Problem 2 (30 Points)

In this problem, you will start by selecting a sorting algorithm to sort 10,000 random integers with values 1-10,000. We will use pthreads in the problem to generate a parallel version of your sorting algorithm. If you select code from the web (and you are encouraged to do so), make sure to cite the source of where you obtained the code. You are also welcome to write your own code from scratch. You will also need to generate the 10,000 random integers (do not time this part of your program). Make sure to print out the final results of the sorted integers to demonstrate that your code works (do not include a printout of the 10,000 numbers). Provide your source code in your submission on Canvas.

a. Run your program with 1, 2, 4, 8 and 32 threads on any system of your choosing. The system should have at least 4 cores for you to run your program on. Report the performance of your program. b. Describe some of the challenges faced when performing sorting with multiple threads. c. Evaluate both the weak scaling and strong scaling properties of your sorting implementation.

*You should include your C/C++ program for this problem. Also include your written answers to questions 2 (a-c) in your homework 1 write-up.

Sources Cited: https://malithjayaweera.com/2019/02/parallel-merge-sort/

Code compiled and run on vector.coe.neu.edu (nproc=80)

Part (a)

Running the program with 1, 2, 4, 8, 16, 32, 64, 128, 256 threads for 10000 elements:

Threads	1	2	4	8	16	32	64	128	256
Latency (ms)	7.706	4.208	2.729	2.42	2.876	3.971	6.412	10.99	20.571

The program shows a decrease in latency as the number of threads increases up intil a certain point. However, the latency does not decrease linearly with the number of threads. The program shows the best performance with 8 threads, after which the latency starts to increase. This is likely due to the overhead of managing more threads than the system can handle efficiently.

Part (b)

Describe some of the challenges faced when performing sorting with multiple threads:

1. **Thread Creation Overhead**: Creating threads is an expensive operation. The overhead of creating threads is likely to be more than the time saved by parallelizing the merge sort algorithm.
2. **Thread Synchronization**: When multiple threads are working on the same data, it is essential to synchronize the threads. This means that access to shared data

structures and resources (like merging sorted subarrays) needs to be synchronized which leads to delays.

3. **Race Conditions**: Multiple threads trying to read from or write to the same memory location at the same time can lead to unexpected values being read or written. This can lead to incorrect elements being merged and thus incorrect sorting. Mutex locking is needed to prevent this but while ultimately lead to some overhead.
4. **Memory Contention**: Multiple threads accessing the same memory locations can lead to memory contention. This can slow down the program due to cache misses and other memory access delays.

Part (c)

Weak Scaling of the Merge Sort Algorithm:

Number of Elements	10000	20000	40000	80000	160000	320000	640000	1280000	2560000
Threads	1	2	4	8	16	32	64	128	256
Latency (ms)	7.726	8.515	10.622	17.217	30.01	52.292	110.53	241.676	segfault

Strong Scaling of the Merge Sort Algorithm:

Number of Elements: 10000

Threads	1	2	4	8	16	32	64	128	256
Latency (ms)	7.706	4.208	2.729	2.42	2.876	3.971	6.412	10.99	20.571

Evaluation of Scaling:

Weak Scaling: The program shows an increase in latency as the number of threads and problem size increase. This shows that the weak scaling of this merge sort algorithm is very poor since the algorithm does not efficiently handle larger problem sizes with more threads. **Strong Scaling:** The program shows a decrease in latency up until 8 threads, where it starts to then increase in latency for threads larger than 8. In other words, the merge sort algorithm benefits from parallelism up until a certain point, after which the overhead of managing more threads becomes a bottleneck. This means that the program exhibits good strong scaling up until 8 threads, after which the scaling starts to degrade.

Miscellaneous

- The program was compiled and run using the following command: `g++ -pthread merge_sort.cpp -o merge_sort` or simply `make`