Hands-On Lab

Using .NET4 Parallel Extensions – Geometric Approximation of Pi

Lab version: 1.0.0

Last updated: 3/13/2010

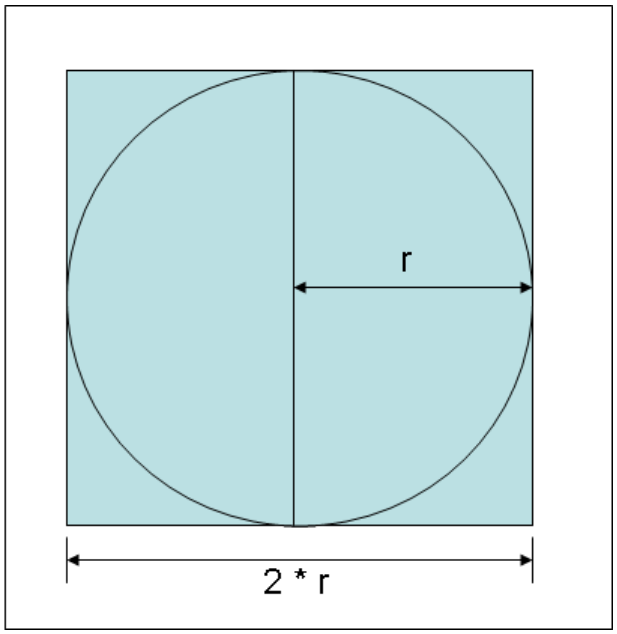
Introduction

Scenario

In the following set of exercises, you will implement an application that calculates PI by using a geometric approximation. Initially, you will perform the calculation in a single-threaded manner; then you will change the code to divide the calculation into discrete operations and perform the calculation by using thread pool threads. Finally, you will change the code to use the Task Parallel Library to perform each operation as a parallel task. In the process, you will uncover some data synchronization issues you need to address.

The algorithm you will implement calculates PI based on some simple mathematics and statistical sampling.

If you draw a circle of radius r and draw a square with sides that touch the circle, the sides of the square are 2 * r in length as shown in the following image:



The area of the square, S, is calculated as follows:

S = (2 * r) * (2 * r)

or

S = 4 * r * r

Rearranging, this gives:

r * r = S / 4

The area of the circle, C, is calculated as follows:

C = PI * r * r

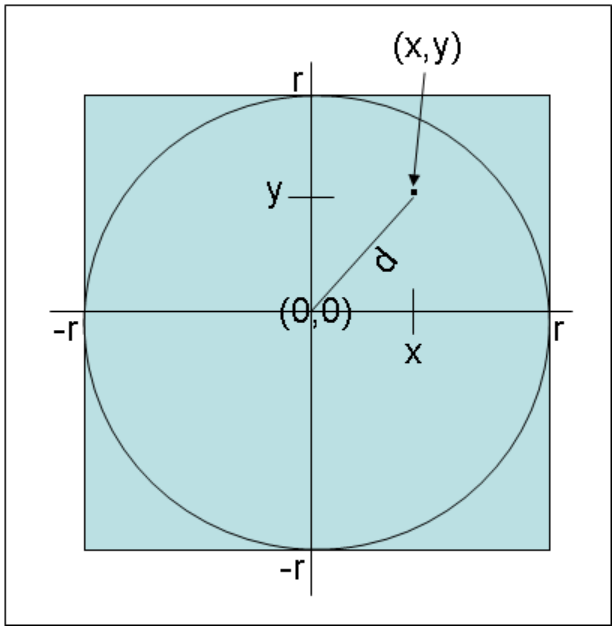
Substituting for r*r and rearranging gives:

PI = 4 * C / S

To calculate PI, the trick is to determine the value of the ratio C / S. This is where the statistical sampling comes in.

To do this, generate a set of random points that lie within the square and count how many of these points also fall within the circle. If you have generated a sufficiently large and random sample, the ratio of points that lie within the circle to the points that lie within the square (and also in the circle) approximates the ratio of the areas of the two shapes, C / S. All you have to do is count them.

How do you determine whether a point lies within the circle? To help visualize the solution, draw the square on a piece of graph paper with the center of the square at the origin, point (0, 0). You can then generates pairs of values, or coordinates, that lie within the range (-r, -r) to (+r, +r). You can determine whether any set of coordinates (x, y) lie within the circle by applying Pythagoras' theorem to calculate the distance d of these coordinates from the origin. You can compute d as the square root of ((x * x) + (y * y)). If d is less than or equal to r, the radius of the circle, then the coordinates (x, y) specify a point within the circle, as shown in the following diagram:



You can simplify matters further by generating only coordinates that lie in the upper right quadrant of the graph so that you only have to generate pairs of random numbers between 0 and r. This is the approach you will take in the exercises.

