Hands-On Lab

An Introduction to Parallel LINQ

Lab version: 1.0.0

Last updated: 3/13/2010

Contents

Introduction

In this lab, we'll be exploring new parallel processing opportunities available using Parallel LINQ, or PLINQ. We will contrast methods of iteration and threading, convert LINQ queries to PLINQ equivalents, discuss common pitfalls and best practices, and demonstrate ways to optimize your query performance.

Lab Objectives

After completing this lab, you'll better understand parallel querying including:

1. Creating parallel queries various ways
2. Dividing work using multiple strategies
3. Ensuring thread safety of your data
4. Understanding the runtime workings of a multithreaded application

Prerequisites

This lab requires the following components:

- Visual Studio 2010 Express Edition or higher
- Microsoft .NET Framework 4 (installed with VS 2010)

This lab assumes intermediate knowledge of threading concepts. This includes the concept of hardware and software threads, thread scheduling, and work partitioning. Familiarity with .NET, C#, and LINQ is necessary. Existing familiarity with Visual Studio 2005 or higher is also very helpful.

An Overview of Language INtegrated Queries (LINQ)

LINQ was introduced with .NET 3.5 as a way to bring data querying into Visual Basic and Visual C# in a natural manner. Instead of crafting queries using various dialects of SQL, LINQ allows developers to query data using language constructs integrated directly into the platform. This prevents errors from query typos, helps to reduce or even eliminate common security pitfalls in query construction, and it allows you to think about data querying in the same terms as the rest of an application.

The mechanism by which LINQ actually converts your code into a query structure varies depending on the specific provider. Some providers create an in-memory representation of the operations in a structure called an expression tree, while others simply chain together operations on a given enumeration. An expression tree is typically used to create a native query for another component, such as a database engine. In this manner, multiple databases can be supported. The same syntax can be used to query in-memory collections of type IQueryable. This allows a developer to query a database or a System.Collections.Generic.List using the same basic syntax.

With the ever-growing base of multicore/multi-processor systems, it has become necessary to enable parallel processing wherever possible. Querying is a natural fit as conditions often are evaluated on each member of a collection, with no adverse interactions allowed between each operation. For example, if a data collection contains 1000 items, LINQ performs any evaluations on one thread in a sequential manner. With PLINQ, these same conditions can now be evaluated in parallel, potentially drastically reducing overall query time.

This lab will help to get you started with parallel queries and show you just how easy they can be.

Exercise 1 – Converting to PLINQ

There are many ways to go from a collection of data to a smaller subset. This may involve iterating over a collection and evaluating a condition against each element, using hash values, or looking something up in an index. For example, in order to find a given customer in a collection, you might perform a string match against the last name for every customer in the list. In this exercise, you will see several ways to convert existing code to PLINQ syntax.

Task 1 – Converting from Simple Iteration to LINQ

In this task, you will see the basic steps to convert a loop-based filter to a LINQ query.

1. Open the **PLINQ-HOL** solution
2. In the **Solution Explorer**, double-click **Ex1.cs** (files/objects are named based on the exercise)
3. Locate the **IsPrime** method

The purpose of **IsPrime** is to check whether a particular integer is a prime. The **IsPrime** method divides a given number by all numbers up to the square root of the given number. If there is no integer remainder, then the number being tested has a factor. The first few lines discard special cases, then odd numbers are tested. We'll use this method throughout Exercise 1.

```
public static bool IsPrime(int n)
{
    // zero, one, and negative numbers are not prime
    if (n < 2) return false;
    // 2 is the only even prime
    else if (n == 2) return true;
    // Check for evens
    else if ((n & 1) == 0) return false;

    double sq = Math.Sqrt(n);
    for (int i = 3; i <= sq; i += 2)
    {
        if (n % i == 0) return false;
    }
```

```
    return true;
}
```

The first way to find primes is by using a *for* loop. A *for* loop lets you iterate over a range of numbers :

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

Another option is to use the *foreach* loop. This requires a type that implements the **IEnumerable**<> interface to iterate over, but this can be created easily by using the **Range** static method from the **Enumerable** class. This returns a new enumeration of numbers satisfying the given range. This is how we will create the initial sequential implementation.

4. Enter the following code after the **IsPrime** method:

```
public static List<int> Task1()
{
    var primes = new List<int>();

    foreach (var item in Enumerable.Range(lowTest, totalTest))
    {
        if (IsPrime(item)) primes.Add(item);
    }

    return primes;
}
```

This is a very straight-forward example of how to iterate over a collection and compare each item against something. The resulting collection will contain only the primes. The next step is to convert this to a LINQ query.

