Why PFX?

Over the past 15 years, CPU manufacturers have shifted from single-core to multicore processors. This is problematic for us as programmers because single-threaded code does not automatically run faster as a result of those extra cores.

Utilizing multiple cores is easy for most server applications, where each thread can independently handle a separate client request, but it's more difficult on the desktop because it typically requires that you take your computationally intensive code and do the following:

1. *Partition* it into small chunks.
2. Execute those chunks in parallel via multithreading.
3. *Collate* the results as they become available, in a thread-safe and performant manner.

Although you can do all of this with the classic multithreading constructs, it's awkward—particularly the steps of partitioning and collating. A further problem is that the usual strategy of locking for thread safety causes a lot of contention when many threads work on the same data at once.

The PFX libraries have been designed specifically to help in these scenarios.

NOTE

Programming to leverage multicores or multiple processors is called *parallel programming*. This is a subset of the broader concept of multithreading.

PFX Concepts

There are two strategies for partitioning work among threads: *data parallelism* and *task parallelism*.

When a set of tasks must be performed on many data values, we can parallelize by having each thread perform the (same) set of tasks on a subset of values. This is called *data parallelism* because we are partitioning the *data* between threads. In contrast, with *task parallelism* we partition the *tasks*; in other words, we have each thread perform a different task.

In general, data parallelism is easier and scales better to highly parallel hardware because it reduces or eliminates shared data (thereby reducing contention and thread-safety issues). Also, data parallelism exploits the fact that there are often more data values than discrete tasks, increasing the parallelism potential.

Data parallelism is also conducive to *structured parallelism*, which means that parallel work units start and finish in the same place in your program. In contrast, task parallelism tends to be unstructured, meaning that parallel work units may start and finish in places scattered across your program. Structured parallelism is simpler and less error prone and allows you to farm the difficult job of partitioning and thread coordination (and even result collation) out to libraries.

PFX Components

PFX comprises two layers of functionality, as shown in Figure 22-1. The higher layer consists of two *structured data parallelism* APIs: PLINQ and the `Parallel` class. The lower layer contains the task parallelism classes—plus a set of additional constructs to help with parallel programming activities.

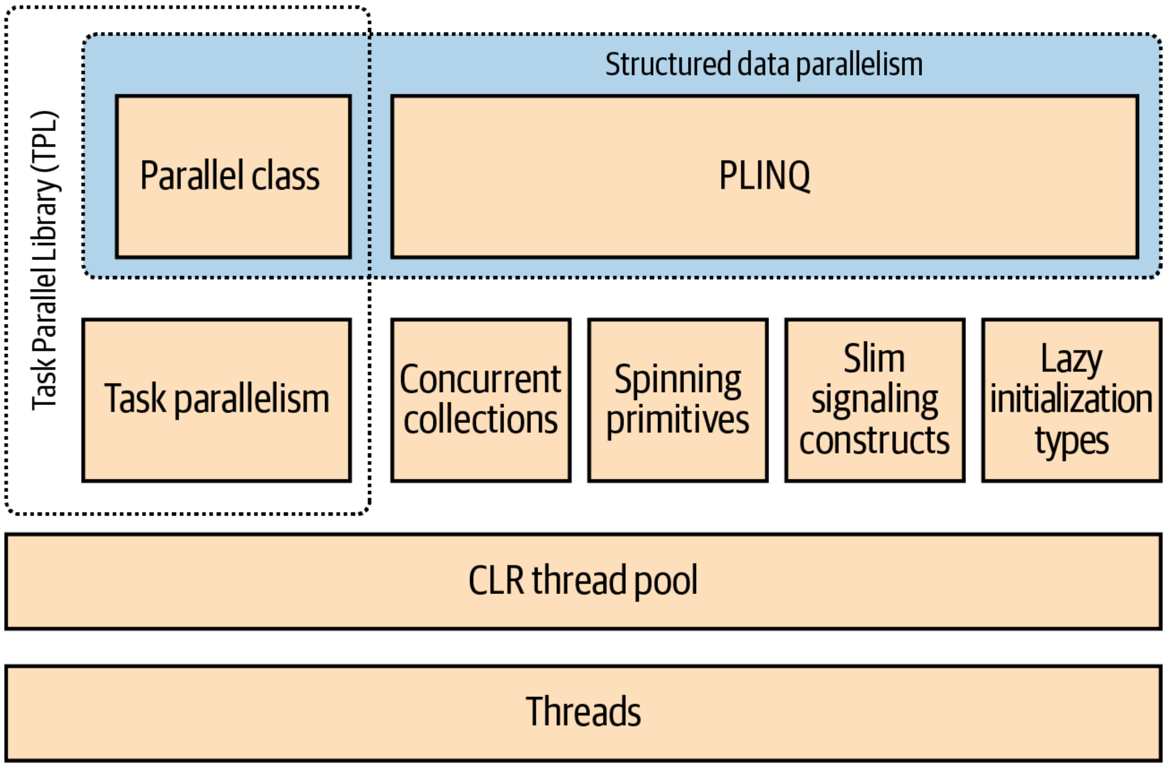
Structured data parallelism
Task Parallel Library (TPL)
Parallel class
PLINQ
Task parallelism
Concurrent collections
Spinning primitives
Slim signaling constructs
Lazy initialization types
CLR thread pool
Threads

Figure 22-1. PFX components

PLINQ offers the richest functionality: it automates all the steps of parallelization—including partitioning the work into tasks, executing those tasks on threads, and collating the results into a single output sequence. It's called *declarative*—because you simply declare that you want to parallelize your work (which you structure as a LINQ query) and let the runtime take care of the implementation details. In contrast, the other approaches are *imperative*, in that you need to explicitly write code to partition or collate.

As the following synopsis shows, in the case of the `Parallel` class, you must collate results yourself; with the task parallelism constructs, you must partition the work yourself, too:

	Partitions work	**Collates results**
PLINQ	Yes	Yes
The `Parallel` class	Yes	No
PFX’s task parallelism	No	No

The concurrent collections and spinning primitives help you with lower-level parallel programming activities. These are important because PFX has been designed to work not only with today’s hardware, but also with future generations of processors with far more cores. If you want to move a pile of chopped wood and you have 32 workers to do the job, the biggest challenge is moving the wood without the workers getting in each other’s way. It’s the same with dividing an algorithm among 32 cores: if ordinary locks are used to protect common resources, the resultant blocking can mean that only a fraction of those cores are ever actually busy at once. The concurrent collections are tuned specifically for highly concurrent access, with the focus on minimizing or eliminating blocking. PLINQ and the `Parallel` class themselves rely on the concurrent collections and on spinning primitives for efficient management of work.

OTHER USES FOR PFX

The parallel programming constructs are useful not only for leveraging multicores, but also in other scenarios:

- The concurrent collections are sometimes appropriate when you want a thread-safe queue, stack, or dictionary.
- `BlockingCollection` provides an easy means to implement producer/consumer structures, and is a good way to *limit* concurrency.
- Tasks are the basis of asynchronous programming, as we saw in Chapter 14.

When to Use PFX

The primary use case for PFX is *parallel programming*: leveraging multicore processors to speed up computationally intensive code.

A challenge in parallel programming is Amdahl's law, which states that the maximum performance improvement from parallelization is governed by the portion of the code that must execute sequentially. For instance, if only two-thirds of an algorithm's execution time is parallelizable, you can never exceed a threefold performance gain—even with an infinite number of cores.

So, before proceeding, it's worth verifying that the bottleneck is in parallelizable code. It's also worth considering whether your code *needs* to be computationally intensive—optimization is often the easiest and most effective approach. There's a trade-off, though, in that some optimization techniques can make it more difficult to parallelize code.

The easiest gains come with what's called *embarrassingly parallel* problems—this is when a job can be easily divided into tasks that efficiently execute on their own (structured parallelism is very well suited to such problems). Examples include many image-processing tasks, ray tracing, and brute-force approaches in mathematics or cryptography. An example of a non-embarrassingly parallel problem is implementing an optimized version of the quicksort algorithm—a good result takes some thought and might require unstructured parallelism.

PLINQ

PLINQ automatically parallelizes local LINQ queries. PLINQ has the advantage of being easy to use in that it offloads the burden of both work partitioning and result collation to .NET.

To use PLINQ, simply call `AsParallel()` on the input sequence and then continue the LINQ query as usual. The following query calculates the prime numbers between 3 and 100,000, making full use of all cores on the target machine:

```
// Calculate prime numbers using a simple (unoptimized) algorithm.

IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

int[] primes = parallelQuery.ToArray();
```

`AsParallel` is an extension method in `System.Linq.ParallelEnumerable`. It wraps the input in a sequence based on `ParallelQuery<TSource>`, which causes the LINQ query operators that you subsequently call to bind to an alternate set of extension methods defined in `ParallelEnumerable`. These provide parallel implementations of each of the standard query operators. Essentially, they work by partitioning the input sequence into chunks that execute on different threads, collating the results back into a single output sequence for consumption, as depicted in Figure 22-2.

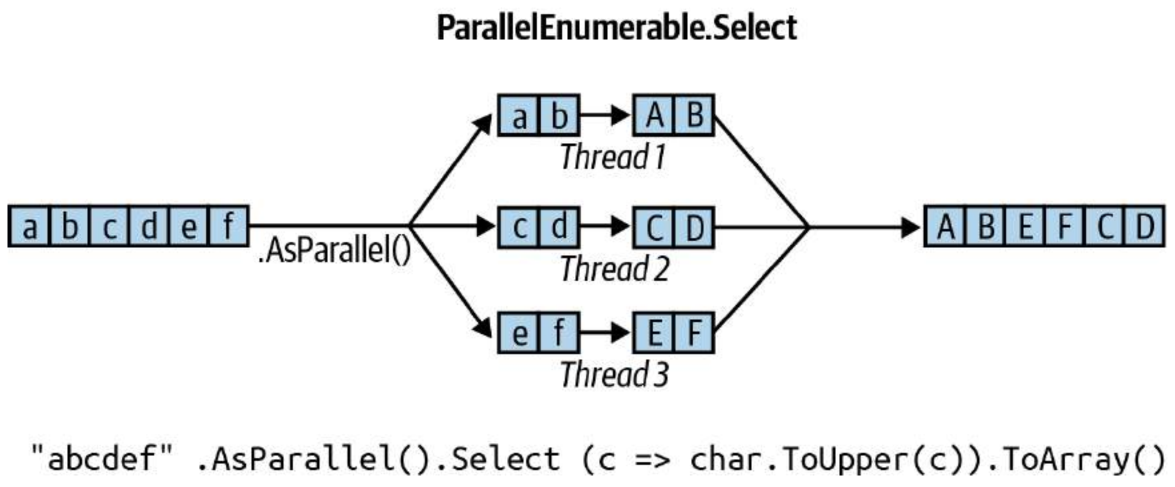
ParallelEnumerable.Select
a b c d e f
.AsParallel()
a b
A B
Thread 1
c d
C D
Thread 2
e f
E F
Thread 3
A B E F C D
"abcdef" .AsParallel().Select (c => char.ToUpper(c)).ToArray()

Figure 22-2. PLINQ execution model

Calling `AsSequential()` unwraps a `ParallelQuery` sequence so that subsequent query operators bind to the standard query operators and execute sequentially. This is necessary before calling methods that have side effects or are not thread-safe.

