

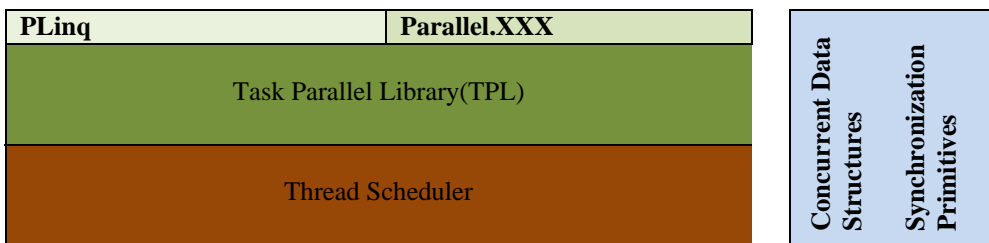
Microsoft Pfx (Parallel Extensions) June 2008 CTP

By Andy Clymer

Over the past five years we have seen a major change in CPU design. Traditionally applications were able to run faster as the clock speeds of the CPU's increased. The increase in CPU speed pretty much followed Moores Law (doubling every 18 months). However, it is now not proving possible to keep increasing performance in this exponential way. In an effort to continue to build machines with greater performance the chip manufacturers have changed strategy and have now moved to multi core based architectures. This results in the machine having a series of execution engines that can run concurrently as opposed to the traditional single core ones that can only execute one instruction at a time. For this to have the desired effect of ever better performance, applications need to be written so that they can take advantage of multiple execution engines, in other words the application has to be structured to make use of multiple threads.

Taking a traditional sequential application and making it use multiple threads; can often be a serious challenge: algorithms have to be restructured so that they can be parallelized; additional plumbing code needs to be written to co-ordinate the parallel algorithm. Microsoft has attempted to solve/assist in the latter by providing a new library inside the System.Threading namespace, known as the Parallel Framework Extensions, or Pfx.

Pfx consists of the following core components



If we start from the bottom up, the Thread Scheduler provides the necessary logic to spawn threads, and to ensure that the appropriate threads are created to keep the core's busy. This layer is not intended for use by application developers directly but as a building block for the parallel framework. The scheduler uses its own pool of threads and does not currently; rely on the normal .NET thread pool. Above the scheduler lives the Task Parallel Library (TPL). This provides the necessary abstractions to create parallel tasks, which will ultimately be scheduled to run on one of the scheduler's threads. Note that creating a task does not mean creating a new thread; the scheduler will create a new thread if it deems there is sufficient compute resource to run the task, or it will simply queue the request waiting for an available core. Tasks can also be structured hierarchically so that Task A may well create sub tasks B and C, waiting for task A to complete by default will ultimately mean waiting for all tasks A,B and C to complete. Below is a code sample that creates a task and then waits for it to complete.

```

using System.Threading.Tasks;
using System.Threading;

namespace SimpleTasks
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task = Task.Create(
                delegate(object o)
                {
                    Console.WriteLine("Task is running on thread : {0} ",
                        Thread.CurrentThread.ManagedThreadId);
                }
            );

            task.Wait();
            Console.WriteLine("All Done");
        }
    }
}

```

While developers can use the TPL, there is an additional layer that sits on top of it that is designed to be the starting point when attempting to parallelize a piece of computational logic. This is the layer that contains Parallel LINQ (PLINQ), Parallel classes and Future<T>, all of this offer a simple means to turn a sequential piece of logic into one that makes use of multiple cores.

Perhaps the simplest API to consider is Parallel.X. This type provides a facade to implement some of the basic parallel patterns. For example, suppose I have a standard `for` loop that simply squares each item in an array. This loop is a good candidate for parallelization as each iteration of the loop is independent of the others and can be executed in any order.

```

private static void SquareArray(int[] elements)
{
    for (int nItem = 0; nItem < elements.Length; nItem++)
    {
        elements[nItem] *= elements[nItem];
    }
}

```

To make use of `Parallel.For` here we simply refactor too

```
private static void ParallelSquareArray(int[] elements)
{
    Parallel.For(0, elements.Length, nItem =>
    {
        elements[nItem] *= elements[nItem];
    });
}
```

`Parallel.For` in this guise takes three parameters: a start index, an end index and an instance of the Action delegate. The Pfx library will schedule the running of the delegate for each iteration of the loop, but in this case taking advantage of multiple cores along the way.

