# A Guide to Parallel Programming

*Design Patterns for Decomposition, Coordination and Scalable Sharing*

PRELIMINARY

April 15, 2010

# 2 Parallel Loops

Use the parallel loop pattern when the same independent operation needs to be performed for each element of a collection or for a fixed number of iterations. Iterations of a loop are independent if they don't write to memory locations or files that are read by other iterations. The syntax of a parallel loop is very similar to the **for** and **foreach** loops you already know, but the parallel loop performs much faster on multicore computers.

The parallel loop pattern independently applies an operation to multiple data elements. This kind of computation is sometimes called *data parallelism*.

Parallel loops express *potential parallelism*. This means that the degree of parallelism doesn't need to be specified by your code. Instead, the run-time environment executes iterations of your loop as concurrently as possible on the available hardware resources. The loop works correctly no matter how many CPU cores are available. If there is only one core then the performance is about the same as a sequential iteration. If there are many cores then performance improves proportionately.

## The Problem

The parallel loop pattern allows an application that uses **for** or **foreach** loops to run faster on multicore computers. A parallel loop does the same work as a sequential loop, only in less time when more than one core is available. You can use both **parallel for** and **parallel foreach** loops, depending on whether you want to iterate over a range of integer indices or over values from a collection.

It's easy to make **for** or **foreach** loops with independent iterations run faster on multicore computers by using their parallel counterparts.

### Parallel For Loops

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

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

This example assumes that iterations of the loop body are independent of one another.

The computation can take advantage of multiple cores by simply replacing the **for** keyword with a call to the **Parallel.For** method.

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

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

**Parallel.For** is one of the parallel extensions provided as part of the .NET 4 Framework. It is a normal static method with three arguments.

> **C++ Note:** Parallel loops in C++ can use the **Concurrency::parallel_for** function that is provided by the Parallel Patterns Library (PPL) in Visual Studio 2010.

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

> **Note:** The code example includes a lambda expression in the form i => ... as the third parameter to the **Parallel.For** method. Lambda expressions denote unnamed or anonymous delegate methods. If you are unfamiliar with the syntax for lambda expressions, check the reference section at the end of this chapter for more information.

> **Anti-pattern**: If you need to use a step size other than one, transform the loop index in the body of the loop. Be aware that a step size other than one can indicate a data dependency, such the calculation of a sum. Analyze such a computation carefully before you convert it into a parallel loop.

## Parallel ForEach

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

```
MyObject[] myEnumerable = ...

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

This example assumes that iterations of the loop body don't write to memory locations or files that are read by other iterations. The loop body only makes updates to fields of the particular instance of the **MyObject** class that is passed to it with each iteration. It doesn't read fields that are updated by other iterations.

Take advantage of multiple cores by replacing the **foreach** keyword with a call to the **Parallel.ForEach** method.

```
MyObject[] myEnumerable = ...

Parallel.ForEach(myEnumerable, obj =>
{
```

```
   // ... do some work ...
});
```

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

**Parallel.ForEach** is one of the parallel extensions provided as part of the .NET 4 Framework. It is a normal static method with two arguments.

> **C++ Note:** Parallel loops in C++ can use the **Concurrency::parallel_for_each** function that is provided by the Parallel Patterns Library (PPL) in Visual Studio 2010.

The first argument contains a collection that provides the **IEnumerable<T>** interface. The second argument is a delegate method that will be invoked for each element of the input collection.

Unlike the sequential case, the order of execution is not guaranteed with either the **Parallel.For** or the **Parallel.ForEach** method.

## Benefits

If you compare the **Parallel.For** loop with the original **for** loop, or the **Parallel.ForEach** loop with the original **foreach** loop, you'll notice that the sequential and parallel loop bodies are the same. The performance, however, is not. If you have more than one core on your computer, as most recently manufactured computers do, then the parallel loop will use them. By default, the degree of parallelism (that is, how many iterations run concurrently in hardware) depends on the number of cores on your computer. The more cores you use, the faster your loop executes. How much faster depends on the kind of work your loop does. See the Design Notes section of this chapter for more information.

If you use a parallel loop, you are planning for a future where the number of cores on a typical computer will increase. Your code will run unchanged, without recompilation, and automatically take advantage of the larger number of cores.

Implementations of the parallel loop pattern ensure that exceptions that are thrown during the execution of a loop body are not lost. For both the **Parallel.For** and **Parallel.ForEach** methods, exceptions that occur are collected into an **AggregateException** object and rethrown in the context of the calling thread. This ensures that all exceptions are propagated back to you.  See the Variations section of this chapter to learn more about exception handling for parallel loops.

Robust exception handling is an important aspect of the parallel loop pattern.

So far you've seen only the most basic parallel loops. There are many variations. There are twelve overloaded methods for **Parallel.For** and twenty overloaded methods for **Parallel.ForEach**. In addition, the Language Integrated Query (LINQ) feature of .the NET framework includes a parallel version named PLINQ (Parallel LINQ). With over 100 extension methods, there are many options and variations on how to express PLINQ queries. See the Variations section of this chapter to learn about some of the most important cases.

The parallel loop pattern is one of the easiest parallel patterns to apply to existing code. Just look for the top-level loops in your application and check that the loop body is independent. If it is, replace **for** or **foreach** with **Parallel.For** or **Parallel.ForEach**.

**Anti-pattern:** If the loop body is not independent, for example, when you use an iteration to calculate a sum, then you may need to apply the variation on a parallel loop that's described in chapter 4, Parallel Aggregation. If your loop has a negative step size (that is, it iterates from high to low values), the iterations are probably not independent.

## An Example

Here's an example of when to use a parallel loop. Fabrikam Shipping extends credit to its commercial accounts. It wants to use 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.

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

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

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

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

It's possible to nest parallel loops. In this situation, the run-time environment coordinates the use of processor resources. Another option to consider is to combine the nested loops into a single loop.

Nesting or unrolling loops is appropriate in cases where the amount of work in each top-level iteration is either very large or of uneven size.

Small loop bodies, even as small as a single multiplication, are not necessarily ineligible for parallel loops. It also depends on the number of iterations. If you have very small loop bodies but a large number of data elements, then a parallel loop may still provide a worthwhile improvement in performance. See Special Handling for Small Loop Bodies in the Variations section of this chapter for more information.

When you're thinking about how and where to add parallel loops, keep in mind the first rule of parallel programming: if in doubt, profile. Performance testing is the only way to guarantee that you are achieving the expected speedup.

If in doubt, profile.

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

## Sequential Prediction

Here is the sequential version of the credit analysis operation.

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

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

## Prediction Using Parallel.ForEach

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