Exercise 1: Configure Event Tracing

In this example you will use the Visual Studio 2010 Parallel Performance Analyzer. This tool needs to install the system module symbols.

The analyzer uses Event Tracing for Windows (ETW) to gather information about system modules and gathers many values during an analyzer session. There is a possibility that some events will be dropped if the system is not configured to have sufficiently large ETW buffers. If the analyzer detects that kernel events have been missed it will issue a fatal error, if it determines that user events have been missed the analysis will complete but a warning will be issued. Either situations mean that the results gathered by the analyzer will not be accurate so the second task in this exercise is to change the size of these buffers so that all events are recorded.

▶ Set up symbols on the computer

1. Click the start button, click *All Programs*, and then click the *Microsoft Visual Studio 2010* folder.

2. Right-click *Microsoft Visual Studio 2010 - ENU to start V*isual Studio, and then click *Run as administrator*.

3. The User Account Control dialog appears.

4. If you are prompted to type a password used the password of an Administrator's account on the computer.

5. Click *Yes*.

   This will start Visual Studio 2010 and run it using the Administrator account. The Parallel Performance Analyzer requires the privileges of the Administrator account.

6. In Visual Studio, on the *Tools* menu, click *Options*.

7. In the *Options* dialog box expand the *Debugging* entry in the tree view control, and then click the *Symbols* entry.
   The right hand side of the dialog shows the symbol settings.

8. In the *Symbol file (.pdb) locations* box, check *Microsoft Symbol Servers*, and then click *OK*.
   This se*tting will ensure that the f*irst time the Perfo*rmance Analyzer* is run the symbols for the sy*s*tem files will be downloaded to the local symbol cache.

9. Close Visual Studio 2010.

▶ Change Event Tracing for Windows buffer size

1. Note: The ETW buffer size issue may be fixed in a future product release. The manual configuration steps are provided just in case they are needed.
2. Click the start menu. In the *Search programs and files* box type **regedit.exe**, and then press Enter.
   If the *User Access Control* dialog appear*s, click Yes* to indicate that you wish to run the program.

3. In the tree view move to the folder *HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0\VSPerf*.
   If this key does not have a key called *Monitor*, c*reate it by follow*ing these steps:
   - In the tree view click *VSPerf.*
   - *On the Edit menu, point to New*, and then click *Key*.
   - For the *name o*f this key type **Monitor**.

4. Click the *Monitor* key.

5. On the *Edit* menu, point to *New* and then click *Key*.

6. For the name of this key, type **EtwConfig**.

7. In the *EtwConfig* key create the following values with the specified contents. For each entry in the table complete the following steps:
   - Click *EtwConfig* in the tree view.
   - On the *Edit* menu, point to *New*, and then click *DWORD (32 bit) Value*.
   - For the name of the value use the name from the *Value Name* column in the following table.
   - On the *Edit* menu, click *Modify* to open the Edit *DWORD (32 bit) Value* dialog.
   - In the *Edit DWORD (32 bit) Value* dialog box, click the *Decimal* radio button.
   - In the *Edit DWORD (32 bit) Value* dialog box, in the *Value data* field, type the corresponding value from the *Value Contents* column in the following table.
   - Click *OK*.

Value Name	Value Contents
FlushTimer	0
BufferSize	256
MinBuffers	512
MaxBuffers	1024

8. Close the *Registry Editor*.

Exercise 2: Calculate PI Using a Serial Procedure

In this exercise, you will create the basic code to calculate the value of PI by using a single thread. You will run this code and measure how long it takes to perform the calculation. You will then run this code under the Parallel Performance Analyzer to learn how to use this tool.

► Add code to calculate PI by using a single thread

1. Click the start button, click *All Programs* and then click the *Microsoft Visual Studio 2010* folder.

2. Right click *Microsoft Visual Studio 2010 - ENU*, and then click *Run as administrator*.

3. The User Account Control dialog appears.

4. If you are prompted to type a password used the password of an Adminsitrator's account on the computer.

5. Click *Yes*.

6. Click *File*, point to *Open*, and then click *Project/Solution...*

7. In the *Open Project* dialog box move to the [*LABFILES*]*Starter**CalculatePI**CalculatePI – Step 1* folder, click *CalculatePI – Step 1.csproj*, and then click *Open.*

   The project and the containing solution are loaded in the Solution Explorer window.

