Working Outline

Title: *A Guide to Parallel Programming: Design Patterns for Decomposition, Coordination and Scalable Sharing*

[*Note:* The book may have a banner that identifies it as being new for .NET 4 Framework and Visual Studio 2010. It is also possible that one of these could be incorporated in the title.]

Foreword

Introductory remarks by a recognized expert, including a mention of hardware trends for concurrency. This foreword can be thought of as a brief synopsis of the material in Herb Sutter's PDC '09 talk. For concision, this is the only place where hardware trends will be described. The author of the foreword has not been selected at this time.

Preface

Who This Book Is For

This book is for working programmers who write managed code in .NET or unmanaged code on the Windows platform. This includes programmers who write in C++, C#, Visual Basic and F#. No prior knowledge of parallel programming techniques is assumed.

Note: Although the material in this book can be used with languages other than C#/C++, it focuses on C# with some callouts for the equivalent C++. Particular attention is given to relevant features in .NET 4.

The examples in this book are written in C# and use the features of the .NET 4 Framework, including the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). However, you can use the concepts presented here with other frameworks and libraries. For example, F# now has language support for concurrency and unmanaged code can be written to use the Parallel Patterns Library (PPL).

Complete code solutions are posted on CodePlex. There is both a C# (or PLINQ) version and a C++ version for every example.

What is not covered

This book focuses more on CPU-bound workloads than I/O-bound workloads. The goal is to make computationally intensive applications run faster by better utilizing the computer's available processors. As a result, the book does not focus as much on the issue of I/O latency.

Nonetheless, some attention is paid to balanced workloads that have are both CPU intensive and have large amounts of I/O (see Chapter 7, data pipelines). There is also an important example for user interfaces in Chapter 5 (Futures and Continuations) that illustrates concurrency.

The book describes parallelism within a single multi-core node with shared memory rather than the cluster (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, since each node of a cluster may have multiple processing units.

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. [This section will be expanded to mention specific VS 2010 features and industry-related motivations for these features.]

The patterns described in this guide can be adapted to platforms such as HPC Server, but we do not show this.

What You Need to Use the Code

[This section will include a description and location of CodePlex site.]

Description of system requirements (.NET 4, VS 2010, etc.)

How to Use This Book

This book presents parallel programming techniques in terms of particular patterns.

[Diagram - Subway map showing chapters]

Introduction

Chapter 1 introduces the common problems faced by developers who need to incorporate parallelism to make their applications run faster.

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.
- **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: Data Aggregation using Map/Reduce**. Patterns for data aggregation are used when the body of a parallel loop includes data dependencies, for example when calculating a sum or searching a collection for a maximum value.
- **Chapter 5: Futures and Continuations**. This pattern arises 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 performed in parallel and some serially, depending on when inputs become available.

Chapters 6 and 7 deal with some of the 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 is called dynamic task decomposition. It applies to several domains, including graph algorithms.
- **Chapter 7: Pipelines**. Use pipelines to feed successive outputs of one component to the input queue of another component, in the style of an assembly line. Parallelism results when the pipeline fills, and all components are simultaneously active.

Supporting Material

In addition to the patterns, there are several appendices.

- **Appendix A: Supporting Patterns**. This appendix gives tips for adapting some of the common object-oriented patterns such as facades, decorators and repositories for concurrency. For example, it shows how "lazy" data access can promote concurrency in a repository. Event-based coordination (“agents”) is another familiar pattern that is also often adapted for parallel scenarios.
- **Appendix B: Debugging and Profiling Parallel Applications**. This appendix gives you an overview of how to debug and profile parallel applications in VS 2010.
- **Appendix C: Technology Roadmap**. A technology roadmap helps you place the various Microsoft technologies and frameworks for parallel programming in context.
- **Appendix D: QuickStart Examples**. This appendix contains very brief code examples for common tasks such as "How-to parallelize a loop over a collection". It is meant as a short-cut reference.

A glossary contains the definition of terms used in this book.

You can read the chapters in this book sequentially. We recommend that all readers read chapters 1, 2 and 3 for an introduction and overview of the basic principles. After reading chapters 1, 2 and 3, you can either jump to topics of interest or read sequentially. Although attention has been paid to presenting the material in a logical order, each chapter, from chapter 4 and up, is independent.

Callouts (with a trap icon) show things you should watch out for. These are sometimes called "anti-patterns."

[Anti-pattern, rendered as callout]: Don't apply the patterns in this book blindly to your applications. (Hammer/nail problem)

Acknowledgments

Acknowledgment of project contributors.

