduce phase is performed by the call to the **parallel_reduce** function, which counts the duplicate candidate IDs. Note that the call to the **FriendMultiSet** function returns an empty multiset that is used as the identity element. The **Union** method combines two multisets.

The **return** statement performs the final postprocessing step that selects the candidates with the highest multiplicities.

Variations

This section contains some common variations of the Parallel Aggregation pattern.

CONSIDERATIONS FOR SMALL LOOP BODIES

If the body of your parallel aggregation loop performs very little work, you may find that performing parallel aggregation takes longer than sequential aggregation. When you have small loop bodies, you can apply the techniques that were described in Chapter 3, "Parallel Loops" in the "Special Handling of Small Loop Bodies" section. These techniques allow you to use sequential aggregation within subranges.

OTHER USES FOR COMBINABLE OBJECTS

The **combinable** class is most commonly used to implement the Parallel Aggregation pattern, but you do not necessarily need to use **combinable** objects for aggregation. You can also use **combinable** instances to create thread-local variables when a thread starts.

Design Notes

If you compare the sequential and parallel versions of the Parallel Aggregation pattern, you see that the design of the parallel version includes an additional step in the algorithm that merges partial results. Figure 1 illustrates the two phases of parallel aggregation.

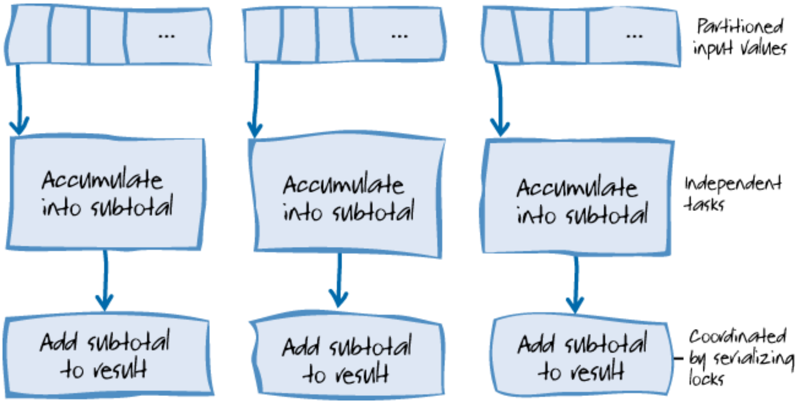

...
...
...
Partitioned input values
Accumulate into subtotal
Accumulate into subtotal
Accumulate into subtotal
Independent tasks
Add subtotal to result
Add subtotal to result
Add subtotal to result
Coordinated by serializing locks

FIGURE 1
Parallel aggregation

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 1). The **local** method of the **combinable** class provides access to the unshared storage for each thread. Each worker thread 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. After all of the partial results have been computed, the combine function of the combinable object merges the local results into the final, global result.

The reason that this approach is fast is that there is very little need for synchronization operations. Calculating the per-task local results uses no shared variables, and therefore requires no locks. The **combine** operation is a separate sequential step and also does not require locks.

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

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 a parallel approach.

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. To do this, you only need to convert sequential **for** to **parallel_for** and add one lock statement.

```
// WARNING: BUGGY CODE. Do not copy this example.
// It will run *much slower* than the sequential version.
// It is included to show what *not* to do.

vector<int> sequence = ...
CRITICAL_SECTION cs;
InitializeCriticalSectionAndSpinCount(&cs, 0x80000400);
```

```
int count = 0;

// BUG -- Do not use parallel_for_each in this case
parallel_for_each(sequence.cbegin(), sequence.cend(),
  [&count, &cs](int i)
  {
    // BUG -- Do not use locking inside of a parallel aggregation
    EnterCriticalSection(&cs);
    // BUG -- Do not use shared variable for parallel aggregation
    count += IsPrime(i) ? 1 : 0;
    LeaveCriticalSection(&cs);
  });

return count;
```

If you forget to enter and exit the critical section, this code fails to calculate the correct sum on a multicore computer. Adding the synchronization code makes this 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! The reason for the poor performance is the cost of synchronization.

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 perform better, 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's a group of patterns related to summarizing data in a collection. Aggregation (also known as Reduce) is one of them. The others include Scan and Pack. 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. For more information about these related patterns, see the section, "Further Reading," at the end of this chapter.

Exercises

1. 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?
2. In the social network example, there's 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?
3. In the standard reference on map/reduce (see the section, "Further Reading"), 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?

Further Reading

Musser et al. explain the standard template library (STL). 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. Toub's examples are in managed code, but the algorithms apply equally to native code. The Wikipedia article describes the mathematical multisets that were used in code example in this chapter.

Dean, J., and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In OSDI '04: Sixth Symposium on Operating System Design and Implementation, 137–150, 2004.

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

McCool, M., "Structured Patterns: An Overview." December 2009.

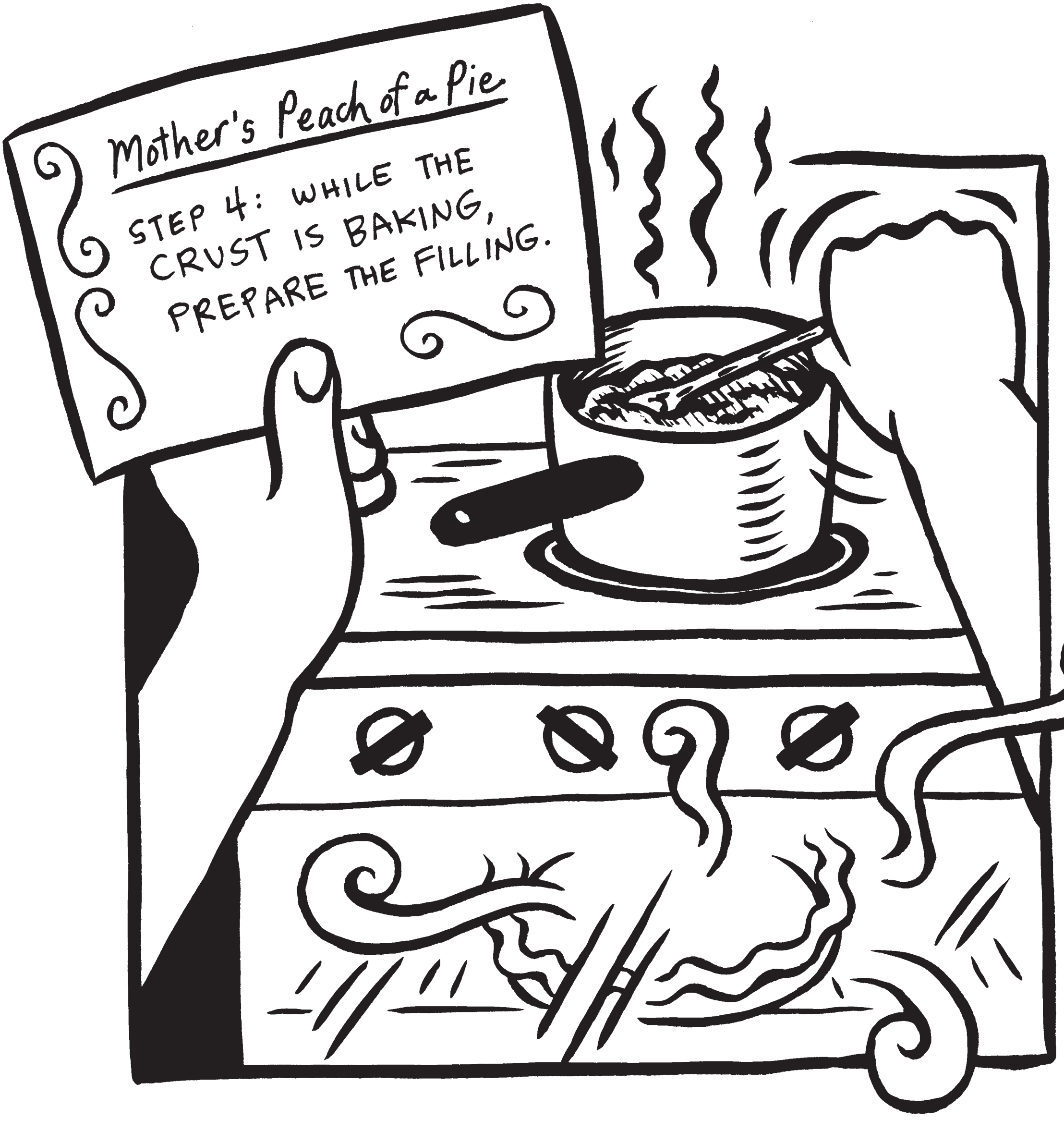
http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=223101515.

Musser, D. R., G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library,* 3rd edition. Addison-Wesley Professional, December 2009.

Toub, S., "Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4." 2009.

http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en.

Wikipedia. "Multiset." http://en.wikipedia.org/wiki/Multiset.


Mother's Peach of a Pie
STEP 4: WHILE THE CRUST IS BAKING, PREPARE THE FILLING.

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. The Futures pattern integrates task parallelism with the familiar world of arguments and return values.

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.

Don't confuse the task-based futures in this chapter with other Future pattern implementations such as the **std::future** implementation in the Standard Template Library (STL) that has been incorporated into the C++0x working paper.

You can think of a future as 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 thread that needs the result 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 pending task may be executed in the current thread context.

The Parallel Patterns Library (PPL) makes it very easy to use the Futures pattern. Here is a minimal futures implementation that illustrates how futures work.

```
template <class T>
class Future
{
  private:
    single_assignment<T> m_val;
```

This implementation of a **Future** class omits features such as the ability to rethrow exceptions when you call the **Result** method multiple times. You can use this implementation in your own applications, but you should be aware that it is not meant to be completely full-featured.

```
    task_group m_tg;

  public:
    template <class Func>
    Future(Func f)
    {
        m_tg.run([f, this]()
        {
          send(m_val, f());
        });
    }

    T Result()
    {
       m_tg.wait();
       return receive(&m_val);
    }
};
```

You can easily implement futures using task groups.

In this example, each new instance of the **Future<T>** class creates a task group and uses the task group's **run** method to add a new task to that task group. The work function of the new task is an argument to the constructor of the **Future<T>** class. The work function returns a value of type **T**.

> **Note:** *The* **single_assignment** *class that is used in the implementation of the* **Future** *class is a type of messaging buffer. The* **send** *and* **receive** *functions allow for concurrency-safe communication of a single data value. For more information about messaging buffers, see Chapter 7, "Pipelines."*

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, just as an assembly line or data stream does. In contrast, with futures, nodes of the task graph are connected by singleton values, similar to arguments and return values.

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.

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 with return values. 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.

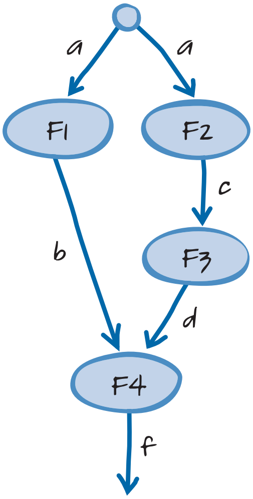
```
int a = 22;

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

Suppose that **F1**, **F2**, **F3**, and **F4** are processor-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.

FIGURE 1
A task graph for calculating *f*



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. For simplicity, the code assumes 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.

```
int a = 22;

Future<int> futureB([a](){ return F1(a); });
int c = F2(a);
int d = F3(c);
int f = F4(futureB.Result(), d);
return f;
```

The **Result** *method either 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** method 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.

```
int a = 22;

Future<int> futureD([a](){ return 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** method. This makes exception handling easy. You can think of futures as either returning a result or throwing an exception. Conceptually, this is very similar to the way any C++ function works. Here is another example of exception handling.

Futures, when they are based on tasks, defer exceptions until the **Result** *method is called.*

```
int a = 22;

Future<int> futureD([a](){ return F3(F2error(a)); });
int b = F1(a);
try
{
  int f = F4(b, futureD.Result());
  printf("  Result = %d\", f);
}
catch (exception& e)
{
  printf("  Exception ‘%s’ is caught as expected.\n",
         e.what());
}
```

If an exception of type **exception** were thrown in **F2** or **F3**, it would be deferred and rethrown when the **Result** method of **futureD** is called. In this example, the invocation of the **Result** method occurs within a **try** block, which means that the exception can be handled in the corresponding **catch** block.

Example: 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 the application has the market data, it 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 System. 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.

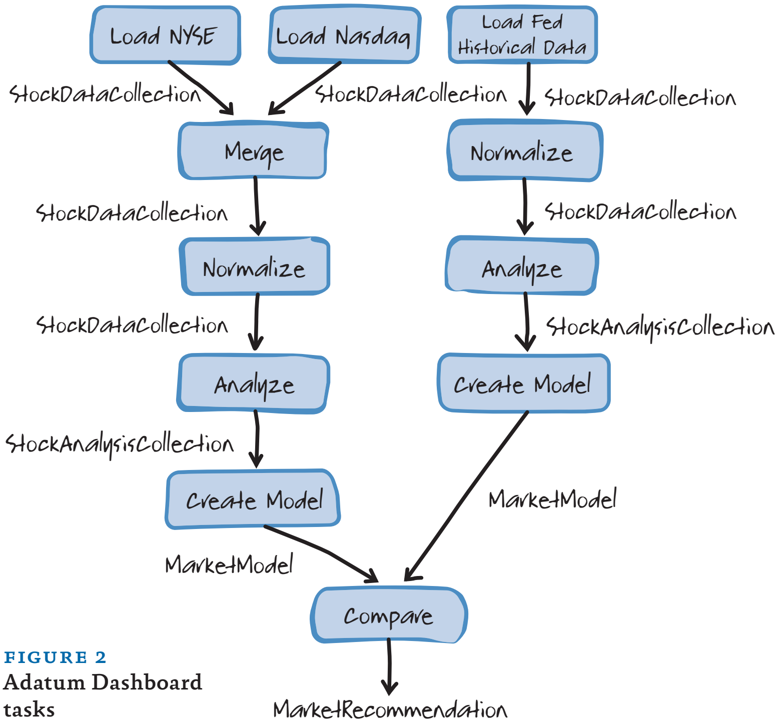

Load NYSE
Load Nasdaq
Load Fed Historical Data
StockDataCollection
StockDataCollection
StockDataCollection
Merge
Normalize
StockDataCollection
StockDataCollection
Normalize
Analyze
StockDataCollection
StockAnalysisCollection
Analyze
Create Model
StockAnalysisCollection
Create Model
MarketModel
MarketModel
Compare
MarketRecommendation

FIGURE 2
Adatum Dashboard tasks

The tasks in this diagram communicate by specific types of business objects. These are implemented as classes in the Adatum Dashboard application.

You can download the source code for the Adatum Dashboard application from the CodePlex site at http://parallelpatternscpp.codeplex.com/ in the Chapter5\A-Dash project. The application consists of three parts: the business object definitions, an analysis engine, and the user interface.

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

The **StockAnalysisCollection** type 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 data accessor method that returns a "buy, hold, or sell" decision.

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. This code differs slightly from the online sample source; details of cancellation handling have been omitted for clarity.

```
MarketRecommendation
DoAnalysisSequential(AnalysisEngineState& engineState) const
{
  engineState.Reset();
  engineState.IsRunning();
  vector<StockDataCollection> stockDatasets;
  vector<MarketModel> models;

  // Current market data tasks

  stockDatasets.push_back(LoadNyseData());
  stockDatasets.push_back(LoadNasdaqData());
  StockDataCollection mergedMarketData =
                               MergeMarketData(stockDatasets);
  StockDataCollection normalizedMarketData =
                               NormalizeData(mergedMarketData);
  StockAnalysisCollection analyzedStockData =
                                AnalyzeData(normalizedMarketData);
  models.push_back(RunModel(analyzedStockData));

  // Historical data tasks

  StockDataCollection fedHistoricalData =
                                      LoadFedHistoricalData();
  StockDataCollection normalizedHistoricalData =
                               NormalizeData(fedHistoricalData);
  StockAnalysisCollection analyzedHistoricalData =
                           AnalyzeData(normalizedHistoricalData);
  models.push_back(RunModel(analyzedHistoricalData));

  // Compare results

  MarketRecommendation result = CompareModels(models);
  engineState.SetMarketRecommendation(result.GetValue());
  engineState.IsStopped();
  return result;
}
```

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 parallel version uses futures for each of the operational steps. Here's the code. This code differs slightly from the online sample source; details of cancellation handling have been omitted for clarity.

```
MarketRecommendation
DoAnalysisParallel(AnalysisEngineState& engineState) const
{
  engineState.Reset();
  engineState.IsRunning();

  // Current market data tasks

  Future<StockDataCollection> future1(
    [this, &engineState]()->StockDataCollection
    {
      scoped_oversubcription_token oversubscribeForIO;
      return LoadNyseData();
    });

  Future<StockDataCollection> future2(
    [this, &engineState]()->StockDataCollection
    {
      scoped_oversubcription_token oversubscribeForIO;
      return LoadNasdaqData();
    });

  Future<StockDataCollection> future3(
    [this, &engineState, &future1, &future2]()
      ->StockDataCollection
    {
      vector<StockDataCollection> stockDatasets;
      stockDatasets.push_back(future1.Result());
      stockDatasets.push_back(future2.Result());
      return this->MergeMarketData(stockDatasets);
    });

  Future<StockDataCollection> future4(
    [this, &engineState, &future3]()->StockDataCollection
    {
      return NormalizeData(future3.Result());
  });

Future< StockAnalysisCollection> future5(
  [this, &engineState, &future4]()
    ->StockAnalysisCollection
  {
    return AnalyzeData(future4.Result());
  });

Future< MarketModel> future6 = Future<MarketModel>(
  [this, &engineState, &future5]()->MarketModel
  {
    return RunModel(future5.Result());
  });

// Historical data tasks

Future<StockDataCollection> future7(
  [this, &engineState]()->StockDataCollection
  {
    scoped_oversubcription_token oversubscribeForIO;
    return LoadFedHistoricalData();
  });

Future<StockDataCollection> future8(
  [this, &engineState, &future7]()->StockDataCollection
  {
    return NormalizeData(future7.Result());
  });

Future<StockAnalysisCollection> future9(
  [this, &engineState, &future8]()->StockAnalysisCollection
  {
    return AnalyzeData(future8.Result());
  });

Future<MarketModel> future10 = Future<MarketModel>(
  [this, &engineState, &future9]()->MarketModel
  {
    return RunModel(future9.Result());
  });

// Compare results

vector<MarketModel> models;
models.push_back(future6.Result());
```

```
  models.push_back(future10.Result());
  MarketRecommendation result = CompareModels(models);
  engineState.SetMarketRecommendation(result.GetValue());
  engineState.IsStopped();
  return result;
}
```

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. On a single-core machine the performance of the parallel version will be approximately the same as the sequential version. On a computer with many cores, the futures will all execute in parallel, constrained by the data dependencies that exist among them.

Several of the futures are long-running I/O-intensive tasks that use a small percentage of a core's processing power. For these futures, the code uses the **scoped_oversubscription_token** class to signal that the task scheduler can use the resources that were allocated to the current task to perform another task concurrently.

Variations

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

Canceling Futures

There are several ways to implement a cancellation model using the Futures pattern.

By default, if you enter the **Result** method of a future from within a task context, canceling that task's task group before the **Result** method exits will implicitly cause the task group in the Future instance to be canceled. See the section, "Canceling a Task," in Chapter 3, "Parallel Tasks" for more information about the propagation of cancellation across task groups.

In addition to implicitly propagated task cancellation, you can also use messaging buffers as a way to implement an explicit cancellation approach for futures. A cancellation strategy based on messaging buffers is shown in the ImagePipeline example in Chapter 7, "Pipelines."

Removing Bottlenecks

The idea of a critical path is familiar from project management. A "path" is any sequence of tasks from the beginning of the work to the end result. A task graph may contain more than one path. For example, look at the task graph that is shown in Figure 2. You can see that there are three paths, beginning with "Load NYSE," "Load Nasdaq," and

"Load Fed Historical Data" tasks. Each path ends with the "Compare" task.

The duration of a path is the sum of the execution time for each task in the path. The critical path is the path with the longest duration. The amount of time needed to calculate the end result depends only on the critical path. As long as there are enough resources (that is, available cores), the noncritical paths don't affect the overall execution time.

If you want to make your task graph run faster, you need to find a way to reduce the duration of the critical path. To do this, you can organize the work more efficiently. You can break down the slowest tasks into additional tasks, which can then execute in parallel. You can also modify a particularly time-consuming task so that it executes in parallel internally using any of the patterns that are described in this book.

The Adatum Dashboard example doesn't offer much opportunity for breaking down the slowest tasks into additional tasks that execute in parallel. This is because the paths are linear. However, you can use the Parallel Loops and Parallel Aggregation patterns to exploit more of the potential parallelism within each of the Analyze tasks if they take the most time. The task graph remains unchanged, but the tasks within it are now also parallelized. The Parallel Loops pattern is discussed in Chapter 2, "Parallel Loops," and the Parallel Aggregation pattern is discussed in Chapter 4, "Parallel Aggregation."

Modifying the Graph at Run Time

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.

However, you can also create dependencies between futures dynamically. For example, if you wanted to update the UI after each of the futures in the Adatum Dashboard example completed in order to show the application's progress, you could create tasks that wait on the futures that make up the task graph of the Adatum Dashboard example. In other words, you can call the **Result** method of the **Future** class as many times as needed. With each invocation, the calling context will be suspended until the values have been computed. Making calls to the **Result** method can occur outside of the context where the futures were originally created.

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

Design Notes

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

Decomposition into Futures

The first design decision is the most obvious one: the Adatum Dashboard introduces parallelism by means of futures. 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 use 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 possible to use an implicit approach. In Chapter 3, tasks communicate with side effects that modify shared data structures. In this case, you rely on the tasks to use control dependencies that block appropriately. However, in general, explicit data flow is less prone to error that implied data flow.

You can see this by analogy. In principle, 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. However, in practice, using return values is considered to be a much less error-prone way to write programs. Programmers tend to make more mistakes with global variables than with return values.

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. In addition, tasks that return values can often require less synchronization than tasks that globally access state variables, and they are much easier to understand.

Futures also promote a natural kind of data isolation similar to what is found in functional programming, which relies on operations that communicate with input and output values. Functional programs are very easy to adapt to multicore environments. In general, futures should only communicate with the outside world by means of their return values. It's also a good practice to use immutable types for return values.

Applications that use arguments and return values to communicate among tasks scale well as the number of cores increases.

Related Patterns

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

PIPELINE PATTERN

The Pipeline pattern is described in Chapter 7, "Pipelines." It differs in several important respects from a task graph. The pipeline focuses on data flow by means of queues (messaging 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 parent/child relationship. The master task creates the worker 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 PPL uses the Master/Worker pattern internally.