Wow! This looks so easy! Unfortunately the performance of the example above is far slower than its sequential version. This is due to the fact that the piece of work to be executed, on each iteration, is so small that the cost of invoking the work via a delegate just swamps the rest of the processing. So when looking for potential loops to parallelize you need to ensure there is sufficient work in each iteration to offset the overhead of invoking the work via a delegate. The most likely candidates are when you have two loops: the outer loop is often parallelized leaving the inner loop to run as a whole on each thread. There is also a `Parallel.Foreach` which works along similar lines as `Parallel.For` but for `IEnumerable<T>` streams.

Even if you can find loops that have sufficient amount of work often loop iterations are not completely independent of each other. Take this example: a method that attempts to approximate

$$\int_0^1 \frac{4}{1+x^2} dx$$

A C# implementation looks like

```

private static void SequentialMethod()
{
    double numberOfSteps = 100000000;
    double step = 1.0 / (double)numberOfSteps;
    double sum = 0;

    for (int nStep = 0; nStep < numberOfSteps; nStep++)
    {
        double x = (nStep + 0.5) * step;

        sum = sum + 4.0 / (1.0 + x * x);
    }

    double pi = step * sum;

    Console.WriteLine(pi);
}

```

While each iteration of the loop is using a different value for `x`, `sum` is being shared. To allow us to continue to use `Parallel.For` we need to use a form that allows each thread to have its own value for `sum`, so there is no contention when `sum` is updated. If each thread has its own local copy of `sum` all then we need to do is to supply some additional logic that combines each local result as each thread completes.

```

private static void ParallelFor()
{
    int numberOfSteps = 100000000;
    double step = 1.0 / (double)numberOfSteps;
    double sum = 0;
    object sumLock = new object();

    Parallel.For(0, numberOfSteps, 1, () => 0.0,
        (nStep, localSum) =>
        {
            double x = (nStep + 0.5) * step;

            localSum.ThreadLocalState = localSum.ThreadLocalState + 4.0 / (1.0 + x * x);
        },
        (lsum) => { lock (sumLock) { sum += lsum; } } );

    double pi = step * sum;

    Console.WriteLine(pi);
}

```

Initialise local sum

Combine local sums

In this version `Parallel.For` takes two additional parameters, the first one (parameter 4) is a delegate that is responsible for initialising the local value; the second one is a delegate responsible for combining the

local results (parameter 6). Note that this delegate will be executed in parallel too so shared variables must be accessed in a thread safe manner.

Again, unfortunately, this implementation of the algorithm does not have sufficient work inside the loop body, and as such is slightly slower than the sequential version. To solve this kind of problem we will need to partition the work into bigger blocks. In earlier CTPs, due to the way `Parallel.For` was implemented, this was not very easy. In the latest CTP `Parallel.For` has been improved to make this kind of partitioning possible. To create larger blocks of work we need to have two loops: an outer loop that is parallelized and an inner loop that represents the parallel task. By doing this there is now sufficient work inside the inner loop to utilise the benefit of multiple cores. The dimensions for the outer loop have been based on the number of cores on the machine to minimise the scheduling overhead, this has been done by working out the block size based on `Environment.ProcessorCount`.

```
private static void OptimisedParallelFor()
{
    object sumLock = new object();
    double numberOfSteps = 100000000;
    double step = 1.0 / (double)numberOfSteps;
    double sum = 0;

    // Assumes numberOfSteps is exactly divisible by number of cores
    int nWorkItems = Environment.ProcessorCount;
    int stride = (int)(numberOfSteps / nWorkItems);

    Parallel.For(0, nWorkItems, 1,
        () => 0.0,
        (nWorkItem, lsum) =>
        {
            int startPoint = (int)((numberOfSteps / nWorkItems) * nWorkItem);

            for (int nStep = startPoint; nStep < stride + startPoint; nStep++)
            {
                double x = (nStep + 0.5) * step;

                lsum.ThreadLocalState = lsum.ThreadLocalState + 4.0 / (1.0 + x * x);
            }
        },
        (lsum) => { lock (sumLock) { sum += lsum; } });

    double pi = step * sum;

    Console.WriteLine(pi);
}
```