Chapter 1: Introduction

The CPU meter shows the problem. One processor is running at 100%, but all of the other processors on your computer are idle. Your application is CPU bound, but you are using only a fraction of the computing power that your multi-core system is capable of. What next?

The answer, in a nutshell, is *parallel programming*. Where you once would have written a simple sequential function, you find that this will no longer meet your performance goals. To use your system's processors efficiently, such operations must now be decomposed into pieces that can run at the same time.

This is easier said than done. Parallel programming has a reputation for being the domain of experts and a minefield of nasty, hard-to-reproduce bugs. Fortunately, help has arrived. Microsoft's Visual Studio 2010 has introduced a new programming model for parallelism, along with supporting libraries, that significantly simplifies the job.

Decomposition, Coordination and Scalable Sharing

In this book you will see that the process of designing and implementing a parallel application involves three aspects: *decomposition* into tasks, methods for *coordination* of tasks and techniques for *scalable sharing*.

- Decomposition into tasks. Decomposition is the process of identifying separable units of computation at an appropriate level of granularity. We call these *tasks*. Decomposition requires that you understand the algorithmic and structural aspects of your application. In general, tasks should be as independent as possible.
- Coordination of tasks. You will notice that operations cannot be run in a completely arbitrary order. Tasks are not fully independent of one another. The order of execution and the degree of concurrency are constrained by the control flow and by data flow characteristics of the application's underlying algorithms.
- Scalable sharing. Tasks often need to share data. There are a number of techniques that allow data to be shared safely and without degradation in performance. Scalable sharing may involve changes to the algorithm. This is why parallel programming has a stronger emphasis on immutable data than serial programs.

Anti-pattern (callout): Unfortunately, the easiest approach, considering all program variables as mutable shared state and wrapping all access to them in serializing locks, is *not* a scalable approach to sharing. As more and more tasks sharing the same data are added waiting for locks dominates the computation. The end result is the additional tasks spend more time waiting than doing useful work.

These aspects are highly interrelated. How you decide to decompose your problem into tasks is highly dependent on what strategy you have chosen for dealing with shared data. How you coordinate the tasks is highly dependent on how you have chosen to decompose your problem into tasks.

The best way to master the techniques of decomposition, coordination and scalable sharing is to use the patterns described in this book. Patterns are a true short cut to mastery of the subject.

A Word about Terminology

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

Concurrent programming, which .NET has a lot of support for already, is about tolerating long-latency events (such as I/O) and is something that occurs even on single processor computers. In contrast, parallel programming focuses on improving the performance of CPU-bound applications when multiple units of execution (such as processors or cores) are available.

[More]

Concurrency results when at least two of the stages have work to be done; not all components need to be simultaneously active, just two; of course, the more that are active, the more opportunity there is for parallelism.

Applying patterns

The first step is figuring out where the opportunities for parallel execution lie by matching characteristics of your application to the patterns described in this book.

Do you have coarse-grained iteration with no need for communication among the steps of the iteration? If so, then the parallel loop pattern described in chapter 2 may be used.

Do you need to aggregate data by applying some kind of combination operator? If so, then you might consider applying the map/reduce pattern described in chapter 4.

One way to familiarize yourself with the possibilities is to read the first page or two of each chapter in this book. This will give you an overview of the kinds of decomposition that have been proven to work in a wide variety of applications.

The Limits of Parallelism

A theoretical result called Amdahl's law says that the amount of performance improvement you see is limited by the amount of sequential processing you do. The result may surprise you. [More]

A Few Tips

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

1. Whenever possible, use a library that does the parallel work for you.
2. Make use of your application server's inherent parallelism; e.g. Web Server or database.
3. Use an API to encapsulate parallelism - e.g. TBB, TPL or PPL. These libraries were written by experts and avoid many of the common problems that arise in parallel programming.
4. Use patterns, like the ones described in this book.
5. Don't share data among concurrent tasks unless absolutely necessary.
6. 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.

7. 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.

Chapter 2: Parallel Loops

The Problem

Use the parallel loops pattern when the same calculation needs to be performed on each element of a collection independently. Implementations of this pattern are sometimes called "delightfully parallel loops. It is one of the easiest parallel patterns to use.

The parallel loop pattern is sometimes called the map pattern, especially when the operation returns a value. (In this case, the elements are "mapped" into a collection of transformed values.) The use of the term map is especially common in functional languages like F#.