```
static void UpdatePredictionsParallel(AccountRepository accounts)
{
  Parallel.ForEach(accounts.GetAllAccounts, account =>
  {
```

```
    Trend trend = SampleUtilities.Fit(account.Balance);
    double prediction = trend.Predict(
                        account.Balance.Length + NumberOfMonths);
    account.ParPrediction = prediction;
    account.ParWarning = prediction < account.Overdraft;
  });
}
```

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

## Prediction with PLINQ

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

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

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

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

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

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

### Performance Comparison

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

## Variations

The credit analysis example is a typical use case for parallel loops, but there can be variations. This section introduces some of the most important variations of the parallel loop pattern. These discussions are condensed summaries of topics presented in a paper by Stephen Toub [Toub09]. See the references section of this chapter for more information.

### Breaking Out of Loops Early

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

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

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

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

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

Calling the **Break** method of the **ParallelLoopState** object begins an orderly shutdown of the loop processing. Any iterations that are running as of the call to **Break** will run to completion. However, you may want to check for a break condition in long-running loop bodies and exit that iteration immediately if a break was requested.

To see if another iteration running in parallel has requested a break, retrieve the value of the parallel loop state's **IsStopped** property. A value of **true** indicates that a break has been requested.

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

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

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

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

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

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

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

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

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

```
}
```

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

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

## External Loop Cancellation

In some scenarios you may want to cancel a parallel loop from the outside, for example, in response to a request from the user interface. In this variation, use a cancellation token derived from a **CancellationTokenSource** instance. Here is an example.

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

  var options = new ParallelOptions
                        { CancellationToken = token };

  try
  {
    Parallel.For(0, N, options, (i) =>
    {
      // ... do some work ...

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

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

If external cancellation has been signaled *and* your loop has called either the **Break** or the **Stop** method of the **ParallelLoopState** object, then a race occurs to see which will be recognized first. The parallel

loop will either throw an **OperationCanceledException** or it will terminate using the mechanism described in the previous section, but not both.

## Exception Handling

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

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

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

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

## Special Handling of Small Loop Bodies

If the body of the loop is very small and each iteration is expected to take the same amount of time, 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 synchronizing between worker threads and the cost of invoking a delegate method. In most situations these are negligible, but with very small loop bodies they can be significant.

Partitioning divides data into sets of non-overlapping regions called partitions; partitions are allocated to available processors.

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

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

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

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

```
int[] result = new int[1000000];
Parallel.ForEach(Partitioner.Create(0, 1000000, 50000),
    (range) =>
    {
       for (int i = range.Item1; i < range.Item2; i++)
       {
         // small, equally sized blocks of work
         result[i] = i * i;
       }
    });
```

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

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

Custom partitioning is an extension point in the API for parallel loops. You can implement your own partitioning strategies. This is covered in more detail in the Implementation Details section of this chapter.

## Small Numbers of Iterations

If you have a very small number of iterations, you may want to invoke them in parallel as a list. Here is an example.

