When Should I Use Parallel.ForEach? When Should I Use PLINQ?

Pamela Vagata
Parallel Computing Platform Group
Microsoft Corporation

Contents

Introduction 2
- Parallel.ForEach 2
- PLINQ 2

Simple Data-Parallel Operation with Independent Actions 3
- Using Parallel.ForEach for Performing Independent Actions 3
- Why not PLINQ? 3

Ordered Data-Parallel Operation 3
- Using PLINQ for Order Preservation 3
- Why not Parallel.ForEach? 4

Streaming Data-Parallel Operation 5
- Using PLINQ for Stream Processing 5
- Why not Parallel.ForEach? 6

Operating over Two Collections 7
- Using PLINQ for Operating over Two Collections 7
- Why not Parallel.ForEach? 7

Thread-Local State 8
- Using PLINQ for Thread-Local State Access 8
- Using Parallel.ForEach for Thread-Local State Access 8

Exiting from Operations 9
- Using Parallel.ForEach to Exit from Operations 9
- ParallelLoopState.Stop() 10
- ParallelLoopState.Break() 10
- Why not PLINQ? 11

Conclusion 11

Introduction

When you need to optimize a program for multi-core machines, a great place to start is by asking if your program can be split up into parts that can execute in parallel. If your solution can be viewed as a compute-intensive operation performed on each element in a large data set in parallel, it is a prime candidate for taking advantage of new data-parallel programming capabilities in .NET Framework 4: `Parallel.ForEach` and Parallel LINQ (PLINQ). This document will familiarize you with `Parallel.ForEach` and PLINQ, discuss how to use these technologies and explain the specific scenarios that lend themselves to each technology.

Parallel.ForEach

The `Parallel` class's `ForEach` method is a multi-threaded implementation of a common loop construct in C#, the `foreach` loop. Recall that a `foreach` loop allows you to iterate over an enumerable data set represented using an `IEnumerable<T>`. `Parallel.ForEach` is similar to a `foreach` loop in that it iterates over an enumerable data set, but unlike `foreach`, `Parallel.ForEach` uses multiple threads to evaluate different invocations of the loop body. As it turns out, these characteristics make `Parallel.ForEach` a broadly useful mechanism for data-parallel programming.

In order to evaluate a function over a sequence in parallel, an important thing to consider is how to break the iteration space into smaller pieces that can be processed in parallel. This partitioning allows each thread to evaluate the loop body over one partition.

`Parallel.ForEach` has numerous overloads; the most commonly used has the following signature:

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

The `IEnumerable<TSource> source` specifies the sequence to iterate over, and the `Action<TSource> body` specifies the delegate to invoke for each element. For the sake of simplicity, we won't explain the details of the other signatures of `Parallel.ForEach`; the list of different overloads can be found here.

PLINQ

Akin to `Parallel.ForEach`, PLINQ is also a programming model for executing data-parallel operations. The user defines a data-parallel operation by combining various predefined set-based operators such as projections, filters, aggregations, and so forth. Since LINQ is declarative, PLINQ is able to step in and handle parallelizing the query on the user's behalf. Similarly to `Parallel.ForEach`, PLINQ achieves parallelism by partitioning the input sequence and processing different input elements on different threads.

While each of these tools deserves an article on its own, this information is beyond to scope of this document. Rather, the document will focus on the interesting differences between the two approaches to data parallelism. Specifically, this document will cover the scenarios when it makes sense to use `Parallel.ForEach` instead of PLINQ, and vice versa.

Simple Data-Parallel Operation with Independent Actions

A data-parallel operation with independent actions involves executing a computationally-expensive action with no return value.

Using Parallel.ForEach for Performing Independent Actions

The independent action pattern can be achieved with the usage of any `Parallel.ForEach` signature, since all `Parallel.ForEach` overloads are unified by this capability; if you use the overload mentioned above, the code would look as follows:

```
public static void IndependentAction(
    IEnumerable<T> source, Action<T> action)
{
    Parallel.ForEach(source, element => action(element));
}
```

In this sample, `action(element)` represents the computationally expensive action you would perform per element.

Why not PLINQ?