An Illustration

Fabrikam Shipping wants to use balance trends to identify at-risk customers. Each card holder will have a projected balance that is calculated using a statistical trend-projection technique. If the three-month projected balance exceeds the credit limit, the account will be flagged for manual review by a credit analyst. Each customer's credit status can be calculated independently. We simply need to replace the serial loop with a parallel loop.

The data consists of each customer's balance histories, indexed by customer. The parallel loop iterates over customers. 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.

[Code sample - before and after. Shows **Parallel.ForEach**]

[Callout mentioning alternatives-- **Parallel.For** and **ParallelEnumeration.ForEach**. These are described more fully in the Implementation Notes section.]

Design Notes

The pattern emphasizes decomposition by data, not tasks. [Concepts described in the text will include the basics of *partitioning, scaling to N processors* and *work stealing algorithms*.]

With partitioning, data is divided into sets of non-overlapping regions; partitions are allocated to available processors.

Scaling to N processors means avoiding an implementation approach where the number of processors, *K*, is hard coded. (This is also known as "the K problem".)

It is common to create a task for each data element being processed, regardless of the number of processors available. The TPL framework optimizes this case for you.

[The concept of work stealing is described in a brief callout. Algorithms for work stealing are described at a high level. The hill-climbing algorithm is deferred until chapter 6. We provide enough information to build intuition but do not give a definitive treatment. (The hill-climbing heuristic is used for work

scheduling by default in TPL. This algorithm dynamically adapts to varying task sizes. TPL allows you to provide your own scheduler.)]

After reading this section, the reader should understand 1) why data isolation is a desirable property and 2) why a naïve application of thread pools is an inadequate design for parallel loops.

Implementation Notes

Additional detail for the classes and methods introduced: **Parallel.For**, **Parallel.ForEach**, **ParallelEnumeration.ForEach.** Overloaded methods that allow you to specify options such as the number of processors to use.

Anti-pattern: Frank changes "for (I = 0;…)" to Parallel.For and his application stops working.

Anti-pattern: Data dependencies in loops. Pull calculations out of loops to avoid dependencies.

Anti-pattern: Processor oversubscription. Don't thrash. Don't have too many tasks per processor [Note: remove, since we have talked about load balancing algorithms that address this problem?]

Anti-pattern: Processor undersubscription. Having too few tasks misses an opportunity to increase parallelism

Anti-pattern: Downward iteration. Attempt to make indices go from high to low. Potentially unexpected results if there are dependencies between array elements (which was why a downward iteration was used in the first place).

Anti-pattern: Stepped iteration. Attempt to increment indices by number other than 1.

Anti-pattern: Very small loop bodies. Task overhead overwhelms benefit of parallel execution.

Anti-pattern: Inappropriate decomposition (too coarse or too fine). You can also make the steps too large.

Anti-pattern: IList<T> implementations that are not thread safe

Anti-pattern: Nested parallelism; e.g. Parallel.ForEach over a ParallelQuery<T> object

Anti-pattern: Violating thread affinity constraints in source data. For example, this can easily happen with user interface controls. Some environments require that methods of UI controls be called in the thread that created the control. [Note: Move to Chapter 3?]

Anti-pattern: Using parallel loops for I/O bound workloads.

Anti-pattern: Dependencies between partitioned chunks. Do not allow data dependencies to occur between parallel data partitions. (Include a discussion of subtle data dependencies that may not be immediately apparent.)

Uses

Delightfully parallel loops occur frequently. [The text mentions several application areas where they arise.]

Variations and Related Patterns

Upper/lower bounds vs. iterators.

PLINQ. PLINQ provides an alternate syntax for parallel loops. [The text shows a source code example of the Fabrikam example coded in PLINQ.]

Callout: Parallel.ForEach and PLINQ work on slightly different threading models in the .NET Framework 4. PLINQ uses a fixed number of threads to execute a query; by default, it uses the number of logical cores in the machine, or it uses the value passed to WithDegreeOfParallelism if one was specified. Conversely, Parallel.ForEach may use a variable number of threads.

Breaking out of loops early

Planned exit

Loop cancellation

Nested loops

Related patterns:

SPMD – Distributed systems (MPI, Batch & SOA)

Master/Worker – Task centric (TPL or PPL)

Fork/Join – Thread centric (TPL or PPL)

Data Parallelism – Algorithm strategy pattern

References

Chapter 3: Parallel Tasks

The Problem

In the previous chapter we saw how a single operation could be applied within a parallel loop. In this chapter we will consider what happens when you have multiple asynchronous operations without iteration.