5. Enter the following code after the **Task1** method:

```
public static List<int> Task1_LINQ()
{
    var primes = from item in Enumerable.Range(lowTest, totalTest)
                 where IsPrime(item)
                 select item;

    return primes.ToList();
}
```

This is a simple LINQ example. Of course, LINQ provides you so many other operators such as complex filtering, projecting into new objects, ordering, and more. At this point, the LINQ query provides little advantage over the *foreach* implementation. Now it's time to explore what it takes to convert this sequential query into a parallel implementation that takes advantage of multiple processor cores.

Task 2 – Moving from LINQ to PLINQ

Moving from a sequential LINQ query to a parallel query is very simple in many cases. The *Enumerable* class just yields integers in the supplied range. An equivalent class, *ParallelEnumerable,* provides parallel behavior by causing the query to do processing on multiple threads, thus making your code instantly multi-threaded without doing any extra work.

1. Make a copy of **Task1_LINQ** and rename it to **Task2_PLINQ**:

```
public static List<int> Task2_PLINQ()
{
    var primes = from item in Enumerable.Range(lowTest, totalTest)
                 where IsPrime(item)
                 select item;

    return primes.ToList();
}
```

With just one small change, you can make a big difference to performance.

2. In the first line of the LINQ query, replace the call to **Enumerable.Range** with **ParallelEnumerable.Range**:

```
var primes = from item in ParallelEnumerable.Range(lowTest, totalTest)
```

Now when the query is processed, different elements may be processed by different threads. This is done completely automatically. You don't need to worry about creating **Thread** objects, dividing up the work, or combining the results at the end.

In order to test the queries, we'll add some code to setup the timer, call the method, and display the results.

3. In **Solution Explorer**, open **MainWindow.xaml.cs** in **Code View**
4. After the "Insert code after this point" content, paste the following code:

```
timer.Restart();
primes = Ex1.Task1_LINQ();
timer.Stop();
ReportProgress("Ex1.Task1_LINQ", primes.Count, timer);
```

This uses a **System.Diagnostics.Stopwatch** object which allows you to start, stop, reset, and read elapsed time. It's a great way to time sections of code with little effort. The **Task1_Linq** method is a LINQ query, but not parallel. This can serve as a baseline as we explore other methods of execution. We will also add a call to the parallel version to see the difference.

5. After the previous code block, paste the following code:

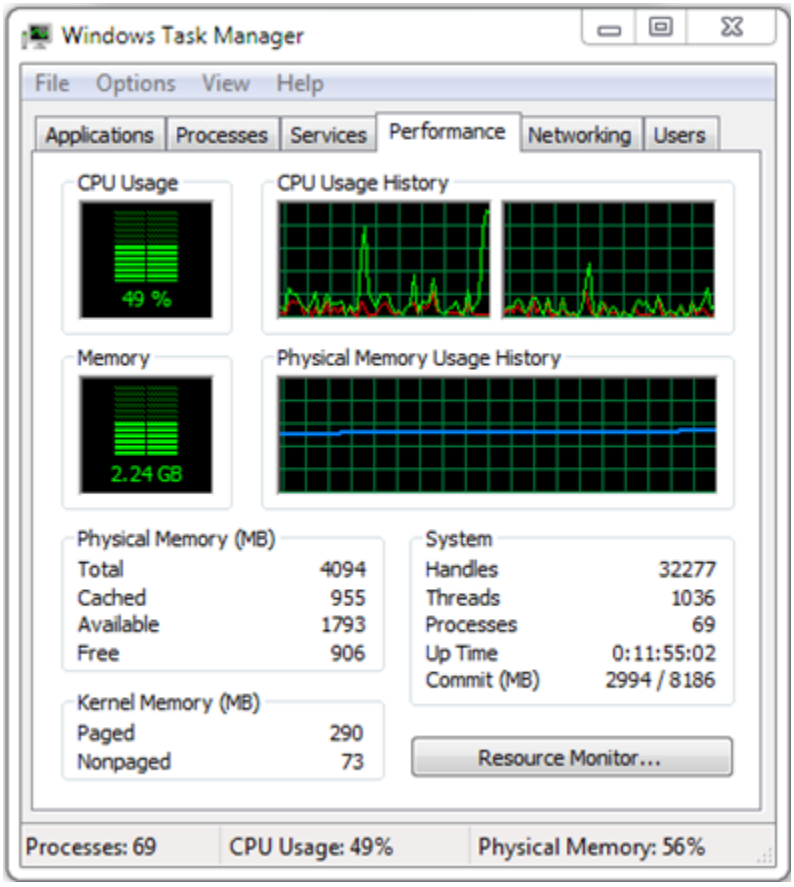
```
timer.Restart();
primes = Ex1.Task2_PLINQ();
timer.Stop();
ReportProgress("Ex1.Task2_PLINQ", primes.Count, timer);
```

We are now ready to test. Assuming this is running on a machine with more than one core, we should expect to see a difference in performance between the two methods. We also expect to see the same number of prime numbers; otherwise something is wrong with the implementation.