DYNAMIC TASK PARALLELISM PATTERN

The Dynamic Task Parallelism pattern is also known as the Divide and Conquer pattern. It 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

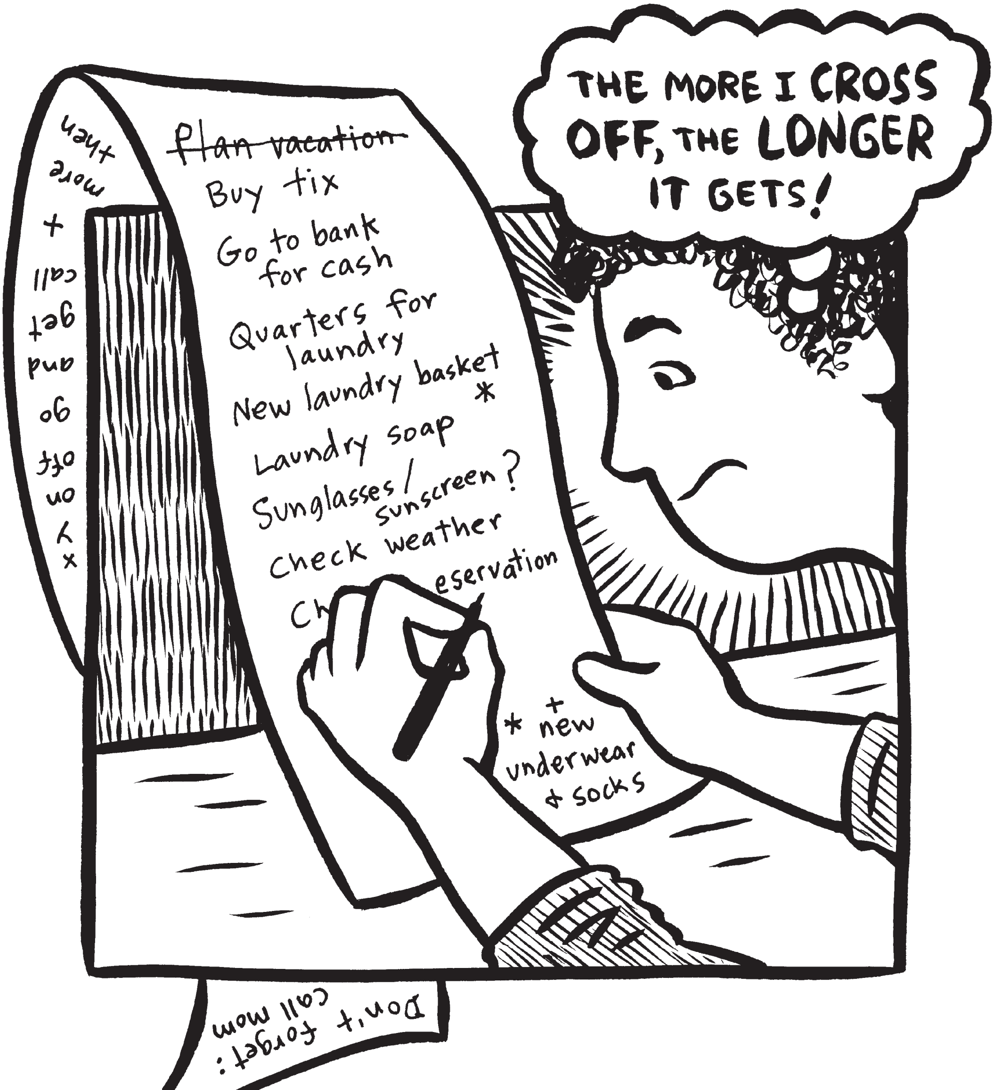
The Discrete Event pattern focuses on sending messages between tasks. There is no limitation on the number of events a task raises or when it raises them. Events can also pass between tasks in either direction; there is no antecedent/dependency relationship. The Discrete Event pattern can be used to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

Exercises

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

   ```
   auto b = F1(a);  auto d = F2(c);  auto e = F3(b,d);
   auto f = F4(e);  auto g = F5(e);  auto 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 the CodePlex at http://parallelpatternscpp.codeplex.com/ so that one of the futures throws an exception. What should happen? Observe the behavior when you execute the modified sample.


THE MORE I CROSS OFF, THE LONGER IT GETS!
~~Plan vacation~~
Buy tix
Go to bank for cash
Quarters for laundry
New laundry basket *
Laundry soap
Sunglasses / sunscreen?
Check weather
Ch eservation
* + new underwear & socks
x
y
on
off
go
and
get
call
+
more
then
Don't forget: call mom

6 Dynamic Task Parallelism

The Dynamic Task Parallelism pattern is applicable to problems that are solved by first solving smaller, related 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. A sequential algorithm that uses recursion can easily be transformed into a computation that uses dynamic task parallelism.

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

Dynamic task parallelism is similar to recursion. Tasks create subtasks on the fly to solve subproblems as needed.

Applications that use data structures such as trees and graphs are typical examples of where you can use dynamic task parallelism. It's also used for applications that have geographic or geometric aspects, where the problem can be partitioned spatially. Dynamic task parallelism differs from the techniques that have been presented so far in this book. It is distinguished by the fact that tasks are added to the work queue as the computation proceeds.

The Basics

The following code shows a binary tree.

```
template<typename T>
struct TreeNode
{
private:
  T m_data;
  shared_ptr<TreeNode<T>> m_left;
  shared_ptr<TreeNode<T>> m_right;
  // ...
}
```

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.

```
template<typename Func>
static void SequentialWalk(shared_ptr<TreeNode<T>> node,
                           Func action)
{
  if (nullptr == node) return;

  action(node->Data());
  SequentialWalk(node->Left(), action);
  SequentialWalk(node->Right(), action);
}
```

The **SequentialWalk** applies the function **action** to each node in the tree in depth-first order. You can also use parallel tasks to walk the tree. This is shown in the following code.

```
template<typename Func>
static void ParallelWalk(shared_ptr<TreeNode<T>> node,
                         Func action)
{
  if (nullptr == node) return;

  parallel_invoke(
    [&node, &action] { action(node->Data()); },
    [&node, &action]
    {
      Tree<T>::ParallelWalk(node->Left(), action);
    },
    [&node, &action]
    {
      Tree<T>::ParallelWalk(node->Right(), action);
    }
  );
}
```

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

When you use dynamic task parallelism to perform a tree walk, you no longer visit nodes in a predictable order. If you need to visit nodes in a sequential order, such as with a preorder, inorder, or postorder traversal, you may want to consider the Pipeline pattern that's described in Chapter 7, "Pipelines."

In this example, the number of tasks is three times the number of nodes in the tree. In an actual scenario, the number of nodes could be very large. The Parallel Pattern Library (PPL) is designed to handle this situation, but you may want to read the section, "Design Notes," later in 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 into sublists, and then it 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(VectorIter begin,
                                VectorIter end,
                                long threshold)
{
  if (distance(begin, end) <= threshold)
  {
    InsertionSort(begin, end);
  }
  else
  {
    VectorIter pivot = partition(begin + 1,
                                 end,
                                 bind2nd(less<int>(), *begin));
    iter_swap(begin, pivot-1);
    SequentialQuickSort(begin, pivot - 1, threshold);
    SequentialQuickSort(pivot, end, threshold);
  }
}
```

In this example, the **VectorIter** typedef expands to the **vector<int>::iterator** method. This method sorts a **vector<int>** instance in place, instead of returning a sorted array. The **begin** and **end** arguments identify the segment that will 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 less than or equal to **threshold**, which is set in a global variable. This optimization applies equally to the sequential and parallel versions of the QuickSort algorithm.

This example uses iterator conventions from the Standard Template Library (STL).

If a segment is longer than **threshold**, the recursive algorithm is used. The **std::partition** method moves all the array elements that are not greater than the element at **pivot** to the segment that precedes **pivot**. It leaves the elements that are greater than **pivot** in the segment that follows **pivot** (**pivot** itself may be moved). Then, the method recursively calls **SequentialQuickSort** on both segments.

The following code shows a parallel implementation of the QuickSort algorithm.

```
static void ParallelQuickSort(VectorIter begin, VectorIter end,
                              long threshold, int depthRemaining)
{
  if (distance(begin, end) <= threshold)
  {
    InsertionSort(begin, end);
  }
  else
  {
    VectorIter pivot = partition(begin + 1,
                                 end,
                                 bind2nd(less<int>(), *begin));
    iter_swap(begin, pivot-1);
    if (depthRemaining > 0)
    {
      parallel_invoke(
        [begin, end, pivot, depthRemaining, threshold] {
          Sort::ParallelQuickSort(begin, pivot - 1,
                                  depthRemaining - 1, threshold);
        },
        [&pivot, begin, end, depthRemaining, threshold] {
          Sort::ParallelQuickSort(pivot, end,
                                  depthRemaining - 1, threshold);
        }
      );
    }
    else
    {
      SequentialQuickSort(begin, pivot - 1, threshold);
      SequentialQuickSort(pivot, end, threshold);
    }
  }
}
```

The parallel 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 an additional optimization besides using insertion sort for subsequences of small size. 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 created only when this argument exceeds zero. The following code shows how to calculate an appropriate depth (that is, the **depthRemaining** argument) from the number of processors.

```
static void ParallelQuickSort(vector<int>& a, long threshold)
{
  const int maxTasks =
    CurrentScheduler::Get()->GetNumberOfVirtualProcessors();

  ParallelQuickSort(a.begin(), a.end(),
    (int)LogN(float(maxTasks), 2.0f) + 4, threshold);
}
```

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. Using arbitrary, unsorted pivots produces segments of unequal size (in fact, the sizes can vary widely). The processing time required to sort each segment depends on the segment's size; therefore, you can expect tasks that are created by using pivots to divide segments to be of uneven duration. To compensate for the uneven durations of the tasks, the formula that calculates the **depthRemaining** argument produces a starting value that will allow more tasks to be created than the number of 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 ^ **depthRemaining**. If you substitute **depthRemaining** = $\log_2$(NCores) + 4 and simplify the expression, you see that the number of tasks is 16 x NCores. (Recall that for any value **a**, 2 ^ (**a** + 4) is the same as 16 times 2^**a** and that if **a** = $\log_2$(**b**), 2^**a** = **b**.)

For other algorithms you might want to use a **depthRemaining** value of 2 or 3, which would correspond to a limit on the number of tasks to 4 x NCores and 8 x NCores respectively. The number of tasks you choose depends on how unequal in duration you expect your tasks to be. The more variability in task durations, the more tasks you will probably want.

Limiting the number of subtasks by measuring the recursion depth is an extremely important technique for ensuring that an appropriate amount of potential parallelism will be introduced. Too many tasks could introduce task-related overhead; too few would result in underutilization of available cores.

> **Note**: *The QuickSort example that is shown in this section was selected to illustrate the principles of dynamic task parallelism. As a sorting algorithm it may or may not be what you want. There are other examples of parallel sorting algorithms in the ConcRT Extras sample pack that may be better suited to your application.*

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 also parallel algorithms for other types of traversals. These techniques rely on concurrent collections to keep track of the remaining work to be done. Here's an example.

```
template<typename T, typename Func>
void ParallelWhileNotEmpty1(
  vector<shared_ptr<TreeNode<T>>> initialValues,
  Func body)
{
  concurrent_vector<shared_ptr<TreeNode<T>>>
    from(initialValues.size());
  for (size_t i = 0; i < initialValues.size(); i++)
    from[i] = initialValues[i];

  while(!from.empty())
  {
    concurrent_vector<shared_ptr<TreeNode<T>>> to;
    function<void (shared_ptr<TreeNode<T>>)> addMethod =
        [&to](shared_ptr<TreeNode<T>> n) { to.push_back(n); };
    parallel_for_each(from.cbegin(), from.cend(),
      [&body, &addMethod](shared_ptr<TreeNode<T>> item)
      {
        body(item, addMethod);
      }
    );
    from = to;
  }
}
```

The **ParallelWhileNotEmpty1** method shows how you can use **parallel_for_each** to process a collection of values that may grow over time. While the **ParallelWhileNotEmpty1** method processes the initial values, additional values to process may be discovered. The additional values are placed in the **to** queue. After the first batch of values is processed, the method starts processing the additional values, which may again result in more values to process. This processing repeats until no additional values are produced.

The **concurrent_vector** class is provided by the Concurrency Runtime as a concurrency-safe implementation of a vector type.

The **ParallelWalkWithWhileNotEmpty1** method uses the **ParallelWhileNotEmpty1** method to walk a binary tree. This is shown in the following code example.

```
template<typename T, typename Func>
void ParallelWalkWithWhileNotEmpty1(shared_ptr<TreeNode<T>> node,
                                    Func action)
{
  if (nullptr == node)
    return;
  vector<shared_ptr<TreeNode<T>>> nodes;
  nodes.push_back(node);

  ParallelWhileNotEmpty1(nodes,
    /* Func body */ [&action](shared_ptr<TreeNode<T>> item,
    function<void (shared_ptr<TreeNode<T>>)> addMethod)
    {
      if (nullptr != item->Left()) addMethod(item->Left());
      if (nullptr != item->Right()) addMethod(item->Right());
      action(item->Data());
    });
}
```

A website tool that checks links is an example of an appropriate place to use the **ParallelWalkWithWhileNotEmpty1** method. The tool 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.

ADDING TASKS TO A PENDING WAIT CONTEXT

In most cases you invoke the **wait** method of a task group only after all of the tasks have been created. In some cases, it is useful to create new tasks after the **wait** method has been invoked but before the previously created tasks have finished. A typical example arises when you are traversing nodes of a graph.

Here is an example of a function that uses parallel tasks of a single task group to traverse a tree.

```
template<typename T, typename Func>
void ParallelSubtreeHandler(task_group& tg,
                            shared_ptr<TreeNode<T>> node,
                            Func action)
{
  while (nullptr != node)
  {
    // Start up processing the left subtree in a new task
    if (nullptr != node->Left())
    {
      tg.run([&tg, node, action]() {
        ParallelSubtreeHandler(tg, node->Left(), action);
      });
    }

    // Process this node
    tg.run([node, action]() { action(node->Data()); });

    // Walk down the right side of the tree
    node = node->Right();
  }
}
```

The **ParallelSubtreeHandler** is called from the top-level function that is shown below.

```
template<typename T, typename Func>
void ParallelTreeUnwinding(shared_ptr<TreeNode<T>> node,
                           Func action)
{
  if (nullptr == node)
    return;

  task_group tg;

  ParallelSubtreeHandler(tg, node, action);

  tg.wait();
}
```

The **ParallelTreeUnwinding** function creates a single task group that is used by the code that handles the subtrees. Here is a code example that shows how the function is called. The lambda expression simply records all the nodes that are visited.

```
  const Tree<int>& tree = ...
  concurrent_vector<int> result;

  ParallelTreeUnwinding(tree.Root(),
    [&result](int itemData)
    {
      DoCpuIntensiveOperation(Time);
      result.push_back(itemData);
    });
```

Dynamically adding new tasks to the task group allows you to use the **task_group** object to track unprocessed nodes instead of using a **concurrent_vector**, as was done in the **ParallelWhileNotEmpty1** code example. The **task_group** also makes the code easier to read because it eliminates the need for a separate data structure to hold unprocessed nodes. Rather than completing when all unprocessed nodes have been removed from the **concurrent_vector**, this example completes when the **task_group** contains no more incomplete tasks.

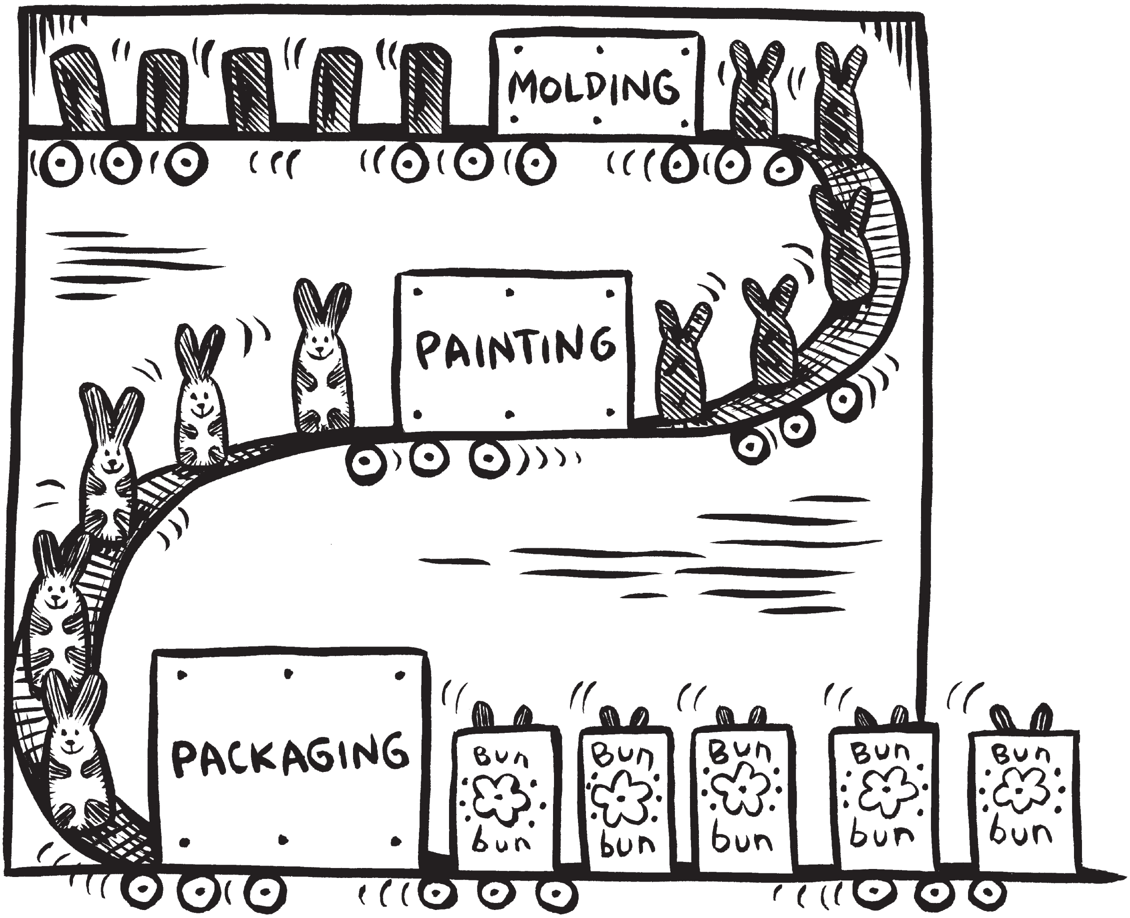
Exercises

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 then observe the execution times for the sequential version to sort different array sizes. What do you expect to see? What's the best value for **threshold** on your system?
2. Use the command line argument to vary the array size, and then 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, besides the number of cores, to limit the number of tasks.

Further Reading

Heineman et al. discuss additional variations on QuickSort and other sorting algorithms.

Heineman, George T., Gary Pollice, and Stanley Selkow. *Algorithms in a Nutshell*. O'Reilly Media, 2008.


MOLDING
PAINTING
PACKAGING
Bun bun
Bun bun
Bun bun
Bun bun
Bun bun

7

Pipelines

The Pipeline pattern allows you to achieve parallelism in cases where there are data dependencies that would prevent you from using a parallel loop. A pipeline is composed of a linear series of producer/consumer stages, where each stage depends on the output of its predecessor. The pipeline is an example of a more general category known as a dataflow network. A dataflow network decomposes computation into cooperating components that communicate by sending and receiving messages.

There are a variety of techniques for implementing pipelines. Those described in this chapter use in-process messaging blocks and asynchronous agents, both of which are provided by the Asynchronous Agents Library.

A pipeline's data flows from its first stage, through intermediate stages and then to a final stage. The stages of a pipeline execute concurrently, but in contrast to a parallel loop, the overall effect is to process the input data in a fixed 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, but more than one item is assembled at a time.

Pipelines occur in many applications. You can use a pipeline when data elements are received from a real-time event stream, such as values on stock ticker tapes, user-generated mouse click events, or packets that arrive over the network. You can also use pipelines to process elements from a data stream, as is done with compression and encryption, or to apply transformation operations to streams of video frames. In all of these cases, it's important that the data elements are processed in sequential order. You can't use a parallel loop for these cases because a parallel loop doesn't preserve the processing order.

A dataflow network is a set of asynchronous components that use messages to communicate with one other.

A data pipeline is a sequence of asynchronous components that are connected by message buffers. Each stage of the pipeline receives input from its predecessor. Use a pipeline when data dependencies prevent you from a using a parallel loop.

In this chapter, the Asynchronous Agents Library's function is to improve performance when there are multiple cores available, but it has other uses. More generally, agents with asynchronous communication can be used to implement concurrency and as a way to organize applications such as simulations.

Types of Messaging Blocks

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

The Asynchronous Agents Library includes the following types of messaging blocks, which are useful in a variety of situations:

- **unbounded_buffer**: a concurrent queue of unbounded size.
- **overwrite_buffer**: a single value that can be updated many times.
- **single_assignment**: a single value that can be set just once.
- **call**: a function that is invoked whenever a value is added to the messaging block.
- **transformer**: a function that is invoked whenever a value is added to the messaging block; the function's return value is added to an output messaging block.
- **choice**: selects a message from a set of sources.
- **join**: waits for more than one source before proceeding.
- **multitype_join**: same as join, except that inputs may have multiple message types.
- **timer**: produces messages based on time intervals.

In this chapter you'll see examples of four types of messaging blocks. They are the **unbounded_buffer<T>**, **overwrite_buffer<T>**, **transformer<T, S>** and **call<T>** messaging blocks. It's also possible to implement your own messaging blocks.

The Basics

The Asynchronous Agents Library provides messaging blocks, agents, and functions that send and receive messages.

In the Asynchronous Agents Library, the buffers that connect stages of a software pipeline are usually messaging blocks, such as instances of the **unbounded_buffer<T>** class. Although the buffer itself is unbounded, the pipeline includes a feedback mechanism that limits the number of pending items. The stages of the pipeline can themselves be implemented with instances of the **agent** class.

Figure 1 illustrates 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 combines the words and phrases into complete sentences, and then it writes the sentences to a disk file.

This section describes an agent-based approach to pipelines that requires a dedicated thread for each pipeline stage. Each stage uses an instance of the **agent** class. See "Asynchronous Pipelines" in this chapter for an important variation of this pattern that does not dedicate a thread to each pipeline stage.

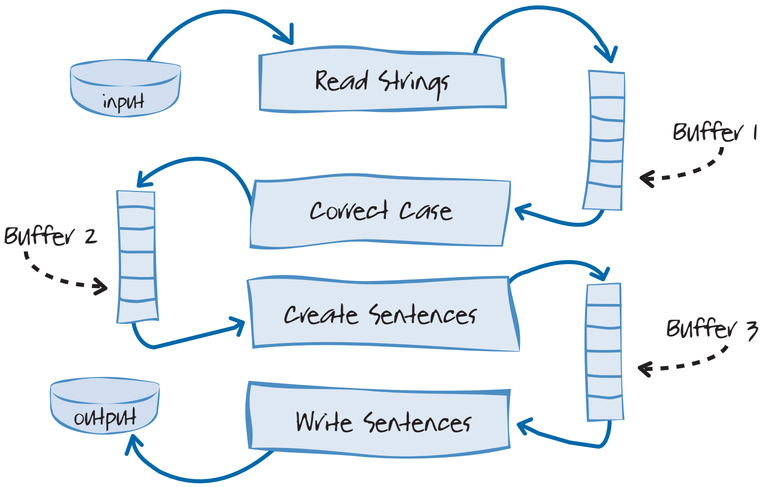


FIGURE 1
Sample pipeline

Stages of the pipeline read from a dedicated input, or *source*, and write to a particular output, or *target*. For example, the "Correct Case" stage uses buffer 1 as its source and writes to buffer 2 as its target. All the stages of the pipeline can execute at the same time because the three messaging blocks buffer any shared inputs and outputs. If there are four available cores, the four stages can run in parallel.

Stages in the pipeline block (that is, wait) on inputs. An input wait is familiar from other programming contexts—if an enumeration or a stream doesn't have a value, the consumer of that enumeration or stream waits until a value is available or an end–of-file condition occurs. Using buffers that hold more than one value at a time compensates for variability in the time it takes to process each value. Buffers allow stages of the pipeline to be asynchronous.

> **Note**: *When using the* **unbounded_buffer<T>** *class you should define a special value as your end-of-file token. This special value is sometimes called the* **sentinel value**. *When using sentinel values you must be careful that the end-of-file signal can never occur as one of the regular messages. This example uses the value given by the* **PhraseSource:: FinishedSentinel ()** *static method to signal the end of the sequence of values.*
>
> *Choosing a sentinel value can be harder than it seems at first. It's often the case that all values of the type* **T** *have meaning as valid payloads of an* **unbounded_buffer<T>** *instance. For example, if your payload type is a string, you might be tempted to*

> *use the empty string as the sentinel value, but this would only be safe if you can guarantee that the empty string is never used as a normal value to be processed by the pipeline. In practice, the null pointer is often used as the sentinel value.*

The following code demonstrates how to implement a pipeline that uses the **unbounded_buffer<T>** class for the buffers and the **agent** class for the stages of the pipeline.

```
unbounded_buffer<wstring> buffer1;
unbounded_buffer<wstring> buffer2;
unbounded_buffer<wstring> buffer3;
PipelineGovernor governor(g_sentencePipelineLimit);

ReadStringsAgent agent1(seed, g_sentenceMax, governor, buffer1);
CorrectCaseAgent agent2(buffer1, buffer2);
CreateSentencesAgent agent3(buffer2, buffer3);
WriteSentencesAgent agent4(g_targetSentence, g_pipelineResults,
                           governor, buffer3);

agent1.start();
agent2.start();
agent3.start();
agent4.start();

agent* agents[4] = { &agent1, &agent2, &agent3, &agent4 };
agent::wait_for_all(4, agents);
```

The first stage generates the input strings and places them in **buffer1**. The second stage transforms the strings. The third stage combines the strings into sentences. The final stage writes the corrected sentences to a file. Note that in this example, the number of input elements (words) is not the same as the number of output elements (sentences); part of the pipeline's functionality is to combine words into sentences.