```
Parallel.Invoke(() => DoWork(0),
                () => DoWork(1),
                () => DoWork(2));
```

The **Parallel.Invoke** method takes a **params** array of **Action** delegates and invokes each in parallel. The method returns when all of the actions have completed.

## Controlling the Degree of Parallelism

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

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

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

This loop will run using at most two worker threads.

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

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

This query will run with eight worker threads.

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

**Anti-pattern**: Be careful if you use parallel loops for I/O-bound workloads. If the I/O wait times are long, you may experience an unbounded growth of worker threads due to a hill-climbing heuristic used by the .NET **ThreadPool** class's thread injection logic. This heuristic steadily increases the number of worker threads when the threads of the current pool are blocked for long periods of time. You can limit the degree of parallelism as a way to prevent this from happening.

**Note:** The **Parallel** class and PLINQ work on slightly different threading models in the .NET 4 Framework. PLINQ uses a fixed number of threads to execute a query; by default, it uses the number

of logical cores in the machine, or it uses the value passed to the **WithDegreeOfParallelism** method if one was specified.

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

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

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

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

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

## Using Thread-Local State in a Loop Body

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

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

// ForEach<TSource, TLocal>(
```

```
//        OrderablePartitioner<TSource> source,
//   ParallelOptions parallelOptions,
//   Func<TLocal> localInit,
//   Func<TSource, ParallelLoopState, long, TLocal, TLocal> body,
//   Action<TLocal> localFinally);
```

```
Parallel.ForEach(
     // source
     Partitioner.Create(0, numberOfSteps),

     // parallelOptions
     new ParallelOptions(),

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

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

     // localFinally
     null);
```

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

> **Anti-pattern**: Be careful that your loop body does not contain hidden dependencies. The example of trying to share an instance of a class such as **Random** that is not thread safe across parallel iterations is an example of a subtle dependency.
>
> If your loop iterates from a high value to a low value with a negative step size, this is an indication that it probably contains dependencies between iterations.

## Using a Custom Task Scheduler for a Parallel Loop

In some cases you may want to substitute custom task scheduling logic for the default task scheduler that uses **ThreadPool** worker threads. This variation occurs, for example, when Single-Threaded Apartment (STA) threads, such as those used to invoke legacy COM components, must be used to perform the body of the parallel loop. By default, Multithreaded Apartment (MTA) worker threads from the .NET thread pool are used to execute parallel loops. You could also use this variation as a way to ensure a processor or node affinity on computers with non-uniform memory architecture (NUMA). These are just a few of the advanced scenarios where a custom task scheduler might be needed.

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

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

For more information on task schedulers, see the Implementation Notes section in chapter 3, Parallel Tasks.

> **Note:** It isn't possible to specify a custom task scheduler for PLINQ queries. If you need a custom scheduler for a PLINQ query, you must implement a custom class that derives from **ParallelQuery<T>**.

## Mixing the Parallel Class and PLINQ

PLINQ queries are instances of the **ParallelQuery<T>** class. This class provides the **IEnumerable<T>** interface, so it is possible to use a PLINQ query as the source collection for a **Parallel.ForEach** loop. This is not recommended.

Instead, you should use PLINQ's **ForAll** extension method for the **ParallelQuery<T>** instance. PLINQ's **ForAll** extension method performs the same kind of iteration as **Parallel**.**ForEach**, but it avoids wasteful merge and repartition steps that would otherwise be required in this case.

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

```
var q = (from d in data.AsParallel() ... select d => F(d));
q.ForAll(item =>
{
  // ... Process item
});
```

## Parallel Loops with Custom Iteration

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

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

You can implement a custom iterator for the **Tree<T>** class.

```
public static IEnumerable<Tree<T>> Iterate<T>(Tree<T> root)
{
```

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

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

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

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

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

## Overriding the Default Behavior of IList<T>

In certain rare cases, the parallel loop's default handling of the **IList<T>** type may not be what you want. This can occur when the **IList<T>** implementation has unfavorable random-access performance characteristics or when random access causes a list with lazy-loading semantics to load all list values into memory.

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

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

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

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

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

**Note:** The **System.Data.Linq.EntitySet<TEntity>** class is an example of a type that shows better performance if **IEnumerable<T>** is used for parallel iteration. This is due to the type's lazy loading semantics.

## Collections with Thread Affinity Requirements

In some rare cases a collection's implementation of the **MoveNext** method may have *thread affinity*. This means that the **MoveNext** method must always be called from a specific thread such as a UI thread. This situation can arise with objects provided by Windows Forms or Windows Presentation Foundation or with objects that pull data from the object model of a Microsoft Office application.