8. In the *Solution Explorer* window, double-click *Program.cs* in the *CalculatePI – Step 1* project.

   The file opens in the *Code and Text Editor* window. This is a console application. The skeleton structure of the application has already been created for you.

9. Scroll to the bottom of the file, and examine the *Main* method. It looks like this:

```
static void Main(string[] args)
{
    Console.WriteLine(
        "Geometric approximation of PI calculated serially: {0}", pi);
    double pi = SerialPI();
    Console.WriteLine("PI = {0}", pi);
}
```

   This method calls the *SerialPI* method, which will calculate PI by using the geometric algorithm described previously. The value is returned as a *double* and displayed.

10. Scroll up and examine the *SerialPI* method.

```
static double SerialPI()
{
    List<double> pointsList = new List<double>(NUMPOINTS);
    Random random = new Random(SEED);
    int numPointsInCircle = 0;
    Stopwatch timer = new Stopwatch();
    timer.Start();

    try
    {
        // TO DO: Implement the geometric approximation of PI
        return 0;
    }
    finally
    {
        long milliseconds = timer.ElapsedMilliseconds;
        Console.WriteLine(
            "SerialPI complete: Duration: {0} ms", milliseconds);
        Console.WriteLine(
            "Points in pointsList: {0}. Points within circle: {1}",
```

```
            pointsList.Count, numPointsInCircle);
    }
}
```

This method generates a large set of coordinates and calculates the distances of each set of coordinates from the origin. The size of the set is specified by the constant *NUMPOINTS* at the top of the *Program* class. The bigger this value is, the greater the set of coordinates and the more accurate is the value of PI calculated by this method.

The method stores the distance of each point from the origin in the *pointsList* variable which is a *List<double>* collection. The data for the coordinates is generated by using the *random* variable. This is a *Random* object, seeded with a constant to generate the same set of random numbers each time you run the program. (This helps you determine that it is running correctly.) The *SEED* constant is declared at the top of the *Program* class.

You use the *numPointsInCircle* variable to count the number of points in the *pointsList* collection that lie within the bounds of the circle. The radius of the circle is specified by the *RADIUS* constant at the top of the *Program* class.

To obtain information about how long the procedure takes to complete, the code creates a *Stopwatch* variable called *timer* and starts it running. The *finally* block determines how long the calculation took and displays the result. The *finally* block also displays the number of items in the *pointsList* collection and the number of points that it found that lay within the circle.

You will add the code that performs the calculation to the *try* block in the next few steps.

11. In the *try* block, delete the comment and remove the *return* statement. (This statement was provided only to ensure that the code compiles.) Add the *for* block and statements shown next in bold to the *try* block:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin =
            Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    }
}
```

This block of code generates a pair of coordinate values that lie in the range *0* to *RADIUS,* and it stores them in the *xCoord* and *yCoord* variables. The code then uses Pythagoras' theorem to calculate the distance of these coordinates from the origin and adds the result to the *pointsList* collection.

Note Although there is a little bit of computational work performed by this block of code, in a real-world scientific application you are likely to include far more complex calculations that will keep the processor occupied for longer. To simulate this situation, this block of code calls another method, *doAdditionalProcessing*. All this method does is occupy a number of CPU cycles as shown in the following code sample.

```
private static void doAdditionalProcessing()
{
   Thread.SpinWait(SPINWAITS);
}
```

SPINWAITS is another constant defined at the top of the *Program* class.

12. In the *SerialPI* method, in the *try* block, add the *foreach* statement shown next in bold after the *for* statement:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        if (datum <= RADIUS)
        {
            numPointsInCircle++;
        }
    }
}
```

This code iterates through the *pointsList* collection and examines each value in turn. If the value is less than or equal to the radius of the circle, it increments the *numPointsInCircle* variable. At the end of this loop, *numPointsInCircle* should contain the total number of coordinates that were found to lie within the bounds of the circle.

13. Add the following statements shown in bold to the *try* block, after the *foreach* statement:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        ...
    }

    double pi = 4.0 * numPointsInCircle / NUMPOINTS;
    return pi;
}
```

These statements calculate PI based on the ratio of the number of points that lie within the circle to the total number of points, using the formula described earlier. The value is returned as the result of the method.

► Test the code

1. On the *Build* menu, click *Build CalculatePI – Step 1*.

Verify that the project builds without generating any errors (if there are any errors, check the code with the code given in the steps above).

2. On the *Debug* menu, click *Start Without Debugging*.

   The program runs and displays its approximation of PI. Write down the *Duration* value and the value for PI. The following text shows some typical output (your value for the Duration will probably be different from that shown)