Use an instance of the **unbounded_buffer<T>** *class to connect stages of a pipeline.*

References to input and output buffers are passed to each agent's constructor. For example, the second stage, **agent2**, which is an instance of the **CorrectCaseAgent** class, uses **buffer1** as its input and **buffer2** as its output. Figure 1 illustrates the resulting connections among the agents.

Stages of the pipeline may not take exactly the same amount of time to process each element. To prevent an excess number of buffered elements, the pipeline uses a mechanism for limiting the number of data elements that may be pending at any given time. This mechanism is provided by the **PipelineGovernor** class, which is defined in the Utilities folder of the Microsoft® Visual Studio® solution for the example code. Only the first and the last stages of the pipeline need to interact with the pipeline governor. When an item exits the pipeline, the last stage of the pipeline asks the governor instance to decrement a counter of in-flight elements. Before placing a new element into the pipeline, the first stage of the pipeline checks to see that the maximum number of in-flight elements hasn't been exceeded. If this is the case, the pipeline stage asks the governor instance to increment the count of in-flight elements and then places the element into the first buffer. If the pipeline is full, the first stage waits for the governor to signal when space in the pipeline becomes available. Internally, the governor uses a messaging block to synchronize between the last stage of the pipeline and the first.

Use a governor to limit the number of in-flight elements in the pipeline.

If you don't use a governor to limit the in-flight elements in a pipeline, their numbers can grow without bound. Adding too many elements at once can result in performance problems or out-of-memory errors. The governor provided in this example is just one approach; you may want to consider others, depending on the needs of your application.

After all of the agents have been created and connected to their respective sources and targets, the code invokes the **start** method of each agent.

Call an agent's **start** *method to begin execution.*

The code calls the **wait_for_all** static method of the **agent** class to defer cleanup until after all stages have completed processing their inputs. In this code example, the memory for the agents is allocated on the stack in the current context, so you can't exit the current context until the agents have finished their work.

Use the **wait_for_all** *method to allow agents to finish processing before proceeding in the current context.*

The first stage of the pipeline is implemented by the **ReadStrings Agent** class. This agent includes a sequential loop that writes to its output buffer. Here is the code for the agent's **run** method, which specifies what the agent should do after it has started.

An agent's **run** *method is invoked when the agent is started. The agent terminates when the* **run** *method invokes the agent's* **done** *method and exits.*

```
class ReadStringsAgent : public agent
{
  // ...
  void run()
  {
    PhraseSource source(m_seed, m_numberOfSentences);
    wstring inputPhrase;
    do
    {
      // ...
      inputPhrase = source.Next();

      // Limit whole sentences in the pipeline not phrases.
      if (phrase == L".")
        m_governor.WaitForAvailablePipelineSlot();
      asend(m_phraseOutput, inputPhrase);
    } while (inputPhrase != PhraseSource::FinishedSentinel());
    done();
  }
};
```

Applications that use the Pipeline pattern require that elements be processed in order. If the processing order is not important, you may consider using another pattern, such as the Parallel Loop pattern, to process your data.

The sequential **do** loop populates the output buffer with values. The loop is sequential in order to preserve the order of elements that are processed, which is one of the requirements of applications that use the Pipeline pattern. The values come from an external data source that's represented by the **PhraseSource** class. Successive values are retrieved by calling the phrase source object's **Next** method.

The **asend** function, named for "asynchronous send," is provided by the Asynchronous Agents Library in the agents.h header file. It schedules a task to propagate the data to the target messaging block.

Use the **send** *or* **asend** *function to place data into the next stage's input buffer.*

A producer can use either the **send** or **asend** function to relay values, which are also referred to as messages, to the target messaging block. The **send** function blocks the current context until the target messaging block accepts or rejects the message. The **send** function returns **true** if the message was accepted and **false** otherwise. The **asend** function does not wait for the target to accept or decline the message before it returns. Use **send** when you must ensure that the value reaches its destination before proceeding in the current context. Use **asend** when you want a "fire and forget" style of message passing.

It's possible in some cases that a message won't be accepted by the target messaging block. This can occur, for example, if you attempt to send more than one message to a single assignment buffer. The second message won't be accepted by the buffer. Additionally, messaging block constructors allow you to provide a custom filter function that determines whether an incoming message will be accepted by that messaging block.

The first stage of the pipeline invokes the **WaitForAvailablePipelineSlot** method of the governor before adding a new element. If the pipeline is full, the governor will block until space becomes available.

The second stage of the pipeline capitalizes words if necessary. It consumes values from its source buffer, transforms them, and places the transformed values into its target buffer. The following code shows the **run** method of the **CorrectCaseAgent** class.

```
class CorrectCaseAgent : public agent
{
  // ...

  void run()
  {
    wstring inputPhrase;
    while(true)
    {
      inputPhrase = receive(m_phraseInput);
      if (inputPhrase == PhraseSource::FinishedSentinel())
      {
        asend(m_phraseOutput, inputPhrase);
        break;
      }
      // ... transform phrase by possibly capitalizing it
      asend(m_phraseOutput, outputPhrase);
    }
    done();
  }
};
```

Use the **receive** *function to wait for input from a messaging block. Use a sentinel value to indicate shutdown.*

The important point of this code is that the **receive** function is called on the messaging block that acts as the source. This allows the consuming agent to wait for values to become available from the producer of those values. The code iterates until the special end-of-file sentinel value is seen.

Some messaging blocks allow multiple consumers and producers.

Some messaging blocks, including the **unbounded_buffer<T>** *class, support multiple producers and consumers.*

The third stage of the pipeline uses the **run** method of the **CreateSentencesAgent** class to read a sequence of phrases and combine them into sentences. When it encounters a phrase that ends with the period character, it knows that the end of the sentence has been reached and writes the sentence to the target messaging buffer. The **CreateSentencesAgent** class shows that it's not always necessary for pipeline stages to consume and produce an identical number of values.

Pipeline stages can summarize or combine values. There's not always a one-to-one correspondence of inputs and transformed outputs in each stage of a pipeline.

The last stage of the pipeline, which is implemented by the **WriteSentencesAgent** class, consumes values from its predecessor in the pipeline but doesn't produce any values. Instead, it writes to an output file stream. Here's the code for the agent's **run** method.

```
class WriteSentencesAgent : public agent
{
  // ...

  void run()
  {
    wofstream fout;
    fout.open(m_outputPath);
    wstring sentence;
    while(true)
    {
      sentence = receive(m_sentenceInput);
      if (sentence == PhraseSource::FinishedSentinel())
        break;
      if (sentence == m_targetSentence)
```

```
        sentence.append(L"       Success!");
      fout << m_currentSentenceCount++ << L" "
                                     << sentence.c_str() << endl;
      sentence.clear();
      m_governor.FreePipelineSlot();

      OutputProgress(m_currentSentenceCount);
    }
    fout.close();
    done();
  }
};
```

The agent reads sentences it receives and compares them to the desired target sentence, **m_targetSentence**. It writes all generated sentences to a file and flags the ones that match the target.

The agent invokes the **FreePipelineSlot** method of the pipeline's governor to signal that space in the pipeline has become available.

One reason that agents and messaging blocks make it easy to write pipelines is that you can rely on familiar sequential techniques such as iteration. There is some synchronization, but it's hidden inside the implementation of the **unbounded_buffer<T>** class.

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

An Example

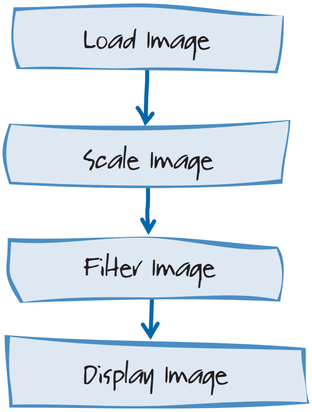
You can't use a parallel loop for this example because the application requires that images be processed in sequence. Parallel loops don't guarantee any particular processing order.

The online samples include an application named ImagePipeline. This application takes a directory of JPEG images and generates thumbnail versions, which are also post-processed with an image-enhancing filter. The resulting processed images are displayed as a slideshow, in alphabetical file name 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, noise is added to the image to create a speckling effect, and then the processed image is displayed as the next picture in the slideshow. Figure 2 illustrates this sequence.

FIGURE 2
Sequential image processing



Here's the code for the sequential version.

```
vector<wstring> filenames = ...

int sequence = kFirstImage;

for_each_infinite(filenames.cbegin(), filenames.cend(),
  [this, &sequence, offset] (wstring file)->bool
{
  ImageInfoPtr pInfo = LoadImage(sequence++, file, offset);
  ScaleImage(pInfo, m_imageDisplaySize);
  FilterImage(pInfo, m_noiseLevel);
  DisplayImage(pInfo);
  return IsCancellationPending();
});
```

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 the capture of performance measurements is omitted. You can refer to the online samples to see those details.

The type **ImageInfoPtr** is a **typedef** abbreviation for **shared_ptr<ImageInfo>**, a Standard Template Library (STL) shared pointer to an **ImageInfo** instance. The **ImageInfo** class is a data record that contains the image bitmap to be processed. Pointers are used as a way to pass data between stages of the pipeline without the overhead of copying the image bitmaps. Buffering ensures that no locks are needed for this "shared" data; each **ImageInfo** instance is guaranteed to be accessed by only one stage of the pipeline at a time.

The function **for_each_infinite** is a helper function that is defined by the sample code. It invokes a function (in this case a function object given by a lambda expression) on each element of a sequence.

When the loop reaches the end of the sequence, it restarts at the beginning; however, if any invocation of the function returns true, iteration stops. In this example, the only way to exit the loop is by throwing an exception or when **IsCancellationPending()** returns **true**. See "Variations" in this chapter for more information on the cancellation model that is used in this example.

THE IMAGE PIPELINE

The sequential loop can process only 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 (like video frames), and within each step are substeps that require inputs from previous substeps. You can't display an image until after the filter is applied to it. You can't apply the filter until after the image is scaled to thumbnail size. You can't do the scaling until after the original image loads.

Even with such strong sequential constraints, the Pipeline pattern can introduce parallelism into this example. Each image will still pass through all four stages, in sequence, but the stages themselves can work on different images at the same time. Figure 3 illustrates the image pipeline.

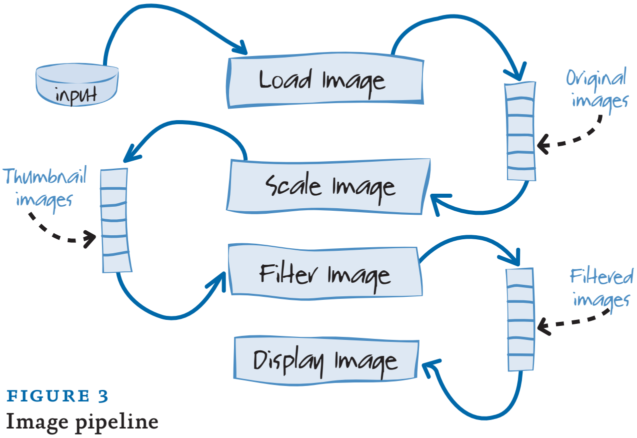


FIGURE 3
Image pipeline

The following code from the ImageAgentPipelineControlFlow.h file shows the parallel version.

```
unbounded_buffer<ImageInfoPtr> buffer1;
unbounded_buffer<ImageInfoPtr> buffer2;
unbounded_buffer<ImageInfoPtr> buffer3;
```

```
ImageScalerAgent   imageScaler(..., buffer1, buffer2);
ImageFiltererAgent imageFilterer(..., buffer2, buffer3);
ImageDisplayAgent  imageDisplayer(..., m_governor,
                              ..., buffer3);

imageScaler.start();
imageFilterer.start();
imageDisplayer.start();

vector<wstring> filenames = ...

int sequence = kFirstImage;

for_each_infinite(filenames.cbegin(), filenames.cend(),
  [this, offset, &buffer1, &sequence] (wstring file)->bool
{
  ImageInfoPtr pInfo = this->LoadImage(sequence++, file, offset);
  if (nullptr == pInfo)
    return true;
  m_governor.WaitForAvailablePipelineSlot();
  asend(buffer1, pInfo);

  return IsCancellationPending();
});

m_governor.WaitForEmptyPipeline();
asend<ImageInfoPtr>(buffer1, nullptr);
agent* agents[3]={&imageScaler, &imageFilterer, &imageDisplayer};
agent::wait_for_all(3, agents);
```

There are three **unbounded_buffer<T>** messaging blocks that act as buffers between the stages of the pipeline. A call to the agent's **start** method launches each processing stage.

The code iterates through the file names to be processed and uses the **LoadImage** method to load each image into memory. This step is the same as in the sequential version of the code. However, instead of proceeding directly to the next operation, the code places a shared pointer to the newly loaded image's data into messaging block **buffer1**, which is the input source of the **ImageScalerAgent** object. The image scaling agent receives the image and begins to process it. Meanwhile, the loop continues with its next iteration and begins loading the next image.

Like the text processing example described in the previous section of this chapter, the image processing example uses the

Be careful about copying large amounts of data between pipeline stages. For example, copying large bitmapped images between stages will unnecessarily consume a large amount of memory. Instead, pass a pointer to a data structure.

PipelineGovernor utility class to limit the maximum number of in-flight elements in the pipeline.

(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.)

PERFORMANCE CHARACTERISTICS

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

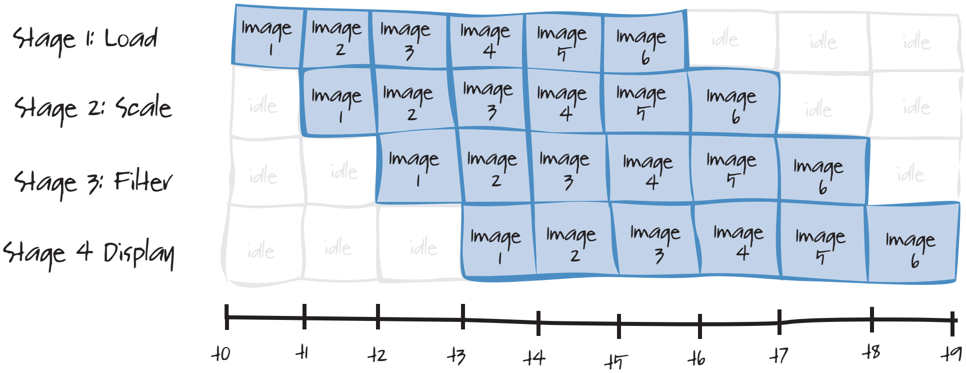

Stage 1: Load
Image 1 | Image 2 | Image 3 | Image 4 | Image 5 | Image 6 | idle | idle | idle
Stage 2: Scale
idle | Image 1 | Image 2 | Image 3 | Image 4 | Image 5 | Image 6 | idle | idle
Stage 3: Filter
idle | idle | Image 1 | Image 2 | Image 3 | Image 4 | Image 5 | Image 6 | idle
Stage 4 Display
idle | idle | idle | Image 1 | Image 2 | Image 3 | Image 4 | Image 5 | Image 6
t0
t1
t2
t3
t4
t5
t6
t7
t8
t9

FIGURE 4

Image pipeline with stages of equal speed

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 Figure 4, t_1 occurs after T units of time have elapsed, t_2 after 2 x T units of time have elapsed, and so on.

If there are enough available cores to allow the pipeline's tasks to run in parallel, Figure 4 shows that the expected execution time for six images in a pipeline with four stages is approximately 9 x T. In contrast, the sequential version takes approximately 24 x T because each of the 24 steps must be processed one after another.

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

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

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

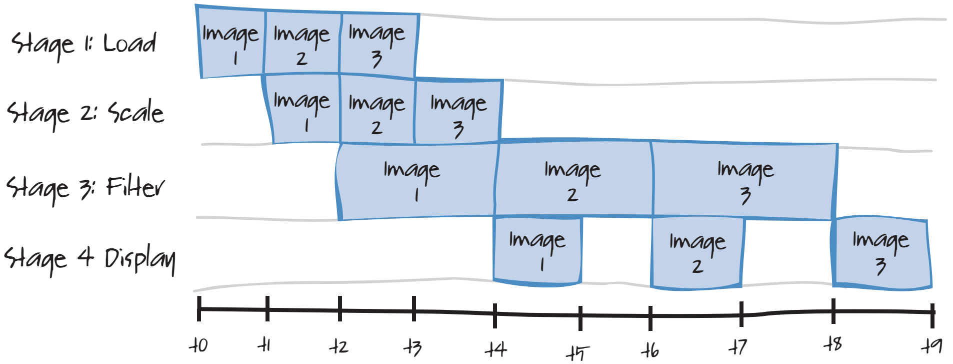


FIGURE 5
Image pipeline with unequal stages

When one of the stages takes 2 x 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 image is 2 x 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, you can see this effect for yourself. The ImagePipeline sample has a user interface (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's 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 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.

ASYNCHRONOUS PIPELINES

The pipelines that have been described so far are synchronous. Producers and consumers are long-running tasks (implemented with the

agent class) that internally use sequential loops to read inputs and write outputs. Agents whose run methods contain sequential loops are sometimes called control-flow agents. They require a dedicated thread for each stage of the pipeline. Dedicating a thread to each stage makes sense if the pipeline follows the recommended practice of dividing the work into stages of equal duration. With an equal division of work, each of the threads will be continuously busy for the duration of the pipeline's run. See the "Performance Characteristics" section of this chapter for more information about the ideal allocation of work to pipeline stages.

You can also have an asynchronous pipeline, where tasks are only created after data becomes available. This style of implementation is more oriented toward dataflow than control flow. The differences between the control flow and dataflow approaches are a matter of coding preference. However, there are some functional and performance distinctions between the two approaches.

The **transformer** *and* **call** *classes are message buffers that act like agents, but unlike agents they don't require dedicated threads. Use* **transformer** *and* **call** *objects to implement asynchronous pipelines.*

Asynchronous pipelines are implemented using the **transformer** class and the **call** class. These classes are messaging blocks in the Asynchronous Agents Library. The **transformer** class and **call** class are queues that a producer puts data into; if there's currently no task processing the queue when data arrives, a new task is created to process the queue, and it's active as long as there's incoming data to process. If it ever finds that there is no more data, the task goes away. If more data arrives, a new task starts. In other words, the **transformer** class or **call** class is a message buffer that acts like an agent but creates tasks as needed to process incoming data values instead of dedicating a thread to this purpose.

Asynchronous pipelines are useful in cases where there are many pipeline stages, and you don't want to dedicate a thread to each stage. They are also efficient in cases where you expect the pipeline to often be empty (for example, while waiting for input). In these cases, transformer messaging blocks can improve application performance due to better utilization of threads.

A drawback to asynchronous pipelines is that the code can be slightly more difficult to write and debug than the agent-based style that was shown earlier in this chapter. The asynchronous style of pipelines may require the use of a separate task scheduler instance in order to keep scheduling latency low. Asynchronous pipelines are limited to pipeline stages that have an equal number of inputs and outputs. See Appendix A for more information about task schedulers.

Here is an example of an asynchronous pipeline from the **ImageAgentPipelineDataFlow** class.

```
void Initialize()
{
  m_scaler = unique_ptr<transformer<ImageInfoPtr, ImageInfoPtr>>(
                new transformer<ImageInfoPtr, ImageInfoPtr>(
                [this](ImageInfoPtr pInfo)->ImageInfoPtr
                {
                    this->ScaleImage(pInfo, m_imageDisplaySize);
                    return pInfo;
                },
                ...
            ));

  m_filterer =
           unique_ptr<transformer<ImageInfoPtr, ImageInfoPtr>>(
                new transformer<ImageInfoPtr, ImageInfoPtr>(
                [this](ImageInfoPtr pInfo)->ImageInfoPtr
                {
                    this->FilterImage(pInfo, m_noiseLevel);
                    return pInfo;
                },
                ...
            ));

  m_displayer = unique_ptr<call<ImageInfoPtr>>(
                new call<ImageInfoPtr>(
                [this](ImageInfoPtr pInfo)
                {
                    this->DisplayImage(pInfo);
                    m_governor.FreePipelineSlot();
                },
                ...
            ));

  m_scaler->link_target(m_filterer.get());
  m_filterer->link_target(m_displayer.get());
}
```

This code creates **transformer** objects that receive and send **ImageInfoPtr** objects. Each transformer declaration specifies a lambda function. The lambda function takes an image pointer as its argument, performs an operation on it, and returns the pointer to the modified image. A transformer has a one-to-one relationship between input and output messages. In other words, for each input value, the transformation function must return a single corresponding output value.

The final stage in the pipeline uses a **call** messaging block. Call messaging blocks are similar to transformers but have no output message. The **m_displayer** variable contains a lambda function that displays the image and updates the pipeline governor but does not produce any output.

You provide a function that performs a transformation on input values as an argument to the transformer class's constructor. The transformation function is invoked by the system when inputs are available; therefore, you should be careful that all exceptions are handled within the transformation function.

Set targets of the **transformer** *and* **call** *objects using the* **link_target** *method.*

The code creates **transformer** and **call** objects that correspond to all stages of the pipeline except the first. The transformer's targets are configured by invoking the **link_target** method. You don't need to set sources because transformer and call objects are themselves a kind of messaging block; they are their own data sources.

The code shows the **run** method of the dataflow-based imaging pipeline.

```
Initialize();
vector<wstring> filenames = ...

int sequence = kFirstImage;
for_each_infinite(filenames.cbegin(), filenames.cend(),
  [this, offset, &sequence](wstring file)->bool
{
  ImageInfoPtr pInfo = this->LoadImage(sequence++, file, offset);
  if (nullptr == pInfo)
    return true;
  m_governor.WaitForAvailablePipelineSlot();
  asend(m_scaler.get(), pInfo);

  return IsCancellationPending();
});

// Allow subsequent stages to terminate
m_governor.WaitForEmptyPipeline();
done();
```

If you compare the code sample with the **run** method of the agent-based image pipeline that was described in the previous section, you can see similarities. In both, a sequential loop loads images and sends them to a messaging block that is the data source for the image scaling stage. In both, the number of in-flight elements is limited by a call to a governor object.

The difference between the two approaches is seen at run time. With the asynchronous pipeline, a new task is created whenever an empty **transformer** messaging block receives a new element (by means of the **send** or **asend** functions). This new task invokes the transformation function (that was passed to it as the first argument of the constructor), and then sends the return value of the transformation function to the messaging block that has been configured as the transformer's target.

At run time, **transformer** *and* **call** *messaging blocks create tasks on demand to process any queued items. An active task is present only when there are elements to process.*

A **call** messaging block behaves like a **transformer** messaging block, except that no target is involved. The **call** block's function is invoked on the new input element. It does not return a value.

CANCELING A PIPELINE

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

A natural place to check for cancellation is at the end of the loop that processes items from the input source of a pipeline stage. In the image processing example, you'll see that the **ImagePipelineDlg** class that controls the user interface provides an instance of the **overwrite_buffer<bool>** class. This object signals that cancellation has been requested from the user interface. Each stage of the pipeline periodically checks the value of the overwrite buffer to see if cancellation has been requested.