For query operators that accept two input sequences (`Join`, `GroupJoin`, `Concat`, `Union`, `Intersect`, `Except`, and `Zip`), you must apply `AsParallel()` to both input sequences (otherwise, an exception is thrown). You don't, however, need to keep applying `AsParallel` to a query as it progresses, because PLINQ's query operators output another `ParallelQuery` sequence. In fact, calling `AsParallel` again introduces inefficiency in that it forces merging and repartitioning of the query:

```
mySequence.AsParallel()           // Wraps sequence in ParallelQuery<int>
          .Where (n => n > 100)   // Outputs another ParallelQuery<int>
          .AsParallel()           // Unnecessary - and inefficient!
          .Select (n => n * n)
```

Not all query operators can be effectively parallelized. For those that cannot (see "PLINQ Limitations"), PLINQ implements the operator sequentially, instead. PLINQ might also operate sequentially if it suspects that the overhead of parallelization will actually slow a particular query.

PLINQ is only for local collections: it doesn't work with Entity Framework, for instance, because in those cases the LINQ translates into SQL, which then executes on a database server. However, you *can* use PLINQ to perform additional local querying on the result sets obtained from database queries.

WARNING

If a PLINQ query throws an exception, it’s rethrown as an `AggregateException` whose `InnerExceptions` property contains the real exception (or exceptions). For more details, see “Working with AggregateException”.

WHY ISN'T ASPARALLEL THE DEFAULT?

Given that `AsParallel` transparently parallelizes LINQ queries, the question arises: Why didn't Microsoft simply parallelize the standard query operators and make PLINQ the default?

There are a number of reasons for the *opt-in* approach. First, for PLINQ to be useful, there must be a reasonable amount of computationally intensive work for it to farm out to worker threads. Most LINQ-to-Objects queries execute very quickly; thus, not only would parallelization be unnecessary, but the overhead of partitioning, collating, and coordinating the extra threads might actually slow things down.

Additionally:

- The output of a PLINQ query (by default) can differ from a LINQ query with respect to element ordering (see "PLINQ and Ordering").
- PLINQ wraps exceptions in an `AggregateException` (to handle the possibility of multiple exceptions being thrown).
- PLINQ will give unreliable results if the query invokes thread-unsafe methods.

Finally, PLINQ offers quite a few hooks for tuning and tweaking. Burdening the standard LINQ-to-Objects API with such nuances would add distraction.

Parallel Execution Ballistics

Like ordinary LINQ queries, PLINQ queries are lazily evaluated. This means that execution is triggered only when you begin consuming the results—typically via a `foreach` loop (although it can also be via a conversion operator such as `ToArray` or an operator that returns a single element or value).

As you enumerate the results, though, execution proceeds somewhat differently from that of an ordinary sequential query. A sequential query is powered entirely by the consumer in a "pull" fashion: each element from the input sequence is fetched exactly when required by the consumer. A parallel query ordinarily uses independent threads to fetch elements from the input sequence slightly *ahead* of when they're needed by the consumer (rather like a teleprompter for newsreaders). It then processes the elements in parallel through the query chain, holding the results in a small buffer so that they're ready for the consumer on demand. If the consumer pauses or breaks out of the enumeration early, the query processor also pauses or stops so as not to waste CPU time or memory.

NOTE

You can tweak PLINQ's buffering behavior by calling `WithMergeOptions` after `AsParallel`. The default value of `AutoBuffered` generally gives the best overall results. `NotBuffered` disables the buffer and is useful if you want to see results as soon as possible; `FullyBuffered` caches the entire result set before presenting it to the consumer (the `OrderBy` and `Reverse` operators naturally work this way, as do the element, aggregation, and conversion operators).

PLINQ and Ordering

A side effect of parallelizing the query operators is that when the results are collated, it's not necessarily in the same order that they were submitted (see Figure 22-2). In other words, LINQ's normal order-preservation guarantee for sequences no longer holds.

If you need order preservation, you can force it by calling `AsOrdered()` after `AsParallel()`:

```
myCollection.AsParallel().AsOrdered()...
```

Calling `AsOrdered` incurs a performance hit with large numbers of elements because PLINQ must keep track of each element's original position.

You can negate the effect of `AsOrdered` later in a query by calling `AsUnordered`: this introduces a "random shuffle point," which allows the query to execute more efficiently from that point on. So, if you wanted to preserve input-sequence ordering for just the first two query operators, you'd do this:

```
inputSequence.AsParallel().AsOrdered()
  .QueryOperator1()
  .QueryOperator2()
  .AsUnordered()       // From here on, ordering doesn't matter
  .QueryOperator3()
  ...
```

`AsOrdered` is not the default because for most queries, the original input ordering doesn't matter. In other words, if `AsOrdered` were the default, you'd need to apply `AsUnordered` to the majority of your parallel queries to get the best performance, which would be burdensome.

PLINQ Limitations

There are practical limitations on what PLINQ can parallelize. The following query operators prevent parallelization by default unless the source elements are in their original indexing position:

- The indexed versions of `Select`, `SelectMany`, and `ElementAt`

Most query operators change the indexing position of elements (including those that remove elements, such as `Where`). This means that if you want to use the preceding operators, they'll usually need to be at the start of the query.

The following query operators are parallelizable but use an expensive partitioning strategy that can sometimes be slower than sequential processing:

`Join`, `GroupBy`, `GroupJoin`, `Distinct`, `Union`, `Intersect`, and `Except`

The `Aggregate` operator's *seeded* overloads in their standard incarnations are not parallelizable—PLINQ provides special overloads to deal with this (see "Optimizing PLINQ").

All other operators are parallelizable, although use of these operators doesn't guarantee that your query will be parallelized. PLINQ might run your query sequentially if it suspects that the overhead of parallelization will slow down that particular query. You can override this behavior and force parallelism by calling the following after `AsParallel()`:

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Example: Parallel Spellchecker

Suppose that we want to write a spellchecker that runs quickly with very large documents by utilizing all available cores. By formulating our algorithm into a LINQ query, we can very easily parallelize it.

The first step is to download a dictionary of English words into a `HashSet` for efficient lookup:

```
if (!File.Exists ("WordLookup.txt")    // Contains about 150,000 words
  File.WriteAllText ("WordLookup.txt",
    await new HttpClient().GetStringAsync (
      "http://www.albahari.com/ispell/allwords.txt"));

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

We then use our word lookup to create a test "document" comprising an array of a million random words. After we build the array, let's introduce a couple of spelling mistakes:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();
```

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Introduce a couple
wordsToTest [23456] = "wubsie";     // of spelling mistakes.
```

Now we can perform our parallel spellcheck by testing `wordsToTest` against `wordLookup`. PLINQ makes this very easy:

```
var query = wordsToTest
  .AsParallel()
  .Select  ((word, index) => (word, index))
  .Where   (iword => !wordLookup.Contains (iword.word))
  .OrderBy (iword => iword.index);

foreach (var mistake in query)
  Console.WriteLine (mistake.word + " - index = " + mistake.index);

// OUTPUT:
// woozsh - index = 12345
// wubsie - index = 23456
```

The `wordLookup.Contains` method in the predicate gives the query some "meat" and makes it worth parallelizing.

NOTE

Notice that our query uses tuples `(word, index)` rather than anonymous types. Because tuples are implemented as value types rather than reference types, this improves peak memory consumption and performance by reducing heap allocations and subsequent garbage collections. (Benchmarking reveals the gains to be moderate in practice, due to the efficiency of the memory manager and the fact that the allocations in question don't survive beyond Generation 0.)

Using ThreadLocal<T>

Let's extend our example by parallelizing the creation of the random test-word list itself. We structured this as a LINQ query, so it should be easy. Here's the sequential version:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

Unfortunately, the call to `random.Next` is not thread-safe, so it's not as simple as inserting `AsParallel()` into the query. A potential solution is to write a function that locks around `random.Next`; however, this would limit concurrency. The better option is to use `ThreadLocal<Random>` (see "Thread-Local Storage") to create a separate `Random` object for each thread. We then can parallelize the query, as follows:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );

string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

In our factory function for instantiating a `Random` object, we pass in a `Guid`'s hashcode to ensure that if two `Random` objects are created within a short period of time, they'll yield different random number sequences.

WHEN TO USE PLINQ

It’s tempting to search your existing applications for LINQ queries and experiment with parallelizing them. This is usually unproductive, because most problems for which LINQ is obviously the best solution tend to execute very quickly and so don’t benefit from parallelization. A better approach is to find a CPU-intensive bottleneck and then consider whether it can be expressed as a LINQ query. (A welcome side effect of such restructuring is that LINQ typically makes code smaller and more readable.)

PLINQ is well suited to embarrassingly parallel problems. It can be a poor choice for imaging, however, because collating millions of pixels into an output sequence creates a bottleneck. Instead, it’s better to write pixels directly to an array or unmanaged memory block and use the `Parallel` class or task parallelism to manage the multithreading. (It is possible, however, to defeat result collation using `ForAll`—we discuss this in “Optimizing PLINQ”. Doing so makes sense if the image processing algorithm naturally lends itself to LINQ.)

Functional Purity

Because PLINQ runs your query on parallel threads, you must be careful not to perform thread-unsafe operations. In particular, writing to variables is *side-effecting* and therefore thread-unsafe:

```
// The following query multiplies each element by its position.
// Given an input of Enumerable.Range(0,999), it should output squares.
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

We could make incrementing `i` thread-safe by using locks, but the problem would still remain that `i` won’t necessarily correspond to the position of the input element. And adding `AsOrdered` to the query wouldn’t fix the latter problem, because `AsOrdered` ensures only that the elements are output in an order consistent with them having been processed sequentially—it doesn't actually *process* them sequentially.