6. Press **CTRL+ALT+DELETE**, then click **Start Task Manager**
7. In **Windows Task Manager**, switch to the **Performance** tab
8. In the **Options** menu, ensure that **Always on Top** is checked

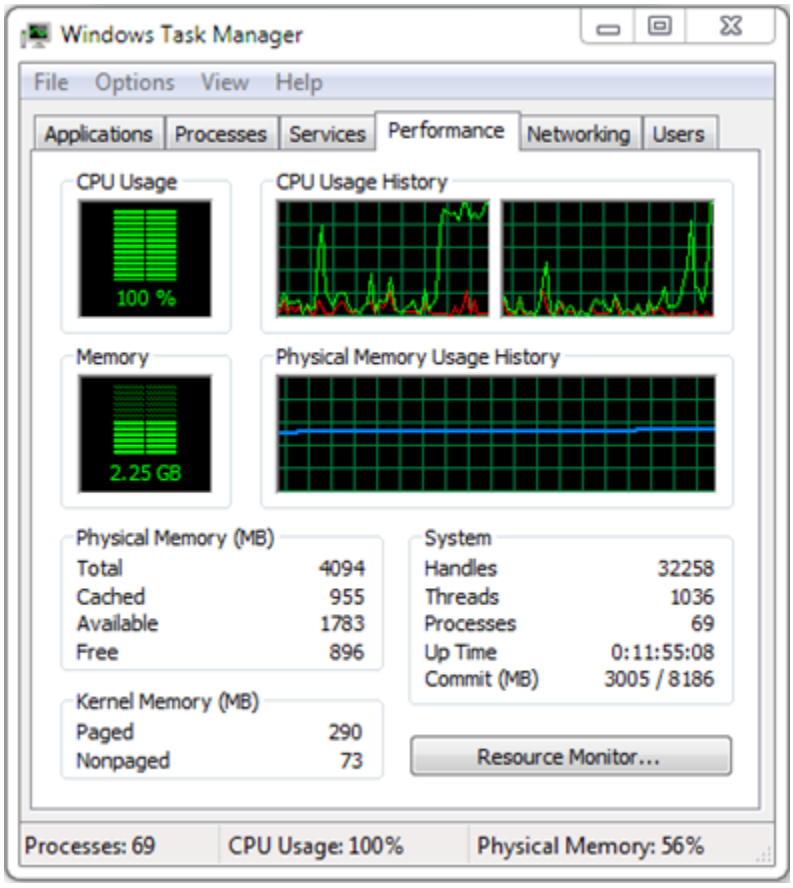
The task manager shows CPU utilization which is constantly fluctuating. Watching the CPU Usage History makes it easier to see the change from running a sequential implementation to a parallel one. Note that these numbers are heavily influenced by other applications running at the same time, so performance numbers can sometimes be misleading. True timings need to be run with controlled environments, and over multiple executions to determine averages. Numbers will almost certainly be different from those shown in these screenshots.

9. Debug the solution by clicking the **Debug | Start Debugging** menu command (**F5**)
10. Click **Start** (There is no progress shown until each item completes)
11. Observe the Task Manager to see that the number hovers around 50-60%

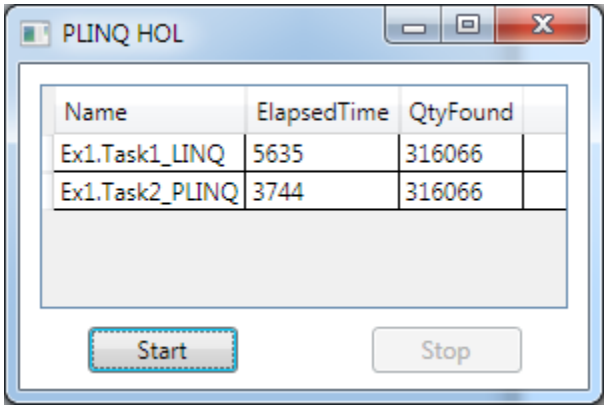


12. Once the first status appears in the **PLINQ HOL** window, the parallel version will be running. Observe that the Task Manager remains in the 90%-100% range during this version. Depending on the speed of processor(s) and number of cores, this could happen quickly. If you don't see it, don't worry about it. The **ElapsedTime** in the main window

summarizes things nicely.



13. When the code completes, it will be clear (assuming that you have more than one core) that the parallel version is faster.



Task 3 – Contrasting Threading with Parallel LINQ

In some cases, your code may already be threaded using a classic threading model. While converting to PLINQ may not affect performance, it will likely simplify the flow of the code resulting in a more maintainable application with less likelihood of mistakes.