For example, the base class, **AgentBase**, which is used to implement the agents in the image processing example, includes the following definitions.

```
class AgentBase : public agent
{
  private:
    // ...
    ISource<bool>& m_cancellationSource;

  public:
    // ...
    AgentBase(HWND dialog,
              ISource<bool>& cancellationSource, ...) :
        m_dialogWindow(dialog),
        m_cancellationSource(cancellationSource),
        ...
    {
      // ...
    }
    bool IsCancellationPending() const
    {
        return ... || receive(m_cancellationSource);
    }

    // ...
}
```

This code shows how the external environment (in this case, a request from the application's user interface) can signal that the pipeline should cancel processing. The agent's constructor includes a parameter that takes an **ISource<bool>** object as a cancellation source. The cancellation source is implemented as an instance of the **overwrite_buffer<bool>** class. Its value is false unless the user requests cancellation, and then the value of the cancellation source becomes **true**. Individual pipeline operations invoke the **IsCancellationPending()** method to effect an orderly shutdown.

How you implement cancellation can affect the performance of your application. You should be careful not to check for cancellation within tight loops, but you should also check for cancellation often enough to keep cancellation latency from becoming noticeable to the user. The Image Pipeline checks for cancellation at the beginning of each pipeline step which, depending on the speed of your computer, corresponds to one check every few hundred milliseconds for each agent thread. Profiling your application can help you determine if polling for cancellation is harming performance.

When there is an 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.

HANDLING PIPELINE EXCEPTIONS

Exceptions are similar to cancellations. The difference between the two is that when an 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.

The base class, **AgentBase,** in the image processing example uses an instance of the **overwrite_buffer<bool>** class to alert all pipeline stages when an exception in one stage has occurred. This is shown in the following code.

```
class AgentBase : public agent
{
  private:
    // ...
    mutable overwrite_buffer<bool> m_shutdownPending;

  public:
    // ...
    AgentBase(...) ...
    {
      send(m_shutdownPending, false);
    }

    void ShutdownOnError(Phases phase, const wstring& filePath,
                         const exception& e) const
    {
      wostringstream message;
      message << e.what();
      SendError(phase, filePath, message.str());
    }

    void SendError(Phases phase, const wstring& filePath,
                   wstring message) const
    {
      // ...
      send(m_shutdownPending, true);
      send(m_errorTarget, ErrorInfo(phase, filePath, message));
      PostMessageW(m_dialogWindow, WM_REPORTERROR, 0, 0);
    }

    bool IsCancellationPending() const
    {
        return receive(m_shutdownPending) ||
               receive(m_cancellationSource);
    }

    // ...
}
```

The stages of the pipeline invoke the application's **ShutdownOnError** method if they catch an exception. Because the pipeline stages run concurrently, the shutdown method is coded in a concurrency-safe manner. It sends values to buffers instead of updating shared variables directly.

The **ShutdownOnError** method sends the value **true** to the overwrite buffer **m_shutdownPending** to signal the other pipeline agents of the imminent shutdown. Next, the method sends a message that contains the error information to the unbounded buffer **m_errorTarget**. Finally, it sends a custom Windows message, **WM_REPORTERROR**, to notify the UI that an error needs to be processed. When the UI thread handles the Windows message, it invokes an application callback method that gets information from the **m_errorTarget** buffer and displays it in a dialog box. The information contains a text description of the exception and the name of the image file that was being processed when the exception occurred.

The **IsCancellationPending** method checks for two conditions: whether shutdown is pending due to an exception, or whether there is a user-initiated cancellation request. Two separate buffers are used because the implementation does *not* reuse the signaling mechanism provided for external cancellation. The pipeline stages can signal that an exception has occurred, but only the user can request cancellation of the operation. The reason is one of scope: operations other than the pipeline might be affected by a cancellation request. Error handling is intended to be local to the pipeline itself.

LOAD BALANCING USING MULTIPLE PRODUCERS

The **unbounded_buffer<T>** class allows you to read values from more than one producer. This feature can be used to implement load balancing for pipeline stages that take longer than other stages.

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

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's 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.

FIGURE 6
Consuming values from multiple producers

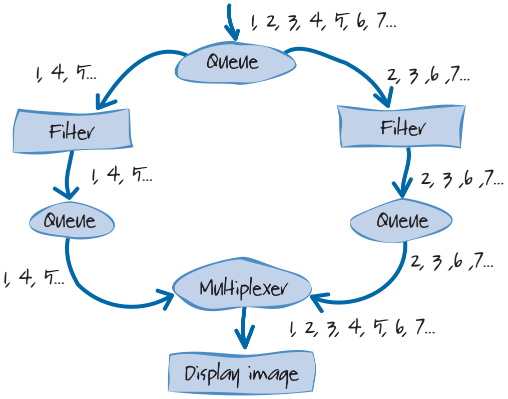


Figure 6 shows what happens when you add an additional filter task. The numbers in the figure represent the sequence numbers of the images being processed. (Recall that the images must be processed in order in this example.) Both of the filter tasks take images produced by the previous stage of the pipeline. 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 target buffer to hold the elements that it produces. The consumer of these queues is a component known as a multiplexer, which combines the inputs from all of the producers. The multiplexer provided in the sample code 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 don't 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.

Here's an example to make this more concrete. Suppose that each image has a unique sequence number that's available by invoking a data accessor method. The image numbers start with 1 and increase sequentially. As Figure 6 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 its own queue. 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're 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. 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 put 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.

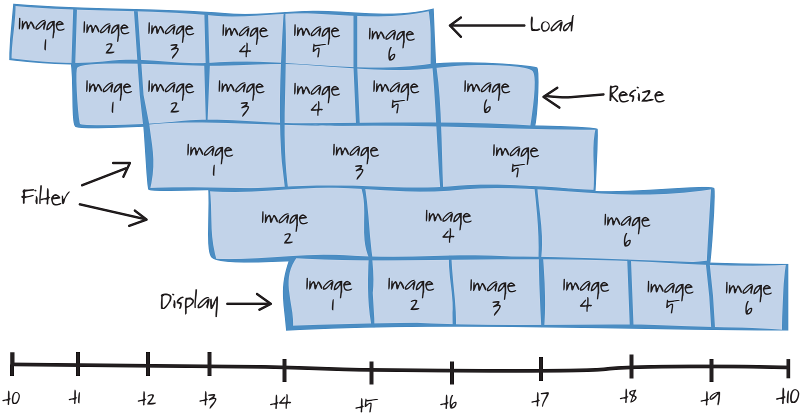


FIGURE 7
Image pipeline with load balancing

If all pipeline stages, except the filter stage, take T units of time to process an image, and the filter stage takes 2 x T units of time, 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 sample and select the **Load Balanced** radio button, you'll see this effect. The speed of the pipeline (after a suitable number of images are 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 displayed on the ImagePipeline sample's UI, indicates the overhead that's 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 message buffers and streams have some similarities. It's sometimes useful to treat a message buffer 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. Suppose that you want to compress a file and then encrypt it. Both compression and encryption are supported by native libraries, but the functions' parameter lists expect streams, not messaging blocks. It's possible to implement a stream whose underlying implementation relies on agents and messaging blocks.

Anti-Patterns

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

COPYING LARGE AMOUNTS OF DATA BETWEEN PIPELINE STAGES

If your data structures are large, you should pass pointers to data, and not the data itself, down the pipeline. Use the Resource Acquisition is Initialization (RAII) patterns to ensure correctness when using pointers. This is especially true for non-linear dataflow networks with multiple endpoints. Only pass small data items by value.

PIPELINE STAGES THAT ARE TOO SMALL

Don't pass very small items of work. The overhead of managing the pipeline will override the gains from parallelism.

FORGETTING TO USE MESSAGE PASSING FOR ISOLATION

Don't use shared data structures, such as locks and semaphores, to share data between agents. Instead, pass messages.

INFINITE WAITS

If a pipeline task catches an exception and terminates, it will no longer take values from its input messaging block. Depending on the logic of your pipeline, you may find that processing is blocked indefinitely. You can avoid this situation by using the technique that was described in the section, "Exception Handling," earlier in this chapter.

UNBOUNDED QUEUE GROWTH

You should be careful to limit the number of elements that can be pending at one time in the pipeline's buffers. Use the techniques described in the previous sections to enforce such a limit. Refer to the **PipelineGovernor** class in the online samples for an example of how to limit the number of in-flight items in a pipeline.

MORE INFORMATION

For more information about this guidance, see Best Practices in the Asynchronous Agents Library on MSDN at http://msdn.microsoft.com/en-us/library/ff601928.aspx.

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 run time as well as the nature of the application you are trying to implement. Unlike most of the other patterns in this book, the Pipeline pattern doesn't automatically scale with the number of cores. This is one of its limitations. (Of course, in some cases you can introduce additional parallelism 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.

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.

The number of in-flight elements in the pipeline is also important for overall performance. If you limit your pipelines to contain only very small numbers of data values, you may find that not all stages of your pipeline are fully occupied with work, especially if data elements take a variable amount of processing time. Allowing the pipeline buffers to hold more data elements accommodates the variability in processing time. The allowed number of in-flight data elements can also depend on the size of the objects being processed. You would probably want to use fewer entries if each element contained an object such as a large bitmapped image that required a great deal of memory.

In general, there should be enough buffering to absorb variability in the pipeline flow, but no more. Use the Visual Studio Concurrency Visualization view to understand the throughput characteristics of the pipeline and modify the pipeline capacity to minimize the amount of time each stage is blocked by I/O waits.

Related Patterns

The Pipeline pattern has much in common with the concepts of pipes and filters that are implemented in operating systems. 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

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

Further Reading

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.

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

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

Appendix A The Task Scheduler and Resource Manager

The Parallel Patterns Library (PPL) and Asynchronous Agents Library rely on features of a lower-level component known as the Concurrency Runtime. The Concurrency Runtime contains a task scheduler and a resource manager. Both are documented on MSDN®, but in this appendix you'll find an overview of their functionality and learn of some of the motivations that shaped their designs.

The Concurrency Runtime enables you to declare sources of potential parallelism in your applications. It is designed both to use existing parallel hardware architectures and to take advantage of future advances in those architectures. In other words, your applications will continue to run efficiently as platforms evolve. The task scheduler determines where and when to run your application's tasks, and it uses cooperative scheduling to provide load balancing across cores. The resource manager prevents the different parts of your application, as well as any libraries they use, from contending for parallel computing resources. It also helps to ensure the best use of resources such as hardware caches. The problems addressed by the Concurrency Runtime are far from trivial. It uses advanced algorithms and benefits from many person-years of experience to optimize parallel performance on multicore architectures.

Figure 1 illustrates how the components in the Concurrency namespace relate to each other. (These components are shown in shaded blue.)

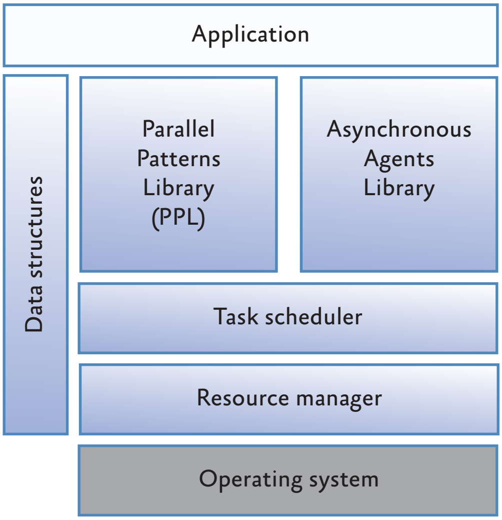


FIGURE 1
Relationships among libraries and run-time components

The following table lists the header files for each component of the Concurrency Runtime. The information in these files gives you a convenient way to learn the capabilities of a component.

Component	C++ Header File
Parallel Patterns Library	ppl.h
Asynchronous Agents Library	agent.h
Data structures	concurrent_vector.h concurrent_queue.h
Task scheduler	concrt.h
Resource manager	concrtrm.h

The Concurrency Runtime is a user-mode layer that sits on top of the operating system. It can manage large numbers of cores, something that is not feasible for an application to do by itself. The Concurrency Runtime is part of the C++ runtime that is included in Microsoft® Visual Studio® 2010 development system. No additional libraries are required.

The Concurrency Runtime abstracts some of the operating system's processor management APIs and provides implementations that automatically use the features of each version of the operating system. For example, on a 64-bit version of Microsoft Windows® 7 operating system, you can scale your application to 256 cores while automatically assigning tasks to cores in a way that respects Non-Uniform Memory Architecture (NUMA) boundaries. (This would not be easy to program.) This same application will run on Windows Vista®, which supports 64 cores.

Note: *All class names and global function names mentioned in this appendix are from the* **Concurrency** *namespace, unless otherwise specified.*

Resource Manager

The resource manager allocates processor cores among the application's task schedulers and ensures that related tasks execute as "locally" as possible. Local execution includes taking advantage of the memory access characteristics of NUMA nodes and hardware caches. The resource manager is a singleton instance of the **Resource Manager** class.

Most programmers won't invoke the resource manager directly, but it's helpful to understand how the resource manager works.

The resource manager is a singleton instance of the **ResourceManager** *class. It allocates processor resources to the application's task schedulers and helps execute tasks as "locally" as possible.*

WHY IT'S NEEDED

The resource manager is especially helpful when there are multiple scheduler instances within a single application. In these situations, the resource manager allows task scheduling components, including the **Scheduler** class as well as components written by third parties, to coexist without contending for cores. For example, if your application has two schedulers, both with default scheduler policies, the resource manager initially divides the available cores on your computer equally between them. The resource manager makes the division of cores along NUMA node boundaries or processor packages, if possible.

The resource manager is also helpful when an application uses parallel libraries from more than one vendor. The resource manager allows these libraries to cooperatively share resources. Microsoft encourages vendors who write libraries for concurrency to build their components on top of the Concurrency Runtime so that all the libraries can take advantage of this feature.

The resource manager also provides dynamic resource management, which adjusts the level of concurrency across components based on core utilization.

HOW RESOURCE MANAGEMENT WORKS

The main abstraction of the resource manager is a virtual processor object that is provided by the **IVirtualProcessorRoot** class. You can think of a virtual processor object as a token that grants a scheduler the right to start (or resume) one thread. The core that runs the thread is chosen based on a *processor affinity mask* that is specified by the virtual processor object. A processor affinity mask is a bit mask that specifies which core(s) the operating system can use to run a particular thread. Cores mean all hardware-supported execution resources, *A virtual processor object grants a scheduler permission to start (or resume) one thread. The core that executes the thread is chosen based on a specific processor affinity mask.*

including hardware threads of simultaneous multithreading ("hyperthreading") architectures.

You can think of a virtual processor object as similar to a concert ticket that has a section assignment but no specific seat number. When the ticket is eventually presented at the door it gives the bearer the right to sit in any seat in the designated section of the concert hall. Similarly, the virtual processor object is a "ticket" that will allow a worker thread to run on any core that meets the requirements of the processor affinity mask.

> **Note**: *To manage threads, the resource manager provides instances of the* **IThreadProxy** *class, which a scheduling component should associate with objects that provide the* **IExecutionContext** *interface.*

The following diagram shows how this works.

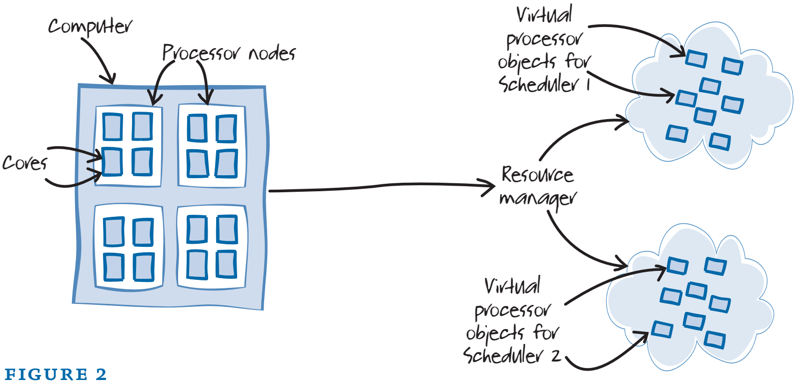

Computer
Processor nodes
Cores
Virtual processor objects for Scheduler 1
Resource manager
Virtual processor objects for Scheduler 2

FIGURE 2
Virtual processor objects

A processor node is an abstraction created by the resource manager to represent NUMA nodes, processor packages or (potentially) other kinds of groupings.

Figure 2 shows that the cores of the computer are grouped into processor packages and NUMA nodes. The resource manager knows how the NUMA nodes and processor packages are laid out in the computer. For a particular process, several scheduler objects ask the resource manager for specific numbers of virtual processor objects that will be used by their worker threads. Some schedulers may want a higher degree of concurrency than other schedulers. Some schedulers want as much concurrency as possible.

The resource manager attempts to satisfy the requests of all scheduler objects, given the fact that there are a fixed number of cores on the computer. In the end, the resource manager gives some virtual processor objects to each scheduler. The virtual processor objects do not issue particular core IDs; instead they specify a processor node, which is an abstraction used by the resource manager to represent NUMA nodes, processor packages or other kinds of groupings of execution resources. In any case, each virtual processor object represents the right for one worker thread to run on that processor node, even though the particular core will be chosen later.

The resource manager maps processor nodes to a set of cores. After a scheduler receives its virtual processor objects, which worker thread will use a particular virtual processor object is still unknown. As it runs, the scheduler associates (and disassociates) worker threads with virtual processor objects. At any given moment, there is one virtual processor object per running thread. The virtual processor object uses the processor affinity mask to determine which cores can be selected by the operating system. The operating system decides the specific core that will be assigned to a worker thread. The worker thread runs on the chosen core.

After the thread begins to run, the virtual processor object can't be reused with another thread unless a *cooperative context switch* occurs. A cooperative context switch happens when a worker thread becomes blocked as a result of a cooperative blocking operation. The blocked thread is disassociated from its virtual processor object, and another worker thread becomes associated with the virtual processor object and is allowed to run. If a scheduler wants more threads to run at the same time, the scheduler must ask the resource manager for additional virtual processor objects. The number of virtual processor objects assigned to a scheduler equals that scheduler's level of concurrency, or the number of worker threads that can run at the same time. The scheduler is free to create as many threads as it wants, but it can only allow as many threads as it has virtual processor objects to *run* at any given time.

The net effect of the interaction between a scheduler and its virtual processor objects is to fix the scheduler's level of concurrency and to cause its threads to run on specific cores.

Dynamic Resource Management

The resource manager uses dynamic resource management to help schedulers cooperate in their use of cores. At run time, the resource manager dynamically monitors the use of execution resources. It knows when virtual processor objects are idle and when they are busy. If it detects use patterns that indicate consistent underutilization of a core, the resource manager might reassign that core to another scheduler. Dynamic resource management only occurs when there are multiple schedulers.

Techniques such as reassigning cores allow the resource manager to place execution resources where they are most needed. As a consequence, you may notice that the allocation of virtual processor objects changes over time.

OVERSUBSCRIBING CORES

If a set of cores has more virtual processor objects than the number of cores (where all hardware threads are considered to be cores), the cores are said to be *oversubscribed*. Normally, the default scheduling policy partitions the available cores without creating more virtual processor objects than there are cores on the computer. In other words, the resource manager avoids oversubscription. When cores are busy with compute-intensive operations, oversubscription results in contention for system caches and reduced throughput. However, there are exceptions to this rule.

For example, the resource manager deliberately uses oversubscription if the aggregate minimum level of concurrency required by all of the application's schedulers exceeds the number of cores. Another situation is when the scheduling policy option **TargetOversubscription Factor** is set to a value greater than one. This policy allows a scheduler to ask for oversubscribed cores at startup.

A third example is when a scheduler requests additional concurrency (in the form of additional virtual processor objects) from the resource manager as the program runs. Finally, a special case arises when the resource manager's dynamic resource management feature observes that a scheduler is underutilizing one of its cores. In this case the resource manager might temporarily oversubscribe the underutilized core with work from other schedulers without removing the corresponding virtual processor object from the first scheduler.

QUERYING THE ENVIRONMENT

While it is unlikely that you will need to program directly against the resource manager, there are a few functions in the concrtrm.h header file that you may find useful. These functions retrieve information about the operating environment. The **GetProcessorCount** and **Get ProcessorNodeCount** functions are particularly useful. They return the number of cores (counting all hardware threads as cores) and the number of processor nodes on your computer. If you see that the number of nodes is greater than one, you can deduce that you are on a machine that has some concept of locality for cores, such as a machine with NUMA or multiple processor packages. The exact meaning of a "processor node" is determined by the resource manager and depends on what kind of computer you have.

Kinds of Tasks

Before discussing the task scheduler it's important that you understand that there are two types of tasks in the Concurrency Runtime. The task scheduler provides its own task abstraction, known as lightweight task, which should not be confused with the tasks provided by PPL.

LIGHTWEIGHT TASKS

The primary interface to lightweight tasks is the **ScheduleTask** method of the **ScheduleGroup** class. The **ScheduleTask** method allows you to add a new, pending lightweight task to the queue of a particular schedule group. You can also invoke the **CurrentScheduler::ScheduleTask** static method if you want the runtime to choose a schedule group for you. Schedulers and schedule groups are described in the "Task Schedulers" section of this appendix.

Do not confuse lightweight tasks provided by the Concurrency Runtime with tasks in PPL. Most programmers will not create lightweight tasks directly.

You can use lightweight tasks when you don't need any of the cancellation, exception handling, or task-wait features of PPL tasks. If you need to wait for a lightweight task to finish, you must implement the wait with lower-level synchronization primitives. If you want to handle exceptions, you need to use STL's **exception_ptr** class and then rethrow captured exceptions at a time of your choosing.

The Concurrency Runtime's interface to lightweight tasks is similar to the Windows thread pool in that it only uses function pointers for its work functions. However, the Concurrency Runtime's sample pack includes, as a convenience, a functor-based way to schedule lightweight tasks that uses wrapper classes named **task_scheduler** and **schedule_group**. If you use lightweight tasks, you will probably find the sample pack's interface to be a more convenient approach.

Most programmers won't use lightweight tasks. Instead, they will use PPL to create tasks. Nonetheless, lightweight tasks can be useful for programming new control structures. Also, you can use lightweight tasks to migrate from threads to tasks. For example, if you have an existing application that uses calls to Windows APIs to create threads, and you want the application to use a task scheduler object as a better thread pool, then you may want to investigate lightweight tasks. However, it's recommended that most programmers should use PPL for task-based applications.

Lightweight tasks are used internally by the messaging block and agent classes of the Asynchronous Agents Library.

The Asynchronous Agents Library uses lightweight tasks to implement the messaging block and agent classes that are described in Chapter 7, "Pipelines."

TASKS CREATED USING PPL

When you use any of the features of PPL to create tasks, such as the **parallel_for**, **parallel_for_each** or **parallel_invoke** functions, or the **task_group::run** or **structured_task_group::run** methods, you create a kind of task that is distinct from a lightweight task. This appendix refers to such tasks as PPL tasks.

Task Schedulers

The Concurrency Runtime's task scheduler is similar to a thread pool. In fact, you can think of the task scheduler as a special type of thread pool that is very good at optimizing large numbers of fine-grained work requests. Unlike many thread pool implementations, the task scheduler does not automatically create a new worker thread when a request arrives and all existing threads are busy. Instead, the scheduler queues the new task and executes it when processor resources become available. The goal is to keep all of the processor cores as busy as possible while minimizing the number of context switches in the operating system and maximizing the effectiveness of hardware caches.

Task schedulers represent a special kind of thread pool that is optimized for fine-grained tasks.

Task scheduling is implemented by the **Scheduler** class. Scheduler instances use worker threads that are associated with the virtual processor objects provided by the resource manager whenever the worker threads are running. There is at most one running thread per virtual processor object granted by the resource manager.

Instances of the **Context** class represent the threads known to a scheduler instance, along with additional per-thread data structures that are maintained by the scheduler for its own use.

MANAGING TASK SCHEDULERS

There is one default scheduler per process, but you can create additional schedulers per context. You can also set the scheduling policy of the default scheduler.

The Concurrency Runtime provides one default scheduler per process. The default scheduler is created when you first make calls into the Concurrency Runtime. You can set scheduler policy for the current context by creating instances of the **Scheduler** class and attaching them to the currently executing thread.

Alternatively, you can invoke the **SetDefaultSchedulerPolicy** static method of the **Scheduler** class at the beginning of your application, before the default scheduler has been created, to specify how the default scheduler will work.