The correct solution is to rewrite our query to use the indexed version of `Select`:

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

For best performance, any methods called from query operators should be thread-safe by virtue of not writing to fields or properties (non-side-effecting, or *functionally pure*). If they're thread-safe by virtue of *locking*, the query's parallelism potential will be limited by the effects of contention.

Setting the Degree of Parallelism

By default, PLINQ chooses an optimum degree of parallelism for the processor in use. You can override it by calling `WithDegreeOfParallelism` after `AsParallel`:

```
...AsParallel().WithDegreeOfParallelism(4)...
```

An example of when you might increase the parallelism beyond the core count is with I/O-bound work (downloading many web pages at once, for instance). However, task combinators and asynchronous functions provide a similarly easy and more *efficient* solution (see "Task Combinators"). Unlike with `Tasks`, PLINQ cannot perform I/O-bound work without blocking threads (and *pooled* threads, to make matters worse).

Changing the degree of parallelism

You can call `WithDegreeOfParallelism` only once within a PLINQ query. If you need to call it again, you must force merging and repartitioning of the query by calling `AsParallel()` again within the query:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
```

```
    .AsParallel().WithDegreeOfParallelism (3)   // Forces Merge + Partition
    .Select (c => char.ToUpper (c))
```

Cancellation

Canceling a PLINQ query whose results you're consuming in a `foreach` loop is easy: simply break out of the `foreach` and the query will be automatically canceled as the enumerator is implicitly disposed.

For a query that terminates with a conversion, element, or aggregation operator, you can cancel it from another thread via a *cancellation token* (see "Cancellation"). To insert a token, call `WithCancellation` after calling `AsParallel`, passing in the `Token` property of a `CancellationTokenSource` object. Another thread can then call `Cancel` on the token source (or we can call it ourselves with a delay). This then throws an `OperationCanceledException` on the query's consumer:

```
IEnumerable<int> tenMillion = Enumerable.Range (3, 10_000_000);

var cancelSource = new CancellationTokenSource();
cancelSource.CancelAfter (100);   // Cancel query after 100 milliseconds

var primeNumberQuery =
  from n in tenMillion.AsParallel().WithCancellation (cancelSource.Token)
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

try
{
  // Start query running:
  int[] primes = primeNumberQuery.ToArray();
  // We'll never get here because the other thread will cancel us.
}
catch (OperationCanceledException)
{
  Console.WriteLine ("Query canceled");
}
```

Upon cancellation, PLINQ waits for each worker thread to finish with its current element before ending the query. This means that any external methods that the query calls will run to completion.

Optimizing PLINQ

Output-side optimization

One of PLINQ's advantages is that it conveniently collates the results from parallelized work into a single output sequence. Sometimes, though, all that you end up doing with that sequence is running some function once over each element:

```
foreach (int n in parallelQuery)
  DoSomething (n);
```

If this is the case—and you don't care about the order in which the elements are processed—you can improve efficiency with PLINQ's `ForAll` method.

The `ForAll` method runs a delegate over every output element of a `ParallelQuery`. It hooks directly into PLINQ's internals, bypassing the steps of collating and enumerating the results. Here's a trivial example:

```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write);
```

Figure 22-3 shows the process.

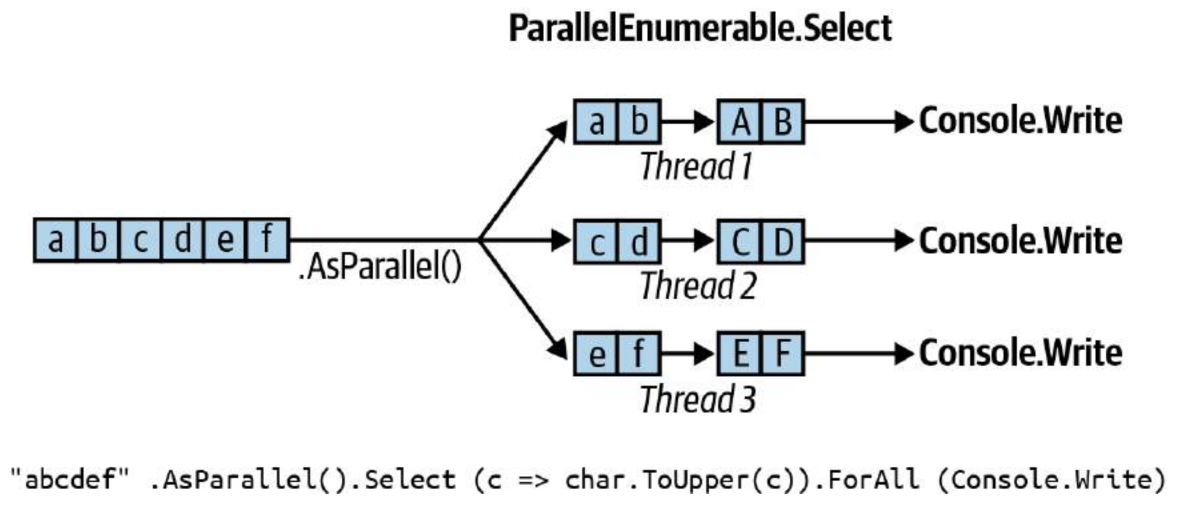

Figure 22-3. PLINQ `ForAll`

NOTE

Collating and enumerating results is not a massively expensive operation, so the `ForAll` optimization yields the greatest gains when there are large numbers of quickly executing input elements.

Input-side optimization

PLINQ has three partitioning strategies for assigning input elements to threads:

Strategy	Element allocation	Relative performance
Chunk partitioning	Dynamic	Average
Range partitioning	Static	Poor to excellent
Hash partitioning	Static	Poor

For query operators that require comparing elements (`GroupBy`, `Join`, `GroupJoin`, `Intersect`, `Except`, `Union`, and `Distinct`), you have no choice: PLINQ always uses *hash partitioning*. Hash partitioning is relatively inefficient in that it must precalculate the hashcode of every element (so that elements with identical hashcodes can be processed on the same thread). If you find this to be too slow, your only option is to call `AsSequential` to disable parallelization.

For all other query operators, you have a choice as to whether to use range or chunk partitioning. By default:

- If the input sequence is *indexable* (if it's an array or implements `IList<T>`), PLINQ chooses *range partitioning*.
- Otherwise, PLINQ chooses *chunk partitioning*.

In a nutshell, range partitioning is faster with long sequences for which every element takes a similar amount of CPU time to process. Otherwise, chunk partitioning is usually faster.

To force range partitioning:

- If the query starts with `Enumerable.Range`, replace that method with `ParallelEnumerable.Range`.
- Otherwise, simply call `ToList` or `ToArray` on the input sequence (obviously, this incurs a performance cost in itself, which you should take into account).

WARNING

`ParallelEnumerable.Range` is not simply a shortcut for calling `Enumerable.Range(…).AsParallel()`. It changes the performance of the query by activating range partitioning.

To force chunk partitioning, wrap the input sequence in a call to `Partitioner.Create` (in `System.Collection.Concurrent`), as follows:

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };
var parallelQuery =
  Partitioner.Create (numbers, true).AsParallel()
  .Where (...)
```

The second argument to `Partitioner.Create` indicates that you want to *load-balance* the query, which is another way of saying that you want chunk partitioning.

Chunk partitioning works by having each worker thread periodically grab small “chunks” of elements from the input sequence to process (see Figure 22-4). PLINQ starts by allocating very small chunks (one or two elements at a time). It then increases the chunk size as the query progresses: this ensures that small sequences are effectively parallelized and large sequences don’t cause excessive round-tripping. If a worker happens to get “easy” elements (that process quickly), it will end up getting more chunks. This system keeps every thread equally busy (and the cores “balanced”); the only downside is that fetching elements from the shared input sequence requires synchronization (typically an exclusive lock)—and this can result in some overhead and contention.

Range partitioning bypasses the normal input-side enumeration and preallocates an equal number of elements to each worker, avoiding contention on the input sequence. But if some threads happen to get easy elements and finish early, they sit idle while the remaining threads continue working. Our earlier prime number calculator might perform poorly with range partitioning. An example of when range partitioning would do well is in calculating the sum of the square roots of the first 10 million integers:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

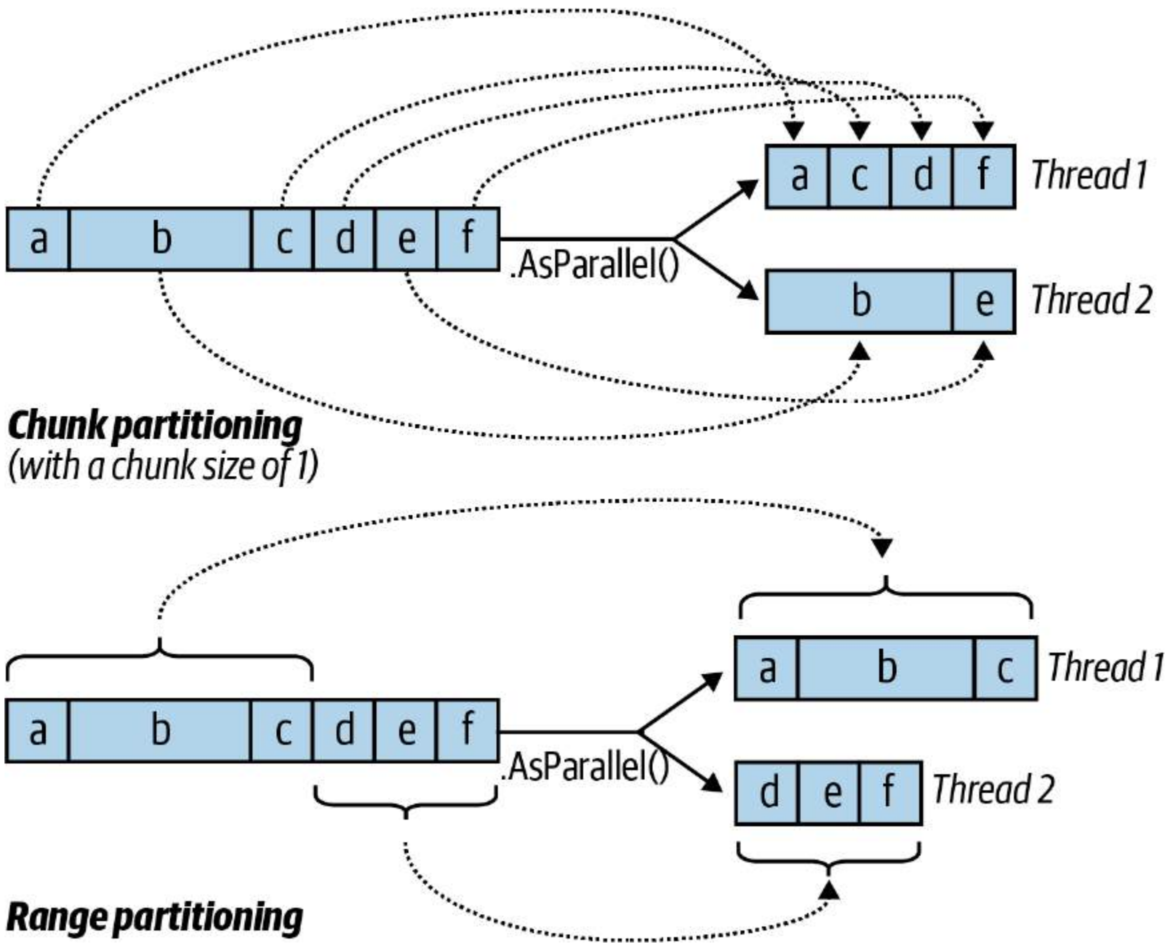

Figure 22-4. Chunk versus range partitioning

`ParallelEnumerable.Range` returns a `ParallelQuery<T>`, so you don't need to subsequently call `AsParallel`.

