CSCI320: Homework 3

Due: Monday, Feb 12th

General Information

Your solutions to the homework assignment should be submitted electronically on Canvas.

- Please submit a ZIP folder containing your files.
  - Only include the source code.
  - Include a README with your name and student ID on two lines, and additional information on lines below. (Collaboration Information).
- Coding Guidelines:
  - Minimize compiler warnings! Be careful with casts!
  - Remember to free memory appropriately
  - **Make sure your program compiles!**

Collaboration Policy

- I expect you might discuss the homework with other members of the class. This is allowed. However, note the other policies!
- I expect you to write your own code.
- You may collaborate, look over code, hunt for bugs, etc, but *you should be the only one typing your own code.*
- If you assist or collaborate with other class members on coding, please note it in the README file.
- (You will not be penalized for this.)

Two Pi. (24 pts)

In this assignment, you will write programs to estimate the value of the extremely popular number, π, in two different ways.

1 Infinite Series. (12 pts)

It is a well known fact that one can compute the value of π using the following formula:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \ldots + (-1)^n \frac{1}{2n+1}\right)$$

This is simply four multiplied by the sum of an infinite series. Of course, we cannot actually compute all the terms of an infinite series in finite time, but we can get close by computing many terms.

In this part, you will write two programs in **pi_seq.c** and **pi_par.c** respectively. The first will be a sequential version of the program, the second will be a parallel version. It is highly suggested that you write the sequential version first!

There is some code provided in **pi_example.c**, this code is meant as a demonstration of how to obtain the wall clock time of your code.

1. Both programs should output 3 lines of text:
   - The estimation of π
   - The number of terms used
   - The (wall clock) time of the computation

   I encourage you to structure your programs like **pi_example.c** to make sure the correct part is timed.

2. In **pi_seq.c** you should write a sequential program which calculates π using the equation above.
   - Your program should accept 1 optional command-line argument.
   - The optional argument is the number of terms to include in the infinite series. The default number should be 1000000.

3. In **pi_par.c** you should write a parallel program which calculates π using the equation above.
   - Your program should accept 2 optional command-line arguments.
   - The first optional argument is the number of terms to include in the infinite series. The default number should be 1000000.
   - The second optional argument is the number of threads to use in the computation. The default value should be 10.

4. Advice and Remarks:
   - Write the sequential program first!
   - In the parallel program, think about how to distribute the terms of the sum.
   - To avoid warnings, use *uintptr_t* appropriately. This does require `<stdint.h>`
   - Test on various different sizes of input and threads.

2 Monte Carlo. (12 pts)

Monte Carlo, a location in the Principality of Monacco, is the site of a famous casino. It is an icon in the world of gambling, much like Las Vegas in the United States. Due to this, a large family of computational processes which involve randomness has *Monte Carlo* in its name. For example, Monte Carlo algorithms[1] and Monte Carlo methods.

Monte Carlo methods involve random sampling to arrive at a computational goal.

Consider Figure 1, which is a square with an inscribed circle.

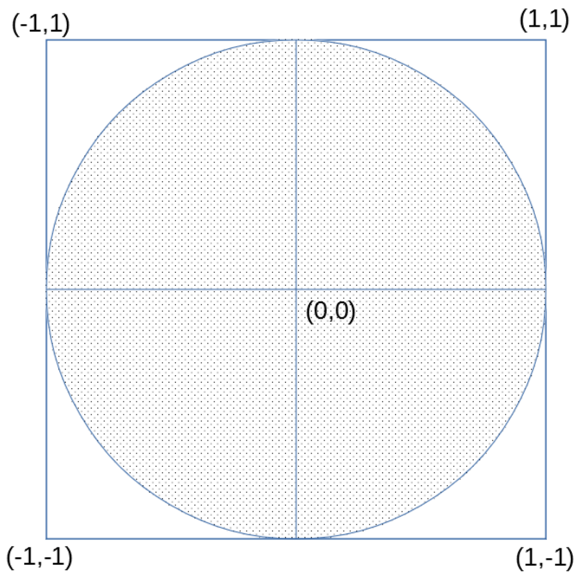

Figure 1: A circle inscribed in a square.

Note that the area of this square is exactly 4 and the area of the inscribed circle is exactly π. The ratio of the areas is $\pi/4$.

This yields the following Monte Carlo method for estimating the value of π:

1. Randomly sample *many* points, (x, y) pairs, with each coordinate uniformly drawn between $[-1, 1]$.
2. Count the number of points which lie within the circle
3. Count the total number of points which lie within the square (all of them)
4. Divide the two, this should be proportional to the ratio of the areas, which is $\frac{\pi}{4}$.
5. Multiply by 4 to obtain π.

We can simplify this process a little by only focusing on the first quadrant instead of all four quadrants - the ratio between the area of the small square and the quarter-circle is still $\pi/4$.

You will write two programs, **monte_seq.c** and **monte_par.c** which uses this idea to estimate the value of π. One will be the sequential version and the other the parallel version.

1. Both programs should output 3 lines of text:
   - The estimation of π
   - The number of samples used
   - The (wall clock) time of the computation

   I encourage you to structure your programs properly to time the correct thing.

[1]Not to be confused with Las Vegas algorithms, which are a different category of randomized algorithms.

2. In **monte_seq.c** you should write the sequential program.
   - Your program should accept 1 optional command-line argument.
   - The optional argument is the number of samples. The default number should be 1000000.
3. In **monte_par.c** you should write the parallel program.
   - Your program should accept 2 optional command-line arguments.
   - The first optional argument is the number of samples. The default number should be 1000000.
   - The second optional argument is the number of threads to use in the computation. The default value should be 10.
4. Advice and Remarks:
   - Write the sequential program first!
   - You should not need to take a square root.
   - **On Random numbers:**
     - There is no function in C to generate random *floats*. You will have to do some division! Make sure you read how `rand()` works.
     - Remember to seed the random number generator (RNG) with `srand(time(NULL))` or something similar.
     - In parallel, each thread must seed its RNG with a different number! Try mixing the current time with the thread number...[2]

[2]Some of this have to do with the C random number generator not strictly being thread-safe sometimes. If say, four threads each generate a sequence of random numbers, all four sequences will actually all be the same! This is because the original *seed* is somewhat shared between the four and thus each thread has their RNG in the same state.