```
Geometric approximation of PI calculated serially
ThreadPoolPI complete: Duration: 44893 ms
Points in pointsList: 10000000. Points within circle: 7853722
PI = 3.1414888
```

> **Note:** the value for PI will not be the actual value, it will be an approximation based on the parameters used in this code. However, this value is important because it will be the benchmark for the other exercises.

3. Press a key to close the console window.

► Analyze the code

1. In Visual Studio, on the *Analyze* menu, click *Launch Performance Wizard*.
2. O*n the Performance Wizard Page 1 of 3* page click *Concurrency*, check *Visualize the behavior of a multithreaded application*, and then click *Next*.

3. *On the Performance Wizard Page 2 of 3* page, confirm that *One or more available projects* is selected and that *CalculatePI – Step 1* is selected in the list box, and then click *Next.*

4. *On the Performance Wizard Page 3 of 3* page, confi*rm that Launch profiling after the wizard finishes* is checked, and then click *Finish*.

   The console application starts running.

   This process will take longer than before, so be patient. The analyzer will automatically close the console when the application has finished. Once the data has been gathered the analyzer will process the data. This may take a few minutes.

> **Note:** If this is the first time that the analyzer has been run the analysis process will take longer because the system module symbols are downloaded from the symbol server. These symbols are cached so that subsequent analyzer sessions will be much quicker.

   Finally you will see the *Concurrency Visualization* window.

5. When the *Concurrency Visualization* window appears, click *Toggle Full Screen*.

6. Click *Threads*.

   This screen shows all the threads used by the process, many of which are of no interest to you in this exercise. You should hide the threads that you do not need to see by using the following step:

- In the column to the left of the timeline right-click the thread that you wish to hide, for example, *Disk 1 Reads*, and then click *Hide*.

Repeat this step for the other *Reads* and *Writes* threads, and the *Debugger Helper Thread*. This will leave the *Main Thread*, *sev*eral *Thread Pool* threads, several *Worker* threads and a *CLR Worker Thread*.

7. Click the entry for the *Main Thread* in the left hand column of the timeline.

   The timeline for each thread changes color according to the thread state. In the bottom left hand corner the *Visible Timeline Profile* gives a legend for these thread states.

8. In the *Visible Timeline Profile* box, click the legend item for *Execution*.

   The *Profile Report* tab shows a tree view with the main modules used in the application.

9. Expand the tree node for *calculatepi.exe*.

   There should be an entry for *SerialPI* indicating that the method is executed on the *Main Thread*.

10. Examine the timeline. There is a slider called *Zoom* which you can use to change the magnification. Zoom in so that you can see more clearly the changing of state for the *Main Thread*.

    Notice that the thread shows that for most of the time the thread switches quite erratically between the *Execution* and *Preemption* state.

11. Adjust the zoom so that you can see two or three such *Execution* segments on the visible timeline. Click one.

    The *Current Stack* tab in the panel below the time line shows the stack, which should include the *SerialPI* method.

12. In the *Current View* drop down list, click *Cores*.

    This view shows a timeline for each processor core in your computer, and it displays the same time range that you set on the *Thread* view. You may see that the different segments of execution that you saw in the last step resulted in a switch between cores. If you find that the code is executing on only one core scroll to a different position where execution switches between cores and then switch back to *Threads* view to confirm that the execution is for the single thread.

13. In the *Threads* view window, click one of the threads marked as *Thread Pool*, and then look at the entries in the *Current Stack* tab.

Confirm that this thread is not used by the code in your program; it is used by the .NET Framework runtime. Do the same on the other threads in the time line and confirm that they too are used by the .NET Framework runtime and not by your code.

14. On the *View* menu, click *Full Screen* to return to the normal docking window view.

15. On the *File* menu, click *Close* to close the *Concurrency Analyzer* window.

    You can see from this exercise that a single thread was used to perform the work in *SerialPI* but this thread does not necessarily run on the same core throughout the execution of the process.

16. Close the *program.cs* window.

Exercise 3: Calculate PI Using a Thread Pool

In this exercise, you will take the code developed in the last exercise and divide it into operations that can be performed on several threads. You will test the code and show that, unless you are very careful, using the Thread Pool can lead to thread synchronisation issues.

► Add code to calculate PI by using ThreadPool threads

1. Select the starter project for this exercise, *CalculatePI – Step 2*, by following these steps:
   - In the *Solution Explorer* right click *CalculatePI – Step 2*, and then click *Set as Startup Project*.
   - In *Solution Explorer*, in the *CalculatePI – Step 2* project, double-click *Program.cs to display the file* in the *Code and Text Editor* window.

2. Locate the *ThreadPoolPI* method. It contains exactly the same code as the *SerialPI* method that you developed in the last exercise.

3. At the top of the *try* block add the following code shown in bold.