While PLINQ has the `ForAll` operator, it may be easier to think in terms of parallel loops rather than parallel queries for this type of scenario. Furthermore, PLINQ may be too heavyweight for a simple independent action. With `Parallel.ForEach`, you can specify `ParallelOptions.MaxDegreeOfParallelism`, which specifies that at most N threads are required. Thus, if the ThreadPool's resources are scarce, even if the number of available threads is less than N, those threads will begin assisting the execution of `Parallel.ForEach`. As more threads become available, those resources will then be used for execution of the loop's body delegate as well. However, PLINQ requires exactly N threads, which is specified by using the `WithDegreeOfParallelism()` extension method. In other words, for PLINQ N represents the number of threads which are actively involved in the PLINQ query.

Ordered Data-Parallel Operation

Using PLINQ for Order Preservation

If your operation requires the output to preserve the ordering of the input data set, you will probably find it easier to use PLINQ rather than `Parallel.ForEach.` With PLINQ, the `AsOrdered()` operator will automatically handle the order preservation of the input sequence when executing the subsequent queries and abstract away the explicit index management and result storing details. It does so by reading the elements from the input data set, partitioning the data set, computing the results in parallel, and finally arranging the results into the correctly-ordered output sequence. For example, if you wanted to transform the frames in a movie from RGB to grayscale, the resulting movie would be incorrect unless the original ordering of the frames was preserved. Here is how you'd use an order-preserving query to process the bitmap frames in a movie in parallel, and then iterate over them in the order in which the corresponding frames appeared in the input movie sequence:

```
public static void GrayscaleTransformation(IEnumerable<Frame> Movie)
{
```

```
    var ProcessedMovie =
        Movie
        .AsParallel()
        .AsOrdered()
        .Select(frame => ConvertToGrayscale(frame));
    foreach (var grayscaleFrame in ProcessedMovie)
    {
        // Movie frames will be evaluated lazily
    }
}
```

The sample above demonstrates the succinctness with which order preservation can be expressed when using PLINQ. The power of increased productivity goes hand in hand with the declarative nature of PLINQ. Instead of worrying about how you will implement your solution such that the results are ordered in the same manner as the input, you simply express what you would like to be accomplished using PLINQ operators. . By calling `AsOrdered()` after `AsParallel()`, the transformed movie sequence will be in the correct order.

Why not Parallel.ForEach?

As mentioned before, order preservation is relevant when it comes to producing results that adhere to the original ordering. While order-preserving data-parallel operations can be implemented using `Parallel.ForEach`, it usually requires a considerable amount of work with the exception of a few trivial cases. As an example of such a trivial case, you can use the following `Parallel.ForEach` overload to accomplish the effects of PLINQ's `AsOrdered()` operator:

```
public static ParallelLoopResult ForEach<TSource >(
              IEnumerable<TSource> source,
              Action<TSource, ParallelLoopState, Int64>
       body)
```

This overload passes the index of the current element into the user delegate. Then, you can write the result out to the same index in the output collection, compute the expensive operation in parallel, and finally use the indices to correctly order the outputs. This sample demonstrates one of many ways to preserve order:

```
public static double [] PairwiseMultiply(
    double[] v1,
    double[] v2)
{
    var length = Math.Min(v1.Length, v2.Lenth);
    double[] result = new double[length];
    Parallel.ForEach(v1,
        (element, loopstate, elementIndex) =>
            result[elementIndex] = element * v2[elementIndex]);
    return result;
}
```

Note that in this sample, the input and output were fixed-size arrays, which made writing the results to the output collection rather simple. However, the disadvantages to this approach become immediately obvious. If the original collection were an `IEnumerable` rather than an array, you have 4 possible ways to implement order preservation:

- The first option involves taking the O(n) hit of calling the `IEnumerable.Count()` which effectively iterates over the entire collection to return the number of elements. Then, you can allocate a properly-sized array before calling `Parallel.ForEach`, and insert each result into the correct position in the output sequence.

- The second option would be to materialize the original collection before using it; in the event that your input data set is prohibitively large, neither of the first two options will be feasible.

- The third option involves some cleverness around your output collection. The output collection could be a hashing data structure, in which case the amount of memory used just to store the output will be at least 2x the size of the input memory in order for the hashing data structure to get around collisions; it is important to note that if your input size is large, the hashing data structure might be prohibitively large, and would drive performance down due to false cache sharing and garbage collection pressure.

- Finally, the last option would be to store the results along with their original indices, and implement your own sorting algorithm to perform a sort on the output collection before returning the results.