1. In **Ex1.cs**, Create the following method after **Task2_PLINQ**:

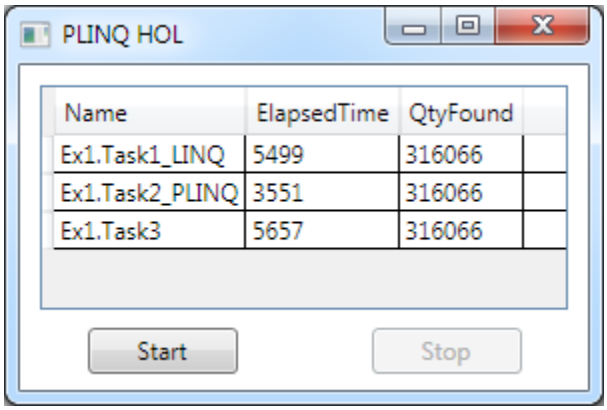
```
public static List<int> Task3()
{
    List<int> primes = new List<int>();
    int complete = 0;
    AutoResetEvent wait = new AutoResetEvent(false);

    for (int i = lowTest; i < highTest; i++)
    {
        ThreadPool.QueueUserWorkItem((o) =>
        {
            if (IsPrime((int)o))
                lock (primes) { primes.Add((int)o); }

            if (Interlocked.Increment(ref complete) == totalTest)
                wait.Set();
        }, i);
    }

    wait.WaitOne();
    return primes;
}
```

2. In **MainWindow.xaml.cs**, in the **w_DoWork** method, paste the following code before the end of the **try** block:

```
timer.Restart();
primes = Ex1.Task3();
timer.Stop();
ReportProgress("Ex1.Task3", primes.Count, timer);
```

This method uses the **QueueUserWorkItem** static method on the **ThreadPool** class. Every number to be tested is added to the work queue, then as threads in the pool are available they will be assigned the next work item. For CPU-bound applications, such as this one, you will often want no more threads than processors. In an I/O bound application where threads are often in a wait state, the correct number of threads could be more or less than the number of processors. There is some fine-tuning to determine the number of threads for a given workload.

In this example, the **ThreadPool** class actually performs worse than the sequential LINQ query:



This is because **ThreadPool** incurs significant overhead to schedule a work item compared to processing one element in Parallel LINQ. Using **ThreadPool**, each element requires a delegate object, boxing the integer argument, and managing the thread queue itself. Though there are additional options that could be used to fine-tune the performance, the default method is considerably slower than the equivalent Parallel LINQ version. Also, the Parallel LINQ version of the code is significantly easier to read, understand and maintain.

Exercise 2 – Fine-Tuning PLINQ Queries

Once you have converted a query into its parallel equivalent, you can take advantage of advanced features to enable cancellation or to exercise fine control over the default heuristics.

Task 1 – Cancelling a PLINQ Query

In many long-running operations, there is a need to be able to cancel execution prior to completion. Perhaps the wrong parameters were passed, a network outage is discovered, or a user needs to update some entries before full processing. Cancelling such operations seems like a simple case of using a shared Boolean flag, but in reality the situation is more complex.

1. In **Solution Explorer**, double-click **Ex2.cs**
   At the start of this exercise, we will be looking at the same **IsPrime** method. Tasks will use the same idea of iterating over numbers and checking for primeness. Because of the potentially large number of numbers to test, it is entirely possible that a user would want to cancel the operation – especially if the user entered more zeros than intended. If you have worked with threading in the past, you may have implemented a shared Boolean flag to accomplish this, but multiple threads writing to shared variables is problematic and locking adds additional overhead. Additionally, there are two problems: the iteration itself should know when an exit is requested, and the work being done also needs to know. A cancellation is different from an error condition, yet either one ends execution early.
2. Look at the **Task1** method:

```
public static List<int> Task1()
{
    var primes = from item in
        ParallelEnumerable.Range(lowTest, totalTest)
                 where IsPrime(item)
                 select item;

    return primes.ToList();
}
```

3. Adding cancellation to the query itself consists of just a few steps. The first thing is to declare a **CancellationTokenSource** object. At the top of the **Ex2.cs** source file, add the following declaration:

```
public static CancellationTokenSource cts = new CancellationTokenSource();
```

4. Next we need to instruct the query to watch a token obtained from this token source for a cancellation request. In the **Task1** method, after the call to **Range()**, add the **WithCancellation** method passing it the token in the **CancellationTokenSource**. The line should now appear thus:

```
var primes =
    from item in ParallelEnumerable.Range(lowTest, totalTest)
        .WithCancellation(cts.Token)
```

5. Finally, we need a way for the user to actually trigger the cancellation. Switch to the **MainWindow.xaml.cs** file
6. In the **buttonStop_Click** method, add the following code:

```
Ex2.cts.Cancel();
Ex2.cts = new CancellationTokenSource();
```