NOTE

Range partitioning doesn't necessarily allocate element ranges in *contiguous* blocks—it might instead choose a "striping" strategy. For instance, if there are two workers, one worker might process odd-numbered elements while the other processes even-numbered elements. The `TakeWhile` operator is almost certain to trigger a striping strategy to avoid unnecessarily processing elements later in the sequence.

Optimizing custom aggregations

PLINQ parallelizes the `Sum`, `Average`, `Min`, and `Max` operators efficiently without additional intervention. The `Aggregate` operator, though, presents special challenges for PLINQ. As described in Chapter 9, `Aggregate` performs custom aggregations. For example, the following sums a sequence of numbers, mimicking the `Sum` operator:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n);   // 6
```

We also saw in Chapter 9 that for *unseeded* aggregations, the supplied delegate must be associative and commutative. PLINQ will give incorrect results if this rule is violated, because it draws *multiple seeds* from the input sequence in order to aggregate several partitions of the sequence simultaneously.

Explicitly seeded aggregations might seem like a safe option with PLINQ, but unfortunately these ordinarily execute sequentially because of the reliance on a single seed. To mitigate this, PLINQ provides another overload of `Aggregate` that lets you specify multiple seeds—or rather, a *seed factory function*. For each thread, it executes this function to generate a separate seed, which becomes a *thread-local* accumulator into which it locally aggregates elements.

You must also supply a function to indicate how to combine the local and main accumulators. Finally, this `Aggregate` overload (somewhat gratuitously) expects a delegate to perform any final transformation on the result (you can achieve this as easily by running some function on the result yourself afterward). So, here are the four delegates, in the order they are passed:

`seedFactory`

Returns a new local accumulator

`updateAccumulatorFunc`

Aggregates an element into a local accumulator `combineAccumulatorFunc`

Combines a local accumulator with the main accumulator

`resultSelector`

Applies any final transformation on the end result

NOTE

In simple scenarios, you can specify a *seed value* instead of a seed factory. This tactic fails when the seed is a reference type that you want to mutate, because the same instance will then be shared by each thread.

To give a very simple example, the following sums the values in a `numbers` array:

```
numbers.AsParallel().Aggregate (
 () => 0,                                      // seedFactory
  (localTotal, n) => localTotal + n,           // updateAccumulatorFunc
  (mainTot, localTot) => mainTot + localTot,   // combineAccumulatorFunc
  finalResult => finalResult)                  // resultSelector
```

This example is contrived in that we could get the same answer just as efficiently using simpler approaches (such as an unseeded aggregate, or better, the `Sum` operator). To give a more realistic example, suppose that we want to calculate the frequency of each letter in the English alphabet in a given string. A simple sequential solution might look like this:

```
string text = "Let’s suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
  int index = char.ToUpper (c) - 'A';
  if (index >= 0 && index < 26) letterFrequencies [index]++;
};
```

NOTE

An example of when the input text might be very long is in gene sequencing. The “alphabet” would then consist of the letters *a*, *c*, *g*, and *t*.

To parallelize this, we could replace the `foreach` statement with a call to `Parallel.ForEach` (which we cover in the following section), but this will leave us to deal with concurrency issues on the shared array. And locking around accessing that array would all but kill the potential for parallelization.

`Aggregate` offers a tidy solution. The accumulator, in this case, is an array just like the `letterFrequencies` array in our preceding example. Here’s a sequential version using `Aggregate`:

```
int[] result =
  text.Aggregate (
    new int[26],                // Create the "accumulator"
    (letterFrequencies, c) =>   // Aggregate a letter into the accumulator
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index < 26) letterFrequencies [index]++;
      return letterFrequencies;
    });
```

And now the parallel version, using PLINQ’s special overload:

```
int[] result =
  text.AsParallel().Aggregate (
   () => new int[26],             // Create a new local accumulator

    (localFrequencies, c) =>       // Aggregate into the local accumulator
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index < 26) localFrequencies [index]++;
      return localFrequencies;
    },
                                   // Aggregate local->main accumulator
    (mainFreq, localFreq) =>
      mainFreq.Zip (localFreq, (f1, f2) => f1 + f2).ToArray(),
    finalResult => finalResult     // Perform any final transformation
  );                               // on the end result.
```

Notice that the local accumulation function *mutates* the `localFrequencies` array. This ability to perform this optimization is important—and is legitimate because `localFrequencies` is local to each thread.

The Parallel Class

PFX provides a basic form of structured parallelism via three static methods in the `Parallel` class:

`Parallel.Invoke`

Executes an array of delegates in parallel

`Parallel.For`

Performs the parallel equivalent of a C# `for` loop

`Parallel.ForEach`

Performs the parallel equivalent of a C# `foreach` loop

All three methods block until all work is complete. As with PLINQ, after an unhandled exception, remaining workers are stopped after their current iteration and the exception (or exceptions) are thrown back to the caller—wrapped in an `AggregateException` (see "Working with AggregateException").

Parallel.Invoke

`Parallel.Invoke` executes an array of `Action` delegates in parallel and then waits for them to complete. The simplest version of the method is defined as follows:

```
public static void Invoke (params Action[] actions);
```

Just as with PLINQ, the `Parallel`.* methods are optimized for compute-bound and not I/O-bound work. However, downloading two web pages at once provides a simple way to demonstrate `Parallel.Invoke`:

```
Parallel.Invoke (
 () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),
 () => new WebClient().DownloadFile ("http://microsoft.com", "ms.html"));
```

On the surface, this seems like a convenient shortcut for creating and waiting on two thread-bound `Task` objects. But there's an important difference: `Parallel.Invoke` still works efficiently if you pass in an array of a million delegates. This is because it *partitions* large numbers of elements into batches that it assigns to a handful of underlying `Task`s rather than creating a separate `Task` for each delegate.

As with all of `Parallel`'s methods, you're on your own when it comes to collating the results. This means that you need to keep thread safety in mind. The following, for instance, is thread-unsafe:

```
var data = new List<string>();
Parallel.Invoke (
 () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),
 () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

Locking around adding to the list would resolve this, although locking would create a bottleneck if you had a much larger array of quickly executing delegates. A better solution is to use the thread-safe collections, which we cover in later sections—`ConcurrentBag` would be ideal in this case.

`Parallel.Invoke` is also overloaded to accept a `ParallelOptions` object:

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

With `ParallelOptions`, you can insert a cancellation token, limit the maximum concurrency, and specify a custom task scheduler. A cancellation token is relevant when you're executing (roughly) more tasks than you have cores: upon cancellation, any unstarted delegates will be abandoned. Any already executing delegates will, however, continue to completion. See "Cancellation" for an example of how to use cancellation tokens.

Parallel.For and Parallel.ForEach

`Parallel.For` and `Parallel.ForEach` perform the equivalent of a C# `for` and `foreach` loop but with each iteration executing in parallel instead of sequentially. Here are their (simplest) signatures:

```
public static ParallelLoopResult For (
  int fromInclusive, int toExclusive, Action<int> body)

public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource> body)
```

This sequential `for` loop:

```
for (int i = 0; i < 100; i++)
  Foo (i);
```

is parallelized like this:

```
Parallel.For (0, 100, i => Foo (i));
```

or more simply:

```
Parallel.For (0, 100, Foo);
```

And this sequential `foreach`:

```
foreach (char c in "Hello, world")
  Foo (c);
```

is parallelized like this:

```
Parallel.ForEach ("Hello, world", Foo);
```

To give a practical example, if we import the `System.Security.Cryptography` namespace, we can generate six public/private keypair strings in parallel, as follows:

```
var keyPairs = new string[6];

Parallel.For (0, keyPairs.Length,
              i => keyPairs[i] = RSA.Create().ToXmlString (true));
```

As with `Parallel.Invoke`, we can feed `Parallel.For` and `Parallel.ForEach` a large number of work items and they'll be efficiently partitioned onto a few tasks.

NOTE

The latter query could also be done with PLINQ:

```
string[] keyPairs =
  ParallelEnumerable.Range (0, 6)
  .Select (i => RSA.Create().ToXmlString (true))
  .ToArray();
```

Outer versus inner loops

`Parallel.For` and `Parallel.ForEach` usually work best on outer rather than inner loops. This is because with the former, you're offering larger chunks of work to parallelize, diluting the management overhead. Parallelizing both inner and outer loops is usually unnecessary.