While this approach works in our example and scales very nicely it does assume that each iteration of the outer loop represents the same amount of work. If the logic was such that the amount of work for the inner loop varied then having the same number of outer iterations as the number of cores wouldn't make best use of the cores. In these cases you will need to determine what is a reasonable values for the outer loop to overcome the overhead of scheduling the work, but have sufficient number of work allocations so that as a core becomes idle it can pick up another piece of work. This will undoubtedly come down to benchmarking and experience.

As well as `Parallel.X` type there is also an implementation of `Parallel IEnumerable` based LINQ, commonly known as PLINQ. PLINQ parallelizations tend to be a little simpler to get increased performance, possibly due to the fact that sequential LINQ already relies heavily on delegate invocation.

Below is a method that takes an XML document that contains the TV program guide for the next 7 days, and an array of favourite programs you just can't afford to miss. The method implements a LINQ query against the XML document to identify any matching programs.

```
private static void FilterGuide(string[] showsToWatch, XDocument guide )
{
    var toWatch = from program in guide.Descendants("programme")
                  where IsShowToWatch( showsToWatch , (string)program.Element("title") )
                  select new
                  {
                      Title = (string)program.Element("title"),
                      Channel = (string)program.Parent.Attribute("id"),
                      When = ToDateTime(
                          (string)program.Parent.Attribute("date"),
                          (string)program.Element("start"))
                  };

    Console.WriteLine("Found {0} ", toWatch.Count());
}
```

To turn this into a parallel query you simply call `AsParallel` on the data source to produce a parallel enumerable source.

```
private static void ParallelFilterGuide(string[] showsToWatch, XDocument guide)
{
    var toWatch = from program in guide.Descendants("programme").AsParallel()
                  where IsShowToWatch(showsToWatch, (string)program.Element("title"))
                  select new
                  {
                      Title = (string)program.Element("title"),
                      Channel = (string)program.Parent.Attribute("id"),
                      When = ToDateTime(
                          (string)program.Parent.Attribute("date"),
                          (string)program.Element("start"))
                  };

    Console.WriteLine("Found {0} ", toWatch.Count());
}
```

The `AsParallel` method call returns an object that is of type `IParallelEnumerable<T>`, which extends `IEnumerable<T>`, but the important thing now is that PLINQ brings to the table its own LINQ extension methods which cause the remaining part of the query to execute in parallel. The PLINQ extension methods are defined in `System.Linq.ParallelEnumerable` and seem cover the majority of LINQ based features.

Finally, and what is new to this release of Pfx, are the concurrent data structures (CDS), these APIs are used throughout the Pfx stack and provide implementations of common data structures that are optimised to work in a multi threaded environment. This new area of concurrent data structures is perhaps one of the most useful areas of Pfx. These data structures are applicable, not just in parallelizing compute based problems but, to concurrency in general.

Perhaps the biggest players in CDS are the implementations of `ConcurrentQueue<T>` and `ConcurrentStack<T>`. Both of these implementations are designed to never block and so provide a `TryDequeue` and a `TryPop`. The use of the try pattern is used as multiple threads are likely to be attempting to remove an item from the collection and rather than throw an exception if there are no longer items in the collection the method simply returns `false`, saving the expense of an exception.

While there are times when that style of access is applicable, there are also times when a straight forward implementation of the producer/consumer pattern would prefer a blocking variant. Rather than provide a separate version of `ConcurrentQueue` and `ConcurrentStack` that supports blocking operations, the Pfx team have deployed a single class called `BlockingCollection`, this type can provide blocking functionality to either the `ConcurrentQueue` or `ConcurrentStack`. The `BlockingCollection` class acts as a decorator, this is achieved by the fact that both `ConcurrentStack` and `ConcurrentQueue` both implement an interface called `IConcurrentCollection<T>`. Not only does the `BlockingCollection` support blocking `Remove` calls it also supports blocking `Add` calls allowing the throttling of producers.