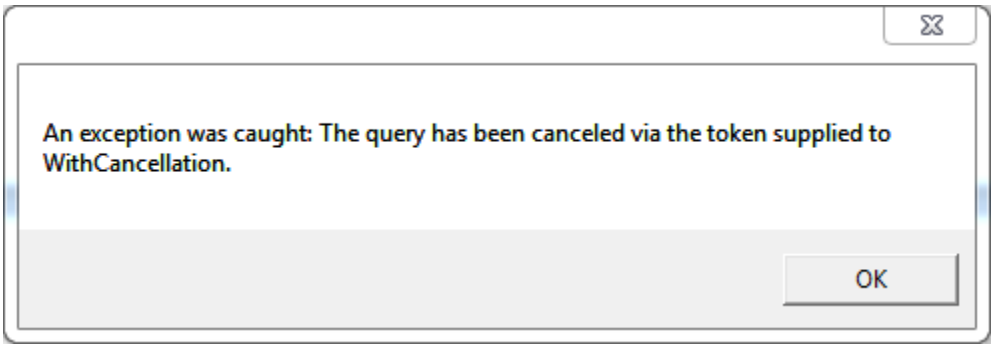
A token can't be reset to a non-cancelled state once it's been cancelled. Just to be safe, we create a new token source at this point so that it's ready for next time.

7. In the **w_DoWork** method, replace the code in the **try** block with the following:

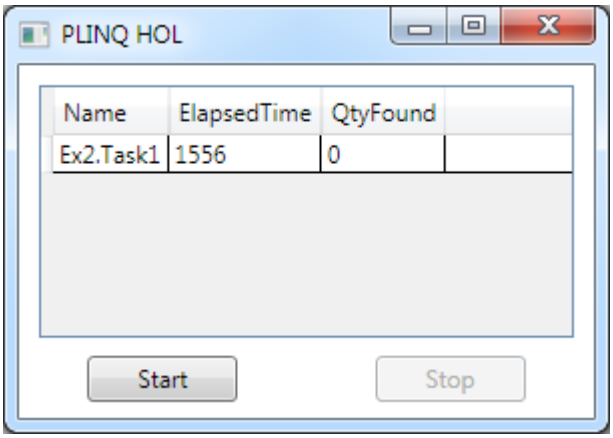
```
timer.Restart();
ReportProgress("Ex2.Task1", primes.Count, timer);
primes = Ex2.Task1();
timer.Stop();
ReportProgress("Ex2.Task1", primes.Count, timer);
primes.Clear();
```

8. Click the **Debug | Debugging** menu command (F5) to debug the application.
9. Click the **Start** button
10. After 2-3 seconds, click the **Stop** button

11. A dialog will appear with an exception based on the cancellation. Click **OK** to dismiss it.



As hoped, the application continues to run, but the query stops running. You can start it again and let it complete, or stop it again.



12. Close the test application when finished.

Task 2 – Cancelling Business Logic

Now that you know how to cancel queries you have a new tool to use. Sometimes, however, the query will take a long time to cancel. This is because cancellation does not affect what is happening within an iteration. Rather, it causes PLINQ to prevent assigning any more work to worker threads once it notices the cancellation request – something that is not even guaranteed to happen after every iteration. This has no effect on worker threads already busy. In other words, if each worker thread has a relatively long operation, they will each complete before cancellation potentially takes effect. This may not be satisfactory.

1. Switch back to **Ex2.cs**
2. The same **CancellationTokenSource.Token** can be used from within your own logic to know when to abort an operation. You just need to check the status from time to time. In the **IsPrime** method, add the following code within the **for** loop so that it appears thus (the new **else** clause:

```
for (int i = 3; i <= sq; i += 2)
{
    if (n % i == 0) return false;
    else ct.ThrowIfCancellationRequested();
}
```

3. Of course you will need to pass the **Token** to the **IsPrime** method as well, so change the method signature to appear thus:

```
public static bool IsPrime(int n, CancellationToken ct)
```

4. The query now needs to pass the **CancellationToken** to **IsPrime**, so modify the call in **Task1** thusly:

```
where IsPrime(item, cts.Token)
```

