|  |
| --- |
|  |
| Parallel Programming with .NET Framework 4.0 |
|  |
| Computation of π Value with Parallel Methodology in .NET |

**INSTRUCTOR: ASST. Prof. dr. Cem ÖZDOğan**

January 14, 2011

Authored by: Atakan Özgün - 200411040

# Abstract

My CENG 471 Parallel Programing course’s project is about Parallel Programming with .NET Framework 4.0. This report covers some of aspects and benefits of parallel programming methodology in .NET. Also it presents how to compute value of pi with parallel .NET.

# Introduction

 Parallel Programming with .NET Framework 4.0 provides new runtime and new class library tools that makes parallelization more efficient and easier than before. In .NET the programmer do not has to work with directly threads or thread pool, which means it requires higher – level manipulations with threads and data.

 Parallelization with .NET 4.0 is a shared memory programming. It comes with 2 main libraries: Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ). Task Parallel Library is responsible for data and task parallelism, Parallel Language Integrated Query is the parallel implementation of LINQ to objects.

 TPL is a public API to System.Threading.Parallel and System.Threading.Tasks namespaces, in which involves 2 main classes: Task and Parallel. In Task class, task operations are defined. In Parallel class, there some are functions like Parallel.For and Parallel.Foreach which provides parallelization for loops. Also, there are some task options defined like implicit creation of a task.

PLINQ is a parallel implementation of LINQ to Objects. PLINQ implements the complete set of LINQ standard query operators as extension methods for the System.Linq namespace and has additional operators for parallel operations. PLINQ queries are also scale in the degree of parallelism based on the capabilities of the host computer.

# Computation of π Value

 Calculation of PI becomes major mathematical phenomenon throughout history. After technology developed and more precision which can’t be reached by hand, for pi value is needed; computers are used. There are many implementations of many equations that each one of them gives a precision in a particular range to compute pi. Today the pi was calculated to 5 trillion [1] digits by a homemade super computer with specifically implemented software called y – cruncher that uses very complex formulas and algorithms. [2]

In my lab project I didn’t needed those astronomical numbers, so I used an approach similar to we used in our lab section. The algorithm to compute pi used here is based on generating random numbers in a unit length square and counting the number of points that fall within the largest circle inscribed in the square. Since the area of the circle (πr2) is equal to π/4, and the area of the square is 1 × 1, the fraction of random points that fall in the circle should approach π/4. The method gives a precision up to 130 digits. [3]

Implementation of the method in C# is below:

With For loop;

public static double SerialForPi(int numSteps)

{

double sum = 0.0;

double step = 1.0 / (double)numSteps;

for (int i = 0; i < numSteps; i++)

{

double x = (i + 0.5) \* step;

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

}

return step \* sum;

}

With LINQ;

public static double SerialLinqPi(int numSteps)

{

double step = 1.0 / (double)numSteps;

return (from i in Enumerable.Range(0, numSteps)

let x = (i + 0.5) \* step

select 4.0 / (1.0 + x \* x)).Sum() \* step;

}

# Parallel Computation of π with .NET

Parallelization of pi value’s computation is very simple with .NET Framework. Code segments and explanations as follow **(See appendix for tables and graphs).**

## Using Parallel Class

For loop is parallelized with this class. Code is below.

public static double ParallelForPi(int numSteps)

{

double sum = 0.0;

double step = 1.0 / (double)numSteps;

object monitor = new object();

Parallel.For(0, numSteps, () => 0.0, (i, state, local) =>

{

double x = (i + 0.5) \* step;

return local + 4.0 / (1.0 + x \* x);

}, local => { lock (monitor) sum += local; });

return step \* sum;

}

Method Definition:

public static ParallelLoopResult For<TLocal>(

int fromInclusive,

int toExclusive,

Func<TLocal> localInit,

Func<int, ParallelLoopState, TLocal, TLocal> body,

Action<TLocal> localFinally

)

* *(int)* **fromInclusive:** The start index, inclusive.
* *(int)* **toExclusive:** The end index, exclusive.
* *(Func<TLocal>)* **localInit:** The function delegate that returns the initial state of the local data for each thread.
* *(Func<int, ParallelLoopState, TLocal, TLocal>)* **body:** The delegate that is invoked once per iteration.
* *(Action<TLocal>)* **localFinally:** The delegate that performs a final action on the local state of each thread.

Each for iteration has a local sum that is added at the end of loop to the global sum.

## Using PLINQ

 LINQ query is parallelized with creation of enumerable range when defining iteration boundary. The commented line is another method for parallelization for the method. However, it is not efficient as used one. Also .WithDegreeOfParallelism<> provides a flexibility to decide how many cores or processors do we want to use.

public static double ParallelLinqPi(int numSteps, int numProcs)

{

double step = 1.0 / (double)numSteps;

return (from i in ParallelEnumerable.Range(0, numSteps) // faster

//from i in Enumerable.Range(0, numSteps).AsParallel()

.WithDegreeOfParallelism(numProcs)

let x = (i + 0.5) \* step

select 4.0 / (1.0 + x \* x)).Sum() \* step;

}

## Using Tasks

 Using tasks library is not crucial in this project. However, I used it anyway because I saw that it makes the program run more smoothly.