If your application uses multiple scheduler objects, you might want to construct and attach all of the schedulers at startup. Allocating schedulers at the outset avoids the overhead that occurs when schedulers are created while work is in progress. This overhead includes reallocating system resources and reassociating threads to processor nodes. For example, if you have one scheduler that does a **parallel_for** loop, it will, by default, use all cores on your machine. If halfway through the run of the **parallel_for** operation you add a scheduler for use by agent-based code, the first scheduler may be asked to reduce its concurrency as it runs. It's more efficient to allocate the division of cores between the two schedulers at the start of the application.

Creating and Attaching a Task Scheduler

Use the factory method **Scheduler::Create** to instantiate a scheduler object. The method accepts a reference to a **SchedulerPolicy** object as its argument. This object contains configuration settings for your scheduler. After creating the scheduler, you attach it to the current context to activate it. The following code is an example of how to do this.

```
#include <concrt.h>
#include <concrtrm.h>
#include <stdio.h>
#include <windows.h>
#include <iostream>

using namespace ::Concurrency;
using namespace ::std;

int main()
{
  SchedulerPolicy myPolicy(2, MinConcurrency, 2,
                              MaxConcurrency, 2);

  Scheduler* myScheduler = Scheduler::Create(myPolicy);

  cout << "My scheduler ID: " << myScheduler->Id() << endl;

  cout << "Default scheduler ID: "
       << CurrentScheduler::Get()->Id() << endl;

  myScheduler->Attach();

  cout << "Current scheduler ID: "
       <<  CurrentScheduler::Get()->Id() << endl;
```

The **Scheduler::Attach** method sets the scheduler that will be used by the current context. The new scheduler in this example alerts the resource manager that it requires a concurrency level of two. You must call the **Attach** method in the thread whose scheduler you want to replace. The **CurrentScheduler** class's **Get** method returns the current context's currently attached scheduler object, or the default scheduler if no user-provided scheduler was previously attached to the current context.

As a convenience, you can use the **Create** method of the **Current Scheduler** class to instantiate a new scheduler object and attach it to the current context with a single call.

Detaching a Task Scheduler

The **Detach** method of the **CurrentScheduler** class reverses the effect of a previous call to the **Scheduler::Attach** method. Attaching and detaching schedulers are stack-based operations. If you call **Detach**, the current scheduler is popped from the stack and the previous scheduler is restored as the current scheduler. The following code shows how to use the **Detach** method.

```
CurrentScheduler::Detach();

cout << "Current scheduler ID: "
     << Concurrency::CurrentScheduler::Get()->Id() << endl;
```

Destroying a Task Scheduler

A scheduler object has reference/release semantics that are independent of the attach/detach process. Attach/detach logically contains a reference/release pair.

You can detach the most recently attached scheduler at any time. The runtime will not destroy the detached scheduler until all references to it have been released and all of its tasks have completed. If you need to be notified when a detached scheduler is eventually destroyed, create a Windows event and register it with the scheduler. The correct shutdown sequence for a scheduler object is shown in the following code.

```
HANDLE schedulerShutdownEvent =
          CreateEvent(NULL, TRUE, FALSE, L"Shutdown Scheduler");
myScheduler->RegisterShutdownEvent(schedulerShutdownEvent);
myScheduler->Release();
WaitForSingleObject(schedulerShutdownEvent, INFINITE);
```

This code blocks the current context until all other users of the scheduler are also finished. Waiting for the scheduler to shut down might be necessary before unloading a DLL, for example.

Scenarios for Using Multiple Task Schedulers

Multiple task schedulers can be helpful when you need quality-of-service guarantees. They can also provide a better user experience by ensuring that the application is responsive even if long-running parallel workloads are executing. For example, you can use specific scheduler instances to reserve cores for high-priority activities such as audio processing.

Message passing is another example of a high-priority activity. If your cores are very busy and message delivery does not have high priority, tasks that depend on receiving data from messaging buffers might not be run, which could cause bugs. A solution is to send messages on a dedicated "message propagation" scheduler. For example, if you wanted to guarantee that a cancellation message gets processed immediately, you could create a new scheduler for it.

Another case is when you're running on a multi-node host in a server farm. For performance reasons, you may not want parallel loops to span multiple processor nodes. Using multiple schedulers allows you to limit your parallel loops to a single processor node.

Multiple task schedulers can separate UI work from background processing. You can have a foreground scheduler in the UI that does some work when you click a button and a background scheduler for work that's ongoing.

Implementing a Custom Scheduling Component

The **Scheduler** class is not extensible. You cannot use it as a base class. This means that you must use the runtime's **Scheduler** class for scheduling work that is created with PPL and the Asynchronous Agents Library.

However, if you are writing your own parallel programming library, you can create your own scheduling component by implementing the **IScheduler** and **IExecutionContext** interfaces.

THE SCHEDULING ALGORITHM

The **Scheduler** class uses a queue-based approach to run tasks. New tasks wait in queues until the scheduler assigns processing resources to them in cooperation with the resource manager and the operating system. Queues emphasize overall throughput at the cost of scheduling fairness for individual tasks. The motivating idea is that a set of related tasks represents the decomposition of a larger problem. Therefore, solving the overall problem in the fastest possible way is the primary goal, even if some tasks have to wait longer than others to run.

Queue-based scheduling emphasizes overall throughput at the cost of "fairness" for individual tasks.

You can contrast the queue-based approach to scheduling with preemptive multitasking that gives time slices to all running threads. Multitasking emphasizes the responsiveness of each thread.

The material in this section can help you understand the performance characteristics that you'll observe when you use the runtime's **Scheduler** class. Be aware that the scheduling algorithm described here represents an implementation choice. Future versions of the Concurrency Runtime might optimize task execution differently.

The behind-the-scenes behavior described in this section applies to Visual Studio 2010 SP1. There's no guarantee that future releases of the runtime won't behave differently.

Schedule Groups

A scheduler object has more than one queue of pending tasks. Pending tasks are tasks that haven't started to run. Internally, the **Scheduler** class uses instances of the **ScheduleGroup** helper class for its queues of pending tasks. For example, a scheduler instance may want separate queues based on the division of cores into processor nodes.

When you add a task to the scheduler, you normally allow the scheduler to choose a schedule group for that task.

The scheduler object maintains several kinds of queues for each of its schedule groups. This is illustrated in Figure 3.

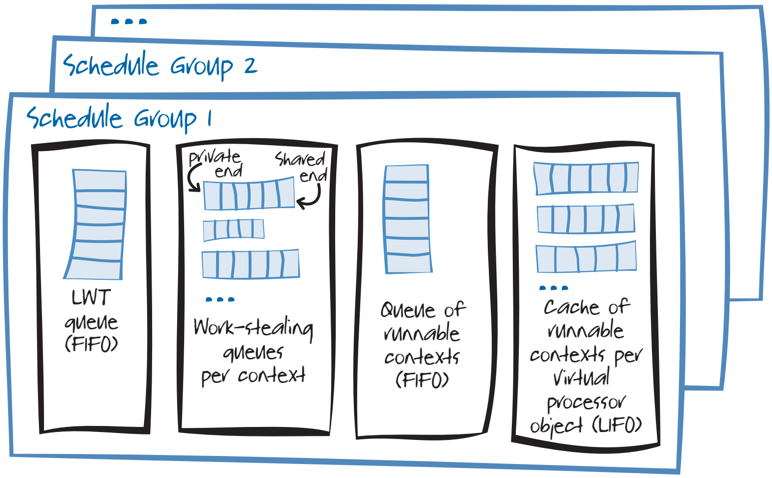

...
Schedule Group 2
Schedule Group 1
LWT queue (FIFO)
private end
shared end
...
Work-stealing queues per context
Queue of runnable contexts (FIFO)
...
Cache of runnable contexts per virtual processor object (LIFO)

FIGURE 3
Data structures of schedule groups

Each schedule group contains one queue for lightweight tasks (LWTs). Each schedule group also contains multiple work-stealing queues of pending PPL tasks. There is one work-stealing queue of pending PPL tasks in the schedule group for each execution context. A work-stealing queue is a mutable list with a private end accessible to a single context and a public end that can be accessed by many contexts. Entries in a work-stealing queue can be added or removed from the private (or local) end with minimal synchronization. Entries can also be added or removed from the public (or shared) end but with higher synchronization costs. Only the context that the schedule group has associated with the work-stealing queue can add to and remove entries from the private end.

Each schedule group contains one main queue of runnable contexts and can also maintain a cache of runnable contexts for each virtual processor object. A runnable context is a thread that was previously interrupted by one of the cooperative blocking operations, but is now unblocked and ready to resume its work.

Adding Tasks

A lightweight task is implicitly created by messaging blocks' operations and by starting an **agent** instance. You can also use the **ScheduleTask** method of the **Scheduler** class, the **CurrentScheduler** class or the **ScheduleGroup** class to create a lightweight task. When you create a new lightweight task, the task is added to the schedule group you specify or to a schedule group chosen by the scheduler. The new task is added to the end of the schedule group's queue of lightweight tasks.

When you create a PPL task, the new task is added to the private end of the work-stealing queue of the current context. The current context may correspond to a thread that is not one of the scheduler's worker threads. In this case, the scheduler associates a work-stealing queue with the current context in one of its schedule groups.

Use the **Attach** and **Detach** methods that were described in the "Managing Task Schedulers" section of this appendix to control which scheduler is the current scheduler for a given context.

Running Tasks

A virtual processor object can become available in one of two ways. One way is that the thread that is currently running on the virtual processor object becomes blocked by one of the cooperative blocking operations. The other way is that the thread completes its current task. When a virtual processor object becomes ready to accept new work, the scheduler uses heuristics to prioritize the possible next steps.

As a general rule, the scheduler tries to resume runnable contexts in preference to starting new pending tasks. A runnable context is a worker thread that previously ran and became blocked by one of the cooperative blocking operations, but is now unblocked and ready to be resumed. The scheduler takes runnable contexts first from the virtual processor object's local cache of runnable contexts in last in first out (LIFO) order and then from the queue of runnable contexts in the current schedule group in first in first out (FIFO) order. It may also look for runnable contexts in the caches of other virtual processor objects and schedule groups. The LIFO cache of runnable contexts improves the likelihood that the data in hardware caches will be relevant. The most recently unblocked context is resumed first, and it is run on the same virtual processor object as the operation that caused the context to become unblocked. You can configure the size of the cache of unblocked tasks with the **LocalContextCacheSize** schedule policy key.

If there are no runnable contexts, the scheduler looks for lightweight tasks in the schedule group's queue of lightweight tasks. It takes the next available lightweight task in FIFO order. Lightweight tasks are usually used to implement message passing and are therefore considered to be of higher priority than PPL tasks.

If there are no lightweight tasks pending in the current schedule group, the scheduler looks for tasks from the public ends of the work-stealing queues of the current schedule group. It looks for tasks in work-stealing queues first within the queues associated with the current processor node, and then across processor nodes.

A special situation arises when a thread enters the **task_group::wait** or the **structured_task_group::wait** methods. Normally, **wait** is considered to be one of the cooperative blocking operations. However, if the call to **wait** requires the current thread to wait for pending tasks that are in the current context's work-stealing queue, then the current context will *not* block immediately. Instead, PPL reuses the current context to run the locally queued tasks using an optimization known as inline execution. The pending tasks are taken from the private end of the current context's work-stealing queue in LIFO order.

The scheduler is unaware of inline execution because the thread that performs inline execution will not become cooperatively blocked by the **wait** method until there are no more tasks in the local work-stealing queue that would satisfy the wait condition. Of course, if inline execution has satisfied the wait condition, the **wait** method will return and the current thread will continue running its top-level task. For more information about inlining, see "Tasks That Are Run Inline" in this appendix.

The **Scheduler** class implements two variations of its scheduling algorithm, which can be selected by setting the **SchedulingProtocol** scheduling policy key. The two scheduling approaches are *enhanced locality* mode (the default) and *forward progress* mode.

Enhanced Locality Mode

In enhanced locality mode, the scheduler tries to execute related tasks in batches. The scheduler attempts to process *all* pending tasks of the first schedule group until the group contains no more tasks to run. Then, the scheduler moves on to the pending tasks of the next schedule group, and so on, eventually starting over with the first schedule group. This order is not strict; there are also heuristics to avoid thread starvation that can occur if a task that would unblock a currently blocked task is not allowed to run. These heuristics may periodically give preference to tasks of other schedule groups. This can occur if the current schedule group has many pending tasks.

Enhanced locality scheduling processes schedule groups in batches so that related tasks execute together.

In enhanced locality mode the runtime assumes that tasks in a schedule group share memory. In order to derive the most benefit from hardware caches, the scheduler attempts to run the tasks of a schedule group as close together chronologically as possible, even if it means that tasks that were added earlier to some other schedule group experience delays. For example, if tasks 1, 2, and 3 are created and assigned to schedule groups A, B, and A, respectively, and the scheduler starts processing schedule group A, then it is likely that task

3 will be run *before* task 2, even though task 3 was created after task 2. In other words, the scheduler attempts to run all tasks of schedule group A (which contains tasks 1 and 3) before proceeding to schedule group B (which contains task 2).

Due to the processor node affinity provided by the division into schedule groups, the tasks will be spatially close as well. When the next schedule group's turn eventually comes, all of its tasks will be executed the same way.

Forward Progress Mode

In forward progress mode, the scheduler executes one pending task from each of its schedule groups in round-robin fashion. Unlike enhanced locality scheduling, there are no caches of runnable contexts per virtual processor object; however, runnable contexts are prioritized over pending tasks.

Forward progress scheduling rotates through schedule groups, executing one task from each.

With forward progress scheduling, the runtime assumes that keeping each schedule group from stalling matters more than running related tasks as a batch. This might occur, for example, in discrete event simulation scenarios where you want all schedule groups to progress by one step before updating a GUI. Forward progress scheduling is a less commonly used approach for task-based applications.

Task Execution Order

The scheduler does not make guarantees about the order in which queued tasks will be executed, and your application should make no assumptions about when a particular task will be allowed to run. In the current implementation, pending tasks are generally processed in FIFO order unless they are inlined. However, given the interaction of the various queues of pending tasks and the optional round-robin or batch-oriented processing of schedule groups, it's not possible to predict the order of execution. If you need tasks to be run in a particular order, then you should use one of the task coordination techniques described in this book. Examples include the **task_group::wait** method and the **receive** function of messaging blocks.

Tasks That Are Run Inline

As was mentioned earlier in the section on "Running Tasks," it is possible that the scheduler will run a task in the thread context of another task that is waiting for that task to complete. For example, if you invoke a task group's **wait** method from within a task context, the runtime knows that the current context will be idle until the tasks of that task group have completed. The runtime can therefore optimize its use of worker threads by reusing the blocked thread to run one or more of the pending tasks. Inlining is a good example of potential parallelism: when dependent tasks run on a machine with many cores, they run in parallel. On a machine with just one available core, they act like sequential function calls.

Inlining allows you to avoid deadlock due to thread starvation. Unlike other kinds of cooperative blocking operations, the **task_group::wait** function provides a hint to the scheduler about which pending tasks will unblock the current context. Inlined execution is a good way to elevate the scheduling priority of tasks that will unblock other tasks.

Only tasks that were created by the current context are eligible to be inlined when you invoke their task group's **wait** method.

You can provide an explicit hint that inlining should occur by calling the **task_group::run_and_wait** method. This method creates a new task and immediately processes it with inline execution.

USING CONTEXTS TO COMMUNICATE WITH THE SCHEDULER

The **Context** class allows you to communicate with the task scheduler. The context object that corresponds to the currently executing thread can be accessed by invoking the static method **Context::CurrentContext**.

The **CurrentContext** method can be called from application threads as well as from a task scheduler's worker threads. The behavior of the context object's methods may differ, depending on whether the current context object corresponds to a worker thread or whether it is an application thread. For example, if you call a cooperative blocking operation from within an application thread (that is, a thread that is not one of the current scheduler's worker threads), the thread will block without allowing one of the scheduler's runnable contexts to resume as is the case when a worker thread blocks.

If you get the context object of a thread that is not one of the scheduler's worker threads, be aware that the behavior of some context-dependent operations will be different than would be the case for the scheduler's worker threads.

Debugging Information

You can get access to useful debugging information from contexts, such as the integer ID for the current context, the schedule group ID and the virtual processor object ID. The information is provided by the static methods **Id**, **ScheduleGroupId,** and **VirtualProcessorId** of the **Context** class.

Querying for Cancellation

You can detect that a task is being cancelled even if you don't have a reference to its task group object by invoking the **Context** class's static **IsCurrentTaskCollectionCanceling** method. Checking for cancellation is useful if you are about to start a long-running operation.

Interface to Cooperative Blocking

A task can communicate with the scheduler by invoking one of the cooperative blocking operations that were described in Chapter 3, "Parallel Tasks." See "Coordinating Tasks with Cooperative Blocking" in Chapter 3 for a list of the built-in operations.

The **Context** class provides an additional, lower-level interface to cooperative blocking through its **Block** and **Unblock** methods. PPL and the Asynchronous Agents Library use the **Block** and **Unblock** methods to implement all of their cooperative blocking operations.

Most programmers should use the higher-level built-in task coordination mechanisms of PPL and messaging blocks rather than implementing their own synchronization primitives.

You can use **Block** and **Unblock** to coordinate your tasks from within custom synchronization primitives. If you program your own parallel synchronization primitives with the **Block** and **Unblock** methods, you must be very careful not to disrupt the runtime's internal use of these operations. **Block** and **Unblock** are not nesting operations, but the order of operations is flexible. If **Unblock** precedes **Block**, the **Block** operation is ignored. You can unintentionally interact with one of PPL's internal operations by allowing an out-of-order **Unblock** call. For example, an out-of-order call to **Unblock** followed by a call to PPL's **critical_section::lock** method can, in certain interleavings, cause a critical section not to be observed.

If you use the **Block** and **Unblock** methods you must be very careful to avoid situations that could unexpectedly interact with PPL's internal use of the methods. Most programmers will not need to use **Block** and **Unblock** to implement their own synchronization primitives.

Block and **Unblock** are low-level methods. Most programmers will want to use the higher-level cooperative blocking operations.

Waiting

The **Concurrency** namespace includes a global **wait** function that allows you to suspend the current context with the guarantee that the suspended task will not be resumed until a time interval that is provided as an argument to the **wait** function has passed. It is possible, due to the queue-oriented scheduling approach used by the task scheduler, that a task will be suspended for a time period that is much longer than the specified wait period.

The **Yield** method of the **Context** class allows a pending lightweight task to run, or if there are no lightweight tasks, it allows one of the runnable contexts to resume. If either of these two conditions is satisfied, then the current worker thread is added to the queue of runnable contexts; otherwise, the **Yield** method is a no-op. The **Yield** method is also a no-op when called from a thread that is not a worker thread of a scheduler.

Calling the **Yield** method in a tight loop while polling for some condition can cause poor performance. Instead, wait for an event.

The Caching Suballocator

You can allocate memory from a memory cache that is local to the current context. This is useful when a task needs to create many small temporary objects on the heap, and you expect that more than one task might create objects of the same size.

The synchronization overhead of coordinating memory allocation and deallocation for a global pool of memory can be a *significant* source of overhead for a parallel application. The Concurrency Runtime provides a thread-private caching suballocator that allows you to allocate and deallocate temporary working storage within a task and potentially to use fewer locks or memory barrier operations than you would if you used the global memory pool. Allocation requests only incur synchronization overhead at the beginning of the run when new memory must be added to the cache.

The caching suballocator is meant for situations where there is frequent allocation of temporary memory within tasks. Invoke its functions only from within a running task. The caching suballocator does not improve performance in all scenarios, and it does not free its memory. Refer to MSDN for more information about its use.

Long-Running I/O Tasks

By default, the scheduler assumes that its tasks are computationally intensive. Dynamic resource management compensates to some extent for I/O-intensive tasks that are not computationally intensive, but if you know that your task is a long-running I/O-intensive task that will use only a fraction of a processor's resources, you should give a hint to the scheduler that it can oversubscribe the current processor node by one additional running thread.

To do this, call the static method **Context::Oversubscribe** with the argument **true**. This tells the scheduler that it should request an additional virtual processor object from the resource manager while using the same processor affinity mask as the current thread is using. This operation increases the level of concurrency in the scheduler without increasing the number of cores that will be used by the scheduler.

Use **Context::Oversubscribe** to add concurrency to the current scheduler without consuming more cores.

When you are done with your long-running I/O-intensive task, call the **Oversubscribe** method with the argument **false** to reset the level of concurrency to the previous value. Be sure to call the **Oversubscribe** method in pairs, and to take exceptions into account. The sample pack includes the **scoped_oversubcription_token** helper class to automatically ensure that the **Oversubscribe** method is called in pairs. You should place calls to the **Oversubscribe** method within the body of the work function of your long-running I/O task.

SETTING SCHEDULER POLICY

There are a number of settings that you can control that will affect how the task scheduler does its job. These are documented on MSDN as values of the **PolicyElementKey** enumeration (and they are also found in the concrt.h header file), but here is a summary of the most important settings. See the section "Creating and Attaching a Task Scheduler" in this appendix for a code example of how to set policies.

Policy Key	Description
MinConcurrency	This is the minimum number of virtual processor objects that the scheduler requires at startup.
MaxConcurrency	This is the maximum number of virtual processor objects that the scheduler requires at startup. There is a special value, **MaxExecutionResources,** that indicates "as many as exist on the computer."
SchedulingProtocol	This indicates which of the two available scheduling algorithms should be used by the scheduler. Choose **EnhanceScheduleGroupLocality** (the default) or **EnhanceForwardProgress**.
TargetOversubscription-Factor	You can use this setting if you know that your operations will ordinarily use only a fraction of the processor's time, as is typical in I/O-bound programs. The runtime multiplies the number of virtual processor objects by the oversubscription factor.

Anti-Patterns

Here are a few things to watch out for.

Multiple Resource Managers

The resource manager is a singleton that works across one process. It does not coordinate processor resources across multiple operating-system processes. If your application uses multiple, concurrent processes, you may need to reduce the level of concurrency in each process for optimum efficiency.

It is possible, in some situations, for more than one resource manager instance to be created within a single process. If this happens, the resource manager instances will contend for resources. This situation arises if a library uses static linking. There will be one resource manager for each statically linked instance of the C++ runtime.

It is also possible to have multiple resource manager instances if you use more than one version of the C++ runtime within a single application. In this case, there will be one resource manager for each version of the runtime.

In general, you should avoid situations where more than one instance of the resource manager can exist.

Avoid static linking for your parallel applications.

Resource Management Overhead

The resource manager optimizes the use of processor resources by dynamically adjusting the concurrency levels of schedulers that are active in the application. In certain scenarios, you may find that dynamic resource management isn't providing optimal results. In these cases you can effectively disable dynamic resource management by setting the minimum and maximum concurrency levels of your scheduler to the same value. You should understand the performance characteristics of your application before doing this. In most scenarios dynamic resource management will result in better overall throughput of your application.

Unintentional Oversubscription from Inlined Tasks

The "Running Tasks" section of this appendix describes how a scheduler may reuse a thread that is waiting for the tasks of a task group to complete to run one or more pending tasks. This is known as inlining.

Inline execution that occurs in a worker thread of a scheduler object has no effect on the level of concurrency of the scheduler. With or without inline execution, the number of worker threads that are allowed to run at the same time is limited by the number of virtual processor objects that have been allocated to the scheduler by the resource manager.

Unlike worker threads, application threads are not managed by a task scheduler instance. Application threads (that is, any threads that are not a scheduler's worker threads) do not count toward the limit on the number of concurrent running threads that is coordinated by the resource manager. Unless blocked by the operating system, application threads are always allowed to run.

If you call the **task_group::run** method from an application thread and subsequently call the **wait** method on the task group from that same thread, inline execution of the task you created may occur on the application thread. The application thread will be allowed to run regardless of the number of running worker threads in the scheduler. Therefore, inline execution in an application may increase the parallel operation's overall level of concurrency by one running thread. If your scheduler had a maximum concurrency level of four virtual processor objects, your application might run five tasks concurrently: four on the worker threads of the scheduler, plus one on the application thread that entered the **wait** method.