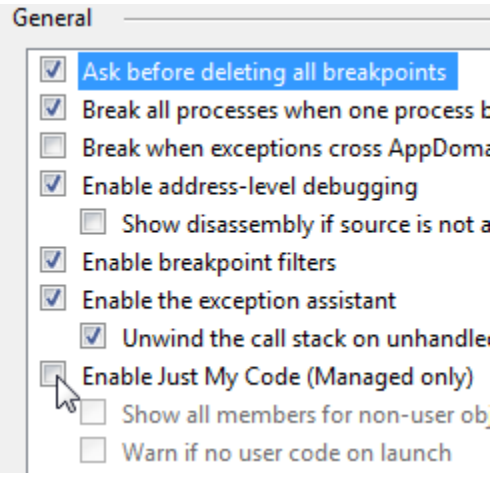
5. Now that our worker threads may throw the **OperationCanceledException**, we can choose to catch it. Add a **try/catch** block so that the entire method appears thus:

```
try
{
    var primes =
        from item in ParallelEnumerable.Range(lowTest,totalTest)
            .WithCancellation(cts.Token)
             where IsPrime(item, cts.Token)
             select item;

    return primes.ToList();
}
catch (OperationCanceledException e)
{
    Console.WriteLine(e.Message);
    return new List<int>();
}
```

Of course, in some cases, you may want the exception to propagate further, or to handle it in some other way. Simply returning a zero-size list makes it unclear if no primes were found or if an error occurred. You have a great degree of control over how cancellation affects your work.

6. When in Debug mode, the **OperationCanceledException** may break the program, even if it is handled by a **catch** block. You can examine the state of the running code and then continue execution by clicking **Continue**. To prevent the break, disable the **Just My Code** debugging by clicking the **Tools | Options** menu command, then navigating to the **Debugging | General** options and unchecking

Enable Just My Code. Then, click **OK**.



7. Click the **Debug | Start Debugging** menu command to debug the application.
8. Click the **Start** button
9. After a second, click the **Stop** button
   The operation is now able to quit even when in the middle of processing business logic. Exactly how to accomplish this will be different in every application. For example, in an iteration you can check every so many cycles, however if you periodically wait on other threads, you may not be awake to detect the request. The **WithCancellation** option will never abort an operation forcibly. This allows you to finish steps, clean up state, and exit gracefully.
10. Close the application

In real-world applications, cancelling is a challenging problem to solve that greatly depends on the nature of the work being done. Any intensive long-running operations should check for cancellation periodically. Transitions between states are also logical places to check.

Task 3 – Setting Execution Modes

One of the benefits of Parallel LINQ is its heuristics in determining the best way to execute your query. In some cases, it may actually determine that the query wouldn't benefit from parallelism and will execute it serially instead. For example, if a query contains only a single user delegate which does little work, it may appear that the overhead of spinning up worker threads will exceed the benefit. Since this isn't based on actual profiling though, you may have a better idea of how the query should be run.

1. In **Ex2.cs**, add the following code to create a new method:

```
public static List<int> Task3()
{
    var primes = from item in ParallelEnumerable.Range(lowTest, totalTest)
                 where IsPrime(item, cts.Token)
                 select item;
```

```
    return primes.ToList();
}
```

2. If you test a given algorithm with and without parallelization and don't see a speed increase, then you can force it by using the **WithExecutionMode** extension method. This can be chained from **AsParallel**, **WithCancellation**, or other extension methods on **ParallelQuery<TSource>**. The **WithExecutionMode** extension method takes an argument of the enumerated type **ParallelExecutionMode** with a value of either **Default** or **ForceParallelism**. Add the call so the **Task3** method appears thus:

```
public static List<int> Task3()
{
    var primes = from item in ParallelEnumerable.Range(lowTest, totalTest)
                 .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                 where IsPrime(item, cts.Token)
                 select item;

    return primes.ToList();
}
```

Because this method already runs in parallel, no speedup will be seen, so the method call will have no effect. In general, it is not recommended to simply use this call on every query. Use profiling first to determine if the methods in use can truly benefit from parallelism. Usually, if PLINQ determines that a query will run better sequentially, then forcing parallelism may actually slow it down. To see this in action, you can add a call to it in the **w_DoWork** method in **MainWindow.xaml.cs**.

Task 4 – Using Merge Options

So far, the labs have all used the **ToList** method on the query to evaluate the expression and return the results. For cases where you need to enumerate the query manually, you have control over how results are organized from worker threads. For example, you may want results as soon as they are available. On the other hand, you may want to wait until all results are available.

1. In **Ex2.cs**, create the following method:

```
public static void Task4()
{
    var primes = from item in ParallelEnumerable.Range(lowTest, totalTest)
                 where IsPrime(item, cts.Token)
                 select item;
}
```

2. Instead of returning the **List**, we will dump out first 50 elements. Add the following code to implement this:

```
int qty = 0;
foreach (int prime in primes)
{
    Console.WriteLine(prime);
    if (qty++ > 50) break;
}
```

3. Switch to **MainWindow.xaml.cs**
4. Since this method doesn't return a collection like the other implementations, a slight different is needed. In the **w_DoWork** method, replace the code after the **Insert code after this point** comment with the following:

```
timer.Restart();
ReportProgress("Ex2.Task4", -1, timer);
Ex2.Task4();
timer.Stop();
ReportProgress("Ex2.Task4", -1, timer);
primes.Clear();
```

5. Press F5 to start debugging
6. Ensure that the **Output** window is showing (click the **View | Output** menu command if it is not)
7. Click **Start** in the application UI

Almost immediately, you should see numbers appearing in the **Output** window. In some runs, you will see two different sequences interleaved, depending on processor count. In other runs the numbers may all be from the same sequence. The key here is that all numbers immediately appear even though the query is still executing. This could have a performance impact as the query is consuming CPU resources at the same time as your main thread is attempting to process the results. In order to force the query to complete first, you need to use the **WithMergeOptions** extension method and the **ParallelMergeOptions** enumerated type.