In this case it can be useful to temporarily *fork* a program's flow of control into tasks that may execute concurrently. The operations are parallel tasks.

An Illustration

Forking into parallel tasks can be used for speculative selection. For example, you can run two routines and use the results of the first one that returns a value. For example, an application maintains a list of

items to be processed using a doubly-linked list. High priority items are placed at the head of the list. Low priority items are placed at the end of the list. The application has an operation to find a recently added item using linear search. It is likely that the desired item is either near the front of the list or near the end of the list. The application creates two tasks for this purpose. When the item is found, the other task is canceled. The example introduces the idea of waiting for tasks and cancellation of pending tasks.

Design

The main new concept is the *task*. [Includes a comparison between tasks and user work items (threads).]

Callout: We recommend that you use tasks for all asynch work instead of user work items (threads from a pool). In .NET Framework 4, a thread consumes a megabyte of stack space, whether or not that space is used for currently executing functions. Also, thread creation and cleanup are very expensive operations. Creating too many threads is a performance killer.

Anti-pattern: The term fork/join has a long history of being applied to threads of execution. Beware: tasks are not threads. (Brief explanation of why tasks are not threads appears as a callout.)

Anti-pattern: False sharing. If you use **Parallel.Invoke** and write results into two adjacent array elements, you may see significant performance degradation. This arises from the behavior of hardware caches. You don't want to have these variables in adjacent locations. Refer to MSDN article on false sharing.

Anti-pattern: C# closure semantics may not be intuitive. Captured variables may not behave as expected. The solution is to introduce an additional temporary variable in the appropriate scope. (This is one reason why you should use **Parallel.For** instead of coding a loop yourself.)

Implementation Notes

Classes and methods introduced: **Parallel.Invoke**, **Task.Factory.StartNew**, **Task.Wait**, **Task.WaitAll, Action<T>**

Callout: **Environment.ProcessorCount** fine print. **ProcessorCount** may be capped at 32 or 64 even when more are available; the value counts hardware threads (not cores).

Uses

Variations and Related Patterns

Speculative selection- Use the results one of two parallel computations depending on a condition. We saw this pattern in this chapter's example

Speculative execution - use heuristics to do work that may or may not be needed. If you guess right, latency is reduced.

Related patterns:

SPMD – Distributed systems (MPI, Batch & SOA)

Master/Worker – Task centric (TPL or PPL)

Loop Parallelism – Data centric (OpenMP)

Task Parallelism – Algorithm strategy pattern

Speculation

References

Chapter 4: Parallel Aggregation with Map/Reduce

The Problem

In chapter 2, we described applying the same operation independently to elements in a collection. However, in some cases there are data dependencies during the iteration. This chapter describes *aggregation*, which is a pattern that may be appropriate when these dependencies exist.

Suppose that you want to find the sum of a collection of numbers. To be efficiently implemented, this requires intermediate results to be kept in task-local state. You don't want an expensive synchronization operation on a single memory location that holds the partially calculated sum.

Aggregation doesn't have to be arithmetic addition. Any operation whose outcome depends on results from different iterations is an example of aggregation. There are many variations of this pattern, such as reduce, stencil, scan and pack. These are described in a separate section at the end of this chapter.

An Illustration

Consider a problem from social networking. Individuals may designate other individuals as friends. Given an individual, recommend candidates for friendship, based on mutual friends. In the parallel phase, compute the intersection of the individual's collection of friends with every other subscriber's friends, obtaining for each subscriber a collection of mutual friends. In the aggregation phase, rank and then select candidates from among the other subscribers, based on their collections of mutual friends.

[Drawing]

Design

The main new design concept is task local state.

We introduce the idea of designing an algorithm that does extra work to avoid data dependencies. (This is counterintuitive to programmers only familiar with serial execution.)

Implementation Notes

Uses

Variations and Related Patterns

There is a cluster of patterns related to summarizing data in a collection: reduce, stencil, scan and pack.

The reduce pattern occurs when a result is accumulated into a single location of memory.

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

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

The pack pattern. Iteration through a collection selecting elements to retain and discard. The result of a pack operation is a subset of the original input collection.

These can be combined, as in the fold and scan pattern.

A well-known example of aggregation is map/reduce. This combines parallel loops (map) with aggregation (reduce).

MapReduce in PLINQ

Outputting a single result vs. a set of results.