In the following example, we'd typically need more than 100 cores to benefit from the inner parallelization:

```
Parallel.For (0, 100, i =>
{
  Parallel.For (0, 50, j => Foo (i, j));   // Sequential would be better
});                                        // for the inner loop.
```

Indexed Parallel.ForEach

Sometimes, it's useful to know the loop iteration index. With a sequential `foreach`, it's easy:

```
int i = 0;
foreach (char c in "Hello, world")
  Console.WriteLine (c.ToString() + i++);
```

Incrementing a shared variable, however, is not thread-safe in a parallel context. You must instead use the following version of `ForEach`:

```
public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource,ParallelLoopState,long> body)
```

We'll ignore `ParallelLoopState` (which we cover in the following section). For now, we're interested in `Action`'s third type parameter of type `long`, which indicates the loop index:

```
Parallel.ForEach ("Hello, world", (c, state, i) =>
{
   Console.WriteLine (c.ToString() + i);
});
```

To put this into a practical context, let's revisit the spellchecker that we wrote with PLINQ. The following code loads up a dictionary along with an array of a million words to test:

```
if (!File.Exists ("WordLookup.txt"))    // Contains about 150,000 words
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Introduce a couple
wordsToTest [23456] = "wubsie";     // of spelling mistakes.
```

We can perform the spellcheck on our `wordsToTest` array using the indexed version of `Parallel.ForEach`, as follows:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

Notice that we had to collate the results into a thread-safe collection: having to do this is the disadvantage when compared to using PLINQ. The advantage over PLINQ is that we avoid the cost of applying an indexed `Select` query operator—which is less efficient than an indexed `ForEach`.

ParallelLoopState: breaking early out of loops

Because the loop body in a parallel `For` or `ForEach` is a delegate, you can't exit the loop early with a `break` statement. Instead, you must call `Break` or `Stop` on a `ParallelLoopState` object:

```
public class ParallelLoopState
{
  public void Break();
  public void Stop();

  public bool IsExceptional { get; }
  public bool IsStopped { get; }
  public long? LowestBreakIteration { get; }
  public bool ShouldExitCurrentIteration { get; }
}
```

Obtaining a `ParallelLoopState` is easy: all versions of `For` and `ForEach` are overloaded to accept loop bodies of type `Action<TSource,ParallelLoopState>`. So, to parallelize this:

```
foreach (char c in "Hello, world")
  if (c == ',')
    break;
  else
    Console.Write (c);
```

do this:

```
Parallel.ForEach ("Hello, world", (c, loopState) =>
{
  if (c == ',')
    loopState.Break();
  else
    Console.Write (c);
});

// OUTPUT: Hlloe
```

You can see from the output that loop bodies can complete in a random order. Aside from this difference, calling `Break` yields *at least* the same elements as executing the loop sequentially: this example will always output *at least* the letters *H*, *e*, *l*, *l*, and *o* in some order. In contrast, calling `Stop` instead of `Break` forces all threads to finish immediately after their current iteration. In our example, calling `Stop` could give us a subset of the letters *H*, *e*, *l*, *l*, and *o* if another thread were lagging behind. Calling `Stop` is useful when you've found something that you're looking for—or when something has gone wrong and you won't be looking at the results.

NOTE

The `Parallel.For` and `Parallel.ForEach` methods return a `ParallelLoopResult` object that exposes properties called `IsCompleted` and `LowestBreakIteration`. These tell you whether the loop ran to completion; if it didn't, it indicates at what cycle the loop was broken.

If `LowestBreakIteration` returns null, it means that you called `Stop` (rather than `Break`) on the loop.

If your loop body is long, you might want other threads to break partway through the method body in case of an early `Break` or `Stop`. You can do this by polling the `ShouldExitCurrentIteration` property at various places in your code; this property becomes true immediately after a `Stop`—or soon after a `Break`.

NOTE

`ShouldExitCurrentIteration` also becomes true after a cancellation request—or if an exception is thrown in the loop.

`IsExceptional` lets you know whether an exception has occurred on another thread. Any unhandled exception will cause the loop to stop after each thread's current iteration: to avoid this, you must explicitly handle exceptions in your code.

Optimization with local values

`Parallel.For` and `Parallel.ForEach` each offer a set of overloads that feature a generic type argument called `TLocal`. These overloads are designed to help you optimize the collation of data with iteration-intensive loops. The simplest is this:

```
public static ParallelLoopResult For <TLocal> (
  int fromInclusive,
  int toExclusive,
  Func <TLocal> localInit,
  Func <int, ParallelLoopState, TLocal, TLocal> body,
  Action <TLocal> localFinally);
```

These methods are rarely needed in practice because their target scenarios are covered mostly by PLINQ (which is fortunate because these overloads are somewhat intimidating!).

Essentially, the problem is this: suppose that we want to sum the square roots of the numbers 1 through 10,000,000. Calculating 10 million square roots is easily parallelizable, but summing their values is troublesome because we must lock around updating the total:

```
object locker = new object();
double total = 0;
Parallel.For (1, 10000000,
              i => { lock (locker) total += Math.Sqrt (i); });
```

The gain from parallelization is more than offset by the cost of obtaining 10 million locks—plus the resultant blocking.

The reality, though, is that we don't actually *need* 10 million locks. Imagine a team of volunteers picking up a large volume of litter. If all workers shared a single trash can, the travel and contention would make the process extremely inefficient. The obvious solution is for each worker to have a private or "local" trash can, which is occasionally emptied into the main bin.

The `TLocal` versions of `For` and `ForEach` work in exactly this way. The volunteers are internal worker threads, and the *local value* represents a local trash can. For `Parallel` to do this job, you must feed it two additional delegates that indicate the following:

1. How to initialize a new local value
2. How to combine a local aggregation with the master value

Additionally, instead of the body delegate returning `void`, it should return the new aggregate for the local value. Here's our example refactored:

```
object locker = new object();
double grandTotal = 0;

Parallel.For (1, 10000000,

  () => 0.0,                        // Initialize the local value.

  (i, state, localTotal) =>         // Body delegate. Notice that it
     localTotal + Math.Sqrt (i),    // returns the new local total.

  localTotal =>                                   // Add the local value
    { lock (locker) grandTotal += localTotal; }   // to the master value.
);
```

We must still lock, but only around aggregating the local value to the grand total. This makes the process dramatically more efficient.

NOTE

As stated earlier, PLINQ is often a good fit in these scenarios. Our example could be parallelized with PLINQ like this:

```
ParallelEnumerable.Range (1, 10000000)
                  .Sum (i => Math.Sqrt (i))
```

(Notice that we used `ParallelEnumerable` to force *range partitioning*: this improves performance in this case because all numbers will take equally long to process.)

In more complex scenarios, you might use LINQ's `Aggregate` operator instead of `Sum`. If you supplied a local seed factory, the situation would be somewhat analogous to providing a local value function with `Parallel.For`.

Task Parallelism

Task parallelism is the lowest-level approach to parallelization with PFX. The classes for working at this level are defined in the `System.Threading.Tasks` namespace and comprise the following:

Class	Purpose
`Task`	For managing a unit for work
`Task<TResult>`	For managing a unit for work with a return value
`TaskFactory`	For creating tasks
`TaskFactory<TResult>`	For creating tasks and continuations with the same return type
`TaskScheduler`	For managing the scheduling of tasks
`TaskCompletionSource`	For manually controlling a task's workflow

We covered the basics of tasks in Chapter 14; in this section, we look at advanced features of tasks that are aimed at parallel programming:

- Tuning a task's scheduling
- Establish a parent/child relationship when one task is started from another
- Advanced use of continuations
- `TaskFactory`

WARNING

The Task Parallel Library lets you create hundreds (or even thousands) of tasks with minimal overhead. But if you want to create millions of tasks, you'll need to partition those tasks into larger work units to maintain efficiency. The `Parallel` class and PLINQ do this automatically.

NOTE

Visual Studio provides a window for monitoring tasks (Debug®Window®Parallel Tasks). This is equivalent to the Threads window, but for tasks. The Parallel Stacks window also has a special mode for tasks.

Creating and Starting Tasks

As described in Chapter 14, `Task.Run` creates and starts a `Task` or `Task<TResult>`. This method is actually a shortcut for calling `Task.Factory.StartNew`, which allows greater flexibility through additional overloads.

Specifying a state object

`Task.Factory.StartNew` lets you specify a *state* object that is passed to the target. The target method's signature must then comprise a single object-type parameter:

```
var task = Task.Factory.StartNew (Greet, "Hello");
task.Wait();  // Wait for task to complete.

void Greet (object state) { Console.Write (state); }   // Hello
```

This avoids the cost of the closure required for executing a lambda expression that calls `Greet`. This is a micro-optimization and is rarely necessary in practice, so we can put the *state* object to better use, which is to assign a meaningful name to the task. We can then use the `AsyncState` property to query its name:

```
var task = Task.Factory.StartNew (state => Greet ("Hello"), "Greeting");
Console.WriteLine (task.AsyncState);   // Greeting
task.Wait();

void Greet (string message) { Console.Write (message); }
```

NOTE

Visual Studio displays each task's `AsyncState` in the Parallel Tasks window, so having a meaningful name here can ease debugging considerably.

TaskCreationOptions

You can tune a task's execution by specifying a `TaskCreationOptions` enum when calling `StartNew` (or instantiating a `Task`). `TaskCreationOptions` is a flags enum with the following (combinable) values:

```
LongRunning, PreferFairness, AttachedToParent
```

`LongRunning` suggests to the scheduler to dedicate a thread to the task, and as we described in Chapter 14, this is beneficial for I/O-bound tasks and for long-running tasks that might otherwise force short-running tasks to wait an unreasonable amount of time before being scheduled.

`PreferFairness` instructs the scheduler to try to ensure that tasks are scheduled in the order in which they were started. It might ordinarily do otherwise because it internally optimizes the scheduling of tasks using local work-stealing queues—an optimization that allows the creation of *child* tasks without incurring the contention overhead that would otherwise arise with a single work queue. A child task is created by specifying `AttachedToParent`.

Child tasks

When one task starts another, you can optionally establish a parent-child relationship:

```
Task parent = Task.Factory.StartNew (() =>
{
  Console.WriteLine ("I am a parent");

  Task.Factory.StartNew (() =>        // Detached task
  {
    Console.WriteLine ("I am detached");
  });

  Task.Factory.StartNew (() =>        // Child task
  {
    Console.WriteLine ("I am a child");
  }, TaskCreationOptions.AttachedToParent);
});
```

A child task is special in that when you wait for the *parent* task to complete, it waits for any children, as well. At which point any child exceptions bubble up:

```
  TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
  var parent = Task.Factory.StartNew (() =>
  {
    Task.Factory.StartNew (() =>   // Child
    {
      Task.Factory.StartNew (() => { throw null; }, atp);   // Grandchild
    }, atp);
  });

  // The following call throws a NullReferenceException (wrapped
  // in nested AggregateExceptions):
  parent.Wait();
```

This can be particularly useful when a child task is a continuation, as you'll see shortly.