Parallel loops, including PLINQ queries, run by default in worker threads of the .NET Framework thread pool. However, there is a way to marshal elements of the collection from the required threading context to the threads that are executing the loop iterations. Here is an example:

```
// run from thread that is allowed to call MoveNext on source
static void ForEachWithEnumerationOnMainThread<T>(
                          IEnumerable<T> source, Action<T> body)
{
  var collectedData = new BlockingCollection<T>();
  var loop = Task.Factory.StartNew(() =>
                 Parallel.ForEach(
                     collectedData.GetConsumingEnumerable(),
                     body));
  try
  {
    foreach (var item in source) collectedData.Add(item);
  }
  finally { collectedData.CompleteAdding(); }
  loop.Wait();
}
```

The **Parallel.ForEach** loop and the sequential **foreach** loop execute concurrently. The two loops are connected by a special concurrent collection that pipes values from one thread to another. The parallel loop may take advantage of more than one worker thread. The sequential **foreach** loop will only run in the same thread as the method that calls it does.

This variation will only run faster if the body of the **Parallel.ForEach** loop is significantly slower than the **MoveNext** method of the collection. Normally, this will be the case, but you should not expect the same level of performance as you would if **MoveNext** were allowed to be called from within the parallel loop.

## Design Notes

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

The implementations of the parallel loop pattern in the .NET Framework's parallel extensions and in the C++ Parallel Patterns Library contain sophisticated algorithms for dynamic work distribution. These loops automatically adapt to the workload and to a particular machine.

The parallel loop pattern expresses only potential parallelism, but does not guarantee it. This allows performance to scale from single-core scenarios to many cores.

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

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

## Implementation Notes

[TBD – will include description of writing a custom partitioner, approximately 3 pages]

## Related Patterns

Related patterns:

> **Note:** The parallel loop pattern is sometimes called the map pattern, especially when the operation returns a value. In this case, the elements of the input collection are "mapped" or transformed into other values. The use of the term map is especially common in functional languages such as F#.

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

[TBD] reference on lambda expressions and delegates.

Many of the cases in the Variations section of this chapter were taken from Stephen Toub's Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#.

References to Mattson, et al. and academic sources.

# 5 Futures and Continuations

In chapter 3 you saw how the parallel task pattern allowed you to fork the flow of control in a program. In this chapter, you will see how control flow and data flow can be integrated with two patterns known as futures and continuations. These patterns allow you to schedule tasks that can accommodate data flow constraints.

A *future* is a task that returns a value. Instead of explicitly waiting for the task to complete, you simply ask the task for its result when you are ready to use it. If the task has already finished by the time you do this, then its result is waiting for you and is immediately returned. If the task is running but has not yet finished, then the current thread blocks until the result value becomes available. If the task has not yet started, the task will be executed in the current thread context. In other words, querying for the result of a future integrates asynchronous control flow with data flow. Futures are also a good way of expressing the principle of potential parallelism: decomposing a sequential operation with futures may result in much faster execution, and in the worst case will be no slower than the serial case.

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

The **Task<T>** class in the .NET Framework supports the futures pattern. The type parameter **T** gives the type of the task's result. If the parallel tasks described in chapter 3 are asynchronous *actions*, then futures are asynchronous *functions*. (Recall that actions don't return values but functions do.)

**C++ Note:** PPL does not provide a futures implementation. However, the documentation in MSDN shows you how to use tasks to implement this pattern for use in your own code.

**Anti-pattern**: As described in chapter 3 Parallel Tasks, tasks, including futures, cannot always be run inline. In these cases, , the latency of a task-based implementation will be more variable than a sequential implementation due to the strict first-in first-out scheduling of tasks that may not be inlined. There are workarounds available if you experience unacceptable scheduling latency with futures. See the "Implementation Details" section of chapter 3, "Parallel Tasks" for more information about task inlining and the scheduling techniques used by the default task scheduler.

A *continuation*, or *continuation task*, is a task that automatically starts when another task, known as its antecedent, completes. In most cases the antecedent is a future whose result value is used as input to the continuation. An antecedent may have more than one continuation, and a continuation may have more than one antecedent.

A continuation is a task that automatically starts when a specified antecedent task finishes.

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

Unlike other kinds of tasks, continuations are never inlined into the current thread context.

## The Problem

The futures and continuations patterns are the solution to some very common problems. When you think about the parallel task pattern described in chapter 3, you see that in many cases the purpose of a task is to calculate a result. In other words, asynchronous operations often act like functions. Of course, tasks can also do other things, such as reordering values in an array, but calculating new values is common enough to warrant a pattern tailored for it.

### Futures

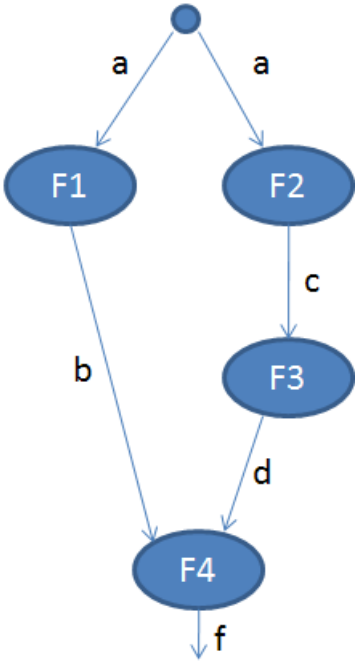
To better understand the problem, consider the following example, taken from the body of a sequential method:

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

Suppose that the functions **F1**, **F2**, **F3** and **F4** are all CPU-intensive computations that interact with each other using arguments and return values instead of updating shared variables in memory.

