Merge Sort

- Project Documentation -

I. Project description

This project aims to accelerate the sequential implementation of the well-known sorting algorithm **MergeSort** by parallelization and multiprocessor execution of the code.

In order to speed up the execution time of the algorithm, the following synchronizing and parallel mechanisms will be used:

- MPI
- Pthreads
- Parallel STL

At least two of the mentioned mechanisms will be used for the parallel implementation of the MergeSort algorithm. The coding language used will be C++ so all the parallelization methods will be available

For the testing part of this project, multiple sets of data will be generated, with different distribution of numbers and with an increasing sample size. The data will be run multiple times so that an average execution time can be observed and noted.

In total, three implementations will be developed, a sequential one and two parallel. This is done in such a way that the results can have a better interpretation in relation to the methods used.

In order to quantify whether the parallel implementations have any actual benefit over the sequential one, a baseline needs to be established. The sequential implementation of the algorithm that will be developed is similar to the one linked in the references.

II. Project environment

All the implementations will be run on a PC with the following specifications:

- CPU
  - Model: Intel i9-9900K
  - 8 Cores, 16 Threads
  - Frequency: 5.0 Ghz
  - Cache: 16 Mb
- RAM
  - Capacity: 32 Gb (4 x 8 Gb DIMMs)
  - Speed: 3200 Mhz
  - Channel: Dual Channel
- GPU: Nvidia GeForce 2070 Super 8 Gb
- Operating System: Windows 10 Pro

III. Sequential Implementation

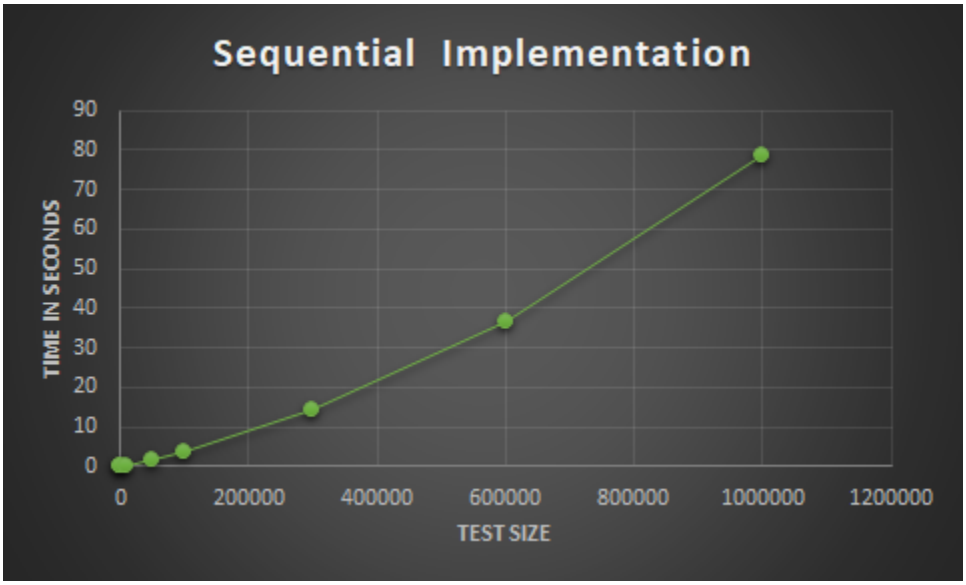
The sequential implementation follows the basic logic for the **Merge Sort**, by dividing the input vector into two halves, and recursively calling itself on the two newly created halves.

The sequential implementation can be seen at the **GitHub** repository linked in the references.
A quick explanation of the functions used can be found below:

- **merge(left, right)**
  This function does the merging between two sorted vectors and returns a single sorted vector. An improvement of note here, would be to use the merge function found in the “algorithm” header.
- **mergeSort(vector)**
  This function is the recursive part of the algorithm, that calls itself after splitting the initial vector in two halves.

For testing the sequential implementation, a smaller application was developed with the sole purpose of generating random sets of data. Those inputs are stored in the **“Input Data”** folder of the repository. In order to save space and time, 8 tests were generated, with varying sizes from 100 numbers to 1000000, with numbers between 1 and 1000000000 uniformly distributed.