8. After the **Start** button re-enables, exit the application
9. Switch back to **Ex2.cs**
10. Add a new line to the query to specify merge options. The query should now appear thus:

```
var primes = from item in ParallelEnumerable.Range(lowTest, totalTest)
                 .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
             where IsPrime(item, cts.Token)
             select item;
```

11. Press F5 to start debugging, then click **Start** in the application UI
    You will observe a noticeable delay between clicking **Start** and seeing any results appear.

12. After the **Start** button becomes re-enabled, you can close the application

The default option in the **ParallelMergeOptions** enumeration is **AutoBuffered** which uses buffers to sometimes return results prior to the entire query completing. Note that the addition of operations such as ordering can partially negate the gain from a parallel implementation since no data can be returned until all results are complete and ordered. Using these options carefully can enable you to fine-tune your performance better.

Task 5 – Handling Exceptions

Exceptions are a fact of programming. Often, dealing with unhandled exceptions is what makes the difference between a good application and a buggy one. In a parallel processing mode, you may want to deal with exceptions in a central way. For example, in processing multiple data lookups, it's important to know the full exception detail for every failure. Instead of creating a complicated custom mechanism, Parallel LINQ already has a facility for it.

1. In **Ex2.cs**, add the following new method:

```
public static List<int> Task5()
{
    var primes =
        from item in ParallelEnumerable.Range(lowTest, totalTest)
                 where IsPrime(item, cts.Token )
                 select item;

    return primes.ToList();
}
```

2. In order to test exception handling, the first step is to throw an exception. In the **IsPrime** method, add the following line of code after the **Throw exception here** comment (before the **return** statement):

```
throw new Exception(string.Format("{0} is not evaluated", n));
```

3. Now, you will need to add exception handling to the **Task5** method. Surround the code block in a **try** block and add a **catch** block so that it matches the following:

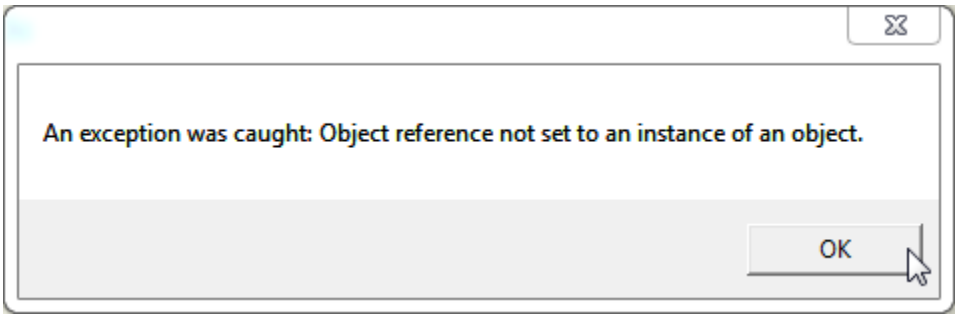
```
public static List<int> Task5()
{
    try
    {
        var primes =
            from item in ParallelEnumerable.Range(lowTest, totalTest)
            where IsPrime(item, cts.Token)
            select item;

        return primes.ToList();
```

```
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        return new List<int>();
    }
}
```

13. In **MainWindow.xaml.cs,** in the **w_DoWork** method, replace the code after the **Insert code after this point** comment with the following:

```
timer.Restart();
ReportProgress("Ex2.Task5", -1, timer);
primes = Ex2.Task5();
timer.Stop();
ReportProgress("Ex2.Task5", -1, timer);
primes.Clear();
```

4. The code is now ready to run. Press F5 to start debugging
5. When the application appears, click **Start**
6. After a second or two, the error will occur. Click **OK** to dismiss the error dialog:



7. This **Start** button will be re-enabled but the **QtyFound** column will contain "0". Note: The **Exception Helper** will appear if **Enable Just My Code** was not disabled previously.

In the **Output** window, among other status messages, you should see the message "One or more errors occurred." This is because with multiple threads running, each one is throwing an exception. Though the **Exception** class has the concept of an inner exception, this is a nested relationship. In parallel programming though, multiple exceptions are at the same level of relevance.