You want to distribute the work across the available CPUs, and you want your code to run correctly no matter how many CPUs are available on your platform. When you study the example, you can see that **F1** can run in parallel with **F2** and **F3** but that **F3** can't start until **F2** is finished. The possible orderings become apparent when you look at the function calls as a graph.

**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.

Futures and continuations are an easy way to introduce asynchrony that is constrained by these kinds of data flow dependencies. Here's an example that shows how it works. The code assumes for simplicity that the values being calculated are integers and that the value of variable **a** has already been supplied, perhaps as an argument to the current method.

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

The **Result** property returns a pre-calculated 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 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, since the results of both functions are required before **F4** can be invoked. (Recall that the **Result** property does not return until the future's value is available.) Note that the calls to **F2**, **F3** and **F4** do not need to be wrapped inside of a future, since a single additional asynchronous operation is all that is needed to fully exploit the potential parallelism of this example.

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

```
Task<int> futureD = Task<int>.Factory.StartNew(
```

```
                                                () => 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 is run asynchronously.

An important point of this example is that exceptions that occur during the execution of a future will be thrown by the **Result** property. This makes exception handling easy, even in cases with many futures and complex chains of continuations.

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

## Continuations

Another very common case occurs when one asynchronous operation invokes a second asynchronous operation and passes data to it. This is described by the continuations pattern.

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

```
TextBox myTextBox = ...;

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

var futureF = Task.Factory.ContinueWhenAll <int, int>(
                 new[] { futureB, futureD },
                 (tasks) => F4(futureB.Result, futureD.Result));

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

This code structures the computation into four tasks, two futures and two continuations.

The **ContinueWhenAll<T, S>** method of the **Task.Factory** object allows you to create a continuation that depends on more than one antecedent task. The **ContinueWith** method creates a continuation task with a single antecedent. The system understands the ordering dependencies between continuations and their antecedent tasks. It makes sure that continuations will only be started after their antecedent tasks have completed.

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

## The Adatum Financial Dashboard

Let's look at an example of how the futures and continuations patterns 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).

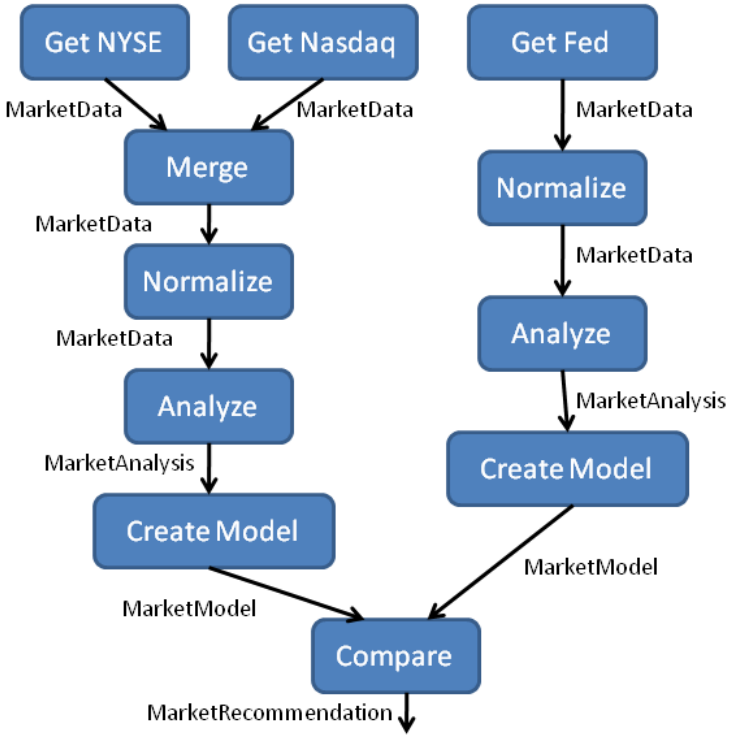
Adatum is a financial services firm that provides a financial dashboard application to its employees. The application, known as the Adatum dashboard allows employees to perform analyses of financial markets. The dashboard application runs on an employee's desktop workstation. The analyses it performs are computationally intensive, but there is also some I/O latency, since the Adatum Dashboard application collects input data from several sources over the network. Employees expect the application to remain responsive regardless of computational load and I/O latencies. Blocking the UI for more than a fraction of a second would be unacceptable.