Pending PPL tasks are only inlined if they reside in the local work-stealing queue of the thread that enters the **wait** method. Therefore, if you wanted to prevent inline execution of tasks in an application thread, you could a use a lightweight task to create the PPL task that would otherwise have been created in the application thread. The pending PPL task would then be placed in the work-stealing queue of whatever worker thread executed the lightweight task and would not be eligible for inline execution on the application thread.

The remarks about inlining in this section also apply to **parallel_for**, **parallel_for_each**, and **parallel_invoke**, as well as tasks created using structured task groups. Inline execution also occurs whenever you use the **task_group::run_and_wait** method.

Deadlock from Thread Starvation

Recall that there are two kinds of blocking operations: cooperative blocking operations proved by the Concurrency Runtime and "noncooperative" blocking operations provided by the operating system.

When a worker thread becomes blocked by a cooperative blocking operation, the scheduler will resume one of its runnable contexts or allow one of the pending tasks to start running. Note that if no idle worker thread is available to run the pending task, a new worker thread will be created. Creating the additional thread does not introduce additional concurrency. At any given time, the number of worker threads that are released to the OS for scheduling never exceeds the number of virtual processor objects allocated to it by the resource manager.

Allowing additional pending tasks to run when a task is cooperatively blocked increases the chance of running the task that would unblock the cooperatively blocked task. This can help avoid deadlock from thread starvation that can occur in systems with many dependent tasks and fixed levels of concurrency.

In contrast to cooperative blocking, noncooperative or OS-level blocking is opaque to PPL and the Concurrency Runtime, unless the blocking operation occurs on a User-Mode Scheduled (UMS) thread. (UMS threads are outside the scope of this book.) When a worker thread becomes blocked by an OS-level blocking operation, the scheduler still considers it to be a running thread. The scheduler does not resume one of its runnable contexts or start one of its pending tasks in response to an OS-level blocking operation. As a consequence, you may end up with deadlock due to thread starvation. For example, in an extreme case, if all worker threads of a scheduler are blocked by noncooperative blocking operations that need pending tasks to run in order to become unblocked, the scheduler is deadlocked.

It is therefore recommended that cooperative blocking operations be used as the primary mechanism of expressing dependencies among tasks.

The number of worker threads in a scheduler is not the scheduler's level of concurrency. At any given time, the number of threads that are released to the OS for execution will not exceed the number of virtual processor objects that have been allocated to the scheduler by the resource manager.

Ignored Process Affinity Mask

As of Windows 7 you can set a process affinity mask that limits the cores on which the threads of your application may run. The version of the Concurrency Runtime that is shipped as part of Visual Studio

2010 SP1 does not recognize this user-selected process affinity mask. Instead, it uses all available cores up to the maximum specified level of concurrency for each scheduler.

This issue is expected to be resolved in a future version of the Concurrency Runtime.

References

For more information about the caching suballocator, see Alloc Function on MSDN at

http://msdn.microsoft.com/en-us/library/dd492420.aspx.

For a general description of NUMA, see Introduction to NUMA on the MSDN Magazine blog at

http://blogs.msdn.com/b/msdnmagazine/archive/2010/05/06/10009393.aspx.

For more information about the PolicyElementKey enumeration, see the entry on MSDN at

http://msdn.microsoft.com/en-us/library/dd492562.

Appendix B Debugging and Profiling Parallel Applications

The Microsoft® Visual Studio® 2010 development system debugger includes two windows that assist with parallel programming: the Parallel Stacks window and the Parallel Tasks window. In addition, 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's 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.

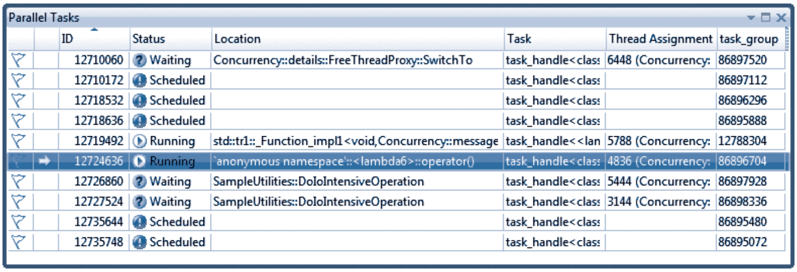
The Parallel Tasks and Parallel Stacks Windows

In Visual Studio, open the parallel guide samples solution. Set the A-Dash project that is discussed in Chapter 5, "Futures," to be the startup project. Open AnalysisEngine.h and find the **AnalysisEngine::DoAnalysisParallel** method, which declares and configures the A-Dash workflow. Each future executes a different stage of the workflow and returns a result that is, in turn, passed to the next future in the workflow. Insert a breakpoint in the declaration of the **future5** lambda.

Start the debugging process. You can either press F5 or click **Start Debugging** on the **Debug** menu. The A-Dash sample begins to run and displays its GUI. On the GUI, select the **Parallel** checkbox, and then click **Calculate**. When execution reaches the breakpoint, all tasks stop and the familiar Call Stack window appears. On the **Debug** menu, point to **Windows**, and then click **Parallel Tasks**. When execution first reaches the breakpoint, the Parallel Tasks window shows a task associated with each future that has been added to the workflow.

Figure 1 illustrates a case where multiple tasks are running. Recall that each task runs in a thread. The Parallel Tasks window shows the assignment of tasks to threads. The ID column identifies the task, while the Thread Assignment column shows the thread. If there is task inlining, more than one task can run in a thread, so it's possible that there will be more tasks than executing threads. The Status column indicates whether the task is running or is in a scheduled or waiting state. In some cases the debugger cannot detect that the task is waiting. In these instances, the task is shown as running. The Location column gives the name of the method that is currently being invoked. Place the cursor over the location field of each task to see a call stack pop-up window that displays only the stack frames that are part of the user's code. To switch to a particular stack frame, double-click on it.

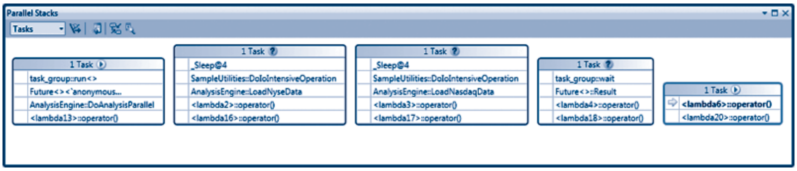
Double-click on a task listed in the Task column to switch the debugger to that task. On the **Debug** menu, point to **Windows**, and then click **Call Stack** to display the complete stack for the thread that is executing the task.



ID	Status	Location	Task	Thread Assignment	task_group
12710060	Waiting	Concurrency::details::FreeThreadProxy::SwitchTo	task_handle<class	6448 (Concurrency:	86897520
12710172	Scheduled		task_handle<class		86897112
12718532	Scheduled		task_handle<class		86896296
12718636	Scheduled		task_handle<class		86895888
12719492	Running	std::tr1::_Function_impl1<void,Concurrency::message	task_handle<<lan	5788 (Concurrency:	12788304
12724636	Running	`anonymous namespace'::<lambda6>::operator()	task_handle<class	4836 (Concurrency:	86896704
12726860	Waiting	SampleUtilities::DoIoIntensiveOperation	task_handle<class	5444 (Concurrency:	86897928
12727524	Waiting	SampleUtilities::DoIoIntensiveOperation	task_handle<class	3144 (Concurrency:	86898336
12735644	Scheduled		task_handle<class		86895480
12735748	Scheduled		task_handle<class		86895072

FIGURE 1
The Parallel Tasks window

On the **Debug** menu, point to **Windows**, and then click **Parallel Stacks**. In the Parallel Stacks window, from the drop-down menu in the upper-left corner, click **Tasks**. The window shows the call stack for each of the running or waiting tasks. This is illustrated in Figure 2.



```
1 Task
task_group::run<>
Future<><`anonymous...
AnalysisEngine::DoAnalysisParallel
<lambda13>::operator()
```

```
1 Task
_Sleep@4
SampleUtilities::DoIoIntensiveOperation
AnalysisEngine::LoadNyseData
<lambda2>::operator()
<lambda16>::operator()
```

```
1 Task
_Sleep@4
SampleUtilities::DoIoIntensiveOperation
AnalysisEngine::LoadNasdaqData
<lambda3>::operator()
<lambda17>::operator()
```

```
1 Task
task_group::wait
Future<>::Result
<lambda4>::operator()
<lambda18>::operator()
```

```
1 Task
<lambda6>::operator()
<lambda20>::operator()
```

FIGURE 2
The Parallel Stacks window

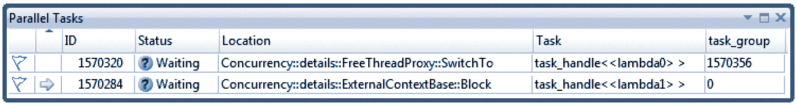
See the "Further Reading" section for references that discuss the Parallel Stacks window in more detail.

If you add additional breakpoints to the other futures defined in **DoAnalysisParallel** and press F5, the contents of the Parallel Tasks and Parallel Stacks windows change as the A-Dash workflow processes data. This is how the application should behave. However, 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 windows can help to identify common concurrency problems such as deadlocks. This behavior is demonstrated in the following code, taken from the ProfilerExamples sample.

```
void Deadlock()
{
    reader_writer_lock lock1;
    reader_writer_lock lock2;

    parallel_invoke(
        [&lock1, &lock2]()
        {
            for (int i = 0; ; i++)
            {
                lock1.lock();
                printf("Got lock 1 at %d\n", i);
                lock2.lock();
                printf("Got lock 2 at %d\n", i);
            }
        },
        [&lock1, &lock2]()
        {
            for (int i = 0; ; i++)
            {
                lock2.lock();
                printf("Got lock 2 at %d\n", i);
                lock1.lock();
                printf("Got lock 1 at %d\n", i);
            }
        }
    );
}
```

This code is a classic example of a deadlock. Each task attempts to acquire a lock. The order in which this occurs leads to a cycle that eventually results in deadlock. At this point, the application stops making progress and there is no more new console output. Once the deadlock occurs, click the **Break All** option on the **Debug** menu, and open the Parallel Tasks window. You'll see something similar to Figure 3. Notice that the status of each task is Waiting instead of Running. Visual Studio also displays a warning dialog "The process appears to be deadlocked (or is not running any user-mode code). All threads have been stopped." Use the Parallel Tasks window to examine each deadlocked task and its call stack to understand why your application is deadlocked. Place the cursor over the Status column to see what the task is waiting for. You can also examine the call stacks and identify the locks, or other wait conditions, that are causing the problem.



Parallel Tasks

		ID	Status	Location	Task	task_group
		1570320	Waiting	Concurrency::details::FreeThreadProxy::SwitchTo	task_handle<<lambda0> >	1570356
	⇨	1570284	Waiting	Concurrency::details::ExternalContextBase::Block	task_handle<<lambda1> >	0

FIGURE 3
Parallel Tasks window showing deadlock

Breakpoints and Memory Allocation

Excessive memory copies can lead to significant performance degradation. Use the debugger to set breakpoints on your class's copy constructor and assignment operators to find unintentional copy and assignment operations.

For example, the ImagePipeline sample passes pointers of type **shared_ptr<ImageInfo>** along the pipeline rather than copies of the **ImageInfo** objects. These are too expensive to copy because they contain large bitmaps. However, **ImageInfo** contains an **ImagePerformanceData** object that is copied once per image. You can use the debugger to verify that no extra copies are being made. Here is how to do this.

Set a breakpoint inside the **ImagePerformanceData** assignment operator in ImagePerformanceData.h and then run the ImagePipeline example. You'll see that an **ImagePerformanceData** object is only assigned once, after it has passed through the pipeline. This occurs during the display phase, when the **ImagePipelineDlg::OnPaint** method creates a copy of the final **ImagePerformanceData** object so that the GUI thread has the latest data available to display.

Use the **Hit Count** feature to count the number of assignments. In the **Breakpoints** window, right-click on the breakpoint and then click **Hit Count** on the shortcut menu. In the Hit Count dialog box, select **break when the hit count is equal to** option from the **When the breakpoint is hit** list. Set the hit count number to 100. Run the sample. The debugger stops on the breakpoint after one hundred images have been processed. Because the number of hits equals the number of images processed so far, you know that only one copy was made for each image. If there were unintentional copies, you would reach the breakpoint after fewer images were processed.

You can also declare the copy constructors and assignment operators as **private** to prevent objects from being unintentionally copied. The **ImageInfo** object has a private copy constructor and assignment operator for just this reason.

The Concurrency Visualizer

The profiler included in the Visual Studio 2010 Premium and Ultimate editions includes the Concurrency Visualizer. This tool 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 is behaving as you intended, and it can help you to diagnose performance problems. This appendix uses the Concurrency Visualizer to profile the ImagePipeline sample from Chapter 7 on a computer with eight logical cores.

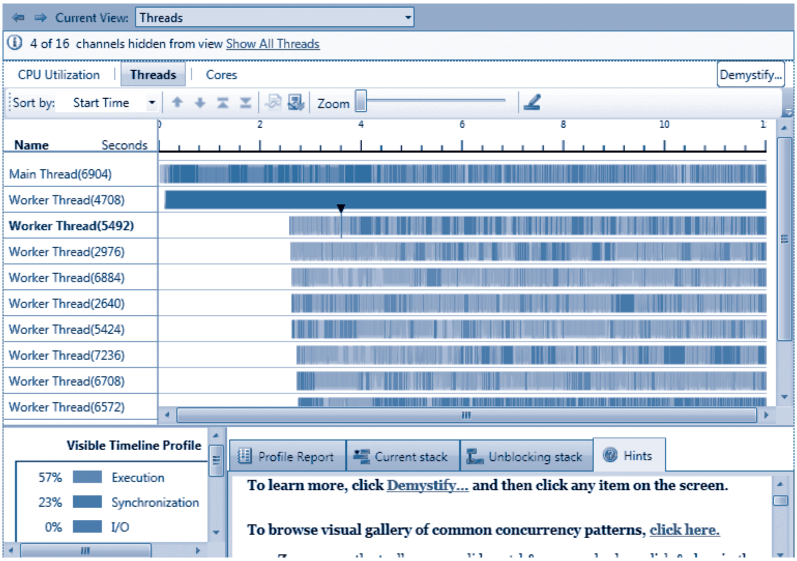
The Concurrency Visualizer has two stages: data collection and visualization. In the collection stage, you first enable data collection and then run your application. In the visualization stage, you examine the data you collected.

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 sample solution in Visual Studio. Click **Start Performance Analysis** on the Visual Studio **Debug** menu. The Performance Wizard begins. Click **Concurrency**, and then 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, which is **Image Pipeline**. Click **Next**. The last page of the wizard asks if you want to begin profiling after the wizard finishes. This check box is selected by default. Click **Finish**. The Visual Studio profiler window appears and indicates that it's currently profiling. The ImagePipeline sample begins to run and opens its GUI window. To maximize processor utilization, select the **Load Balanced** option, and then click **Start**. In order to collect enough data to visualize, let the **Images** counter on the GUI reach at least a hundred. Then click **Stop Profiling** in the Visual Studio profiler window.

During data collection, the performance analyzer takes frequent data samples (known as snapshots) that record the state of your running parallel code. The analyzer also uses Event Tracing for Windows (ETW) to collect all context switches and some other relevant events. 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 in size. 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 data files are available. You don't need to run Visual Studio as an administrator to do this. There are several ways to begin visualization. Unless you've changed the default, visualization automatically starts 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 after the data is collected and analyzed. The summary report shows the different views that are available. These include a Threads view, a CPU Utilization view, and a Cores view.

Figure 4 shows the Threads view. Each task is executed in a thread. The Concurrency Visualizer shows the thread for each task (remember that there may be more than one task per thread because of inline tasks).


Current View: Threads
4 of 16 channels hidden from view Show All Threads
CPU Utilization
Threads
Cores
Demystify...
Sort by: Start Time
Zoom
Name
Seconds
2
4
6
8
10
Main Thread(6904)
Worker Thread(4708)
Worker Thread(5492)
Worker Thread(2976)
Worker Thread(6884)
Worker Thread(2640)
Worker Thread(5424)
Worker Thread(7236)
Worker Thread(6708)
Worker Thread(6572)
Visible Timeline Profile
57% Execution
23% Synchronization
0% I/O
Profile Report
Current stack
Unblocking stack
Hints

To learn more, click Demystify... and then click any item on the screen.

To browse visual gallery of common concurrency patterns, click here.

FIGURE 4
Threads view of the Concurrency Visualizer

The Concurrency Visualizer screens contain many details that may not be readable in this book's figures, which are reduced in size and are not in full color. The full color screen shots from this appendix are available on the CodePlex site at http://parallelpatternscpp.codeplex.com/.

The Threads View also contains a couple of other useful features. Clicking on different segments of an individual thread activity timeline in the upper part of the screen allows you to see the current stack for that activity segment. This allows you to associate code with individual activity segments. Clicking on the different activity types in the Visible Timeline Profile on the bottom left displays a Profile Report. This allows you to discover which functions in your application are involved in each activity type for the currently selected portion of the timeline. You can also click the **Demystify** feature to get further help on what different colors mean and on other features of the report.

Figure 5 illustrates the CPU Utilization view. The **CPU Utilization** view shows how many logical cores the entire application (all tasks) uses, as a function of time. On the computer used for this example, there are eight logical cores. Other processes not related to the application are also shown as an aggregated total named Other Processes. For the application process, there's a graph that shows how many logical cores it's 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 rather than an integer such as 0, 1, or 2, because each point represents an average calculated over the sampling interval.

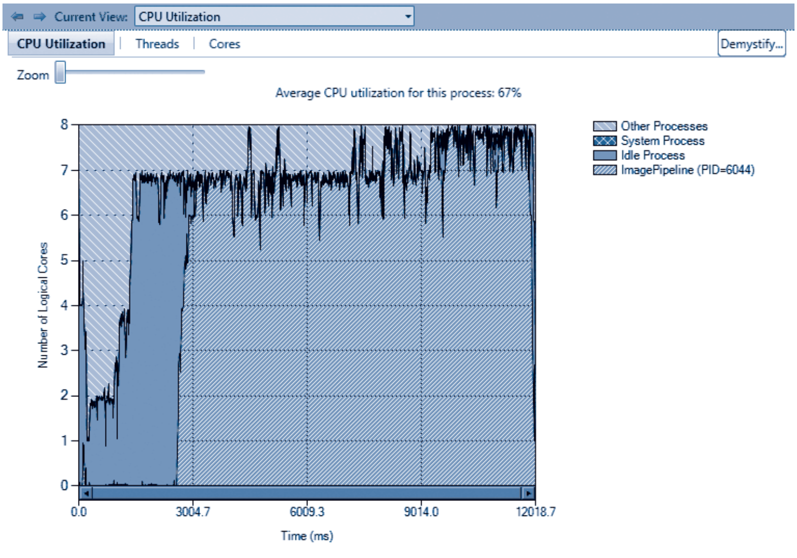
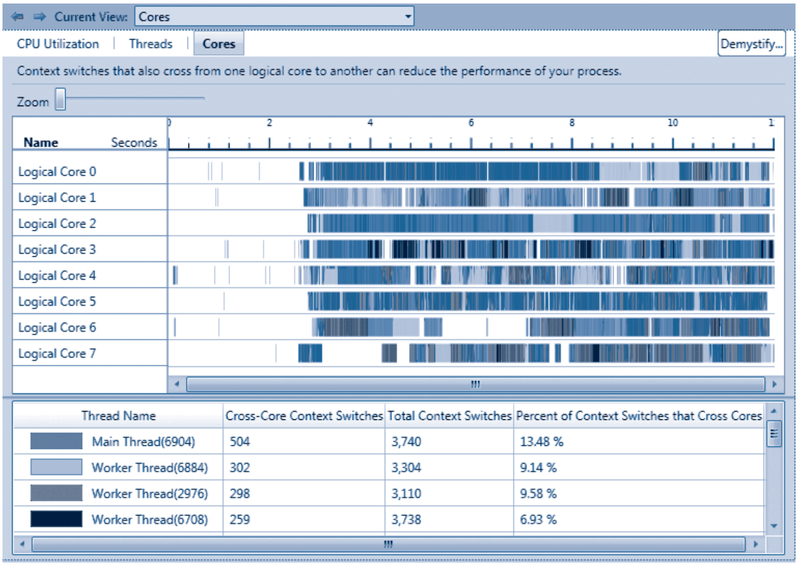

Current View: CPU Utilization
CPU Utilization
Threads
Cores
Demystify...
Zoom
Average CPU utilization for this process: 67%
8
7
6
5
4
3
2
1
0
Number of Logical Cores
0.0
3004.7
6009.3
9014.0
12018.7
Time (ms)
Other Processes
System Process
Idle Process
ImagePipeline (PID=6044)

FIGURE 5
Detail of CPU Utilization view

Figure 6 illustrates the Cores view. 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.) In this example, between 0 and 2.5 on the time scale, the application is idle with little or no work running on any core. Between 2.5 and 12 the pipeline is filled and more tasks are eligible to run than there are cores. Several threads alternate on each core and the table beneath the graph shows that there is some context switching across cores.

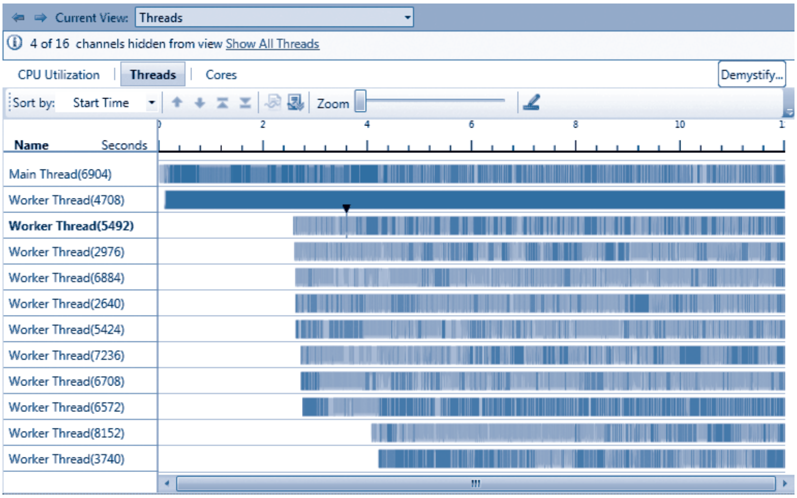

Current View: Cores
CPU Utilization
Threads
Cores
Demystify...
Zoom
Name
Seconds
0
2
4
6
8
10
1
Logical Core 0
Logical Core 1
Logical Core 2
Logical Core 3
Logical Core 4
Logical Core 5
Logical Core 6
Logical Core 7

Context switches that also cross from one logical core to another can reduce the performance of your process.

Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
Main Thread(6904)	504	3,740	13.48 %
Worker Thread(6884)	302	3,304	9.14 %
Worker Thread(2976)	298	3,110	9.58 %
Worker Thread(6708)	259	3,738	6.93 %

FIGURE 6
Detail of Cores view

Figure 7 illustrates the Threads view. The Threads view shows how each thread spends its time. The upper part of the view is a timeline with color-coded bands that indicate different types of activity. For example, red indicates when the thread is synchronizing (waiting for something). In this example, the Threads view initially shows the Main Thread and Worker Thread 4708. Later, more threads are added to the thread pool as the pipeline starts processing images. Not all threads are visible in the view pictured here. (You can hide individual threads by right-clicking the view and then clicking **Hide**). This view also shows that the main thread is active throughout; the green-brown color indicates user interface activity. Other threads show segments of green, denoting execution, red indicating synchronization, and yellow for preempted treads.



Current View: Threads

4 of 16 channels hidden from view Show All Threads

Sort by: Start Time

CPU Utilization
Threads
Cores
Demystify...
Zoom
Name
Seconds
0
2
4
6
8
10
1
Main Thread(6904)
Worker Thread(4708)
Worker Thread(5492)
Worker Thread(2976)
Worker Thread(6884)
Worker Thread(2640)
Worker Thread(5424)
Worker Thread(7236)
Worker Thread(6708)
Worker Thread(6572)
Worker Thread(8152)
Worker Thread(3740)

FIGURE 7
Detail of Threads view

After 2.5 on the timeline, some pipeline threads execute frequently but others execute almost continuously. There are more pipeline threads than cores, so some pipeline threads must alternate between running and being preempted.