There are 3 tasks in this code and all of them are created via Factory class. First task is manages the function that calculates pi. Second task is a kind of scheduler that takes over after first task and makes some changes in user interface. If there is an error, it’s been thrown in this task. Third task is a child task of the first one. It basically writes pi and elapsed time values to the related labels. It also arranges the label values by showing the related label after task finishes. Normally, you have to wait for all procedure. Task codes are below:

// Nested Tasks

private void createTasks(int numSteps, int numProcs)

{

// Parent Task

var piTasks = Task.Factory.StartNew(() =>

{

double piValue;

TimeSpan elapsed;

// No tasks are defined here. Just send actions (code between brackets) to the Work method.

// Only setValue method creates a child task attached to parent.

elapsed = Work(() =>

{

piValue = ComputePi.SerialForPi(numSteps);

setValue(lblSerialForPI, piValue.ToString());

});

setValue(lblSerialForTime, elapsed.ToString());

elapsed = Work(() =>

{

piValue = ComputePi.ParallelForPi(numSteps);

setValue(lblParallelForPI, piValue.ToString());

});

setValue(lblParallelForTime, elapsed.ToString());

elapsed = Work(() =>

{

piValue = ComputePi.SerialLinqPi(numSteps);

setValue(lblSerialLINQPI, piValue.ToString());

});

setValue(lblSerialLINQTime, elapsed.ToString());

elapsed = Work(() =>

{

piValue = ComputePi.ParallelLinqPi(numSteps, numProcs);

setValue(lblParallelLINQPI, piValue.ToString());

});

setValue(lblParallelLINQTime, elapsed.ToString());

}, TaskCreationOptions.AttachedToParent);

// Attached (Continued) Task

var finish = piTasks.ContinueWith(resultTask =>

{

progressBar1.Visible = false;

btnCalculate.Enabled = true;

if (resultTask.IsFaulted)

{

MessageBox.Show(resultTask.Exception.ToString());

}

}, CancellationToken.None, TaskContinuationOptions.AttachedToParent, \_uiScheduler);

}

// Child Task

private void setValue(Label l, String s)

{

Task.Factory.StartNew(() =>

{

l.Text = s;

}, CancellationToken.None, TaskCreationOptions.AttachedToParent, \_uiScheduler);

}

# Summary

 Parallel library of .NET Framework 4.0 brings a practical approach for programmers who develop software with C# and .NET. It’s a little bit complex, but parallelization of LINQ to objects with PLINQ is a huge plus.

# References

* Parallel Programming in .NET Framework, MSDN Library

[http://msdn.microsoft.com/en-us/library/dd460693(VS.100).aspx](http://msdn.microsoft.com/en-us/library/dd460693%28VS.100%29.aspx)

* Task Parallel Library, MSDN Library,

<http://msdn.microsoft.com/en-us/library/dd460717.aspx>

* Task Parallelism, , MSDN Library,

<http://msdn.microsoft.com/en-us/library/dd537609.aspx>

* Data Parallelism, MSDN Library,

<http://msdn.microsoft.com/en-us/library/dd537608.aspx>

* Parallel LINQ (PLINQ), MSDN Library,

<http://msdn.microsoft.com/en-us/library/dd460688.aspx>

* Lambda Expressions, MSDN Library,

<http://msdn.microsoft.com/en-us/library/dd460699.aspx>

Citations

1. McCormick Grad Sets New Pi Record, North Western University, <http://www.mccormick.northwestern.edu/news/articles/article_743.html>
2. Numerical Approximation of pi, Wikipedia, [http://en.wikipedia.org/wiki/Numerical\_approximations\_of\_π](http://en.wikipedia.org/wiki/Numerical_approximations_of_%CF%80)
3. Language and Algorithms, Y – Cruncher, Number World,

<http://www.numberworld.org/y-cruncher/algorithms.html>

# Appendix

## For vs. Parallel For

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Iterations** | **100000000** | **200000000** | **300000000** | **400000000** | **500000000** |
| **For** | 2,1468676 | 4,2943751 | 6,3688588 | 8,4529540 | 10,5321711 |
| **Parallel.For** | 1,0628901 | 2,1405434 | 3,2144211 | 4,3752545 | 5,5206022 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **100000000** | **200000000** | **300000000** | **400000000** | **500000000** |
| **Speed Up** | 2,0198397 | 2,0062079 | 1,9813393 | 1,9319914 | 1,9077939 |
| **Efficiency** | 1,0099198 | 1,0031040 | 0,9906696 | 0,9659957 | 0,9538969 |

## LINQ vs. PLINQ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Procs** | **100000000** | **200000000** | **300000000** | **400000000** | **500000000** |
| **1** | 5,4897684 | 10,5839385 | 16,1879512 | 21,4278669 | 27,0589289 |
| **2** | 3,0862212 | 5,9359600 | 9,0236640 | 11,9426365 | 15,0535736 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **100000000** | **200000000** | **300000000** | **400000000** | **500000000** |
| **Speed Up** | 1,7787994 | 1,7830205 | 1,7939444 | 1,7942325 | 1,7975087 |
| **Efficiency** | 0,8893997 | 0,8915103 | 0,8969722 | 0,8971163 | 0,8987543 |

Notes About Appendix

* LINQ and PLINQ with parallelization degree (procs) 1 are nearly same. So I didn’t use LINQ data.
* My computer has only 2 cores so as the graphs and tables.