The application gathers market data from several sources and then merges the data sets together. The application normalizes the merged market data and then performs an analysis step. After the analysis, it creates a market model. It also performs these same steps for Fed historical market data. After the current and historical models are ready, the application compares the two models and makes a market recommendation of "buy," "sell" or "hold."

> **Note:** The Adatum dashboard analyzes historical data, not a stream of real-time price data.

You can visualize these steps as a graph. This is shown in the following diagram.

**Adatum Dashboard Tasks**




The tasks in this diagram communicate by specific types of business objects, such as **MarketData**, **MarketAnalysis**, **MarketModel** and **MarketRecommendation**. These are implemented as .NET classes in the Adatum dashboard application.

> **Note:** You can download the complete source code for the Adatum dashboard application from the CodePlex site http://parallelpatterns.codeplex.com. The application consists of three parts: the business object definitions, an analysis engine and the user interface.

## The Dashboard's Analysis Engine

The Adatum dashboard's **AnalysisEngine** class could use either parallel or sequential modes of execution to produce a market recommendation.

The sequential process is shown in the following code.

```
public MarketRecommendation DoAnalysisSequential()
{
   return CompareModels(
          new[]
          {
               CreateModel(
```

```
                    AnalyzeData(
                       NormalizeData(
                           MergeMarketData(
                              new [] { LoadNyseData(),
                                       LoadNasdaqData() }))),
                CreateModel(
                    AnalyzeData(
                       NormalizeData(
                              LoadFedHistoricalData())))
                      });
}
```

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

The parallel version uses futures and continuations. It creates futures for each of the operational steps and connects them to each other with continuations. You can look online for the complete example, but here are some highlights.

```
public AnalysisTasks StartParallelAnalysis()
{
    // Create each task
    loadNyseData = Task<MarketData>.Factory.StartNew(...);

    // ...

    return new AnalysisTasks()
    {
        LoadNyseData = loadNyseData,
        LoadNasdaqData =  ... ,
        MergeMarketData =  ... ,
        NormalizeMarketData =  ... ,
        LoadFedHistoricalData =  ... ,
        NormalizeHistoricalData =  ... ,
        AnalyzeMarketData = ... ,
        ModelMarketData =  ... ,
        AnalyzeHistoricalData =  ... ,
        ModelHistoricalData =  ... ,
        CompareModels =  ...
    };
}
```

In the parallel case, the application uses an asynchronous operation implemented by the **StartParallelAnalysis** method.

The **StartParallelAnalysis** method returns an **AnalysisTasks** object that contains futures for each of the operations in the task graph. For example, the **CompareModels** property contains a future that returns

a market recommendation value. The future is implemented as a **Task<MarketRecommendation>** object. To get the final result of the analysis, use the following code.

```
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
AnalysisEngine ae = new AnalysisEngine(token)
AnalysisTasks tasks = ae.StartParallelAnalysis();
MarketRecommendation recommendation = tasks.CompareModels.Result;
```

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

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

Now let's look at how each of the tasks is created.

### Loading External Data

The methods that gather the external data from the network are long-running, I/O intensive operations. Unlike the other steps, they are not particularly CPU intensive. Most of their time is spent waiting for I/O operations to complete. You create these tasks using a factory object and you use an argument to specify that the tasks are I/O intensive and of long duration.

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

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

Note that the **Task<MarketData>.Factory** object creates futures that return values of type **MarketData**. The **TaskCreationOptions.LongRunning** enumerated value tells the task library that the operations are not CPU-intensive and are expected to run for a long time. To prevent underutilization of CPU resources, the task library may choose to run tasks like these in additional threads.

> Use "long running" tasks for tasks that are not CPU-intensive, such as long-running I/O operations.

### Merging

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

```
Task<MarketData> mergeMarketData =
    Task.Factory.ContinueWhenAll<MarketData, MarketData>(
        new[] { loadNyseData, loadNasdaqData },
```

```
        (tasks) => MergeMarketData(
            from t in tasks select t.Result));
```

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

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

### Normalizing

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

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

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

### Analysis and Model Creation

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

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

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

Analysis and model creation are two additional examples of continuations with a single antecedent task.

### Processing Historical Data

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

### Comparing Models

```
Task<MarketRecommendation> compareModels =
   Task.Factory.ContinueWhenAll<MarketModel,
MarketRecommendation>(
        new[] { modelMarketData, modelHistoricalData },
```

```
        (tasks) => CompareModels(
           from t in tasks select t.Result));
```

The “compare models” task compares the current and historical market models and produces the final result.

## The Dashboard’s User Interface

Futures and continuations are also used in the Adatum dashboard’s user interface. It uses the Windows Presentation Foundation (WPF) framework.

The Adatum dashboard user interface is designed so that the result of each analysis step can be viewed by the user as the computation progresses. There are individual buttons for each of the steps. As each result becomes available, the buttons are individually enabled. It is also possible to cancel the analysis from the user interface.

