**Kailey Cozart**

**Distributed, Scalable Computing**

**LM1**

**Part 1:** Run your program a few times. Describe your output, why are the tid values not sequential?

My output ends up spitting out too many threads on the first line, and some of threads get cut in half mid-line. Also, the thread number greetings are out of order. The tid numbers are probably not sequential because they get initialized at different times, and then run and finish at different times. For example, when I get something like:

2Konichiwa! I am thread 1Konichiwa!

I am thread Konichiwa! I am thread 3Konichiwa!

I am thread 4 with a number 5

This is because every time you use the << for the cout statement, it is sending something new to the terminal as an output. Therefore, if one thread is sending the second part of it’s cout command to the terminal, but then another thread sends the first part of it’s cout statement to the terminal, all of the outputs will get mixed together.

**Part 2:** 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?

I got wildly different values for Y. Sometimes I got 2, 3, 8, or something else. This is probably varied because different threads are running and finishing in a fairly random way, so the value for y doesn’t end up being calculated the same each time, even though the problem is still consistent.

**Part 3:** How did you get data to and from the threads? Why might that not work for all algorithms?

I got data to and from the threads by passing by reference. This might not always work due to race conditions and resource contention in other algorithms.

**Part 4:** How do you think we could do this even faster? What are the limitations for using threads to do this calculation?

We might possibly be able to make this even faster by assigning different tasks to different threads, instead of simply giving all of the threads the same tasks. Additionally, an algorithm other than the Monte Carlo method may be faster. One supposedly fast way to approximate pi would be to use something like the Machin-like formula, which uses Taylor series. Simply adding more threads might not make this any faster.

**Bonus:** Calculate and record the performance (time) 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?

*For Part 3:*

1 Thread == 2.14811 sec

2 Threads == 1.53569 sec

3 Threads == 1.65943 sec

4 Threads == 1.69055 sec

5 Threads == 1.42156 sec

6 Threads == 2.27452 sec

7 Threads == 1.41344 sec

8 Threads == 1.64814 sec

9 Threads == 1.56126 sec

10 Threads == 1.68788 sec

*For Part 4:*

1 Thread == 9.35811 sec

2 Threads == 11.1093 sec

3 Threads == 10.3919 sec

4 Threads == 9.84759 sec

5 Threads == 10.0549 sec

6 Threads == 9.61304 sec

7 Threads == 9.89478 sec

8 Threads == 9.80921 sec

9 Threads == 10.0214 sec

10 Threads == 9.86636 sec

There are 4 cores on the machine that I am using.

While there might be some slight positive correlation between thread number and performance, it is hard to tell. Even when I ran part 3 and 4 with 100 threads, the performance didn’t really change.