ToArray / ToList / ToDictionary / ToLookup

Single-value aggregations in PLINQ

The Aggregate method (PLINQ)

The "Take the top N" pattern

Callout: Discussion of techniques for thread-local state; benefits of thread-local state (**ThreadLocal<T>** class which replaces the **ThreadStatic** custom attribute and **Thread.SetData**).

Overloaded methods for specifying thread-local storage

Related patterns:

- Non work-efficient parallelism
- Map/Reduce

References

Chapter 5: Futures and Continuations

The Problem

When one task provides a result that becomes the input to another task, tasks execute serially and in parallel based on data dependencies. Task dependencies form a directed acyclic graph. (Drawing: DAG of task dependencies for b = F(a); c = G(a), d = H(c), f = H(b, d))

A future is a task that returns a value. The **Task<T>** class in the .NET Framework implements the future pattern. This class allows a result to be calculated in parallel with other operations.

An Illustration

[Asynchronous UI actions in a financial dashboard, with data dependencies and context constraints from the UI framework. This was mentioned in Stephen Toub's PDC talk. It is a generalization of the earlier style of asynchronous background workers, which are limited to a single instance and may not have data dependencies.]

Design

The main new concept is using continuations to implement task ordering based on dataflow.

Anti-pattern: When are continuations used vs. .NET 2.0 background workers? Answer: Background worker is not good when you are merging the results from multiple concurrent calculations. (Also, possibly, when there is more than one background task needed, regardless of merging.)

Implementation

Classes and methods introduced: **Task<T>** class (i.e., tasks that return values), **Task.ContinueWith**, **TaskCompletionSource**

Uses

Continuations are very useful for user interfaces.

Callout: Comparison with existing .NET asynch UI programming patterns

Variations Related Patterns

An important use of continuations is in programming user interfaces.

Architecture callout: the default task scheduler in TPL vs. the WPF-specific task scheduler in TPL

References

Chapter 6: Dynamic Task Parallelism

The Problem

In the previous chapters, we were able to determine in advance how many tasks would be needed. In this chapter, we look at cases where tasks are dynamically added to the queue of work as the computation proceeds.

An Illustration

Sorting a list, using a recursive algorithm such as Quicksort, which divides the list, then sorts and recombines the pieces.

Design

Concept: depth limit.

Callout: More on work stealing. We described work stealing in chapter 2. Work stealing helps mostly when tasks create subtasks, as in recursive divide and conquer. It is also important when the duration of tasks can't be predicted and some tasks take a lot longer than others.

Concept: techniques for passing data: 1) Using closures to pass data, 2) using state objects to pass data, 3) using state objects with member methods. (Refer to the anti-pattern described earlier about counterintuitive closure semantics and the need for additional temporary variables.)

Implementation Notes

Classes and methods introduced:

Anti-pattern: False sharing was mentioned in Chapter 3, but it is worth mentioning here as well, since recursive task parallelism may be prone to it.

Anti-pattern: Recursion without thresholds (see Toub p. 44)

Uses

Variations and Related Patterns

Parallel-while-not-empty pattern. Implements "from" and "to" collections using ConcurrentQueue<T> class. Typical of tree walks (such as breadth first traversal).

Task chaining via parent/child tasks. From Taub: "**Parallel.Invoke**, and the Task Wait functionality on which it's based, attempt what's known as inlining, where rather than simply blocking waiting for another thread to execute a task, the waiter may be able to run the waitee on the current thread, thereby improving resource reuse, and improving performance as a result. Still, there may be some cases where tasks are not inlinable, or where the style of development is better suited towards a more asynchronous model." This uses the **TaskCreationOptions.AttachedToParent** property and the **Task<T>.Unwrap** method.

Related patterns:

Recursive Splitting or Divide and Conquer

Recursive Data

References

Chapter 7: Pipelines

The Problem

You can think of a pipeline as an assembly line in a factory. To implement pipelines in software we use concurrent queues that act as buffers. (Drawing: assembly line with queues between components)

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

Pipelines are an example of a more general pattern known as producer/consumer.

An Illustration

Exposure correcting, resizing and sharpening images into thumbnails (using a transfer stream). Each image processing operation is a stage in the pipeline. The sample code can use the **System.Drawing.Images** namespace.

Design

We introduce the concept of a blocking collection. Queue length and block size may vary.

Pipeline length affects the latency/throughput tradeoff.

Suffers from the K-parallel problem typically.

Anti-pattern: Shared queues may have subtle ordering effects (from Miller PDC talk).

