# CS 315 LAB #1 Threads and Synchronization

## NAME: DUE DATE: Sep 15 .

**GRADE:**

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| EX01\_01: Vector Sum with C++11 Threads | 24 | 25 |
| EX01\_02: Vector Sum with PPL | 24 | 25 |
| EX01\_03: Estimate PI with C++11 Threads | 25 | 25 |
| EX01\_03: Estimate PI with PPL | 25 | 25 |
| **TOTAL** | 98 | 100 |

**EX01\_01 and EX 01\_02 What if the DIM\_SIZE is not evenly divisible by the number of threads… how will the extra elements be added together?**

**Learning Goals for this Exercise**

* Practice the use of C++11 threads and Windows Parallel Patterns Library.

## Vector Operations

Vectors are used extensively in mathematics, physics, engineering, and computer science. A vector is just a list of numbers, each of which describes a quantity in an orthogonal-axis of some geometric space, e.g. (4, 6, 2).

We use vectors to describe the direction between two points in a space, by specifying a value obtained from subtracting the values of the 2 points along each of two non-parallel axes (typically our axes are orthogonal).

**For example**: given 2 points in a three-dimensional space described by A (1, 1, 1) and B (2, 3, 5), the directional vector AB is given by (2-1, 3-1, 5-1) 🡪 (1, 2, 4). We call this component-wise subtraction operation **vector subtraction**.

Now given any 2 directional vectors, we can also perform a component-wise sum of each value in the vectors to obtain another vector that describes the overall, combined direction. We call this component-wise addition operation **vector addition**.

Graphically, we illustrate this with the 2 directional vectors ***a*** and ***b***, and the combined vector (***a+b***) as shown below:

***a***

***b***

***a + b***

(a1, a2, …. an)

+ = (a1+b1, a2+b2, …, an+bn)

(b1, b2. …, bn)

Engineers and computer scientists use vectors to model complex phenomena, often requiring the use of very high-dimensional state spaces. These state spaces can often contain 1000s of dimensions. Consequently, in such cases, vector addition and subtraction may need to be performed on very large vectors.

For our first lab exercise, you will need to parallelize vector addition and subtraction using threads and PPL.

### Preliminary Setup

Copy [\\cs1\CS\_ClassData\315\Labs\LAB01\_01](file:///\\cs1\CS_ClassData\315\Labs\LAB01_01) to [\\CS1\home\<yourusername>\CS315-1\LAB01\_01\](file:///\\CS1\home\%3cyourusername%3e\CS315-1\LAB01_01\)

This folder contains a Visual Studio project with vector addition and subtraction implemented in the functions **vector\_add** and **vector\_subtract**. Open, compile and run the visual studio project to see how it works.

**Without modifying the main function**, your task is to parallelize **vector\_add** and **vector\_subtract** using **10** **threads**.

### Parallelization strategy

You will need to create two **lambda functions,** one for vector addition, and the other for subtraction**.**

Each lambda function should be defined in the **vector\_add** and **vector\_subtract** functions respectively. Additionally, the functions will need to **capture** the input and output vector variables, i.e. **a**, **b**, and **result**. Finally, the lambda function should be designed in such a way as to allow it to **only update a portion of the output vector**, depending on its thread ID, in the range 0 to 9.

In the function **vector\_add** and **vector\_subtract**, spawn 10 threads to execute your lambda function responsible for implementing the relevant vector operation.

Please attempt to design a solution first, and then look at the **hint** below if you are stuck.

### EX01\_01:

Copy any code from the startup project to a new visual studio project stored in [\\CS1\home\<yourusername>\CS315-1\LAB01\_01\](file:///\\CS1\home\%3cyourusername%3e\CS315-1\LAB01_01\)

Use **C++11 threads** in your implementation.

### EX01\_02:

Copy any code from the startup project to a new visual studio project stored in [\\CS1\home\<yourusername>\CS315-1\LAB01\_02\](file:///\\CS1\home\%3cyourusername%3e\CS315-1\LAB01_02\)

Use **parallel\_for()** in your implementation.