The application uses continuations instead of registering callbacks. This has the advantage of not requiring the analysis layer to refer to any types defined by the user interface.

Here’s a walkthrough of how the notification works.

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

  void OnRequestCalculate()
  {
     // ...

     this._cancellationTokenSource =
                          new CancellationTokenSource();
     CancellationToken token =
                        this._cancellationTokenSource.Token;

     // Start the analysis
     var analysisEngine = new AnalysisEngine(token);
     AnalysisTasks tasks =
                    analysisEngine.StartAnalysisParallel();

     AddButtonContinuations(tasks, token)
  }

  // ...
}
```

The user interface uses the Model View ViewModel (MVVM) pattern. The main window’s view model has a **Calculate** command that invokes the **OnRequestCalculate** method in response to a user-interface button press.

The cancellation token is passed as an argument to the data analysis object.

Rather than starting background tasks with a method such as **QueueUserWorkItem** from the **ThreadPool** class, the view model asks the analysis engine to create a record that contains tasks corresponding to each independently viewable analysis result.

This architecture demonstrates decoupling. The person who wrote the analysis layer of the application was able to do this without any knowledge of how other parts of the application would use the results of the analysis.

Next, the handler creates user interface-specific continuations with the **AddButtonContinuations** method. This is shown in the following code.

```
public void AddButtonContinuations(AnalysisTasks tasks)
{
  TaskScheduler s =
     TaskScheduler.FromCurrentSynchronizationContext();

  tasks.LoadNyseData.ContinueWith(
    (t) => { this.NyseMarketData = t.Result; }, s);
  tasks.LoadNasdaqData.ContinueWith(
    (t) => { this.NasdaqMarketData = t.Result; }, s);
  tasks.LoadFedHistoricalData.ContinueWith(
    (t) => { this.FedHistoricalData = t.Result; }, s);
  tasks.MergeMarketData.ContinueWith(
    (t) => { this.MergedMarketData = t.Result; }, s);
  tasks.NormalizeHistoricalData.ContinueWith(
    (t) => { this.NormalizedHistoricalData = t.Result; }, s);
  tasks.NormalizeMarketData.ContinueWith(
    (t) => { this.NormalizedMarketData = t.Result; }, s);
  tasks.AnalyzeHistoricalData.ContinueWith(
    (t) => { this.AnalyzedHistoricalData = t.Result; }, s);
  tasks.AnalyzeMarketData.ContinueWith(
    (t) => { this.AnalyzedMarketData = t.Result; }, s);
  tasks.ModelHistoricalData.ContinueWith(
    (t) => { this.ModeledHistoricalData = t.Result; }, s);
  tasks.ModelMarketData.ContinueWith(
    (t) => { this.ModeledMarketData = t.Result; }, s);
  tasks.CompareModels.ContinueWith(
    (t) => {
              this.Recommendation = t.Result;
              this.StatusTextBoxText =
                 (this.Recommendation == null) ?
                 "Canceled" : this.Recommendation.Recommendation;
              this.ModelState = State.Ready;
             }, s);

}
```

The method creates continuations that will automatically run *in the user-interface thread* after each of the tasks finishes and has results ready to view. When the final "compare models" task finishes, the user interface view model will be updated with the final recommendation. Callbacks registered with the view model will notify the user interface of the changes so that these changes will be reflected in the UI.

The result of the **FromCurrentSynchronizationContext** method of the **TaskScheduler** class is a **ThreadScheduler** object that will only allow its tasks to run in the current thread (that is, the user interface thread).

### Continuations for User Interfaces

The Adatum dashboard application demonstrates how an application can use continuations to keep the user interface up to date.

Before adopting parallelism, Adatum used background worker threads to handle the computationally intensive parts of applications such as the Adatum dashboard. However, the Adatum dashboard application has some requirements, such as the use of WPF for the user interface, that make continuations more appropriate than background worker threads.

One of the reasons that the futures and continuations patterns works for the Adatum dashboard is because it satisfies the constraints of thread affinity. Some frameworks place this constraint on objects they expose. For example, in WPF you have to run all methods of a user-interface object on the same thread that you used to create that object.

The futures and continuations pattern makes it easy to deal with thread affinity constraints. Continuation tasks can be configured with a task scheduler that only runs the task on a chosen thread. Antecedents of the task do not need to run on the same thread as the continuation. This allowed Adatum's developers to distribute computationally intensive work among many CPUs while allowing the calculated values to appear in the user interface without violating WPF's thread affinity constraint.

Tasks make it easy to satisfy thread affinity constraints.

Task scheduling has been optimized for CPU-intensive operations. It is more efficient to run fewer threads than tasks. With futures and continuations, the amount of parallelism you can achieve is only limited by your design and the number of available CPUs. This eliminates a restriction of .NET 2.0 background worker threads which are limited to a single instance and cannot have data dependencies. Therefore, they are not appropriate when you are merging the results from multiple concurrent calculations.

**Anti-pattern**: When are continuations used rather than .NET Framework 2.0 background worker threads? Answer: A background worker thread is inappropriate when you are merging the results from multiple concurrent calculations. (Also, possibly, when there is more than one background task needed, regardless of merging.)

### Modifying the Graph at Runtime

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

The extension of the analysis tasks in the user interface layer is an example of dynamic task creation. This user interface augments the graphs of tasks by adding continuations programmatically, *outside of the context where these tasks were originally created*.

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

## Support for Cancellation

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

The application checks for this condition at various checkpoints. If cancellation has been requested, the operation is aborted.

# Variations

So far you've seen some of the most common ways to use futures and continuations to create tasks. Here are some other ways to use them.

## Canceling Futures and Continuations

There are several patterns relating to the cancellation of futures and continuations. [TBD] You can handle cancellation entirely from within the task, as Adatum's dashboard does or you can pass cancellation tokens as an option when the tasks are created.

> **Anti-pattern**: If you pass a cancellation token as an argument to a task's constructor (or to the **StartNew** method of the **Task.Factory** object), you should also make sure that every continuation task is passed that same cancellation token when it is created. If you don't do this consistently, you may experience unhandled **OperationCancelled** exceptions.

## Handling Exceptions in a Continuation

Using a continuation to handle exceptions thrown by the antecedent can be a useful pattern. Here is an example:

```
var t1 = Task<int>.Factory.StartNew(() => F(token));
```

```
var t2 = t1.ContinueWith<int>((t) =>
  {
    try
    {
       return t2.Result;
    }
    catch (AggregateException ae)
    {
       ae.Handle((e) =>
       {
         if (e is MyException)
         {
            Console.WriteLine("MyException caught: " +
e.Message);
            return true;
         }
         return false;
       });
    }
  });
var t3 = t2.ContinueWith((t) => G(t.Result));

return t3.Result;
```

In this example, the continuation **t2** passes the result of its antecedent to the next continuation, unless it catches an exception of type **MyException**.

## Continue When "At Least One" Antecedent Completes

It is possible to invoke a continuation when the first of multiple antecedents completes. To do this, use the **Task<T>.Factory** object's **ContinueWhenAny** method.

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

This is useful when the result of any of the tasks will do. For example each task queries a web service which gives the local weather. The first answer is returned to the user.

## Converting a .NET Asynchronous Call into a Future

It is possible to convert Asynch calls into futures. [TBD]

Futures and continuations are similar in some ways to asynchronous methods that use the .NET **IAsynchResult** interface. In general, futures are easier to use than **IAsynchResult**. See "Variations and Related Patterns" in this chapter for more information.

Futures are easier to use than **IAsynchResult** and benefit from easier exception handling.

## Design Notes

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

### Decomposition into Futures and Continuations

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

### Functional style

There are two approaches to synchronizing data between tasks. In this chapter, the examples have used an explicit approach. Data is passed between tasks as parameters, which makes the data dependencies very obvious to the programmer. Alternatively, as you saw in chapter 2, "Parallel Tasks", it is possible for tasks to communicate with side effects on  the tasks could act on shared data structures. In this case, and rely on thettasks use control dependencies to block appropriately and control flowfor the (implied) data flow.

You can use control flow constraints and side effects with shared data structures for legacy applications and in cases. In general, explicit data flow is less prone to error.

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

Similarly, using futures (tasks that return values) can reduce the possibility of error in a parallel program as compared to tasks that communicate results by a modified shared global state. Tasks that return values can often require less synchronization than tasks that globally accessible state variables.

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

The design of the Adatum Dashboard uses the functional style of programming with a focus on operations that communicate using input and output values. This is in contrast to programs that modify structures in place. The functional pattern is well suited to parallel programming. Functional programs are very easy to adapt to multicore environments. Let’s examine a few of the assumptions.

The first is a commitment to scalable sharing of data. This means that futures should generally only communicate with the outside world by means of their return values. In general, they should be as free as possible of side effects such as writes to mutable shared variables. If you do read and write to shared

variables you will need to serialize your program using synchronization objects or be extraordinarily careful when you write to shared state.

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

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

### Functional Programming using Value Types

If you use the futures and continuations pattern, you may want to use value types. [TBD]

## Implementation Notes

The Adatum Dashboard example introduced **Task<T>** class (i.e., tasks that return values), **Task.ContinueWith**, and **Task.Factory.ContinueWhenAll** methods.

Callout: Comparison with existing .NET asynch UI programming patterns. [TBD]

## Related Patterns

There are a number of variations and related patterns for the continuation pattern. [TBD]

## References

Leijen (OOPSLA 09) describes the motivation for including futures in TPL and has references to other work, especially in functional languages.

Introductory reference to functional programming concepts, targeted at C#.

Reference to NModel implementation of immutable collections (set, bag, seq, map) (http://nmodel.codeplex.com).