Scenario Markers

You can use the Scenario library to mark different phases of complex applications. The following code shows an example. (The Scenario library is a free download on the MSDN® Code Gallery website. For more information, see "Scenario Marker Support" on MSDN at http://msdn.microsoft.com/en-us/library/dd984115.aspx.)

```
#include “Scenario.h”

// ...

shared_ptr<Scenario> myScenario = shared_ptr<Scenario>(new
Scenario());
myScenario->Begin(0, L”Main Calculation”);

// Main Calculation Phase...

myScenario->End();
```

These markers will be displayed in all three views. They appear as a band across the top and bottom of the timeline. Move the mouse over the band to see the marker name. You can see an example of this later on in Figure 11. Don't use too many markers as they can easily overwhelm the visualization and make 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 focus primarily on ways to express potential parallelism. However, there are other types of patterns that are useful in parallel development. The human mind is very good at recognizing visual patterns, and the Concurrency Visualizer takes advantage of this. You can learn to identify some common visual patterns that occur when an application has specific performance problems. This section describes visual patterns that will help you to recognize and fix oversubscription, lock contention, and load imbalances.

More examples of common patterns indicating poorly behaved parallel applications are discussed on MSDN. See the "Further Reading" section for more information. You can also access this content from a link in the **Hints** tab in the Threads view. As seen earlier, Figure 4 shows the link in this tab.

OVERSUBSCRIPTION

Oversubscription occurs when there are more threads than logical processors to run them. Oversubscription can cause poor performance because of the high number of context switches, each of which takes some processing time and which can decrease the benefits provided by memory caches.

Oversubscription is easy to recognize in the Concurrency Visualizer because it causes large numbers of yellow regions in the profiler trace. Yellow means that a thread was preempted (the thread was switched out). When profiled, the following code yields a quintessential depiction of oversubscription.

```
void Oversubscription()
{
  task_group tasks;

  for (unsigned int p = 0; p < (GetProcessorCount() * 4); p++)
  {
    tasks.run([]()
    {
      // Oversubscribe in an exception safe manner
      scoped_oversubcription_token oversubscribe;
      // Do work
```

```
   delay(1000000000);
  });
 }
 tasks.wait();
}
```

Figure 8 illustrates the Threads view from one run of this function on a system with eight logical cores. It produces a very distinct pattern.

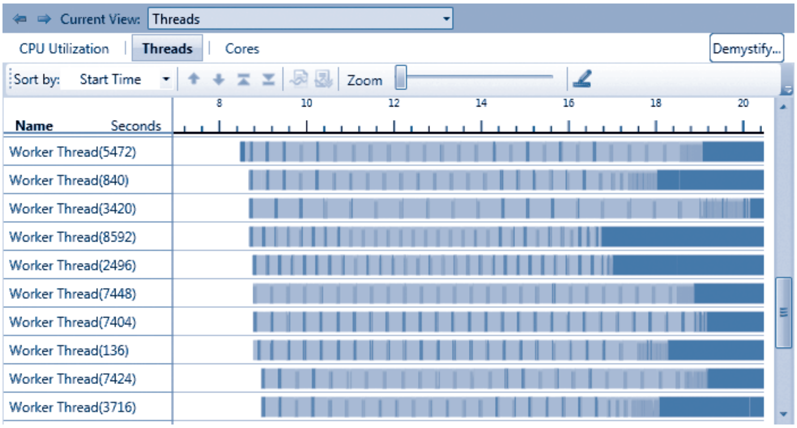

Current View: Threads
CPU Utilization
Threads
Cores
Demystify...
Sort by: Start Time
Zoom
Name
Seconds
8
10
12
14
16
18
20
Worker Thread(5472)
Worker Thread(840)
Worker Thread(3420)
Worker Thread(8592)
Worker Thread(2496)
Worker Thread(7448)
Worker Thread(7404)
Worker Thread(136)
Worker Thread(7424)
Worker Thread(3716)

FIGURE 8
Threads view that shows oversubscription

LOCK CONTENTION AND SERIALIZATION

Contention occurs when a thread attempts to acquire a lock that is held by another thread. In many cases, this results in the second thread blocking until the lock is released. The Threads view of the Concurrency Visualizer depicts blocking in red. It is often a sign of decreased performance. In extreme cases, an application can be fully serialized by one or more locks, even though multiple threads are being used. You can see this in Figure 9, where the narrow bright green areas representing execution only appear in one worker thread at a time. In such cases, the performance may be much worse than in a conventional, serial version of the application.

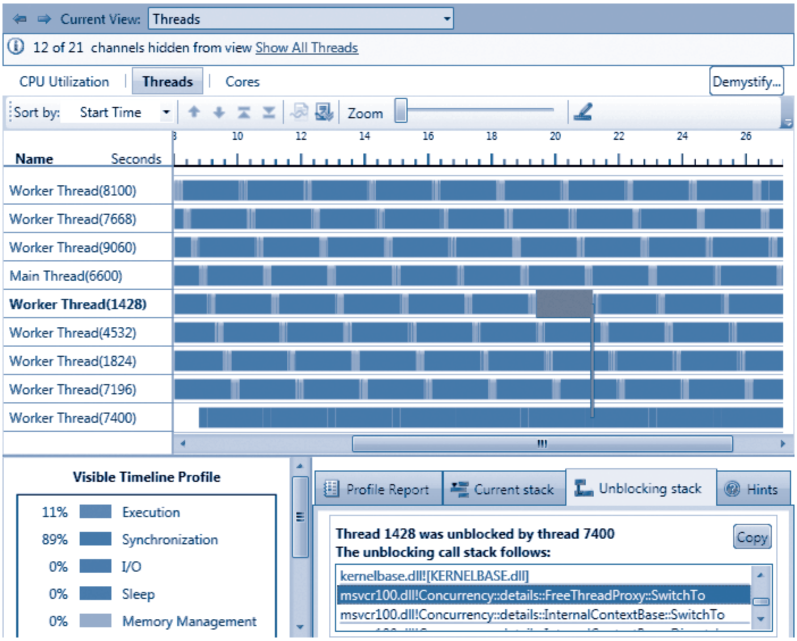
The following **LockContention** method produces a lock convoy, which leads to significant lock contention and serialization of the program even though multiple threads are in use. A lock convoy is a performance problem that occurs when multiple threads contend for a frequently shared resource.

```
void LockContention()
{
 task_group tasks;
 reader_writer_lock lock;
```

```
  for (unsigned int p = 0; p < GetProcessorCount(); p++)
  {
    tasks.run([&lock]()
    {
      for (int i = 0; i < 10; i++)
      {
        // Do work
        delay(100000);

        // Do protected work
        lock.lock();
        delay(100000000);
        lock.unlock();
      }
    });
  }
  tasks.wait();
}
```

Figure 9 illustrates the pattern this code produced in the Threads view of the Concurrency Visualizer.


Current View: Threads
CPU Utilization
Threads
Cores
Demystify...
Sort by: Start Time
Zoom
Name
Seconds
8
10
12
14
16
18
20
22
24
26
Worker Thread(8100)
Worker Thread(7668)
Worker Thread(9060)
Main Thread(6600)
Worker Thread(1428)
Worker Thread(4532)
Worker Thread(1824)
Worker Thread(7196)
Worker Thread(7400)
Profile Report
Current stack
Unblocking stack
Hints
Copy

12 of 21 channels hidden from view Show All Threads

Visible Timeline Profile

11%	Execution
89%	Synchronization
0%	I/O
0%	Sleep
0%	Memory Management

Thread 1428 was unblocked by thread 7400
The unblocking call stack follows:

```
kernelbase.dll![KERNELBASE.dll]
msvcr100.dll!Concurrency::details::FreeThreadProxy::SwitchTo
msvcr100.dll!Concurrency::details::InternalContextBase::SwitchTo
```

FIGURE 9
Threads view showing lock convoy

Clicking on one of the Synchronization (red) blocks highlights it and also indicates which thread blocked it. You can examine the stack of the unblocking thread by clicking the **Unblocking stack** tab. This allows you to see the call stack of the thread that unblocked the blocked thread. In this example, the maroon block on thread 1428 indicates that it was waiting for thread 7400, which was executing a call to the **ThreadProxy::SwitchTo** method. After thread 1428 is unblocked it continues to execute and is displayed as a green block in the Threads view. You can click on any red synchronization block to examine its unblocking stack, if one is available. When examining call stacks, remember that you can quickly view the source by double-clicking on the stack frame. This allows you to associate segments on the Thread view with the code that was executing during that segment.

LOAD IMBALANCE

A load imbalance occurs when work is unevenly distributed across all the threads that are involved in a parallel operation. Load imbalances mean that the system is underutilized because some threads or cores are idle while others finish processing the operation. The visual pattern produced by a load imbalance is recognizable in several of the Concurrency Visualizer views. The following code creates a load imbalance.

```
void LoadImbalance()
{
  const int loadFactor = 20;

  parallel_for_fixed(0, 100000, [loadFactor](int i)
  {
    // Do work
    delay(i, loadFactor);
  });
}
```

Although most of the parallelism support in the PPL uses dynamic partitioning to apportion work to a pool of tasks, the **parallel_for_fixed** method included in concrt_extras.h, which is available at http://code.msdn.microsoft.com/concrtextras, uses fixed partitioning of iterations, without range stealing. The code example shown here, when run on a system with eight logical cores, causes elements [0, 12499] to be processed by one task, elements [12500, 24999] to be processed by another task, and so on, through the entire range. The body of the workload iterates from 0 to the current index value, which means that the amount of work to be done is proportional to the index. Workers that process lower ranges will have significantly less work to do than the workers that process the upper ranges. As a consequence, the tasks for each sub-range finish at different times. This load imbalance is an inefficient use of the processors. Figure 10, which is the CPU Utilization view in the Concurrency Visualizer, illustrates this.

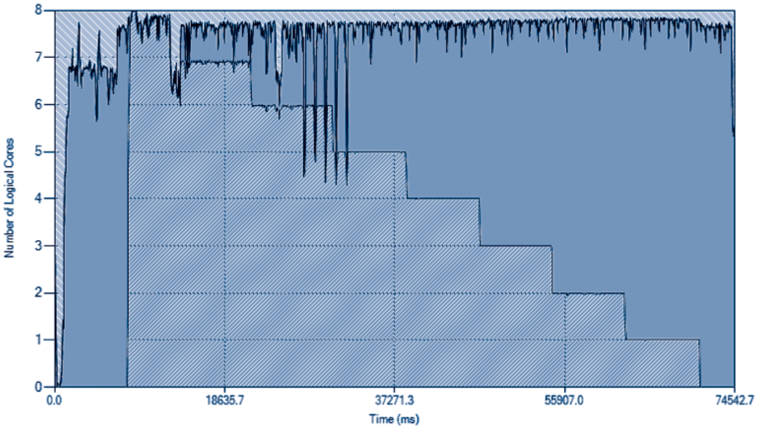
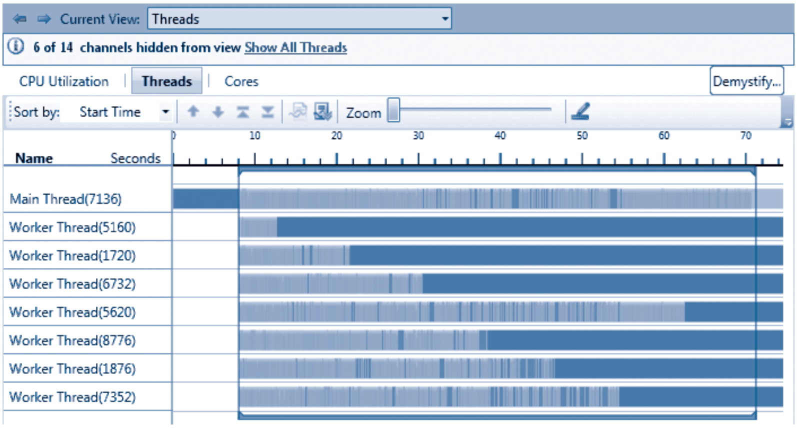


FIGURE 10
CPU view that shows a load imbalance

When the **LoadImbalance** method begins to execute, all eight logical cores on the system are being used. However, after a period of time, usage drops as each core completes its work. This yields a stair-step pattern, as threads are dropped after they complete their portion of the work. The Threads view confirms this analysis. Figure 11 illustrates the Threads view.


Current View: Threads

6 of 14 channels hidden from view Show All Threads

Sort by: Start Time

CPU Utilization
Threads
Cores
Demystify...
Zoom
Name
Seconds
0
10
20
30
40
50
60
70
Main Thread(7136)
Worker Thread(5160)
Worker Thread(1720)
Worker Thread(6732)
Worker Thread(5620)
Worker Thread(8776)
Worker Thread(1876)
Worker Thread(7352)

FIGURE 11
Threads view that shows a load imbalance

The Threads view shows that after completing a portion of the work, the worker threads were idle while they waited for the Main Thread, 7136, to complete the remaining work. The example is a console application and, in this case, the runtime used the main thread to run one of the tasks.

Further Reading

The Parallel Performance Analysis blog at MSDN discusses many techniques for analyzing your code and includes many examples. MSDN also provides information about the Scenario library.

Parallel Development in Visual Studio 2010 blog on MSDN:
http://blogs.msdn.com/b/visualizeparallel/.

"Concurrency Visualizer" on MSDN:
http://msdn.microsoft.com/en-us/library/dd537632.aspx.

"Performance Tuning with the Concurrency Visualizer in Visual Studio 2010" on MSDN:
http://msdn.microsoft.com/en-us/magazine/ee336027.aspx.

Scenario Home Page on MSDN:
http://code.msdn.microsoft.com/scenario.

The Parallel Tasks and Parallel Stacks windows, and other features of the debugger to support parallel programming:
http://www.danielmoth.com/Blog/Parallel-Debugging.aspx
http://msdn.microsoft.com/en-us/library/ms164746.aspx.

"Debugging Task-Based Parallel Applications in Visual Studio 2010" on MSDN:
http://msdn.microsoft.com/en-us/magazine/ee410778.aspx.

Common Patterns for Poorly-Behaved Multithreaded Applications:
http://msdn.microsoft.com/en-us/library/ee329530.aspx.

Improve Debugging And Performance Tuning With ETW:
http://msdn.microsoft.com/en-us/magazine/cc163437.aspx.

Appendix C Technology Overview

Appendix C describes some of the parallel computing resources offered by Microsoft® that are not covered in this book. The "Further Reading" section includes URLs for websites that have more information. Figure 1 illustrates the different offerings and how they are related.

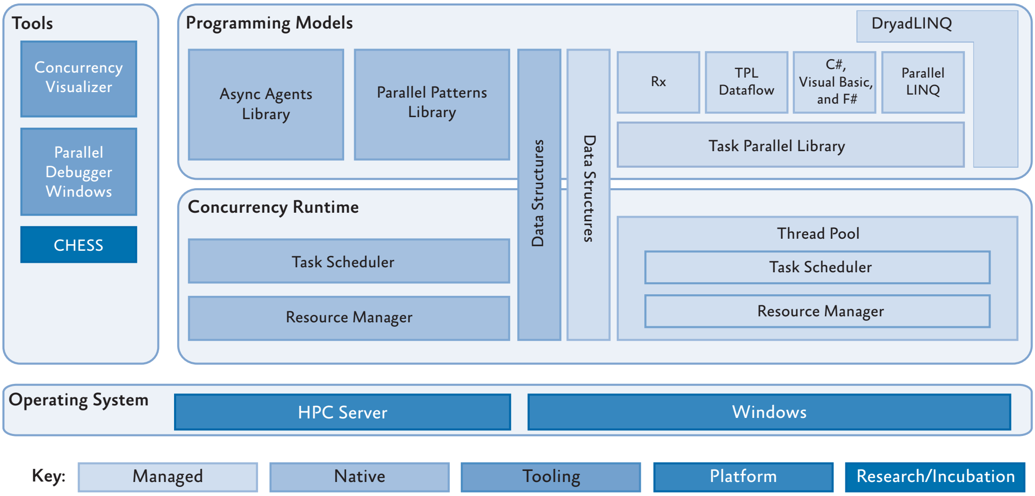

Tools
Concurrency Visualizer
Parallel Debugger Windows
CHESS
Programming Models
DryadLINQ
Async Agents Library
Parallel Patterns Library
Data Structures
Data Structures
Rx
TPL Dataflow
C#, Visual Basic, and F#
Parallel LINQ
Task Parallel Library
Concurrency Runtime
Task Scheduler
Resource Manager
Thread Pool
Task Scheduler
Resource Manager
Operating System
HPC Server
Windows
Key:
Managed
Native
Tooling
Platform
Research/Incubation

FIGURE 1
Microsoft parallel programming resources

This book covers the Parallel Patterns and Asynchronous Agents libraries. These libraries use the native Concurrency Runtime, which includes a task scheduler and a resource manager that execute native parallel workloads on multicore architectures.

The Microsoft Visual C#®, Visual Basic®, and Parallel LINQ (PLINQ) languages ship with the Microsoft Visual Studio® 2010 development system. These languages use the Task Parallel Library (TPL) to support parallelism. The F# language also ships with Visual Studio. It exposes a more functional approach to parallelism than the other languages, and it emphasizes immutable data types. However, the F# runtime libraries build on and integrate with TPL, and the F# PowerPack includes parallelization support that is built on top of PLINQ. How to develop parallel applications that use the Microsoft .NET Framework 4, TPL, and PLINQ is covered in the companion to this book, *Parallel Programming with Microsoft .NET*. Another library, Reactive Extensions (Rx), allows you to use observable collections to compose asynchronous and event-based programs.

The Accelerator API (not shown in the diagram) provides a functional programming model for implementing array-processing operations. Accelerator handles all the details of parallelizing and running the computation on the selected target processor, which includes graphics processing units and multicore CPUs. The Accelerator API is largely processor independent so, with only minor changes, the same array-processing code can run on any supported processor.

Visual Studio 2010 contains several tools for debugging and profiling parallel applications. For examples of how to use them, see Appendix B, "Debugging and Profiling Parallel Applications." You can also use the CHESS tools from Microsoft Research to detect bugs in your parallel code.

CHESS and Accelerator 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're encouraged to download them, evaluate them, and provide the respective teams with feedback.

DryadLINQ is a programming environment for writing large-scale data parallel applications that run on High Performance Computing (HPC) clusters. DryadLINQ combines two important pieces of Microsoft technology: the Dryad distributed execution engine and the .NET Language Integrated Query (LINQ).

All of the above technologies, with the exception of DryadLINQ, are largely for parallelism on a single computer. Windows HPC Server targets clusters of servers and supports scale-out across many computers. Although 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, see:

http://msdn.microsoft.com/concurrency.

For information about F#, including the language reference and walk-throughs, see the Microsoft F# Developer Center at:

http://msdn.microsoft.com/fsharp.

Windows HPC Server 2008 R2 product information and developer resources are available on the Windows HPC Server site at:

http://www.microsoft.com/hpc.

A Community Technology Preview (CTP) of DryadLINQ is available for download at:

http://technet.microsoft.com/library/ee815854(WS.10).aspx.

Details of the Accelerator project are available at:

http://research.microsoft.com/accelerator.

Descriptions and downloads for Rx are available at:

http://msdn.microsoft.com/en-us/data/gg577609.

CHESS is a Microsoft DevLabs project, and is available at:

http://msdn.microsoft.com/devlabs.

A more in-depth overview of possible future directions for support of parallel programming in Visual Studio can be found in Stephen Toub's talk at TechEd Europe 2010 at:

http://www.msteched.com/2010/Europe/DEV208.

Glossary

agent. See *asynchronous agent*.

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.

associative operation. A binary operation is associative if, for a sequence of operations, the order in which the operations are performed does not change the result. For example (a + b) + c = a + (b + c).

asynchronous. An operation that that does not block the current thread of control when the operation starts.

asynchronous agent. A software component that works asynchronously with other agents as part of a larger computation. Often shortened to *agent*.

asynchronous pipeline. A pipeline in which tasks are only created when data becomes available.

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. Compare to *foreground thread*.

barrier. A synchronization point where all participating threads must stop and wait until every thread reaches it.

block. To pause execution while waiting for some event or condition.

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 from an enclosing lexical scope.

commutative operation. A binary operation is commutative if changing the order of the operands does not change the result. For example, a + b = b + a. Examples of commutative operations are scalar addition and scalar multiplication.

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 safe. When a block of code can be run on multiple cores simultaneously without introducing errors.

Concurrency Visualizer. An addition to the Microsoft® Visual Studio® development system's 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.

control flow. A basis for coordination whereby tasks execute according to the steps of an algorithm, as in a parallel loop.

control-flow agent. Agents whose run methods contain sequential loops that process incoming values from a messaging block.

cooperative blocking. A programming idiom whereby the current context waits for a resource to become available or for a signal to occur. The task scheduler is notified when a cooperative blocking operation occurs. Compare with *non-cooperative blocking*.

cooperative cancellation. A programming idiom that uses cooperative blocking to implement operations that are capable of being canceled before they are completed.

cooperative context switch. When a worker thread becomes blocked as a result of a cooperative blocking operation. See *cooperative blocking*.

coordination. Arranging for tasks to work together to ensure a correct outcome. Coordination can 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.

data flow. A basis for coordination where tasks execute when data becomes available, as in a pipeline or task graph. Compare to *control flow*.

data parallelism. A form of parallel processing whereby the same computation executes in parallel on different data. Data parallelism is supported in the Parallel Patterns Library (PPL) by the **parallel_for** and **parallel_for_each** functions. 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 reads and updates a variable without synchronization.

deadlock. When execution stops and cannot resume because the system is waiting for a condition that cannot occur. Threads can deadlock when one holds resources that another needs. Compare to *livelock*.

decomposition. To break a problem into smaller parts. For parallel processing, decomposition can be by data or by task.

degree of parallelism. The number of parallel tasks that may execute concurrently at any one time.

dependency. When one operation uses the results of another. When there is a dependency between operations, they cannot run in parallel. Compare to *independent*.

double-checked locking. Process in which one first tests a condition, then, only if the condition is true, acquires a lock and tests the same condition again, this time to determine whether to update shared data. This maneuver can often avoid the expensive operation of acquiring a lock when it will not be useful.

dynamic partitioning. Data partitioning whereby the parts are selected as the parallel tasks execute. Compare to *static partitioning*.

enhanced locality mode. When a scheduler tries to execute related tasks in batches.

foreground thread. A thread that keeps a process running. After all its foreground threads stop, the process shuts down. Compare to *background thread*.

fork/join. A parallel computing pattern that uses task parallelism. Fork occurs when tasks start; join occurs when all tasks finish.

forward progress mode. When a scheduler executes one pending task from each of its schedule groups in round-robin fashion.

future. A task that returns a value.

function object. See *functor*.

functor. A language feature that allows an instance of a class to be invoked as though it were a function. In C++ *functors* are defined as a class with a definition for **operator()**.

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. Simultaneous multithreading (also sometimes known as hyperthreading) enables more than one hardware thread to execute on a single core. Each hardware thread is considered a separate logical processor.

hyperthreading. See *simultaneous multithreading*.

immutable. Property of data that means it cannot be modified after it's created. For example, strings provided by some libraries 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. Compare to *dependency*.

kernel mode. The mode of execution in which the Microsoft Windows® kernel runs and has full access to all resources. Compare to *user mode*.

lambda expression. An anonymous function that can capture variables from its enclosing lexical scope.

livelock. When execution continues but does not make progress toward its goal. Compare to *deadlock*.

load balancing. When similar amounts of work are assigned to different tasks so that the available processors are used efficiently. Compare to *load imbalance*.

load imbalance. When different amounts of work are assigned to different tasks so that some tasks don't have enough work to do, and the available processors are not used efficiently. Compare to *load balancing*.

lock. A synchronization mechanism that ensures that 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.

manycore. Multicore, usually with more than eight logical processors.

map. A parallel computation where multiple tasks independently perform the same transformation on different data. An example of *data parallelism*.

map/reduce. 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. Compare to *set*.

multiset union. An operation that combines *multisets* by merging their elements and adding the multiplicities of each element.

mutable. Property of data that means that it can be modified after it is created. Not to be confused with the C++ **mutable** keyword. Compare to *immutable*.

mutable type. A type whose instances are mutable.

nested parallelism. When one parallel programming construct appears within another.