**Hint**: assuming the size of the vectors is given by **size**, and the thread has ID **tid**, than a thread is responsible for updating the output vector with the **start** and **end** index determined by:

int start = size/10 \* tid;

int end;

if (tid == 9) // last thread

end = size;

else

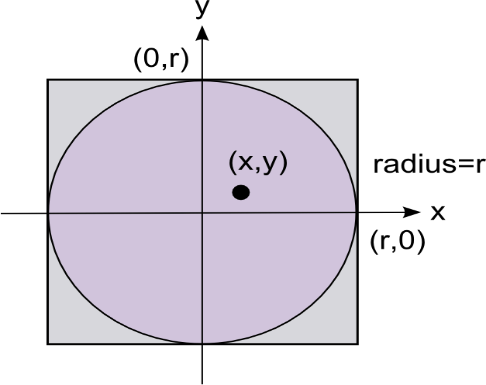
end = size/10 \* (tid+1);

Note that the value assigned to **end** in the last thread (9) is **size**. This is to account for any remainders in the integer division if **size** is not exactly divisible by 10 (the number of threads spawned).

## Estimate PI

We can approximate the value of PI using a technique called the **Monte Carlo** method. The Monte Carlo method is a probabilistic technique that relies on generating many random values.

Consider the following diagram (a circle with radius **r** and a square with sides **2r**):



If we randomly generate the coordinate x and y within the range **[-r, r]**, what is the probability that the (x,y) coordinate point will fall in the circle? If you think about it, it is the he probability given by the ratio:

Area of Circle

------------------

Area of Square

Now if we generate millions of random (x,y) coordinates, than we can claim the ratio of the number of random coordinates actually falling within the circle will also be this ratio, or something pretty close to it. Now since we know the formula for an area of the square is (2r)2 and the area of the circle is πr2, then

π r2 π

------ 🡪 -- = ratio of random coordinate points falling in circle

(2r)2 4

Thus,

π = 4 \* ratio of random coordinate points falling in circle

### Preliminary Setup

Copy [\\cs1\CS\_ClassData\315\Labs\LAB01\_02](file:///\\cs1\CS_ClassData\315\Labs\LAB01_02) to [\\CS1\CS\_Students\<yourname>\CS315-1\LAB01\_02\](file:///\\CS1\CS_Students\%3cyourname%3e\CS315-1\LAB01_02\)

This contains a **serial** implementation of the Monte Carlo method for approximating **PI**. Open, compile, and run the visual studio project.

Your task is to parallelize this **Monte Carlo** implementationusing **10** **threads**.

### Parallelization Strategy

The serial implementation contains a **for** loop that repeats 100 million (i.e. 1e9) times to generate the random coordinate points. We want to **divide this work across 10 threads**. Therefore each thread should generate only 10 million (i.e. 1e8) random coordinates.

Each thread will need to keep track of the total points generated, and the number of points found in the circle. Therefore you will need to first declare and initialize 2 vectors in your **main** function as follows.

vector <int> inCircleCount(10, 0); // Allocate 10 elements and initialize each with 0

vector <int> totalPointsCount(10, 0); // Allocate 10 elements and initialize each with 0

Now design a **lambda function** that captures these 2 vectors. This lambda function will be executed by each thread in your implementation, and it will use these 2 vectors to keep track of the total points generated, and the points found in the circle. The vector element where each thread will update will depend on its threads ID, in the range 0 to 9. E.g. the thread with ID **2** will update **inCricleCount[2],** and **totalPointsCount[2].**

When all your threads complete successfully, sum all the elements in each vector, and generate the ratio required to estimate PI.

### EX01\_03:

Copy any code from the startup project to a new visual studio project stored in [\\CS1\CS\_Students\<yourname>\CS315-1\EX01\_03\](file:///\\CS1\CS_Students\%3cyourname%3e\CS315-1\EX01_03\)

Use **C++11 threads** in your implementation.

### EX01\_04:

Copy any code from the startup project to a new visual studio project stored in [\\CS1\CS\_Students\<yourname>\CS315-1\EX01\_04\](file:///\\CS1\CS_Students\%3cyourname%3e\CS315-1\EX01_04\)

Use **parallel\_for()** in your implementation.