```
try
{
    int numThreads = 2 * Environment.ProcessorCount;
    int calcsPerThread = NUMPOINTS / numThreads;
```

This code determines how many thread pool threads will be used assuming on average that we want two threads per processor core. The code then divides the number of calculations evenly between the threads.

Note: This code is only for illustration purposes and is deliberately kept simple. In this example, the number of calculations can be divided equally between the threads because the number of calculations was selected to allow this. In a real case it may not be possible to evenly distribute the work like this, requiring more complicated code.

4. Add a **for** loop around the code that calculates PI by adding the following code shown in bold:

```
for (int poolThread = 0; poolThread < numThreads; poolThread++)
{
    for (int points = 0; points < calcsPerThread; points++)
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    }
}
```

   You have done two things here. Firstly, you have changed the inner *for* loop so that the loop repeats the number of times indicated by *calcsPerThread*. Secondly, you added the outer *for* loop so that the inner code is repeated the number of times indicated by *numThreads*.

5. Now add code to perform each inner calculation on a separate thread pool thread by adding the code below highlighted in bold.

```
for (int poolThread = 0; poolThread < numThreads; poolThread++)
{
    ThreadPool.QueueUserWorkItem((object o) =>
    {
        for (int points = 0; points < calcsPerThread; points++)
        {
            int xCoord = random.Next(RADIUS);
            int yCoord = random.Next(RADIUS);
            double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
            pointsList.Add(distanceFromOrigin);
            doAdditionalProcessing();
        }
    }, null);
}
```

   This code encloses the previous code as an anonymous delegate which is then passed to *QueueUserWorkItem* to initialize a *ThreadPool* thread.

6. On the *Build* menu, click *Build CalculatePI – Step 2*.

   Verify that the project builds without generating any errors (if there are any errors, check the code with the code given in the steps above).

7. On the *Debug* menu, click *Start Without Debugging*.

   The program runs but note that it stops immediately with zero items in *pointList*. The reason for this is that when *QueueUserWorkItem* is called the thread pool thread is started immediately but there is no code to indicate that the thread has completed its work. This means that the *for* loop initializing the thread pool threads completes very quickly and then the code to access the items in *pointList* is called before any calculations are performed, and finally the method returns and the application stops.

8. Press any key to close the console window, and return to Visual Studio.

9. Above the code to initialize the thread pool threads add the two lines shown below in bold.

```
AutoResetEvent signal = new AutoResetEvent(false);
int threadsStillWorking = numThreads;

for (int poolThread = 0; poolThread < numThreads; poolThread++)
{
```

This code creates an event synchronisation object that can be used to indicate when the main thread should perform the calculation to count the number of items in *pointsList* that are in the circle. The *threadsStillWorking* variable maintains a count of the thread pool threads that are working, and when this counter is decremented to zero you know that the calculation is complete.

10. Add the code shown below in bold to synchronize the main thread to the thread pool threads:

```
for (int poolThread = 0; poolThread < numThreads; poolThread++)
{
    ThreadPool.QueueUserWorkItem(delegate(object o)
    {
        for (int points = 0; points < calcsPerThread; points++)
        {
            ...
        }

        if (Interlocked.Decrement(ref threadsStillWorking) == 0)
        {
            signal.Set();
        }
    }, null);
}

signal.WaitOne();

foreach (double datum in pointsList)
{
    if (datum <= RADIUS)
    {
        numPointsInCircle++;
    }
}
```

The call to *WaitOne* is performed on the main thread and it blocks until the event synchronization object is signalled. When the anonymous delegate code has completed the associated thread will stop working, and to indicate that there is one less thread working *Interlocked.Decrement* is called to reduce the value of the *threadsStillWorking* counter in a thread-safe manner. When this counter reaches zero the event is signalled to allow the main thread to continue.

11. On the *Debug* menu, click *Start Without Debugging*.

The program runs. The following text shows some typical output (your results may be different):

```
Geometric approximation of PI calculated in parallel with the Thread Pool
ThreadPoolPI complete: Duration: 23797 ms
```