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.

noncooperative blocking. Lower-level blocking operations that are provided by the operating system. The task scheduler is unaware of noncooperative blocking; to the scheduler, the blocked tasks are indistinguishable from running tasks. See *cooperative blocking*.

NUMA (non-uniform memory access). A computer memory architecture in which memory access time depends on the memory location relative to a processor.

object graph. A data structure consisting of objects that reference each other. Object graphs are often of 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, when 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 known as 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 Compare to *thread*, *task*.

processor affinity mask. A bit mask that tells the scheduler which processor(s) a thread should be run on.

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) and returns only 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 situation in which the observable behavior of a program depends on the order in which parallel operations complete. Race conditions are usually errors; good programming practices prevent them.

recursive decomposition. In parallel programming, refers to the situation in which the tasks themselves can start more tasks.

reduce. A kind of aggregation whereby data is combined by an operation that is associative, which often makes it possible to perform much of the reduction in parallel.

round robin. A scheduling algorithm whereby 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.

runnable context. A thread that was previously interrupted by a cooperative blocking operation but is now unblocked and ready to resume its work.

scalable. A parallel computation whose performance improves when more processors are available is said to be scalable.

schedule group. An internal data structure used by the scheduler that represents a queue of pending tasks.

semaphore. A synchronization mechanism that ensures that 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. Read/write access to mutable shared data requires synchronization.

single-threaded data type. A type that 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 whereby the parts are selected before the program executes. Compare to *dynamic partitioning*.

sentinel value. A user-defined value that acts as an end-of-file-token.

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 with the PPL and Asynchronous Agents libraries. Compare to *thread, process*.

task graph. Can be seen as a directed graph when tasks provide results that are the inputs to other tasks. The nodes are tasks, and the arcs are values that act as inputs and outputs of the tasks.

task group. A group of tasks that are either waited on or cancelled together. The **task_group** class is thread-safe.

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 whereby different computations execute in parallel on different data. 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. Compare to *process, task*.

thread affinity. When certain operations must only be performed by a particular thread.

thread pool. A collection of threads managed by the system that are used to avoid the overhead of creating and disposing threads.

thread safe. See *concurrency safe*.

thread starvation. When tasks are unable to run because there are no virtual processor objects available to run them in the scheduler.

thread-local state. Variables that are accessed by only one thread. No locking or other synchronization is needed to safely access thread-local state. The **combinable<T>** class is a recommended way to establish thread-local state when using the PPL and Asynchronous Agents libraries.

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 be accessed by only one thread at a time, and its operations are atomic with respect to multiple threads. Compare to *single-threaded data type*.

torn read. When reading a variable requires more than one machine instruction, and another task writes to the variable between the read instructions. Compare to *torn write*.

torn write. When writing a variable requires more than one machine instruction, and another task reads the variable between the write instructions. Compare to *torn read*.

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 poor performance.

undersubscription. When there are fewer tasks than there are processors available to run them, so processors remain idle.

user mode. The mode in which user applications run but have restricted access to resources. Compare to *kernel mode*.

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 functions. The arguments to **parallel_invoke** or **task_group::run**.

work stealing. When a thread executes a task queued for another thread, in order to remain busy.

Index

A

Accelerator API, 150
accumulate function, 47-48
acknowledgments, xxi-xxii
Adatum Dashboard, 65-70
AgentBase class, 101-102
agent class, 86
agent::wait_for_*methods, 33
agent::wait method, 33
Aggregation pattern, 57
alpha blending, 29
Amdahl's law, 8-10, 16
    performance, 96-97
AnalysisEngine class, 67-70
anti-patterns
    parallel_for function, 23
    Parallel Loop pattern, 23-24
    parallel tasks pattern, 37-39
    Pipeline pattern, 107
    resource manager, 129-131
appendixes *see* parallel application debugging and profiling; task scheduler and resource manager; technology overview
asend function, 33
Asynchronous Agents Library, xvi, 1
    Pipeline pattern, 85
asynchronous pipelines, 97-101
audience, xv

B

Block method, 127
bottlenecks, 70-71
breakpoints and memory allocation, 136-137
business objects, 66-67

C

caching suballocator, 127-128
call class, 98-101
call messaging block, 86
cancellation
    futures, 70
    is_canceling method, 34-35
    IsCancellationPending method, 104
    pipelines, 101-102
    querying for, 126
    task group cancellation, 33-35
    unintended propagation of cancellation requests, 38
cartoons, xix, 12, 25, 44, 60, 74, 84
CHESS, 150
choice messaging block, 86
closures, 37-38
code examples, 2
combinable class, 46-49, 55-56
combinable objects, 55
concurrency
    visualizer, 137-142
    vs. parallelism, 8

Concurrency namespace, 31-34
Concurrency Runtime
    components, 112
    sample pack, 47
Context::Block method, 33
Context class, 126-127
Context::Oversubscribe method, 128
Context::Yield method, 33
contributors and reviewers, xxi-xxii
cooperative blocking, 31-33
    interface to, 127
    scheduling interactions with, 24
cooperative context switch, 115
coordination, 3
Cores view, 139-140
CorrectCaseAgent class, 90-91
CPU Utilization view, 139
CPU view that shows a load imbalance, 146
credit analysis parallel review example, 18-19
credit analysis sequential review example, 18
critical_section::lock method, 32
CurrentContext method, 126
CurrentScheduler class, 119

D
data accessors, 16
dataflow network defined, 85
data parallelism, 27
data pipeline defined, 85
data structures of schedule groups, 122
deadlocks, 5, 135-136
    from thread starvation, 131
debugging
    information, 126
    parallel application debugging and profiling, 133-147
decomposition
    Futures pattern, 72
    introduction, 3-6
    overview, 3
degree of parallelism, 22
dependent loop bodies, 16-17
depthRemaining argument, 79
design
    approaches, 6
    notes, 72
Detach method, 120
Discrete Event pattern, 73
Divide and Conquer pattern, 75
DoAnalysisParallel method, 70
DoCpuIntensiveOperation function, 37
DryadLINQ, 150
duplicates in the input enumeration, 23
dynamic resource management, 115
Dynamic Task Parallelism pattern, 75-83
    adding tasks to a pending wait context, 81-83
    basics, 75-77
    depthRemaining argument, 79
    exercises, 83
    further reading, 83
    Futures pattern, 73
    ParallelQuickSort method, 78-79
    ParallelSubtreeHandler function, 82
    ParallelTreeUnwinding function, 82
    ParallelWhileNotEmpty1 method, 80-81
    QuickSort algorithm, 78
    sequential code, 76
    SequentialWalk method, 76
    variations, 80-83
    wait method, 81-82

E
enhanced locality mode, 124-125
EnterCriticalSection function, 40
environment, 116
event::wait method, 33
exception handling, 20-21
    parallel tasks pattern, 35
exercises
    Dynamic Task Parallelism pattern, 83
    Futures pattern, 73
    introduction, 11
    parallel tasks pattern, 42
    Pipeline pattern, 109

F

Fit method, 18
Fork/Join pattern, 27
forward progress mode, 124-125
FreePipelineSlot method, 92
FriendMultiSet type, 51-55
FriendsSet type, 51-52
functional style, 72
further reading
    Dynamic Task Parallelism pattern, 83
    Parallel Aggregation pattern, 58-59
    parallel application debugging and profiling, 147
    Parallel Loop pattern, 25
    parallel tasks pattern, 42
    Pipeline pattern, 109
    technology overview, 151
Future class, 61-62
future defined, 61
Futures pattern, 61-73
    Adatum Dashboard, 65-70
    AnalysisEngine class, 67-70
    basics, 62-70
    bottleneck removal, 70-71
    business objects, 66-67
    decomposition into futures, 72
    design notes, 72
    Discrete Event pattern, 73
    Divide and Conquer pattern, 73, 75
    DoAnalysisParallel method, 70
    Dynamic Task Parallelism pattern, 73
    exercises, 73
    functional style, 72
    Future class, 61-62
    futures
        canceling, 70
        defined, 61
    MarketRecommendation object, 68-70
    Master/Worker pattern, 73
    modifying the graph at run time, 71
    Pipeline pattern, 73
    related patterns, 72-73
    Result method, 64-65
    scoped_oversubscription_token class, 70
    single_assignment class, 62
    StockDataCollection type, 66-67
    task graph, 62
    variations, 70-72

G

GetParent method, 16
GetProcessorCount function, 116
GetProcessorNodeCount function, 116
glossary, 153-162
goals, xx
granularity, 3-4, 16
graphs, 71
guide, xvii

H

hard coding, 2
Hey, Tony, xi-xii

I

ignored process affinity mask, 131-132
ImageAgentPipelineDataFlow class, 98-101
ImagePerformanceData object, 136-137
image pipeline, 94-96
    with load balancing diagram, 106
ImagePipelineDlg class, 101-102
ImagePipeline sample application, 92
IncrementIfPrime function, 47-48
infinite waits, 107
interface to cooperative blocking, 127
introduction, 1-12
    Amdahl's law, 8-10
    decomposition, coordination, and scalable sharing, 3-6
    exercises, 11
    more information, 11
    table of patterns, 7
    terminology, 8
is_canceling method, 34-35
IsCancellationPending method, 104
isolation, 107
IVirtualProcessorRoot class, 113

J
join messaging block, 86

L
LeaveCriticalSection function, 40
lightweight tasks
    parallel tasks pattern, 38, 41
    task scheduler and resource manager, 116-117, 122-123
loads
    balancing with multiple producers, 104-106
    imbalance, 145-147
LockContention method, 143-145
locks, 5-6
long-running I/O tasks, 128
loop-carried dependence, 17
loops
    breaking out of early, 19-20
    dependent loop bodies, 16-17
    hidden loop body dependencies, 23
    parallel loops vs. pipelines, 86
    see also Parallel Loop pattern

M
map phase, 50
Map/Reduce pattern, 49-55
MarketRecommendation object, 68-70
Master/Worker pattern, 73
    parallel tasks pattern, 27
MaxConcurrency policy key, 129
memory allocation and breakpoints, 136-137
messages, 107
messaging blocks, 86
Microsoft parallel programming resources, 149
MinConcurrency policy key, 129
multisets, 50
    union, 51
multitype_join messaging block, 86

N
Non-Uniform Memory Architecture (NUMA), xiii, 112-114

O
operator(), 29
out of scope material, xx, 131
oversubscription, 142-143
    cores, 116
overwrite_buffer messaging block, 86

P
Pack pattern, 57-58
Parallel Aggregation pattern, 45-59
    accumulate function, 47-48
    Aggregation pattern, 57
    basics, 46-49
    combinable class, 46-49, 55-56
    Concurrency Runtime sample pack, 47
    considerations for small loop bodies, 55
    design notes, 55-57
    example, 49-55, 58
    exercises, 58
    FriendMultiSet type, 51-55
    FriendsSet type, 51-52
    further reading, 58-59
    IncrementIfPrime function, 47-48
    map phase, 50
    Map/Reduce pattern, 49-55
    multiset, 50
    multiset union, 51
    other uses for combinable objects, 55
    Pack pattern, 57-58
    parallel_reduce function, 48, 53-55
    parallel_transform function, 53-55
    postprocessing phase, 50
    Reduce pattern, 57
    reduce phase, 50
    related patterns, 57
    Scan pattern, 57-58
    Stencil pattern, 58
    variations, 55

- parallel application debugging and profiling, 133-147
  - breakpoints and memory allocation, 136-137
  - concurrency visualizer, 137-142
  - Cores view, 139-140
  - CPU Utilization view, 139
  - CPU view that shows a load imbalance, 146
  - deadlocks, 135-136
  - further reading, 147
  - ImagePerformanceData object, 136-137
  - load imbalance, 145-147
  - lock contention and serialization, 143-145
  - LockContention method, 143-145
  - oversubscription, 142-143
  - Parallel Stacks window, 133-136
  - Parallel Tasks window, 133-136
  - Scenario library, 141-142
  - scenario markers, 141-142
  - Threads view, 140-141
  - Threads view of the Concurrency Visualizer, 138
  - Threads view showing lock convoy, 144-145
  - Threads view that shows a load imbalance, 146
  - visual patterns, 142-147
- parallel_for_each function, 14-16
  - controlling the degree of parallelism, 22
- parallel_for function, 14-16, 21-22
  - anti-patterns, 23
  - controlling the degree of parallelism, 22
  - overloading, 21-22
  - Parallel Loop pattern, 23
- parallel_invoke function, 28-30, 33, 41
- parallelism
  - degree of parallelism, 22
  - limits of, 8-10
  - potential parallelism described, 2
  - vs. concurrency, 8
- Parallel Loop pattern, 13-25
  - anti-patterns, 23-24
  - breaking out of loops early, 19-20
  - credit analysis parallel review example, 18-19
  - credit analysis sequential review example, 18
  - data accessors, 16
  - degree of parallelism, 22
  - dependent loop bodies, 16-17
  - duplicates in the input enumeration, 23
  - exception handling, 20-21
  - exercises, 24
  - Fit method, 18
  - further reading, 25
  - GetParent method, 16
  - hidden loop body dependencies, 23
  - interactions with cooperative blocking, 24
  - loop-carried dependence, 17
  - parallel_for_each function, 14-16
    - controlling the degree of parallelism, 22
  - parallel_for function, 14-16, 21-22
    - anti-patterns, 23
    - controlling the degree of parallelism, 22
    - overloading, 21-22
  - parallel_for function overloading, 21-22
  - profiling, 17
  - related patterns, 24
  - sequential credit review example, 18
  - small loop bodies with few iterations, 23
  - special handling of small loop bodies, 21-22
  - UpdatePredictionsParallel method, 18-19
  - UpdatePredictionsSequential method, 18
  - variations, 19-22
  - when to use, 17-19
- parallel loops vs. pipelines, 86
- Parallel Patterns Library (PPL), xv, 1
- parallel programming, 1
  - basic precepts, 10-11
  - patterns, xvii

ParallelQuickSort method, 78-79
parallel_reduce function, 48, 53-55
ParallelSubtreeHandler function, 82
parallel tasks pattern, 27-42
    agent::wait_for_* methods, 33
    agent::wait method, 33
    alpha blending, 29
    anti-patterns, 37-39
    asend function, 33
    basics, 28-29
    cancellation requests, 38
    Concurrency namespace, 31-34
    Context::Block method, 33
    Context::Yield method, 33
    cooperative blocking, 31-33
    critical_section::lock method, 32
    data parallelism, 27
    design notes, 39-41
    DoCpuIntensiveOperation function, 37
    EnterCriticalSection function, 40
    event::wait method, 33
    example, 29-31
    exception handling, 35
    exercises, 42
    Fork/Join pattern, 27
    further reading, 42
    how tasks are scheduled, 40-41
    is_canceling method, 34-35
    LeaveCriticalSection function, 40
    lightweight tasks, 38, 41
    Master/Worker pattern, 27
    operator(), 29
    parallel_invoke function, 28-30, 33, 41
    reader_writer_lock::lock method, 32
    reader_writer_lock::lock_read method, 32
    reader_writer_lock::scoped_lock constructor, 32
    reader_writer_lock::scoped_lock_read constructor, 32
    receive function, 33
    Resource Acquisition is Initialization (RAII) pattern, 34
    run_and_wait method, 31
    SearchLeft function, 36
    section::scoped_lock constructor, 32
    send function, 33
    speculative execution, 36-39
    structured task groups and task handles, 41
    synchronization costs, 39
    task group calling conventions, 39
    task group cancellation, 33-35
    task_group::cancel method, 33-35
    task_group class, 27, 39
    task_group::run method, 29
    task_group::wait method, 31-32, 35
    task parallelism, 27
    tasks and threads, 40
    variables captured by closures, 37-38
    variations, 31-35
    wait(...) function, 33
Parallel Tasks window, 133-136
parallel_transform function, 53-55
ParallelTreeUnwinding function, 82
ParallelWhileNotEmpty1 method, 80-81
performance, 96-97
    Amdahl's law, 8-10, 16
PipelineGovernor class, 88-89, 95-96, 107
Pipeline pattern, 85-109
    AgentBase class, 101-102
    agent class, 86
    anti-patterns, 107
    Asynchronous Agents Library, 85
    asynchronous pipelines, 97-101
    basics, 86-92
    call class, 98-101
    call messaging block, 86
    choice messaging block, 86
    copying large amounts of data between stages, 107
    CorrectCaseAgent class, 90-91
    dataflow network defined, 85
    data pipeline defined, 85
    design notes, 108
    example, 92-97
    exercises, 109
    forgetting to use message passing for isolation, 107

FreePipelineSlot method, 92
further reading, 109
Futures pattern, 73
ImageAgentPipelineDataFlow class, 98-101
image pipeline, 94-96
ImagePipelineDlg class, 101-102
ImagePipeline sample application, 92
image pipeline with load balancing diagram, 106
infinite waits, 107
IsCancellationPending method, 104
join messaging block, 86
load balancing using multiple producers, 104-106
more information, 107
multitype_join messaging block, 86
overwrite_buffer messaging block, 86
performance characteristics, 96-97
pipeline cancellation, 101-102
pipeline exceptions, 102-104
PipelineGovernor class, 88-89, 95-96, 107
pipelines and streams, 106
pipeline stages that are too small, 107
pipelines vs. parallel loops, 86
ReadStringsAgent class, 89
receive function, 91
related patterns, 109
run method, 89-91, 100
sample pipeline, 87
scalability, 108
sequential image processing, 92-94
ShutdownOnError method, 103
single_assignment messaging block, 86
source, 87
target, 87
timer messaging block, 86
transformer class, 98-101
transformer messaging block, 86
types of messaging blocks, 86
unbounded_buffer messaging block, 86
unbounded_buffer<T> class, 87-88, 104
variations, 97-106
wait_for_all method, 89
WriteSentencesAgent class, 91-92

pipelines
    asynchronous, 97-101
    canceling, 101-102
    data pipeline defined, 85
    exceptions, 102-104
    image pipeline with load balancing diagram, 106
    sample, 87
    stages, 107
    and streams, 106
    vs. parallel loops, 86
    see also image pipeline
PolicyElementKey enumeration, 128-129
postprocessing phase, 50
PPL tasks, 117, 122-123
prerequisites, xv-xvi
processor affinity mask, 113-114
profiling, 17

Q

QuickSort algorithm, 78

R

ray tracers, 4
reader_writer_lock::lock method, 32
reader_writer_lock::lock_read method, 32
reader_writer_lock::scoped_lock_read constructor, 32
ReadStringsAgent class, 89
receive function
    parallel tasks pattern, 33
    Pipeline pattern, 91
recursive decomposition, 75
Reduce pattern, 57
reduce phase, 50
references, 132
Resource Acquisition is Initialization (RAII) pattern, 34

resource management overhead, 129-130
resource manager, 113-116, 118-119, 121, 128
  anti-patterns, 129-132
ResourceManager class, 113
Result method, 64-65
reviewers, xxi-xxii
run_and_wait method, 31
run method, 89-91, 100

S
sample pipeline, 87
scalable sharing, 5-6
Scan pattern, 57-58
Scenario library, 141-142
scenario markers, 141-142
ScheduleGroup class, 117, 121-122
schedule groups, 121-122
scheduler, 126-128
  policy, 128-129
  see also task scheduler and resource manager
Scheduler::Attach method, 119
Scheduler class, 118
Scheduler::Create method, 119
ScheduleTask method, 117
SchedulingProtocol policy key, 129
scoped_oversubscription_token class, 70
SearchLeft function, 36
section::scoped_lock constructor, 32
send function, 33
sequential code, 76
sequential credit review example, 18
sequential image processing, 92-94
SequentialWalk method, 76
SetDefaultSchedulerPolicy method, 118
sharing, 3
ShutdownOnError method, 103
single_assignment class, 62
single_assignment messaging block, 86
small loop bodies, 55
  with few iterations, 23
  special handling of, 21-22
source, 87
speculative execution, 36-39
Stencil pattern, 58
StockDataCollection type, 66-67
streams and pipelines, 106
structured task groups and task handles, 41
Sutter, Herb, xiii-xiv
synchronization
  costs, 39
  issues of, 5-6
system requirements, xv-xvi

T
table of patterns, 7
target, 87
task execution order, 125
task graph, 62
task_group::cancel method, 33-35
task_group class, 27
  parallel tasks pattern, 27, 39
task groups
  calling conventions, 39
  cancellation, 33-35
task_group::wait method, 31-32, 35
task parallelism, 27
  see also parallel tasks
tasks
  adding, 122-123
  adding to a pending wait context, 81-83
  coordinating, 4
  how they are scheduled, 40-41
  kinds of, 116-117
  running, 123-124
  that are run inline, 125-126
  and threads, 40
  understanding, 3-4
  vs.threads, 4
task scheduler and resource manager, 111-132
  Block method, 127
  caching suballocator, 127-128
  Concurrency Runtime, 111-113, 116-118, 121, 128, 131-132
  Concurrency Runtime components, 112
  Context class, 126-127
  Context::Oversubscribe method, 128
  cooperative context switch, 115

- CurrentContext method, 126
- CurrentScheduler class, 119
- data structures of schedule groups, 122
- deadlock from thread starvation, 131
- debugging information, 126
- Detach method, 120
- dynamic resource management, 115
- enhanced locality mode, 124-125
- forward progress mode, 124-125
- GetProcessorCount function, 116
- GetProcessorNodeCount function, 116
- ignored process affinity mask, 131-132
- interface to cooperative blocking, 127
- IVirtualProcessorRoot class, 113
- lightweight tasks, 116-117, 122-123
- long-running I/O tasks, 128
- MaxConcurrency policy key, 129
- MinConcurrency policy key, 129
- oversubscribing cores, 116
- PolicyElementKey enumeration, 128-129
- PPL tasks, 117, 122-123
- processor affinity mask, 113-114
- querying for cancellation, 126
- querying the environment, 116
- references, 132
- resource management overhead, 129-130
- resource manager, 113-116, 118-119, 121, 128
- resource manager anti-patterns, 129-131
- ResourceManager class, 113
- scenarios for multiple task schedulers, 120-121
- ScheduleGroup class, 117, 121-122
- schedule groups, 121-122
- Scheduler::Attach method, 119
- Scheduler class, 118
- Scheduler::Create method, 119
- schedulers
    - and contexts, 126-128
    - policy, 128-129
- ScheduleTask method, 117
- SchedulingProtocol policy key, 129
- SetDefaultSchedulerPolicy method, 118
- TargetOversubscriptionFactor policy key, 129
- task execution order, 125
- task runnability, 123-124
- tasks
    - adding, 122-123
    - kinds of, 116-117
    - long-running I/O tasks, 128
    - that are run inline, 125-126
- task schedulers, 118-129
    - creating and attaching, 119
    - destroying, 120
    - detaching, 120
    - managing, 119-121
- Unblock method, 127
- unintentional oversubscription from inlined tasks, 130-131
- virtual processor, 113-115
- wait function, 127
- Yield method, 127

task schedulers, 118-129
- creating and attaching, 119
- destroying, 120
- detaching, 120
- managing, 119-121

team, xxi-xxii

technology overview, 149-151
- Accelerator API, 150
- CHESS, 150
- DryadLINQ, 150
- further reading, 151
- Microsoft parallel programming resources, 149
- Visual Studio 2010, 150

terminology, 8
- glossary, 153-162

threads
- starvation, 8
- vs. tasks, 4

Threads view, 140-141
    showing lock convoy, 144-145
    that shows a load imbalance, 146
timer messaging block, 86
transformer class, 98-101
transformer messaging block, 86

U
Unblock method, 127
unbounded_buffer messaging block, 86
unbounded_buffer<T> class, 87-88, 104
unintended propagation of cancellation requests, 38
UpdatePredictionsParallel method, 18-19
UpdatePredictionsSequential method, 18

V
variables captured by closures, 37-38
variations
    parallel tasks pattern, 31-35
    Pipeline pattern, 97-106
virtual processor, 113-115
visual patterns, 142-147
Visual Studio 2010, 150

W
wait_for_all method, 89
wait(...) function, 33
wait function, 127
wait method, 81-82
WriteSentencesAgent class, 91-92

Y
Yield method, 127