Implementation

Classes and methods introduced: **IProducerConsumerCollection<T>** (Generalizing blocking collection to data structures other than a queue, incl. **ConcurrentStack<T>** and **ConcurrentBag<T>**), **TaskGroup**,

Uses

Pipelines and blocking collections are also for handling streams (ticker tapes, user-generated UI events like mouse clicks, packets being received over the network, etc.)

Pipelines are used for encryption/compression.

Examples of producer/consumer abound: Thread pools, UI marshalling, The Rendezvous pattern, SynchronizationContext, Subscribing to system events.

Variations and Related Patterns

Streaming data (incl. Pipes and filters). Up to this point, we have described collections with a fixed number of elements. Streams are an important extension to fixed collections. [Describe BlockingCollection.GetConsumingEnumerable and Parallel.For. Note: we locate this topic here because of its use of blocking collection.]

Nonlinear pipelines. Sometimes a component in a pipeline can consist of more than one component running in parallel. This is useful in the case where you have two operations, A and B. If A takes 1 unit of time on average and B takes 2 units of time, then you want a pipeline that has A followed by two instances of B (since A creates outputs twice as fast as an instance of B can consume them). (Add drawing).

Related patterns:

Producer/Consumer

Shared Queue

References

Appendix: Supporting Patterns

Facades, decorators and repositories are examples of familiar patterns used in object-oriented programs. Although they are not specific to parallel programs, you may find them useful when implementing the parallel patterns described in this book. For example, in Chapter 7 we showed how the decorator pattern can be used to implement a transfer stream. In this appendix, we give a few examples and ideas for using these and other supporting patterns in parallel programs, along with some things to watch out for.

Topic: Thread-safe lazy evaluation. Facades, decorators and repositories often implement lazy evaluation. There are special considerations for making lazy data structures thread safe (i.e., accessible from more than one task). Lazy data structures delay calculation until data is requested). Uses the double-checked locking pattern. Introduces **Lazy<T>** class.

Topic: Lazy initialization, including initialization races by design and asynchronous laziness. Initialization races are a technique that allows object initialization code to be called more than once, in parallel. Only the result of the first to complete will be used. (Pattern is supported by LazyInitializer<T> class as an option). With asynchronous laziness you receive notification when initialization completes (example of using continuations and using Task<T>)

Callout: How do native and managed stacks affect concurrency? Native and managed stacks exist in separate threads. If you run parallel tasks in both native and managed, you may end up with oversubscription of processors. (Use TPL!)

Callout: **IDisposable** for tasks. The Task type implements IDisposable. This gives you an option of cleaning up task memory yourself. However, this is only recommended in simple cases where you are absolutely sure that no references exist.

Handling cross-language scenarios. Libraries are important because we want to support parallelism across multiple languages. Note that there is no support for lambda expressions in C++ CLI.GC considerations-- Garbage collection stops all threads. Therefore, in concurrent applications you need to pay more attention to how much memory you allocate temporarily. Caching delegates can reduce GC overhead.

GCServer, turned on by default in ASP.NET, can also be optionally enabled for client applications. This version of the garbage collector creates distributed heaps and also uses multiprocessing internally.

Appendix: Debugging and Profiling Parallel Programs

The Debugger

Debug vs. release builds- Debug vs release does not affect parallelism. Can affect debugging experience.

Use concurrency visualizer to find memory bottlenecks
Using the parallel stacks window in VS 2010. Describes how to find inappropriate synchronization using the parallel stacks window

The Profiler

Anti-pattern: Only do a minute or two of profiling. The amount of data generated by the profiler is very large.

Appendix: Technology Roadmap

An outline of the various Microsoft technologies which support parallelism and how they fit together; TPL, PPL, ConcRT, DryadLINQ, F#, HPC Server, Azure (Add diagram)

(to do, fill this out with more libraries.)

Appendix: Quick Examples

Contains very short code fragments for common scenarios. It is meant as a reference.

Glossary

Asynchrony, background thread, barrier, blocked convoy, blocking collection, concurrency profiler, core, data partitioning, data race, deadlock, dynamic partitioning, foreground thread, hardware thread, hyperthread, livelock, load imbalance, lock convoy, manycore, memory barrier, multi-core, nested parallelism, node, nonblocking algorithm, overlapped I/O, oversubscription, parallel programming, priority inversion, race, race condition, round robin, simultaneous multithreading (SMT), socket, static partitioning, thread, thread pool, thread-local state, torn read, torn write, two-step dance, undersubscription, virtual core, ….

Index