For this reason, there is a new type, **AggregateException** which contains a collection of any exceptions that have occurred.

8. Modify the **catch** clause so that it appears thus:

```
public static List<int> Task5()
{
```

```
    try
    {
        var primes =
            from item in ParallelEnumerable.Range(lowTest, totalTest)
                     where IsPrime(item, cts.Token )
                     select item;

        return primes.ToList();
    }
    catch (AggregateException e)
    {
        foreach (Exception ex in e.InnerExceptions)
        {
            Console.WriteLine(ex.Message);
        }
    }

    return new List<int>();
}
```

9. Now run the application again, then click the **Start** button
10. Press F5 as needed to resume after exceptions.
11. Finally, in the **Output** window, you should see the details of more than one exception. The number of exceptions displayed is highly dependent on how many threads are running and how many began executing before the first exception was shown.

By catching **AggregateException** you can gain more insight into issues beyond simply knowing that something crashed. Once an exception is thrown, the query will abort, but any further exceptions thrown before threads exit are preserved.

Exercise 3 – Extensibility

Though LINQ and Parallel LINQ are great implementations, they are not complete for every purpose. Fortunately you can add extra functionality yourself through the creation of new extension methods and custom providers.

Task 1 – Creating New Operators

In this task, see how you can augment your queries with your own extension methods that are just as easy to use in a query as the built-in operations.

Note: There isn't currently a way to create a custom Parallel LINQ operator, so we'll call **AsParallel**() on the results to take advantage of parallelism.

In a large dataset, you may not want to look at every element. For example, in a collection of students, perhaps you only want to query every 10th student. You could create a condition in the query to filter these out, but for reasons of expressiveness it is clearer to simply specify the sampling number. This also allows you to create a query that looks at a sampling of data which can be modified to look at a larger or smaller sample or the entire collection by only changing the extension method call.

1. From **Solution Explorer**, open **ExtensionMethods.cs**
   Extension methods actually create methods that are attached to existing classes. These work even if the class is private. You don't have access to any private or internal data, but it allows greater expressiveness in code.
2. Extension methods are available on any class, but to operate on a collection for querying, you need to extend **IEnumerable**. Create the following method:

```
public static IEnumerable<T> WithSampling<T>(this IEnumerable<T>
source, int every)
{
    int idx = 0;
    foreach (var item in source)
    {
        if (++idx % every == 0) yield return item;
        else continue;
    }
}
```

Without knowing the entire collection size (impossible on an enumerable type), it won't operate with a sampling percentage. Instead, it operates by skipping values based on a configurable **every** parameter – in other words, it's like saying "take ever 5th value." It iterates over the given **source** collection object, ignores **every** minus one item, then yields the next item. If too few items are passed in, none will be yielded.

3. Using the new extension method is just a matter of appending it to an enumerable type in your query.
4. From the **Solution Explorer**, open **Ex3.cs**
5. Add the following code:

```
public static void Task1()
{
    var values = from num in Enumerable.Range(1, 100).AsParallel()
                 where num < 50
                 select num;

    foreach (int val in values)
        Console.WriteLine(val);
}
```

As the code currently is, the numbers from 1 through 100 will be created in an enumeration then parallelized. The query will then discard numbers greater than or equal to 50. The **foreach** iteration then simply displays the numbers.

6. In **MainWindows.xaml.cs**, in the **w_DoWork** method, remove the code after the **Insert code after this point** comment, then add the following:

```
ReportProgress("Ex3.Task1", -1, timer);
Ex3.Task1();
```

This method sends information to the **Output** window, so be sure that it's visible while debugging.

7. Press F5 to begin debugging
8. Click **Start**
   Every value from 1-49 will be displayed. If we were performing more complex operations on these values, we might want to reduce the size of the dataset.
9. Close the **PLINQ HOL** application
10. Switch to **Ex3.cs**
11. In **Task1**, add the following call to **WithSampling** to the **Range** call so that it appears thus:

```
var values =
    from num in Enumerable.Range(1, 100).WithSampling(5).AsParallel()
```

12. Press F5 to begin debugging
13. Click **Start**
    Now, as expected, only values from 5-45 in multiples of five are appearing.
14. Close the **PLINQ HOL** application

By using multiple extension methods, you can create the ideal collection of input data for a query. This allows you to focus your conditions on business logic. In this example, instead of a condition of "where num < 50", it would have been even more appropriate to have added the **Take** extension method after **WithSampling** in order to reduce the number of returned items further.

Lab Summary

In this lab, we learned about Parallel LINQ. We started with different ways of processing collections of data and saw how to convert them to parallel implementations. We also learned about options to specify to achieve the best performance. We covered various tips and tricks to ensure data integrity and proper handling. Finally, we saw how extension methods could enable new operators and how providers work.