```
Points in pointsList: 9999976. Points within circle: 9230594
PI = 3.6922376
```

The result of the *ThreadPoolPI* method does not look right. There are two problems; first, there are less than *NUMPOINTS* items in *pointsList*, and second, the value of PI is not the same as obtained before. The random number generator is seeded with the same value as that used by the *SerialPI* method in the previous exercise, so the *Random* object should produce the same sequence of random numbers with the same result and the same number of points within the circle. Consequently, *ThreadPoolPI* should produce the same result as *SerialiPI*.

Note If the pointsList collection actually contains *NUMPOINTS* items, run the application again. You should find that it contains fewer items than expected in most (but not necessarily all) runs.

12. Press a key to close the console window, and return to Visual Studio.

    The cause of the problems is that the types used by *ThreadPoolPI* are not thread-safe, and more than one thread may attempt to modify data managed by these types simultaneously.

13. Examine the code in the anonymous delegate executed on each thread pool thread and identify the statements that reference items that are accessed by multiple threads.

    The following code shows the offending statements highlighted in bold. In this example, the **Next** method of the **Random** class and the **Add** method of the **List** collection both manipulate shared data types. Neither the **Random** class nor the **List** collection are thread-safe, so if these statements are run concurrently by two or more threads, they can become corrupt.

```
for (int points = 0; points < calcsPerThread; points++)
{
    int xCoord = random.Next(RADIUS);
    int yCoord = random.Next(RADIUS);
    double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
    pointsList.Add(distanceFromOrigin);
    doAdditionalProcessing();
}
```

14. Correct the synchronisation issue with the **List** collection first. After the declaration of the event object declare a semaphore object by adding the code highlighted in bold below.

```
AutoResetEvent signal = new AutoResetEvent(false);
int threadsStillWorking = numThreads;
SemaphoreSlim semaphore = new SemaphoreSlim(1);

for (int poolThread = 0; poolThread < numThreads; poolThread++)
{
```

    The *SemaphoreSlim* class is a new lightweight implementation of a semaphore provided by the .NET Framework 4.0.

15. Use this semaphore to protect the collection from concurrent access by adding the code highlighted in bold in the following code.

```
semaphore.Wait(-1);
```

```
pointsList.Add(distanceFromOrigin);
semaphore.Release();
doAdditionalProcessing();
```

The first line blocks the current thread until the semaphore is free, and which point the current thread gains exclusive access. The parameter of -1 means that this method call will wait infinitely if necessary. After the thread sensitive code has run the code releases the semaphore object so another thread can gain ownership.

16. On the *Debug* menu, click *Start Without Debugging*.

    The number of items in *pointsList* is now the same as *NUMPOINTS* showing that all attempts to put items in the collection are successful.

    Look at the value for PI and compare it to the value you obtained in the previous exercise using *SerialPI*. The two values still do not agree. The reason is because the code calls the *Next* method of a *Random* object which is not thread-safe and so can result in two threads obtaining the same value. There is no concurrent version of the *Random* class, so you must serialize access to the *Next* method. Since the code in the anonymous method is short it is possible to use the same semaphore object to synchronize the entire code block.

17. Press a key to close the console window, and return to Visual Studio.

18. Move the line that waits for ownership of the semaphore object from its current position before the access to *pointsList* to above the first call to *Random.Next* as highlighted in bold in the following code.

```
semaphore.Wait(-1);
int xCoord = random.Next(RADIUS);
int yCoord = random.Next(RADIUS);
double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
pointsList.Add(distanceFromOrigin);
semaphore.Release();
doAdditionalProcessing();
```

19. On the *Debug* menu, click *Start Without Debugging*.

    Compare the value for PI obtained with this method with the value obtained by *SerialPI* and confirm that the two values agree. Make a note of the time taken and compare this with the value for *SerialPI*. You should find that the *ThreadPoolPI* method executed in less time than *SerialPI* did.

20. Press a key to close the console window, and return to Visual Studio.

► Analyze the code

1. In Visual Studio, on the *Analyze*, click *Launch Performance Wizard.*

2. On the *Performance Wizard Page 1 of 3* page, click *Concurrency*, check *Visualize the behavior of a multithreaded application*, and then click *Next*.

3. On the *Performance Wizard Page 2 of 3* page, confirm that *One or more available projects* is selected and that *CalculatePI – Step 2* is selected in the list box, and then click *Next.*

4. *On the Performance Wizard Page 3 of 3* page con*firm that Launch profiling after the wizard finishes* is checked, and then click *Finish.*

5. Wait while the console application runs.

6. When the *Concurrency Visualizer* window appears, click *Toggle Full Screen*.

7. Click *Threads*. Only s*how the Main Thread* item and the *CLR Worker Thread* items are relevant for this analysis. Hide the other threads.

   The *Thread Pool* items shown in the timeline are threads in the unmanaged thread pool for this process. In this exercise we are only interested in managed thread pool threads and these appear in the thread list as *CLR Worker Thread* items.

8. Examine the timeline for the *Main Thread* in relation to the *CLR Worker Thread* items that represent the thread pool threads.

   Notice that the majority of the *Main Thread* timeline is spent in the *Synchronization* state. During this time the *CLR Worker Thread* items are in a mixture mostly of *Execution*, *Synchronization* and *Preemption* states. There may also be one or more *CLR Worker Thread* items that are mostly in the *Synchronization* state, these are used by the .NET infrastructure and are not relevant to this exercise.

9. Click the *Synchronization* portion of the *Main Thread* so that it is highlighted and examine the *Current Stack* tab. Confirm that the synchronization was cause by the call to *Threading.WaitHandle.WaitOne* in the *ThreadPoolPI* method.

10. Right-click the following entry, and then click *View Source*:

```
calculatepi.exe!CalculatePI.Program.ThreadPoolPI
```

    The *Program.cs* file opens in the *Code and Text Editor* window and the following line, highlighted in bold, is highlighted in the *Code and Text Editor* window:

```
signal.WaitOne();
```

    This shows that the method call, *WaitOne*, blocks the *Main Thread* thread for the majority of the application execution time. A blocked thread is a thread that takes up resources but does no work.

11. Return to the *Concurrency Visualizer* window.
12. Notice that at the end of the highlighted portion of the *Main Thread* item there is a vertical bar, this bar starts at the Main Thread item and ends on one of the *CLR Worker Thread* items. This bar indicates the code that stopped the blocking of the *Main Thread* item.

At this point the *CLR Worker Thread* item is at the end of a section of *Execution* and all the other *CLR Worker Thread* items have finished all their execution and are in the *Synchronization* state. After this point in time the *Main Thread* thread started executing again. This shows that the last thread pool thread completed its work and called the *Set* method on the event object.

13. Zoom in so that you can see a section that shows some of the *CLR Worker Thread* items in the *Execution*, *Synchronization* and *Preemption* states.

    Notice that there is a checkerboard pattern of these states. When one thread enters the *Pre*-emption state another thread starts executing.

14. In the left hand column click, one of the *CLR Worker Thread* items. In the *Visible Timeline Profile* panel click *Execution*.

15. On the *Profile Report* tab expand the *calculatepi.exe* node.

    This entry shows a line with the name of a method in the *Program* class. The name of this method will not be familiar to you; it will look something like the following:

```
CalculatePI.program.<>c__DisplayClass4.<ThreadPoolPI>b__0
```

16. Right click this method, and then click *View Source*.

    The cursor should be placed on the opening brace of the anonymous delegate executed by the *ThreadPool* thread.

17. Return to the *Concurrency Vizualizer* window.

18. Click one of the synchronization sections in the timeline for one of the *CLR Worker Thread* items. Notice that the *Current Stack* tab shows that the cause of this synchronization is a call to *Wait* method on a *SemaphoreSlim* object.

    These steps show that there are two mechanisms involved here.

    - The operating system schedules the threads to run and since all the *CLR Worker Thread* objects have the same priority the operating system gives each thread a similar amount of time to run. A thread is placed in a *Preemption* state when the operating system decides to give another thread some time to execute.
    - Sometimes two threads will attempt to access the code protected by the semaphore object at the same time. At this point one thread will execute and the other thread will be placed in the *Synchronization* state. This is thread contention.

19. Close the *Concurrency Visualizer* window.

20. On the *View* menu, click *Full Screen*.

Exercise 4: Calculate PI Using Task Parallelization

In this exercise you will add code to perform the routine to calculate PI by using the Task Parallel Library. You will examine the code in the concurrency analyzer to determine the source of the improved performance of this new code.

▶ Add code to calculate PI in Parallel Tasks

1. Load the starter project for this exercise, *CalculatePI – Step 3*, by following these steps:
   - On the *Window* menu, click *Close All Documents* to close all the code files currently open.
   - In the *Solution Explorer*, right-click *CalculatePI – Step 3*, and then click *Set as Startup Project*.
   - In *Solution Explorer*, in the *CalculatePI – Step 3* project, double-click *Program.cs* to display the file in the *Code and Text Editor* window.

2. At the top of the file, add the following statement shown in bold to bring the **System.Threading.Tasks** namespace into scope. This namespace contains the types for the Task Parallel Library.

```
using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

3. Locate the *ParallelTasksPI* method. It contains exactly the same code as the *SerialPI* method that you developed in the second exercise.