The program was designed in such a way that it takes all the input defined and sorts them, then prints the time it took to sort the input to the console.
The execution time of the sequential algorithm was stored and compiled in the following graph.



As we can see, the execution time increases proportionally with the input size. This is expected, and for now, until we have the parallel implementations, we can not say anything more about this graph.

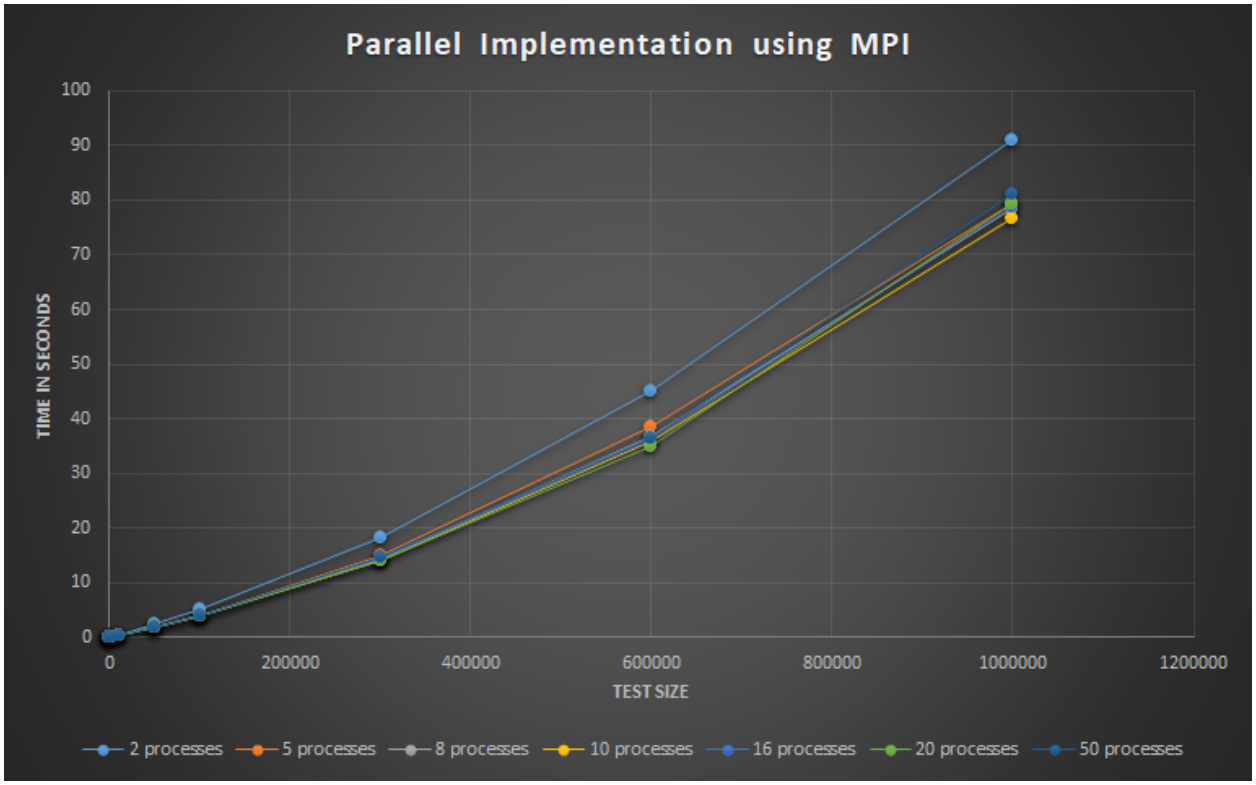
IV. Parallel Implementation I (MPI)

For this parallel implementation, I chose to use MPI in order to send messages between processes. The way in which the MergeSort problem was tackled, is a rather basic one:

The initial array is split in equal subarrays and then each of them is sent to a process in order to be sorted. Each subarray is sorted using the same MergeSort algorithm used for the sequential implementation. After all subarrays are sorted, they are then sent to a single process which will sort them again as a whole.