Perhaps the neatest method on `BlockingCollection` is `GetConsumingEnumerable` this allows you to process the collection as an `IEnumerable<T>`, allowing the consumer code to simply be a `foreach` loop that terminates when the producer signals that it no longer wishes to add new items.

Below is an example of a producer that simply places numbers entered by the user into a queue. When no more numbers are to be processed, it calls `CompleteAdding` to signal no more values will arrive on the queue, it then waits for the consumer thread to finish.

```

static void Main(string[] args)
{
    ConcurrentQueue<int> queue = new ConcurrentQueue<int>();
    BlockingCollection<int> blockingQueue = new BlockingCollection<int>(queue);

    Thread consumer = new Thread(EnumerableConsumer);
    consumer.Start(blockingQueue);

    string numStr = string.Empty;
    do
    {
        Console.Write("Enter number:");
        numStr= Console.ReadLine();

        if (numStr != string.Empty)
        {
            blockingQueue.Add(int.Parse(numStr));
        }
    }
    while (numStr != string.Empty);

    // Producer signals no more items to be added to the queue
    blockingQueue.CompleteAdding();
    Console.WriteLine("Finished");

    Console.WriteLine("Queue empty...finishing off last items");
    consumer.Join();
    Console.WriteLine("System shutdown ok..");
}

```

The consumer thread code is as follows

```

private static void EnumerableConsumer(object o)
{
    BlockingCollection<int> stream = (BlockingCollection<int>)o;

    foreach (int number in stream.GetConsumingEnumerable())
    {
        Console.WriteLine("GOT : {0}", number);
        Thread.Sleep(2000);
    }

    Console.WriteLine("No more numbers...");
}

```

The implementation above only has one consumer thread, having multiple can be as simple as changing the `foreach` to a `Parallel.Foreach` and, voila, multiple consumer threads...

CDS also has a new type called `LazyInit<T>` that provides a way to initialise an object when it is first used as opposed to when its container is created. This implementation of `Lazy<T>` uses best practices for double check locking which, in this day and age, still appears to be a black art to many.

One of the other new types supported is a `CountdownEvent` in pure managed code. Countdown events are typically used when you have a series of tasks that are running and you wish know when they are all done. The countdown event is created with an initial value that represents the number of tasks. The main thread then issues a `Wait` until the counter hits zero. As each of the tasks completes they decrement the countdown event, when the count reaches zero the event is raised and the main thread is unblocked.

The following code downloads various web pages asynchronously with the main thread waiting to be notified when all the downloads have completed. To do this there is a `CountdownEvent` that is set to the number of pages to be downloaded, as each page is downloaded the countdown event is decremented. The main thread calls `Wait` on the countdown event, when the countdown event reaches zero the event is deemed signalled and the main thread unblocks.

```
string[] urls = {
    "http://www.google.co.uk" ,
    "http://www.develop.com" ,
    "http://www.bbc.co.uk",
};

CountdownEvent workOutstanding = new CountdownEvent(urls.Length);

foreach( string url in urls )
{
    WebRequest request = WebRequest.Create( url );

    request.BeginGetResponse(
        ar =>
        {
            WebResponse response = request.EndGetResponse( ar );
            string content = (new StreamReader( response.GetResponseStream() )).ReadToEnd();
            workOutstanding.Decrement();
        } , null);
}

Console.WriteLine("Downloading..");
workOutstanding.Wait();
Console.WriteLine("All Downloaded");
```

That concludes a brief look at the new release of Pfx. Pfx is starting to reach a high level of usability providing not only parallelization constructs that fit very neatly into the framework but also providing the necessary concurrent data structures that .NET has been crying out for since its very inception. One final note when benchmarking your parallel code: ensure that you run the benchmark at least twice inside the same process; otherwise you will be timing not just the code itself but also the overhead for loading the Pfx assembly and the necessary JIT compilation.