Waiting on Multiple Tasks

We saw in Chapter 14 that you can wait on a single task either by calling its `Wait` method or by accessing its `Result` property (if it's a `Task<TResult>`). You can also wait on multiple tasks at once—via the static methods `Task.WaitAll` (waits for all the specified tasks to finish) and `Task.WaitAny` (waits for just one task to finish).

`WaitAll` is similar to waiting out each task in turn, but is more efficient in that it requires (at most) just one context switch. Also, if one or more of the tasks throw an unhandled exception, `WaitAll` still waits out every task. It then rethrows an `AggregateException` that accumulates the exceptions from each faulted task (this is where `AggregateException` is genuinely useful). It's equivalent to doing this:

```
// Assume t1, t2 and t3 are tasks:
var exceptions = new List<Exception>();
try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
if (exceptions.Count > 0) throw new AggregateException (exceptions);
```

Calling `WaitAny` is equivalent to waiting on a `ManualResetEventSlim` that's signaled by each task as it finishes.

As well as a timeout, you can also pass in a *cancellation token* to the `Wait` methods: this lets you cancel the wait—*not the task itself*.

Canceling Tasks

You can optionally pass in a cancellation token when starting a task. Then, if cancellation occurs via that token, the task itself enters the "Canceled" state:

```
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
cts.CancelAfter (500);

Task task = Task.Factory.StartNew (() =>
{
  Thread.Sleep (1000);
  token.ThrowIfCancellationRequested();  // Check for cancellation request
}, token);

try { task.Wait(); }
catch (AggregateException ex)
{
  Console.WriteLine (ex.InnerException is TaskCanceledException);  // True
  Console.WriteLine (task.IsCanceled);                             // True
  Console.WriteLine (task.Status);                             // Canceled
}
```

`TaskCanceledException` is a subclass of `OperationCanceledException`. If you want to explicitly throw an `OperationCanceledException` (rather than calling `token.ThrowIfCancellationRequested`), you must pass the cancellation token into `OperationCanceledException`'s constructor. If you fail to do this, the task won't end up with a `TaskStatus.Canceled` status and won't trigger `OnlyOnCanceled` continuations.

If the task is canceled before it has started, it won't get scheduled—an `OperationCanceledException` will instead be thrown on the task immediately.

Because cancellation tokens are recognized by other APIs, you can pass them into other constructs and cancellations will propagate seamlessly:

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;

Task task = Task.Factory.StartNew (() =>
{
  // Pass our cancellation token into a PLINQ query:
  var query = someSequence.AsParallel().WithCancellation (token)...
  ... enumerate query ...
});
```

Calling `Cancel` on `cancelSource` in this example will cancel the PLINQ query, which will throw an `OperationCanceledException` on the task body, which will then cancel the task.

NOTE

The cancellation tokens that you can pass into methods such as `Wait` and `CancelAndWait` allow you to cancel the *wait* operation and not the task itself.

Continuations

The `ContinueWith` method executes a delegate immediately after a task ends:

```
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedent.."));
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

As soon as `task1` (the *antecedent*) completes, fails, or is canceled, `task2` (the *continuation*) starts. (If `task1` had completed before the second line of code ran, `task2` would be scheduled to execute immediately.) The `ant` argument passed to the continuation's lambda expression is a reference to the antecedent task. `ContinueWith` itself returns a task, making it easy to add further continuations.

By default, antecedent and continuation tasks may execute on different threads. You can force them to execute on the same thread by specifying `TaskContinuationOptions.ExecuteSynchronously` when calling `ContinueWith`: this can improve performance in very fine-grained continuations by lessening indirection.

Continuations and Task<TResult>

Just like ordinary tasks, continuations can be of type `Task<TResult>` and return data. In the following example, we calculate `Math.Sqrt(8*2)` using a series of chained tasks and then write out the result:

```
Task.Factory.StartNew<int> (() => 8)
  .ContinueWith (ant => ant.Result * 2)
  .ContinueWith (ant => Math.Sqrt (ant.Result))
  .ContinueWith (ant => Console.WriteLine (ant.Result));   // 4
```

Our example is somewhat contrived for simplicity; in real life, these lambda expressions would call computationally intensive functions.

Continuations and exceptions

A continuation can know whether an antecedent faulted by querying the antecedent task's `Exception` property—or simply by invoking `Result` / `Wait` and catching the resultant `AggregateException`. If an antecedent faults and the continuation does neither, the exception is considered *unobserved* and the static `TaskScheduler.UnobservedTaskException` event fires when the task is later garbage-collected.

A safe pattern is to rethrow antecedent exceptions. As long as the continuation is `Wait`ed upon, the exception will be propagated and rethrown to the `Waiter`:

```
Task continuation = Task.Factory.StartNew     (()  => { throw null; })
                                .ContinueWith (ant =>
  {
    ant.Wait();
    // Continue processing...
  });

continuation.Wait();    // Exception is now thrown back to caller.
```

Another way to deal with exceptions is to specify different continuations for exceptional versus nonexceptional outcomes. This is done with `TaskContinuationOptions`:

```
Task task1 = Task.Factory.StartNew (() => { throw null; });

Task error = task1.ContinueWith (ant => Console.Write (ant.Exception),
                                 TaskContinuationOptions.OnlyOnFaulted);

Task ok = task1.ContinueWith (ant => Console.Write ("Success!"),
                              TaskContinuationOptions.NotOnFaulted);
```

This pattern is particularly useful in conjunction with child tasks, as you'll see very soon.

The following extension method "swallows" a task's unhandled exceptions:

```
public static void IgnoreExceptions (this Task task)
{
  task.ContinueWith (t => { var ignore = t.Exception; },
    TaskContinuationOptions.OnlyOnFaulted);
}
```

(This could be improved by adding code to log the exception.) Here's how it would be used:

```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

Continuations and child tasks

A powerful feature of continuations is that they kick off only when all child tasks have completed (see Figure 22-5). At that point, any exceptions thrown by the children are marshaled to the continuation.

In the following example, we start three child tasks, each throwing a `NullReferenceException`. We then catch all of them in one fell swoop via a continuation on the parent:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
})
.ContinueWith (p => Console.WriteLine (p.Exception),
                    TaskContinuationOptions.OnlyOnFaulted);
```

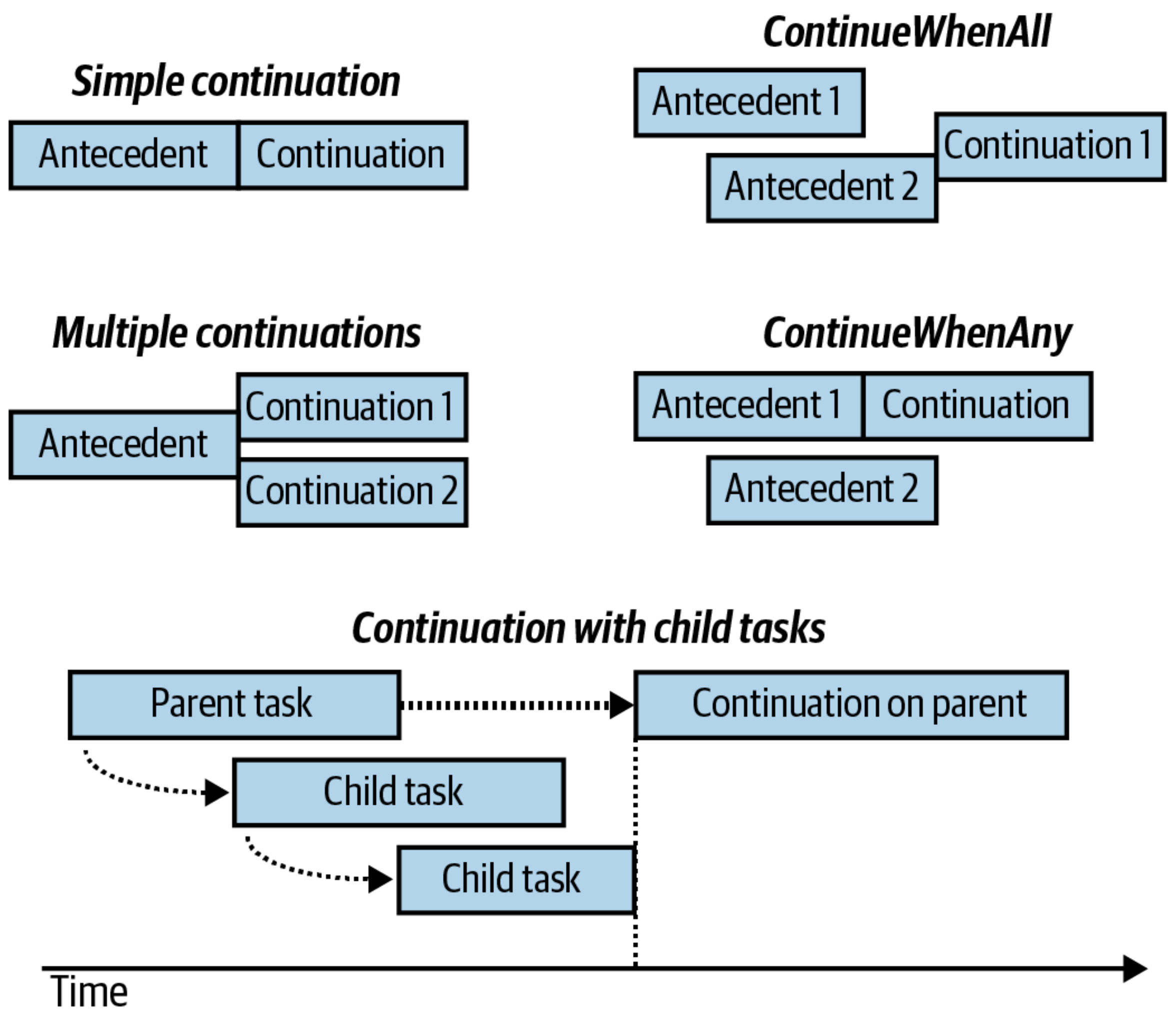

Figure 22-5. Continuations

Conditional continuations

By default, a continuation is scheduled *unconditionally*, whether the antecedent completes, throws an exception, or is canceled. You can alter this behavior via a set of (combinable) flags included within the `TaskContinuationOptions` enum. Following are the three core flags that control conditional continuation:

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

These flags are subtractive in the sense that the more you apply, the less likely the continuation is to execute. For convenience, there are also the following precombined values:

```
OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

(Combining all the `Not*` flags [`NotOnRanToCompletion`, `NotOnFaulted`, `NotOnCanceled`] is nonsensical because it would result in the continuation always being canceled.)

"RanToCompletion" means that the antecedent succeeded without cancellation or unhandled exceptions.

"Faulted" means that an unhandled exception was thrown on the antecedent.

"Canceled" means one of two things:

- The antecedent was canceled via its cancellation token. In other words, an `OperationCanceledException` was thrown on the antecedent, whose `CancellationToken` property matched that passed to the antecedent when it was started.
- The antecedent was implicitly canceled because *it* didn't satisfy a conditional continuation predicate.

It's essential to grasp that when a continuation doesn't execute by virtue of these flags, the continuation is not forgotten or abandoned—it's canceled. This means that any continuations on the continuation itself *will then run* unless you predicate them with `NotOnCanceled`. For example, consider this:

```
Task t1 = Task.Factory.StartNew (...);

Task fault = t1.ContinueWith (ant => Console.WriteLine ("fault"),
                              TaskContinuationOptions.OnlyOnFaulted);
```

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"));
```

As it stands, `t3` will always get scheduled—even if `t1` doesn't throw an exception (see Figure 22-6). This is because if `t1` succeeds, the `fault` task will be canceled, and with no continuation restrictions placed on `t3`, `t3` will then execute unconditionally.