4. In the *try* block replace the *for* statement with the *Parallel.For* statement as shown next in bold:

```
try
{
    Parallel.For (0, NUMPOINTS, (x) =>
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
```

This construct is the parallel analog of the code in the *for* loop in the *SerialPI* method. The body of the original *for* loop is wrapped in a lambda expression.

The Task Parallel Library code sections the work into tasks, and allocates these tasks to threads. However, unlike the *ThreadPoolPI* method in the last exercise, the Task Parallel Library determines how many threads to create based on the current workload, the memory available, the number of CPUs available, and the availability of other critical system resources; you do not have to make decisions about how many threads should be used to run these tasks.

The Task Parallel Library does not guarantee the order that each task is executed.

> **Note** Although each iteration of the For loop runs as a separate task, it is unlikely that the Task Parallel Library will create a new thread for each iteration (unless you have a computer with a large number of processors and a good amount of memory). Instead, the Task Parallel Library will create sufficient threads to keep the processors in your computer occupied. This may or may not be the same as the number of CPUs in your computer, as the Task Parallel Library can dynamically adjust the number of threads used based on the work being performed by each thread.

Notice that you have a similar situation to the *ThreadPoolPI* method; several threads can attempt to concurrently access data structures that are not thread-safe.

5. Synchronize access to these objects by using a *SemaphoreSlim* object. Add the code highlighted in bold in the following code.

```
try
{
    SemaphoreSlim semaphore = new SemaphoreSlim(1);

    Parallel.For (0, NUMPOINTS, (x) =>
    {
        semaphore.Wait(-1);
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        semaphore.Release();
        doAdditionalProcessing();
    });
}
```

This is all you need to do to parallelize the code.

6. On the *Debug* menu, click *Start Without Debugging*.

Verify that the value for PI obtained with this method is the same as obtained by *SerialPI* and *ThreadPoolPI*. Make a note of the time taken and compare this with the values for *SerialPI* and *ThreadPoolPI*. In most cases you will find that the *Parallel TasksPI* method executed in less time than either of the other previous methods, although the speed improvement over the *ThreadPoolPI* method may be marginal. However, the key advantage of the Task Parallel Library is the ease of programming over using the thread pool manually, and the automatic scalability that the abstractions provided by the Task Parallel Library provide.

7. Press a key to close the console window, and return to Visual Studio.

▶ Analyze the code

1. In Visual Studio, on the *Analyze* menu, click *Launch Performance* Wizard.

2. *On the Performance Wizard Page 1 of 3 page* click *Concurrency*, check *Click Visualize the behavior of a multithreaded application*, and then click *Next*.

3. On the *Performance Wizard Page 2 of 3* page, confirm that *One or more available projects* is selected and that *CalculatePI – Step 3* is selected in the list box, and then click *Next.*

4. On the *Performance Wizard Page 3 of 3* page, con*firm that Launch profiling after the wizard finishes is checked* and then click *Finish.*

5. Wait while the console application runs.

6. When the *Concurrency Visualizer* window appears, click *Toggle Full Screen*.

7. Click *Threads*. Hide all threads except the *Main Thread* item and the *CLR Worker Thread* items.

8. Examine the timeline for the *Main Thread* item.

   The most apparent difference is that the main thread is not simply blocked, as was the case with the thread pool exercise. Instead it shows the same checkerboard pattern as was the case for the thread pool threads in the last exercise, with periods in the *Synchronization*, *Execution* and *Preemption* states.

9. Click the *Main Thread* item in the left hand column (do not click the timeline). In the *Visible Timeline Profile* panel, click *Execution*.

10. In *Execution Profile* on the *Profile Report* tab expand the *calculatepi.exe* to show the *Main* method and then expand the *Main* node to show the code executing in the program.

    Verify that there are two branches under this node, one corresponding to the call to *Parallel.For* and the other with a generated name that corresponds to an anonymous delegate. The anonymous delegate will have a name similar to the following:

    ```
    CalculatePI.program.<>c__DisplayClass3.<ParallelTasksPI>b__0
    ```

11. Right-click the anonymous delegate method in the *Execution Profile* panel, and then click *View Source*.

    Confirm that the cursor is positioned on the opening brace of the lambda expression used to initialize the parallelized tasks.

    This shows that the Task Parallel Library uses CLR thread pool threads and the process main thread. When a thread is blocked it performs no work and so cannot contribute to the performance of the application. The Task Parallel Library attempts to minimize such blocking and this is one of the reasons for the improved performance over the thread pool example earlier in this lab.

12. On the *View* menu, click *Full Screen*.

13. On the *File* menu, click *Exit* to close Visual Studio.

When you see the *Microsoft Visual Studio* dialog requesting *Save changes to the following items?* click *No*.