With PLINQ, the user simply asks for order preservation, and the query engine will handle the messy details to ensure that the results come out in the correct order. Since PLINQ's infrastructure allows the `AsOrdered()` operator to process streaming input, in other words, PLINQ allows the input to be lazily materialized. When using PLINQ, materializing the entire output sequence is the worst case pitfall, and you can simply sidestep all the other potential pitfalls mentioned above and perform your data-parallel operation with order preservation by using `AsOrdered()`.

Streaming Data-Parallel Operation

Using PLINQ for Stream Processing

PLINQ offers the capability of processing the query output as a stream. This ability is extremely valuable for several reasons:

1. The results don't have to be materialized in an array, thus, not all of it will have to coexist in memory at any given point in time.
2. You can enumerate over the results on a single thread as they are getting produced.

Continuing with the Analyze Stocks example from above, suppose that you wanted to compute the risk of each stock in a collection of stocks, filter out only the stocks which met a certain criteria of risk analysis, and then perform some computation over the filtered results. The code would look something like this in PLINQ:

```
public static void AnalyzeStocks(IEnumerable<Stock> Stocks)
{
    var StockRiskPortfolio =
        Stocks
        .AsParallel()
        .AsOrdered()
        .Select(stock => new { Stock = stock, Risk = ComputeRisk(stock)})
        .Where(stockRisk => ExpensiveRiskAnalysis(stockRisk.Risk));

    foreach (var stockRisk in StockRiskPortfolio)
    {
        SomeStockComputation(stockRisk.Risk);
        // StockRiskPortfolio will be a stream of results
    }
```

```
}
```

In the example above, the elements are distributed into partitions, processed with multiple threads and then rearranged; it is important to understand that these stages will take place concurrently, and as soon as some results are available, the single threaded consumer in the `foreach` loop can begin its computation. PLINQ optimizes for throughput instead of latency, and uses buffers internally; so it may be the case that even if a particular result has been computed, it may sit in an output buffer until the output buffer is fully saturated before it is allowed to be processed. This can be alleviated by using PLINQ's `WithMergeOptions` extension method, which allows you to specify the output buffering. `WithMergeOptions` takes a `ParallelMergeOptions` enumeration as the parameter; with this, you can specify how the query yields its final output when consumed by a single thread. The choices you have are:

- `ParallelMergeOptions.NotBuffered`: this specifies that each processed element is to be returned from each thread as soon as it's produced.
- `Parallel.MergeOptions.AutoBuffered`: this specifies that the elements be collected into a buffer, and the buffer's contents are periodically yielded to the consuming thread all at once.
- `ParallelMergeOptions.FullyBuffered`: this specifies buffering the entire output sequence, which might allow the results to be produced faster than using the other options, but may also take longer to yield the first element to the consuming thread.

An example on how to use `WithMergeOptions` is available on MSDN.

Why not Parallel.ForEach?

Let's set aside the disadvantages of using `Parallel.ForEach` for order preservation. For the unordered case of processing the result of a computation as a stream with `Parallel.ForEach`, the code would look as follows:

```
public static void AnalyzeStocks(IEnumerable<Stock> Stocks)
{

    Parallel.ForEach(Stocks,
        stock => {
            var risk = ComputeRisk(stock);
            if(ExpensiveRiskAnalysis(risk)
            {
                 // stream processing
                 lock(myLock) { SomeStockComputation(risk) };
                 // store results
            }

}
```

This is nearly identical to the PLINQ sample above, with the exception of explicit locking and less elegant code. Note, in this situation `Parallel.ForEach` expects you to store your results in a thread-safe manner, while PLINQ handles those details for you.

Recall that `Parallel.ForEach`'s body delegate is an `Action` <T> which returns void. In order to store results, we have 3 options; the first is to store the resulting values in a collection which isn't thread safe and therefore will require locking on each write. The second option for storing the resulting values is to do so with a collection which is lock-free and concurrent. Thankfully, instead of having to implement your own, .NET Framework 4 provides a rich set of these collections in the `System.Collections.Concurrent` namespace. The third option for storing result data involves using the thread-local signature of `Parallel.ForEach` and will be discussed in detail later.

Each of these options involves explicitly managing your side effects for simply writing results, all of which are abstracted away by the PLINQ infrastructure.