If we want `t3` to execute only if `fault` actually runs, we must instead do this:

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),
                              TaskContinuationOptions.NotOnCanceled);
```

(Alternatively, we could specify `OnlyOnRanToCompletion`; the difference is that `t3` would not then execute if an exception were thrown within `fault`.)

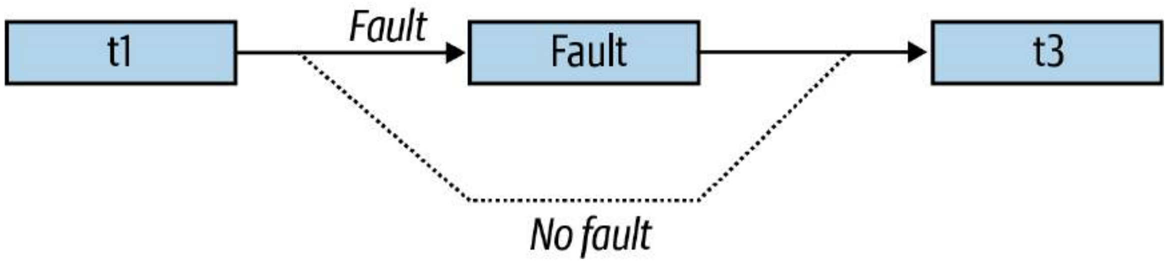

Figure 22-6. Conditional continuations

Continuations with multiple antecedents

You can schedule continuation to execute based on the completion of multiple antecedents with the `ContinueWhenAll` and `ContinueWhenAny` methods in the `TaskFactory` class. These methods have become redundant, however, with the introduction of the task combinators discussed in Chapter 14 (`WhenAll` and `WhenAny`). Specifically, given the following tasks:

```
var task1 = Task.Run (() => Console.Write ("X"));
var task2 = Task.Run (() => Console.Write ("Y"));
```

we can schedule a continuation to execute when both complete as follows:

```
var continuation = Task.Factory.ContinueWhenAll (
  new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));
```

Here's the same result with the `WhenAll` task combinator:

```
var continuation = Task.WhenAll (task1, task2)
                       .ContinueWith (ant => Console.WriteLine ("Done"));
```

Multiple continuations on a single antecedent

Calling `ContinueWith` more than once on the same task creates multiple continuations on a single antecedent. When the antecedent finishes, all continuations will start together (unless you specify `TaskContinuationOptions.ExecuteSynchronously`, in which case the continuations will execute sequentially).

The following waits for one second and then writes either `XY` or `YX`:

```
var t = Task.Factory.StartNew (() => Thread.Sleep (1000));
t.ContinueWith (ant => Console.Write ("X"));
t.ContinueWith (ant => Console.Write ("Y"));
```

Task Schedulers

A *task scheduler* allocates tasks to threads and is represented by the abstract `TaskScheduler` class. .NET provides two concrete implementations: the *default scheduler* that works in tandem with the CLR thread pool, and the *synchronization context scheduler*. The latter is designed (primarily) to help you with the threading model of WPF and Windows Forms, which requires that user interface elements and controls are accessed only from the thread that created them (see "Threading in Rich Client Applications"). By capturing it, we can instruct a task or a continuation to execute on this context:

```
// Suppose we are on a UI thread in a Windows Forms / WPF application:
_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

Assuming `Foo` is a compute-bound method that returns a string and `lblResult` is a WPF or Windows Forms label, we could then safely update the label after the operation completes, as follows:

```
Task.Run (() => Foo())
  .ContinueWith (ant => lblResult.Content = ant.Result, _uiScheduler);
```

Of course, C#'s asynchronous functions would more commonly be used for this kind of thing.

It's also possible to write our own task scheduler (by subclassing `TaskScheduler`), although this is something you'd do only in very specialized scenarios. For custom scheduling, you'd more commonly use `TaskCompletionSource`.

TaskFactory

When you call `Task.Factory`, you're calling a static property on `Task` that returns a default `TaskFactory` object. The purpose of a task factory is to create tasks; specifically, three kinds of tasks:

- "Ordinary" tasks (via `StartNew`)
- Continuations with multiple antecedents (via `ContinueWhenAll` and `ContinueWhenAny`)
- Tasks that wrap methods that follow the defunct APM (via `FromAsync`; see "Obsolete Patterns")

Another way to create tasks is to instantiate `Task` and call `Start`. However, this lets you create only "ordinary" tasks, not continuations.

Creating your own task factories

`TaskFactory` is not an *abstract* factory: you can actually instantiate the class, and this is useful when you want to repeatedly create tasks using the same (nonstandard) values for `TaskCreationOptions`, `TaskContinuationOptions`, or `TaskScheduler`. For example, if we want to repeatedly create long-running *parented* tasks, we could create a custom factory, as follows:

```
var factory = new TaskFactory (
  TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
  TaskContinuationOptions.None);
```

Creating tasks is then simply a matter of calling `StartNew` on the factory:

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

The custom continuation options are applied when calling `ContinueWhenAll` and `ContinueWhenAny`.

Working with AggregateException

As we've seen, PLINQ, the `Parallel` class, and `Tasks` automatically marshal exceptions to the consumer. To see why this is essential, consider the following LINQ query, which throws a `DivideByZeroException` on the first iteration:

```
try
{
  var query = from i in Enumerable.Range (0, 1000000)
              select 100 / i;
  ...
}
catch (DivideByZeroException)
{
  ...
}
```

If we asked PLINQ to parallelize this query and it ignored the handling of exceptions, a `DivideByZeroException` would probably be thrown on a *separate thread*, bypassing our `catch` block and causing the application to die.

Hence, exceptions are automatically caught and rethrown to the caller. But unfortunately, it's not quite as simple as catching a `DivideByZeroException`. Because these libraries utilize many threads, it's actually possible for two or more exceptions to be thrown simultaneously. To ensure that all exceptions are reported, exceptions are therefore wrapped in an `AggregateException` container, which exposes an `InnerExceptions` property containing each of the caught exception(s):

```
try
{
  var query = from i in ParallelEnumerable.Range (0, 1000000)
              select 100 / i;
  // Enumerate query
  ...
}
catch (AggregateException aex)
{
  foreach (Exception ex in aex.InnerExceptions)
    Console.WriteLine (ex.Message);
}
```

NOTE

Both PLINQ and the `Parallel` class end the query or loop execution upon encountering the first exception—by not processing any further elements or loop bodies. More exceptions might be thrown, however, before the current cycle is complete. The first exception in `AggregateException` is visible in the `InnerException` property.

Flatten and Handle

The `AggregateException` class provides a couple of methods to simplify exception handling: `Flatten` and `Handle`.

Flatten

`AggregateException`s will quite often contain other `AggregateException`s. An example of when this might happen is if a child task throws an exception. You can eliminate any level of nesting to simplify handling by calling `Flatten`. This method returns a new `AggregateException` with a simple flat list of inner exceptions:

```
catch (AggregateException aex)
{
  foreach (Exception ex in aex.Flatten().InnerExceptions)
    myLogWriter.LogException (ex);
}
```

Handle

Sometimes, it's useful to catch only specific exception types, and have other types rethrown. The `Handle` method on `AggregateException` provides a shortcut for doing this. It accepts an exception predicate which it runs over every inner exception:

```
public void Handle (Func<Exception, bool> predicate)
```

If the predicate returns `true`, it considers that exception "handled." After the delegate has run over every exception, the following happens:

- If all exceptions were "handled" (the delegate returned `true`), the exception is not rethrown.
- If there were any exceptions for which the delegate returned `false` ("unhandled"), a new `AggregateException` is built up containing those exceptions and is rethrown.

For instance, the following ends up rethrowing another `AggregateException` that contains a single `NullReferenceException`:

```
var parent = Task.Factory.StartNew (() =>
{
  // We'll throw 3 exceptions at once using 3 child tasks:
  int[] numbers = { 0 };

  var childFactory = new TaskFactory
   (TaskCreationOptions.AttachedToParent, TaskContinuationOptions.None);

  childFactory.StartNew (() => 5 / numbers[0]);   // Division by zero
  childFactory.StartNew (() => numbers [1]);      // Index out of range
  childFactory.StartNew (() => { throw null; });  // Null reference
});

try { parent.Wait(); }
catch (AggregateException aex)
{
  aex.Flatten().Handle (ex =>   // Note that we still need to call Flatten
  {
    if (ex is DivideByZeroException)
    {
      Console.WriteLine ("Divide by zero");
      return true;                           // This exception is "handled"
    }
    if (ex is IndexOutOfRangeException)
    {
      Console.WriteLine ("Index out of range");
      return true;                           // This exception is "handled"
    }
    return false;    // All other exceptions will get rethrown
  });
}
```

Concurrent Collections

.NET offers thread-safe collections in the `System.Collections.Concurrent` namespace:

Concurrent collection	Nonconcurrent equivalent
`ConcurrentStack<T>`	`Stack<T>`
`ConcurrentQueue<T>`	`Queue<T>`
`ConcurrentBag<T>`	(none)
`ConcurrentDictionary<TKey,TValue>`	`Dictionary<TKey,TValue>`

The concurrent collections are optimized for high-concurrency scenarios; however, they can also be useful whenever you need a thread-safe collection (as an alternative to locking around an ordinary collection). There are some caveats, though:

- The conventional collections outperform the concurrent collections in all but highly concurrent scenarios.
- A thread-safe collection doesn't guarantee that the code using it will be thread-safe (see "Locking and Thread Safety").
- If you enumerate over a concurrent collection while another thread is modifying it, no exception is thrown—instead, you get a mixture of old and new content.
- There's no concurrent version of `List<T>`.
- The concurrent stack, queue, and bag classes are implemented internally with linked lists. This makes them less memory-efficient than the nonconcurrent `Stack` and `Queue` classes, but better for concurrent access because linked lists are conducive to lock-free or low-lock implementations. (This is because inserting a node into a linked list requires updating just a couple of references, whereas inserting an element into a `List<T>`-like structure might require moving thousands of existing elements.)

