CSCI320: Homework 5

Due: Friday, March 22nd

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. (10 pts)

In this part of the assignment, you will write programs to estimate the value of the extremely popular transcendental number, π, in two different ways. Seeing as you probably already have some experience with this sort of program, this part should not be much work. Also this time you get to use OpenMP!

However, we will care (a little bit) about performance. You do not need to write programs with extremely good span for this first part. However:

- Your parallel programs should be faster than your previous **sequential** programs.
- It may take some tweaking of parameters to achieve this performance.
- Additionally, your programs should take no more than a few minutes (high estimate) to run on an input size of $10,000,000,000$ for the infinite series and $1,000,000,000$ for the Monte-Carlo.

1 Infinite Series. (5 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 write one new parallel program to compute π in **pi_openmp.c**.

I encourage you to retain the sequential and parallel programs from an earlier assignment for performance comparisons!

1. Your program should output 3 lines of text:
    - The estimation of π
    - The number of terms used
    - The (wall clock) time of the computation. Use the correct omp function here.
    - 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.

2 Monte Carlo. (5 pts)

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

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

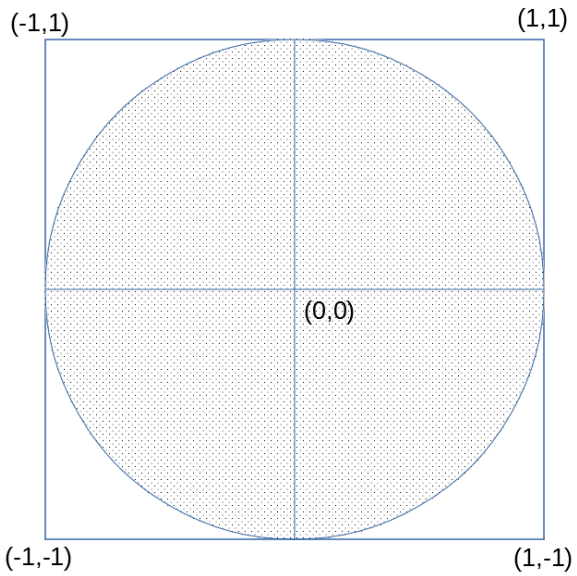

Figure 1: A circle inscribed in a square.

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 one new parallel program in **monte_openmp.c**.

1. Your program should output 3 lines of text:
   - The estimation of π
   - The number of samples used
   - The (wall clock) time of the computation
2. Your program should accept 2 optional command-line arguments.
3. The first optional argument is the number of samples. The default number should be 1000000.
4. The second optional argument is the number of threads to use in the computation. The default value should be 10.

3 Basic Sorting. (14pts)

Sorting is a fundamental task in computer science. In this part of the assignment, you will implement a specific sorting algorithm. We do care a lot about performance so be careful with your code.

You may find the parallel portion of this assignment much easier once we talk about parallel for loops next week.

Before you can begin to write the sorting algorithms, familiarize yourself with the provided code.

- The **random_numbers.c** program is used for generating a *numbers.txt* file which will contain a lot of random numbers in it. It accepts a single command-line argument which determines how many numbers to generate. You should use this to create the input files for your testing purposes. You should run this program a few times with different inputs to see how the generated files differ. The generated file will be formatted as follows:
  - A single number on the first line, indicating that there are n numbers to expect
  - An empty line (separator)
  - n additional lines which each contain one random number
- The provided **sort_skeleton.c** file contains a sketch of how you might organize your code. This code roughly follow this procedures.
  - Reading in the numbers from an input *numbers.txt* and storing it into an array
  - Sorting the actual array
  - Your code should display the time elapsed for the sorting process. Be sure to time the correct part of the code! Do not time the process of reading in the numbers! **This is the only output.**
  - (You **may** want to write a quick check to ensure that your array is actually sorted and also display True or False).
- **You must first write some code to read the input file.** Complete the **Populate** function which takes in a string filename and a pointer to an unsigned integer *size*. This function should read the specified file (generated by the random_numbers.c program) and return an array of the corresponding numbers, in addition to setting the size variable to be the correct count of numbers in the array. Pay attention to how it is used in the skeleton code.

Now, let us actually get to sorting. We will sort our array in *ascending* order for this assignment.

3.1 Bubble

Bubble Sort is a very somewhat inefficent sorting algorithm that is nevertheless fun to implement. A description of the bubble sort algorithm is as follows:

1. Let the array to be sorted be array A, with n total elements within it.
2. Perform n passes through the array. During each pass:
   (a) Let i be 0, the index of the first element.
   (b) Compare $A[i]$ and $A[i+1]$, if the ordering is incorrect (that is, if $A[i] > A[i+1]$) then swap the position of the two elements. Let's refer to this operation as a *bubble* operation.
   (c) Now increment i by 1.
   (d) Repeat steps (b) and (c) until we have completed one pass through the entire array, that is, until $i == n$.
3. Once n passes have been performed, the array will have been sorted.

Note that in each pass, the largest element in the array is eventually moved to the rear. Therefore, in the worst case we will need n passes to sort the array.

The *bubble* operation is what swaps pairs of elements when necessary.

In **bsort.c** please implement Bubble Sort. Make sure to check that your implementation actually sorts the array.

There are several optimization you can do to Bubble sort, such as quitting if an entire pass occurs with no swaps. For this assignments you do not need to include them.

Remember, your code should do the following:

- Your code should read in the numbers from an input *numbers.txt* and store it into an array
- Your code should sort the array
- Your code should display the time elapsed for the sorting process. Be sure to time the correct part of the code. Do not time the process of reading in the numbers.

3.2 Odd and Even

Bubble sort is rather inefficent and very sequential. Let's think about how we can parallelize this sorting idea.

The key operation in bubble sort is the bubble operation, which happens n times during each pass through the array. Perhaps we can do every bubble operation during each pass in parallel?

Unfortunately, if we bubble $A[0]$ and $A[1]$, we cannot bubble $A[1]$ and $A[2]$ at the same time! That's a race!

However, this problem can be avoided if we only bubble *every other pair* of elements in each pass. Thus:

- During the zeroth pass, we will bubble the pairs $(A[0], A[1]), (A[2], A[3]), (A[3], A[4]) \ldots$
- During the first pass, we will bubble the pairs $(A[1], A[2]), (A[3], A[4]), (A[5], A[6]) \ldots$
- In general, on every *even* numbered pass, we bubble every *even* indexed element with the element after it. On every *odd* numbered pass we bubble ever *odd* element with the element after it
- All the bubble operations in a single pass are independent and thus can be done in parallel

This process will sort the input in n passes, leading to the following algorithm. Note that it is very vital to bubble different elements on each pass.

1. Let the array to be sorted be array A, with n total elements within it.
2. Perform n passes through the array. During pass i:
   (a) If i is an odd pass, bubble every odd indexed element with the next one, in parallel
   (b) If i is an even pass, bubble every even indexed element with the next one, in parallel

In **oesort.c** please implement this sorting algorithm. Make sure that your implementation actually sorts the array.

Just as before, your code should do the following:

- Your code should read in the numbers from an input *numbers.txt* and store it into an array
- Your code should sort the array
- Your code should display the time elapsed for the sorting process. Be sure to time the correct part of the code. Do not time the process of reading in the numbers!

Finally. The parallel sort should perform better than the sequential sort on reasonable input sizes of roughly $300,000$ elements or so.

There are several optimization you can do to Odd-Even sort, such as quitting if an entire pass occurs with no swaps. For this assignments you do not need to include them.