Operating over Two Collections

Using PLINQ for Operating over Two Collections

PLINQ's Zip operator specifically handles performing a computation while iterating over two separate collections concurrently. Since it is composable with other query operators, you can perform any number of complex operations on each collection in parallel before combining the collections. For example:

```
public static IEnumerable<T> Zipping<T>(IEnumerable<T> a, IEnumerable<T> b)
{
    return
        a
        .AsParallel()
        .AsOrdered()
        .Select(element => ExpensiveComputation(element))
        .Zip(
             b
             .AsParallel()
             .AsOrdered()
             .Select(element => DifferentExpensiveComputation(element)),
             (a_element, b_element) => Combine(a_element,b_element));
}
```

The sample above demonstrates that each data source can be processed concurrently with different computations; once the results are available from both sources, they can be combined with the Zip Operator.

Why not Parallel.ForEach?

Zipping can essentially be accomplished using the index-aware overload of `Parallel.ForEach` like so:

```
public static IEnumerable<T> Zipping<T>(IEnumerable<T> a, IEnumerable<T> b)
{
    var numElements = Math.Min(a.Count(), b.Count());
    var result = new T[numElements];
    Parallel.ForEach(a,
        (element, loopstate, index) =>
        {
            var a_element = ExpensiveComputation(element);
            var b_element = DifferentExpensiveComputation(b.ElementAt(index));
            result[index] = Combine(a_element, b_element);

        });
    return result;
}
```

However, there are potential pitfalls and disadvantages associated with attempting to combine two collections in this fashion. The disadvantages include the same ones listed above in using `Parallel.ForEach` for order preservation, and one of the pitfalls includes walking over the end of one collection since you are explicitly managing the indices.

Thread-Local State

Using PLINQ for Thread-Local State Access

When performing concurrent data parallel operations, an attractive solution to reducing the cost of synchronization is to use thread-local storage to store the results from each thread's execution, and combine the results at the end, thereby reducing the number of times one needs to incur the overhead of using a synchronization primitive. An option for accessing thread-local state from PLINQ involves using an overload of PLINQ's `Aggregate()` operator, which looks like this:

```
public static TResult Aggregate<TSource, TAccumulate,TResult>(
            this ParallelQuery<TSource> source,
            Func<TAccumulate> seedFactory,
            Func<TAccumulate,TSource,TAccumulate> updateAccumulatorFunc,
            Func<TAccumulate,TAccumulate,TAccumulate>
combineAccumulatorFunc,
            Func<TAccumulate, TResult> resultSelector)
```

Here's how we can implement a geometric mean operator using this overload:

```
public static double GeometricMean(this IEnumerable<int> source)
{
    return
        source
        .AsParallel()
        .Aggregate(
        () => new double[]{1.0,0.0},
        (accumulator, element) =>
            {
                accumulator[0] *= element;
                accumulator[1]++;
                return accumulator;
            },
            (accumulator1, accumulator2) =>
                {
                    accumulator1 [0] *= accumulator2[0];
                    accumulator1[1] += accumulator2[1];
                    return accumulator1;
                },
            accumulatedValues =>
                Math.Pow(accumulatedValues[0], (1/accumulatedValues[1])));
}
```

Although the amount of work per element is likely too small for reaping the benefits of parallelization, the example is designed to demonstrate how to use the Aggregate overload with thread-local access.

Using Parallel.ForEach for Thread-Local State Access

With `Parallel.ForEach`'s, the most general thread-local selection signature looks like this:

```
public static ParallelLoopResult ForEach<TSource,
TLocal>(
             IEnumerable<TSource> source,
             Func<TLocal> localInit,
```

```
                Func<TSource, ParallelLoopState, TLocal,
        TLocal> body,
                Action<TLocal> localFinally)
```

The following example demonstrates how to use this signature to obtain the geometric mean of a sequence:

```
public static double GeometricMean(IEnumerable<int> source)
{
    double Products = 1.0;
    double count = 0.0;
    object mylock = new object();
    Parallel.ForEach(source,
        () => new double[] { 1.0, 0.0 },
        (element, loopstate, tlocal) =>
        {
            tlocal[0] *= element;
            tlocal[1]++;
            return tlocal;
        },
            (final) =>
            {
                lock (mylock)
                {
                    Products *= final[0];
                    count += final[1];
                }
            });
    return Math.Pow(Products, 1 / count);
}
```