In other words, these collections are not merely shortcuts for using an ordinary collection with a lock. To demonstrate, if we execute the following code on a *single* thread:

```
var d = new ConcurrentDictionary<int,int>();
for (int i = 0; i < 1000000; i++) d[i] = 123;
```

it runs three times more slowly than this:

```
var d = new Dictionary<int,int>();
for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;
```

(*Reading* from a `ConcurrentDictionary`, however, is fast because reads are lock-free.)

The concurrent collections also differ from conventional collections in that they expose special methods to perform atomic test-and-act operations, such as `TryPop`. Most of these methods are unified via the `IProducerConsumerCollection<T>` interface.

IProducerConsumerCollection<T>

A producer/consumer collection is one for which the two primary use cases are:

- Adding an element (“producing”)
- Retrieving an element while removing it (“consuming”)

The classic examples are stacks and queues. Producer/consumer collections are significant in parallel programming because they’re conducive to efficient lock-free implementations.

The `IProducerConsumerCollection<T>` interface represents a thread-safe producer/consumer collection. The following classes implement this interface:

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

`IProducerConsumerCollection<T>` extends `ICollection`, adding the following methods:

```
void CopyTo (T[] array, int index);
T[] ToArray();
bool TryAdd (T item);
bool TryTake (out T item);
```

The `TryAdd` and `TryTake` methods test whether an add/remove operation can be performed; if so, they perform the add/remove. The testing and acting are atomically performed, eliminating the need to lock as you would around a conventional collection:

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

`TryTake` returns `false` if the collection is empty. `TryAdd` always succeeds and returns `true` in the three implementations provided. If you wrote your own concurrent collection that prohibited duplicates, however, you'd make `TryAdd` return `false` if the element already existed (an example would be if you wrote a concurrent *set*).

The particular element that `TryTake` removes is defined by the subclass:

- With a stack, `TryTake` removes the most recently added element.
- With a queue, `TryTake` removes the least recently added element.
- With a bag, `TryTake` removes whatever element it can remove most efficiently.

The three concrete classes mostly implement the `TryTake` and `TryAdd` methods explicitly, exposing the same functionality through more specifically named public methods such as `TryDequeue` and `TryPop`.

ConcurrentBag<T>

`ConcurrentBag<T>` stores an *unordered* collection of objects (with duplicates permitted). `ConcurrentBag<T>` is suitable in situations for which you *don't care* which element you get when calling `Take` or `TryTake`.

The benefit of `ConcurrentBag<T>` over a concurrent queue or stack is that a bag's `Add` method suffers almost *no* contention when called by many threads at once. In contrast, calling `Add` in parallel on a queue or stack incurs *some* contention (although a lot less than locking around a *nonconcurrent* collection). Calling `Take` on a concurrent bag is also very efficient—as long as each thread doesn't take more elements than it `Add`ed.

Inside a concurrent bag, each thread gets its own private linked list. Elements are added to the private list that belongs to the thread calling `Add`, eliminating contention. When you enumerate over the bag, the enumerator travels through each thread's private list, yielding each of its elements in turn.

When you call `Take`, the bag first looks at the current thread's private list. If there's at least one element,[1] it can complete the task easily and without contention. But if the list is empty, it must "steal" an element from another thread's private list and incur the potential for contention.

So, to be precise, calling `Take` gives you the element added most recently on that thread; if there are no elements on that thread, it gives you the element added most recently on another thread, chosen at random.

Concurrent bags are ideal when the parallel operation on your collection mostly comprises `Add`ing elements—or when the `Add`s and `Take`s are balanced on a thread. We saw an example of the former previously, when using `Parallel.ForEach` to implement a parallel spellchecker:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
```

```
      misspellings.Add (Tuple.Create ((int) i, word));
  });
```

A concurrent bag would be a poor choice for a producer/consumer queue because elements are added and removed by *different* threads.

BlockingCollection<T>

If you call `TryTake` on any of the producer/consumer collections we discussed in the previous section, `ConcurrentStack<T>`, `ConcurrentQueue<T>`, and `ConcurrentBag<T>`, and the collection is empty, the method returns `false`. Sometimes, it would be more useful in this scenario to *wait* until an element is available.

Rather than overloading the `TryTake` methods with this functionality (which would have caused a blowout of members after allowing for cancellation tokens and timeouts), PFX's designers encapsulated this functionality into a wrapper class called `BlockingCollection<T>`. A blocking collection wraps any collection that implements `IProducerConsumerCollection<T>` and lets you `Take` an element from the wrapped collection—blocking if no element is available.

A blocking collection also lets you limit the total size of the collection, blocking the *producer* if that size is exceeded. A collection limited in this manner is called a *bounded blocking collection*.

To use `BlockingCollection<T>`:

1. Instantiate the class, optionally specifying the `IProducerConsumerCollection<T>` to wrap and the maximum size (bound) of the collection.

2. Call `Add` or `TryAdd` to add elements to the underlying collection.

3. Call `Take` or `TryTake` to remove (consume) elements from the underlying collection.

If you call the constructor without passing in a collection, the class will automatically instantiate a `ConcurrentQueue<T>`. The producing and consuming methods let you specify cancellation tokens and timeouts. `Add` and `TryAdd` may block if the collection size is bounded; `Take` and `TryTake` block while the collection is empty.

Another way to consume elements is to call `GetConsumingEnumerable`. This returns a (potentially) infinite sequence that yields elements as they become available. You can force the sequence to end by calling `CompleteAdding`: this method also prevents further elements from being enqueued.

`BlockingCollection` also provides static methods called `AddToAny` and `TakeFromAny`, which let you add or take an element while specifying several blocking collections. The action is then honored by the first collection able to service the request.

Writing a Producer/Consumer Queue

A producer/consumer queue is a useful structure, both in parallel programming and general concurrency scenarios. Here's how it works:

- A queue is set up to describe work items—or data upon which work is performed.
- When a task needs executing, it's enqueued, and the caller gets on with other things.
- One or more worker threads plug away in the background, picking off and executing queued items.

A producer/consumer queue gives you precise control over how many worker threads execute at once, which is useful not only in limiting CPU consumption but other resources, as well. If the tasks perform intensive disk I/O, for instance, you can limit concurrency to avoid starving the operating system and other applications. You can also dynamically add and remove workers throughout the queue's life. The CLR's thread pool itself is a kind of producer/consumer queue, optimized for short-running compute-bound jobs.

A producer/consumer queue typically holds items of data upon which (the same) task is performed. For example, the items of data may be filenames, and the task might be to encrypt those files. By making the item a delegate, however, you can write a more general-purpose producer/consumer queue where each item can do anything.

At *http://albahari.com/threading*, we show how to write a producer/consumer queue from scratch using an `AutoResetEvent` (and later, using `Monitor`'s `Wait` and `Pulse`). However, writing a producer/consumer from scratch is unnecessary because most of the functionality is provided by `BlockingCollection<T>`. Here's how to use it:

```
public class PCQueue : IDisposable
{
  BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();

  public PCQueue (int workerCount)
  {
    // Create and start a separate Task for each consumer:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public void Enqueue (Action action) { _taskQ.Add (action); }

  void Consume()
  {
    // This sequence that we're enumerating will block when no elements
    // are available and will end when CompleteAdding is called.

    foreach (Action action in _taskQ.GetConsumingEnumerable())
      action();     // Perform task.
  }

  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

Because we didn't pass anything into `BlockingCollection`'s constructor, it instantiated a concurrent queue automatically. Had we passed in a `ConcurrentStack`, we'd have ended up with a producer/consumer stack.

Using Tasks

The producer/consumer that we just wrote is inflexible in that we can't track work items after they've been enqueued. It would be nice if we could do the following:

- Know when a work item has completed (and `await` it)
- Cancel a work item
- Deal elegantly with any exceptions thrown by a work item

An ideal solution would be to have the `Enqueue` method return some object giving us the functionality just described. The good news is that a class already exists to do exactly this—the `Task` class, which we can generate either with a `TaskCompletionSource` or by instantiating directly (creating an unstarted or *cold* task):

```
public class PCQueue : IDisposable
{
  BlockingCollection<Task> _taskQ = new BlockingCollection<Task>();

  public PCQueue (int workerCount)
  {
    // Create and start a separate Task for each consumer:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public Task Enqueue (Action action, CancellationToken cancelToken
                                          = default (CancellationToken))
  {
    var task = new Task (action, cancelToken);
    _taskQ.Add (task);
    return task;
  }

  public Task<TResult> Enqueue<TResult> (Func<TResult> func,
            CancellationToken cancelToken = default (CancellationToken))
  {
    var task = new Task<TResult> (func, cancelToken);
    _taskQ.Add (task);
    return task;
  }

  void Consume()
  {
    foreach (var task in _taskQ.GetConsumingEnumerable())
      try
      {
          if (!task.IsCanceled) task.RunSynchronously();
      }
      catch (InvalidOperationException) { }  // Race condition
  }

  public void Dispose() { _taskQ.CompleteAdding(); }
}
```

In `Enqueue`, we enqueue and return to the caller a task that we create but don’t start.

In `Consume`, we run the task synchronously on the consumer’s thread. We catch an `InvalidOperationException` to handle the unlikely event that the task is canceled in between checking whether it’s canceled and running it.

Here’s how we can use this class:

```
var pcQ = new PCQueue (2);    // Maximum concurrency of 2
string result = await pcQ.Enqueue (() => "That was easy!");
...
```

Hence, we have all the benefits of tasks—with exception propagation, return values, and cancellation—while taking complete control over scheduling.

---

[1] Due to an implementation detail, there actually needs to be at least two elements to avoid contention entirely.