Introduction to Threads

Class: CS315 – Distributed Scalable Computing
Whitworth University, Instructor: Scott Griffith
with heavy reference from Kent Jones

Last Modified: 8/13/2018

Part 1: 'Hello Threads' (15pts)

I have spent enough time talking. Now it is your time do something with these threads. For this first part, you are going to need to write a program that utilizes threads to print out a message.

You are going to spawn **10 threads**. I would suggest you utilize a vector of threads and a for loop to aid in this. You should not be making ten thread variables and initializing each one individually.

Each thread should display **"Hello from Thread <tid>"** where tid is the index of the thread. The first thread to be spawned should be 0, the last one spawned should be 9.

Make sure to clean up your threads by utilizing the **thread.join()** method. This 're-joins' the thread to the main program once it has finished executing.

Question (5pts):
Run your program a few times. Describe your output, why are the tid values not sequential?:

Part 2: Introduction of Lambda Functions (15pts)

Lambda functions are interesting beasts. These functions are kind of the same as traditional functions, but operate differently in a couple of key ways: 1) mostly defined as inline 2) can be passed as a function into other functions/methods. These were introduced with threads in C++11.

Lambda functions have the following syntax:

```
name = [capture specification] (parameter list){
// function body
};
```

The **capture specification** is kind of like a static variable in a traditional class. If called with [foo, bar], variables are copied from the initial scope of the Lambda definition into future calls of the Lambda. If called with [foo, &bar], variable foo would be copied and bar's pointer would be passed (passing by reference).

The **parameter list** works just like a normal function, for our particular lambda we probably want (int tid) to pass in a thread index when called.

Grab the lm1_part2_lambda.cpp file from \\cs1\CS_ClassData\315\LM1_Intorduction_to_Threads\ and start filling in code. Don't delete, or modify the code that is already there.

The **caller** function is going to accept a Lambda function as an argument and start **10 threads** running that function, passing in the thread index to each Lambda. I have provided high level pseudo code, you are going to fill in the actual code.

Your lambda function should accept an integer thread index, as well as capturing Y and problem variables. In the function **Y will be assigned to the tid**, and will assign **100*tid to problem[tid]**. Your function should also print out **"Hello from Thread <tid>"** like from part 1.

After running, your output for problem should be: 0:0, 1:100, 2:200, 3:300 etc.

Question (5pts):
What value do you get for Y at the end? Is that consistent every time you run the program? Why is that varied, but problem consistent?

Part 3: Getting Data Out of a Thread (30pts)

Let's do something productive, not just printing to the screen.

Grab the **lm1_part3_average.cpp** file from \\cs1\CS_ClassData\315\LM1_Intorduction_to_Threads\ and start filing in code. Don't delete or modify anything that is already there, unless stated otherwise.

As you can see **problem** is full of **random ints**. You are going to **average** all of the values in problem, but with multiple threads!

Your program should be able to handle **different numbers of threads**. At the start I would suggest using one or two threads and then testing for a general solution. You are free to use either function based threads (part 1) or Lambda based threads (part 2, this is going to be a little cleaner, maybe).

Each thread will calculate the average of a subset of problem, and report back with the sub average. These averages will be calculated together and reported to the output.

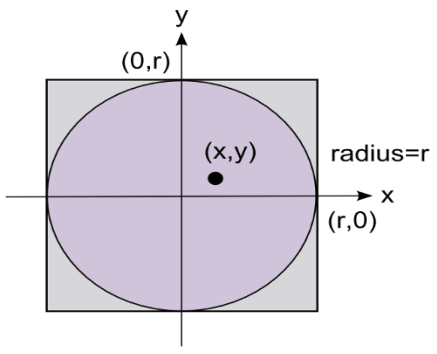
Questions (5pts):
How did you get data to and from the threads? Why might that not work for all algorithms?

Part 4: Algorithm, Approximating Pi (30pts)

Now that you understand how to pass data around, we are going to slightly alter the algorithm. For the most part the program structure should stay the same as in part 3. We are going to use a Monte Carlo method of calculating PI.

The basic premise of a **Monte Carlo method** is to utilize random values to drive a probabilistic solution. If you throw enough darts at a wall, you will eventually get the answer to the question you are looking for.

So consider the diagram (a circle with radius **r** and a square with sides **2r**):



If we randomly generate x / y values within the range [-r,r], what is the probability that the (x,y) point will fall inside the circle?

The probability is the ratio of the area of the circle related to the area of the square: Area of Circle / Area of Square

If we generate millions of random (x,y,) points, we can argue that the ratio of points falling within the circle will be equivalent to this ratio. Armed with this experimentally driven ratio, we can use what we know about the area of circles and squares to figure out PI.

Area of the square: (2*r)^2, Area if the circle: pi*r^2

$$\frac{\pi r^2}{(2r)^2} = \frac{\pi r^2}{4 r^2} = \frac{\pi}{4} = \text{ratio of random points falling in circle} \quad \textbf{or} \quad \underline{\pi = 4 * \text{ratio of points falling in circle}}$$

Go grab **LM1_part4_pi.cpp** from \\cs1\CS_ClassData\315\LM1_Intorduction_to_Threads\ and start filling in code. I went through and left some notes and pseudo code.

Same as before, we are going to let the **user decide how many threads to spawn**. This means you must have a general solution figured out, starting with one or two threads is going to be easier than 10.

We are going to be generating **100 million points**. That is a lot. Feel free when debugging to drop this number down.

Question (5pts):
How do you think we could do this even faster? What are the limitations for using threads to do this calculation?

Bonus Points:
10pts: Calculate and record the performance difference between calculating with different thread numbers (1 – 10) for both Part 3 and Part 4. How many CPU cores are on the machine you are running? Is there a correlation?