CSCI320: Homework 5

Due: Friday, November 13th. **5pm**

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 up your 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.)

1 Sorting. (30 pts)

Sorting is a fundamental task in computer science.

In this part of the assignment, you will implement some sorting algorithms and then implement some parallel sorting algorithms with OpenMP. You should implement the algorithms through your own code in this assignment, so do not just call the qsort function from the C library.

Before you can begin to write actual sorting algorithms, familiarize yourself with the provided code. It is a similar setup as the ones given for the *Bubble Sort* on the previous assignment.

- 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. The 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 **msort.c** and **qsort.c** files contain a sketch of how you might organize your code.
- **You must first put in the code to read the input file!** Complete the **Populate** function which takes in a string filename and a pointer to an unsigned integer size. You probably already have this part from the previous assignment.

We will sort our array in *ascending* order for this assignment.

1.1 Quicksort and Mergesort. (8 pts)

Quicksort and **Mergesort** are efficient sorting algorithms, though they are trickier to implement. I will not describe the algorithms in detail here as you probably already have a good idea of how they work.

1.1.1 Merge Sort

- In **msort.c** please implement Mergesort.
- Notice how the *my_msort* function calls a helper after allocating a temporary array. This is to minimize the amount of additional calls to *malloc* which would otherwise be necessary. You should not need to call *malloc* aside from this call.
- My suggestion is to also implement some additional functions, such as a *merge* function.

1.1.2 Quick Sort

- In **qsort.c** please implement Quicksort. You should choose to use a random pivot element.
- I have not given you code to create temporary arrays, etc. You have to use your own judgment on this part.
- The provided *partition* is probably a helpful function to implement.

1.1.3 Notes

- There is a quick *is_sorted* check provided to you. However, you also might want to do some additional checking of your results. It is very easy to make mistakes and create a lot of duplicates in the array!
- Your program should output the time elapsed after checking to that the result is sorted. Make sure to time the correct parts! (Only the actual sorting.)

1.2 Parallel Mergesort. 4 pts.

For this part of the assignment you should implement a parallel Mergesort.

- You should write your program in *msortpar.c*
- Here, you **do not** need to implement the good merge (with good span). You just need to make a small modification to make recursive calls in parallel.
- I suggest making a copy of your merge sort code from the previous part and just working from there.

1.3 Parallel Quicksort. 18 pts in total.

- You should write your program in *qsortpar.c* for this part!
- Here, you will need to implement the good partition algorithm.
- I outline some of the steps you will need to implement in the following subsections.

1.3.1 Prefix Sum. 8 pts.

- There is a function *psum_seq* where you can implement the prefix sum. I suggest you start with a sequential version to make sure everything works before choosing one of the parallel versions to implement in *psum_par*
- Either the one with $O(n \log n)$ work or the one with $O(n)$ work is fine to implement.
- There is a *test_psum* function which might be helpful. It checks whether the sequential prefix sum agrees with the parallel prefix sum.

1.3.2 Partition. 6 pts.

- The provided *partition_par* is a good place to implement a parallel partition.
- Here, you will need to implement the good partition algorithm.
- This should involve some parallel for loops and two prefix sum calls (or so).

1.3.3 Sort. 4 pts.

- Here, you will need to put together the previous two parts to make an actual sorting algorithm.
- We do care about performance, so be careful, some advice:
  - Avoid making too many malloc calls. You do need to make an auxiliary array, but you should only need to make one large auxiliary array?
  - Use the various clauses of OpenMP for performance reasons - loop scheduling, if-clause, final-clause.

2 Remarks

To receive full credit:

- All algorithms should produce correct results.
- **Be aware of races! ESPECIALLY in merge sort!**
- Your parallel merge sort algorithm should be faster than the corresponding sequential one at large input sizes. (Try 30,000,000).

- Your parallel quick sort algorithm does **not** need to be faster than the corresponding sequential one. In my implementation I was able to get good speedup. My expectation is that you will see some speedup, but I do not want to make a requirement. (You'll need to be really careful with implementing the prefix sums, etc).
- You will **need** to make efficiency optimizations in the parallel versions! That is, run non-parallel code for small inputs, don't create tasks when inputs are small, etc etc...