For testing, the same data inputs were used in order to be consistent and have a comparable baseline between the methods. Also, in order to see the actual gain over how many processes should be used, the parallel implementation was run with 2, 5, 8, 10, 16, 20 and 50 processes in order to get a better understanding of how the algorithm performs.

The following chart shows us the runtime of each test based on the process count.



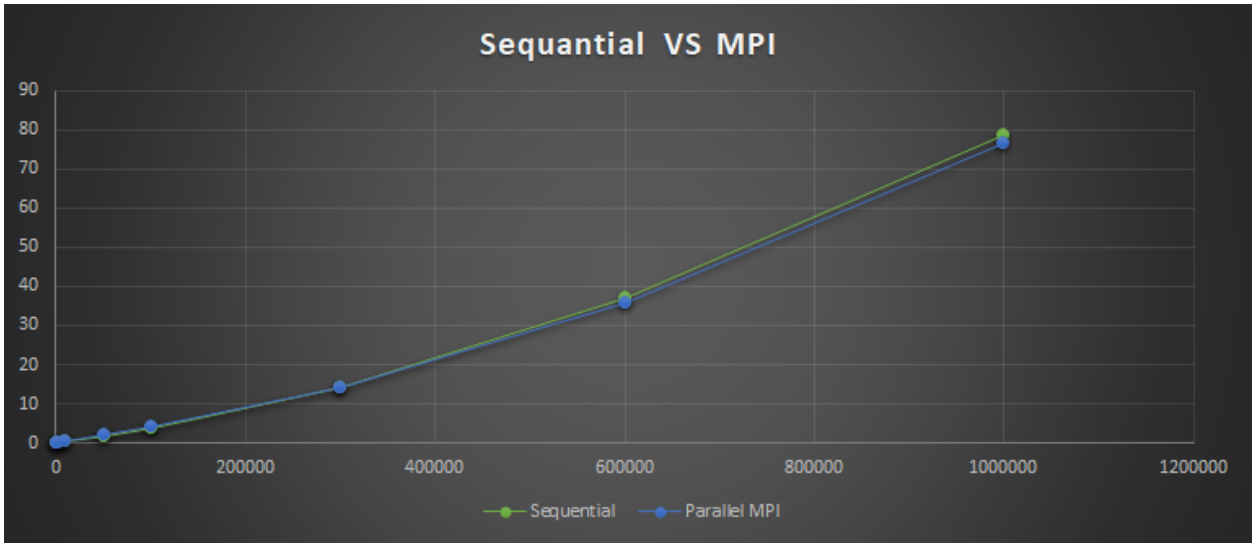
As we can see, the initial run with 2 processes is very slow compared to the other runs. As we increase the processor count to 5 and 8 we can see the algorithm progress to a more stable state regarding the runtime. Going up to 20 and 50 processes, the runtime on smaller tests increases while the runtime on the bigger tests seems to stay consistent within a margin of error.

While the number of tests that were done on this can be categorized as minimal to average, it does show a visible trend regarding the number of processes used for this program. While using little to almost no parallelization, the performance of the program suffers badly, but when trying to add more processes we see diminishing returns and instability with the results. This instability can also be attributed to background tasks that have a bigger, more noticeable impact on the program when run with such a large amount of processes.

From what we can tell from the data and the chart, the middle ground of 10 processes seems to make full use of the parallelization and offer the best time. But, since there is such a small difference between the 8, 10 and 16 processes run, the middle ground

between the 3 seems to be a good tradeoff keeping in mind both the performance and the cost.

Now, this data needs to be compared with the one from the sequential implementation in order to get a better understanding of how the MPI implementation performed. The following chart shows a comparison between the sequential implementation and the parallel one.



What is noticeable right away from this table is the fact that the two implementations seem very alike regarding the execution timings. Therefore we need to look more closely at the data. Whilst the parallel implementation seems to have an edge on bigger tests, this seems rather insignificant and can be attributed to margin of error. The same can be said for the sequential execution of the algorithm on smaller tests.

In conclusion, the MPI parallelization method seems to offer no visible improvement over the sequential implementation, and the only visible improvement is on bigger datasets, but this is marginal, at best.

V. References

GitHub Repository
Sequential MergeSort
MPI Tutorials
Pthreads Information
Parallel STL methods