This example is designed to illustrate that when using `Parallel.ForEach`, thread-local state can be used to dramatically reduce the synchronization costs. By using thread local state, the lock is taken N times, where N is the number of threads used during execution; if we had used a non-thread-local state overload of `Parallel.ForEach` instead, the lock would have been taken for each time a product was computed, an expensive price to pay for a small operation. It is important to realize that with `Parallel.ForEach`, you need to explicitly manage the synchronization mechanism when storing the result, whereas with PLINQ, the synchronization details are abstracted away from the user. As such, in the case where your aggregation operation requires complex synchronization, PLINQ may be more suitable; however, if your aggregation synchronization could instead be performed with a lightweight and simple mechanism, for example, an interlocked operation, using PLINQ might lead to longer running times as you would be relying on PLINQ's general mechanism. Furthermore, it is important to keep in mind the aspects to the implementation details differences between `Parallel.ForEach` and PLINQ such as the degree of parallelism during execution (mentioned above). Depending on the details of the operation you would like to express, these factors will help in determining which technology fits the situation better.

Exiting from Operations

Using Parallel.ForEach to Exit from Operations

For cases in which control flow is relevant, it is important to realize that exiting from a `Parallel.ForEach` loop allows you to achieve the same effect as a control flow statement in a sequential loop while ensuring correctness. Recall, some overloads of `Parallel.ForEach` allow you to track the ParallelLoopState, the most commonly used signature looks like:

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

The ParallelLoopState provides support for breaking out of a loop early by way of two separate methods, which will be discussed in the next two sections.

ParallelLoopState.Stop()

`Stop()` informs the loop that it is unnecessary to start and execute more iterations and the loop will cease as soon as possible; the `ParallelLoopState.IsStopped` property allows each iteration to detect when another iteration has called `Stop()`. `Stop()` is typically useful in the case when your loop is performing an unordered search, and can be exited once it has found what it is looking for. For example if you're searching for any instance of an object in a collection of objects, the code would look like this:

```
public static boolean FindAny<T,T>(IEnumerable<T> TSpace, T match) where T:
IEqualityComparer<T>
{
    var matchFound = false;
    Parallel.ForEach(TSpace,
        (curValue, loopstate) =>
            {
                if (curValue.Equals(match) )
                {
                    matchFound = true;
                    loopstate.Stop();
                }
            });

    return matchFound;

}
```

While the functionality of this operation can be achieved with PLINQ, this example serves to demonstrate how to use `ParallelLoopState.Stop()` as a control flow statement.

ParallelLoopState.Break()

`Break()` informs the loop that iterations belonging to elements preceding the current element still need to be run, but iterations belonging to the elements that follow the current element need not be. The lowest iteration from which `Break()` is called takes effect; this number can be retrieved from the `ParallelLoopState.LowestBreakIteration` property. `Break()` is typically useful in the case that your loop is performing an ordered search. In other words, some search criteria defines a given point up to which data needs to be processed. For example, if you had a sequence with non-unique elements, and wanted to return the lowest index of a matched object, the code would look something like this:

```
public static int FindLowestIndex<T,T>(IEnumerable<T> TSpace, T match) where
T: IEqualityComparer<T>
{
```

```
    int matchedIndex = -1;
    Parallel.ForEach(TSpace,
        (curValue, loopstate, curIndex) =>
            {
                if (curValue.Equals(match) )
                {
                    matchedIndex = curIndex;
                    loopstate.Break();
                }
            });
    return matchedIndex;
}
```

The example above would loop until an instance of the object was found, and signal `Break()`, indicating that only elements with a lower index than the matched instance from the original sequence need to be iterated over; if another match is found in this set of elements, `Break()` is signaled again, and this is repeated until there are no more elements left , at which point the lowest iteration at which `Break()` was called is the lowest index of the element which matched.

Why not PLINQ?

Even though PLINQ provides support for exiting from within a query execution, the differences between PLINQ's exit mechanism and `Parallel.ForEach` are substantial. To exit from a PLINQ query, supply the query with a cancellation token, as explained here. With `Parallel.ForEach`, exiting flags are polled on every iteration. In PLINQ, the cancellation token is polled once every so often, so you cannot assume that a cancelled query will stop quickly.

Conclusion

`Parallel.ForEach` and PLINQ are great tools to quickly inject parallelism into your applications without a steep learning curve. When using these tools, keep in mind the differences and tips explained in this article, and always choose the right tool for your problem.

This material is provided for informational purposes only. Microsoft makes no warranties, express or implied.
©2009 Microsoft Corporation.