CSC 6220: Parallel Computing I: Programming
Term Project - Fall 2022

Neema Sadry

Introduction

The purpose of this project is to create a parallelized algorithm for the Odd-Even Sort algorithm using MPI. In this algorithm, the array will initially be partitioned into subarrays of equal sizes. These subarrays are subsequently distributed to the processes such that one process receives one subarray. There will be 8 processes in total. Process 0 will serve as the primary process, which includes generating an array of the following sizes: 2^{16}, 2^{20}, and 2^{24}. Once an array is created with one of the given sizes, they are subsequently filled with a random integer, which are in the range $[0, 128]$. Next, the subarrays are distributed in a manner such that one subarray array will be issued to each of the remaining processes. These processes will individually complete their respective calculations and send their results back to Process 0, where the final array is assembled.

The final array is outputted to a file called: **`result.txt`**. Moreover, the output of the program, which contains the execution time for the program, can be found in the text file: **`output.txt`**. Finally, this report will compare and analyze the execution time for three algorithms:

1) Modified Odd-Even Sort parallel algorithm
2) Standard Odd-Even Sort algorithm (Section 6.3.5 in our textbook)
3) Serial Quicksort

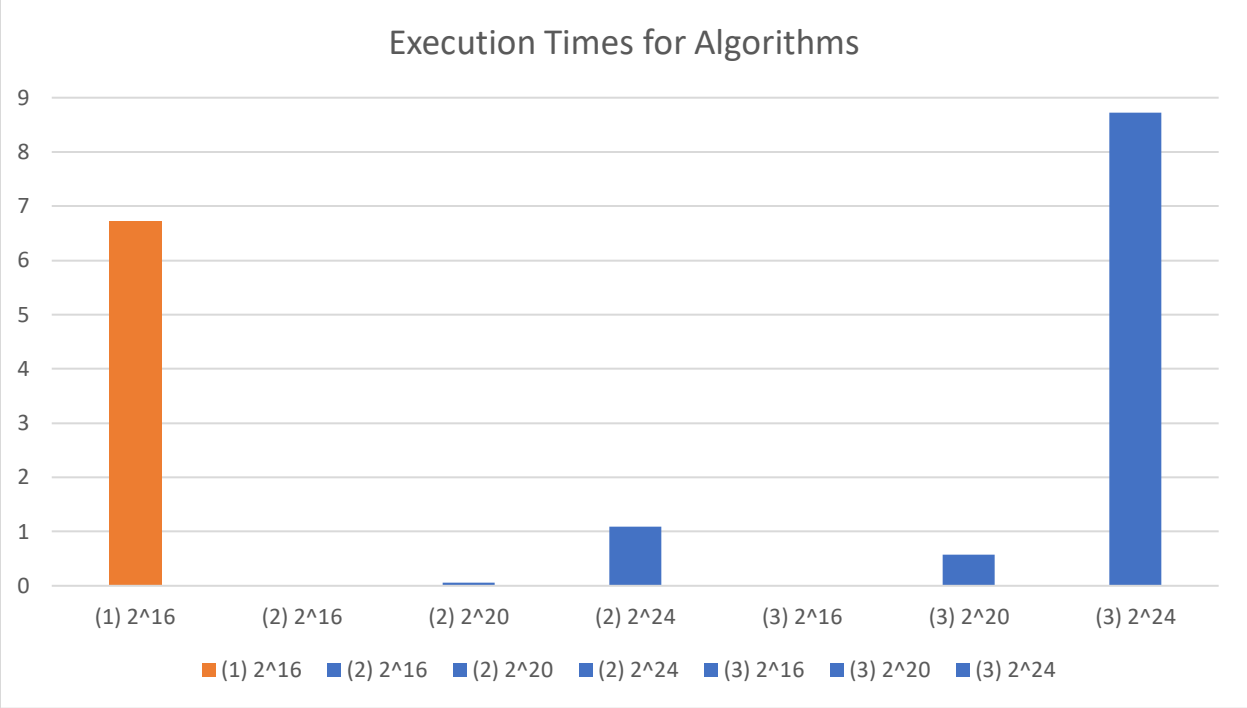
Implementation

The build on the implementation details outlined so far, the program requires an integer constant, N, to be set, which serves to define the size of the array generated; three values were used in separate program executions: (1) $2^{16} = \mathbf{65,536}$, (2) $2^{20} = \mathbf{1,048,576}$, and (3) $2^{24} = \mathbf{16,777,216}$. Like Section 6.3.5 in our textbook for their Odd-Even Sorting implementation, the program was simplified by assuming N is divisible by p, which represents the number of processors being used.

Using this assumption, Process 0 can distribute N/p subarrays to all other processes. These processes will sequentially sort the subarrays individually. However, the algorithm works further to alternate between *odd* and *even* phases. During the *odd* phase, the odd-numbered processor(s), *i*, will communicate with their previous even-numbered processor, $(i - 1)$, in such a way where the subarrays between the two processes are combined. The processor with the higher number will keep the upper-half of the subarray while the processor with the lower number processor retains the lower-half of the subarray. This process is true for the *even* phase, where the even-numbered processor, *i*, will communicate with the following odd-numbered process $(i + 1)$.

Results & Conclusion

Three separate programs were created in order to ease the compilation and execution process: `oeparallel.c`, `oestandard.c`, and `squicksort.c`. All were compiled and executed on both the local machine and on the WSU Grid. Keep in mind that the value of the array sizes was changed in subsequent compilation/execution runs.



The results shown in the graph above are for the execution times of all the algorithms that were successfully able to run on the WSU Grid (blue). The orange-colored data is the only exception given that it was run locally on an M1 Pro. The speedups between the different array sizes for each of their respective algorithms is significant.

The `oestandard.c` program ran successfully on both machines using all the array sizes. Unfortunately, the `oeparallel.c` only the first array size managed to execute and run successfully on the local machine, but not the two larger array sizes. Interestingly, none of the three array sizes ran successfully for the `oeparallel.c` on the Grid. It should be noted that there were no errors listed in the error files generated by the Grid to aid in figuring out the root cause for these unsuccessful executions.