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
- C++ threads
- 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

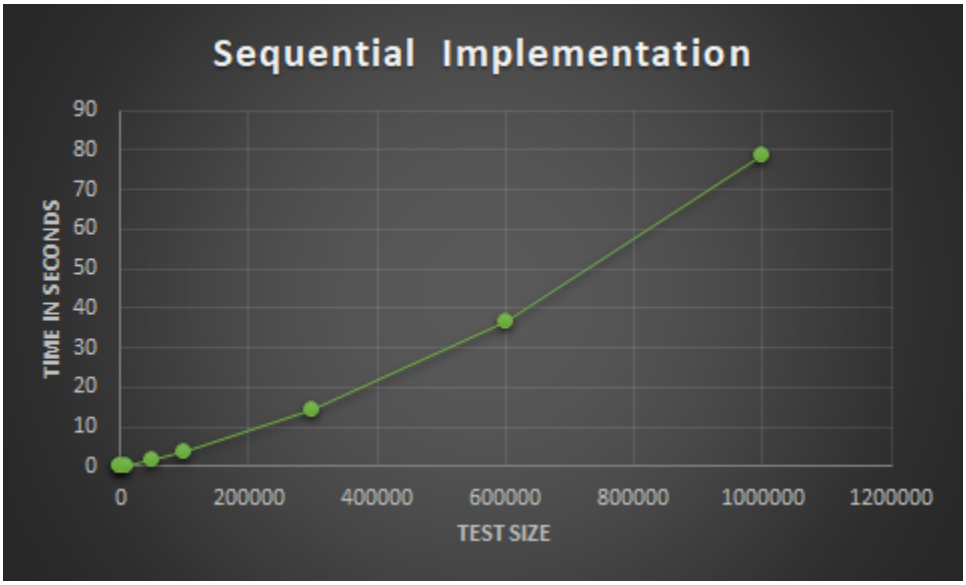
The sequential implementation follows the basic logic for the **Merge Sort**, by dividing the input vector into two halves, and recursively calling itself on the two newly created halves.

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**)

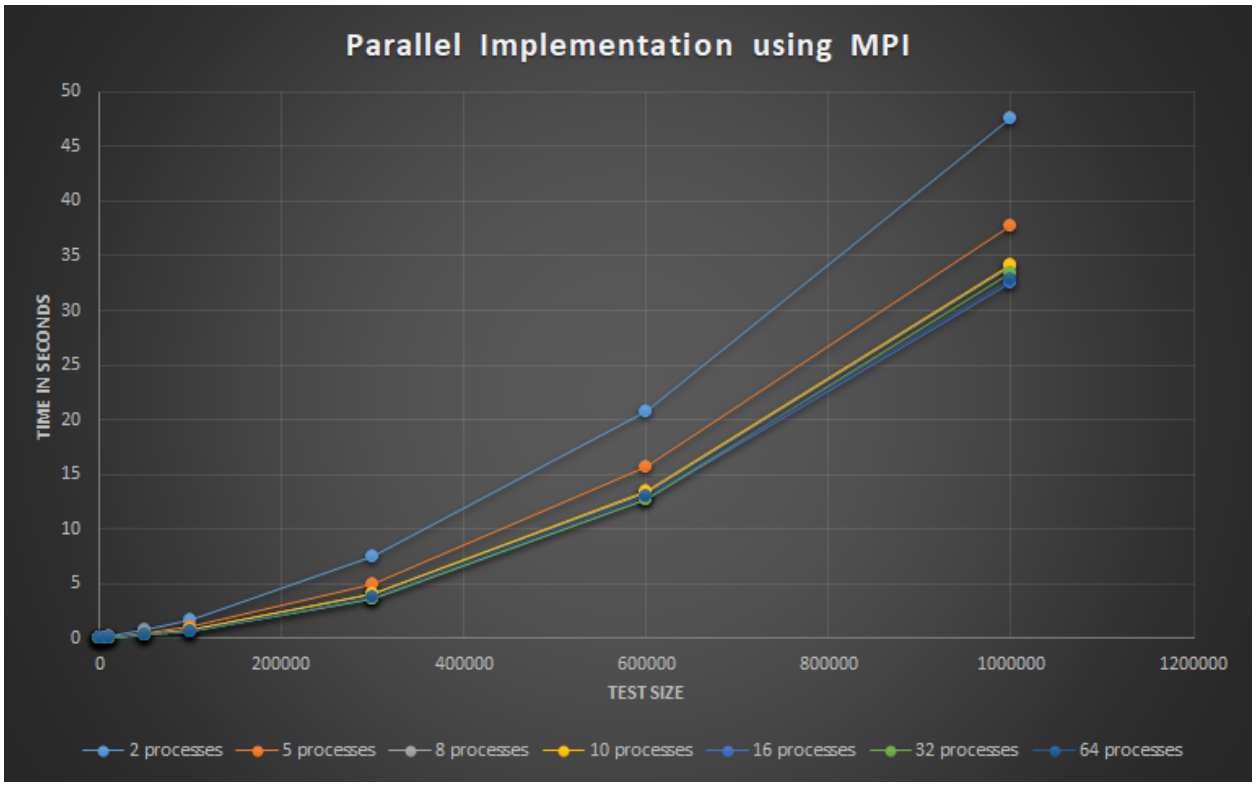
For this parallel implementation, I chose to use MPI in order to send messages between processes. The way in which the problem was tackled in a parallel implementation is the following:

An initial process receives an array, then, if any processes are available, he splits the array in half and sends one part to a “child” process and the other half is split again. If there are any available processes left, the split repeats itself until there are no child processes left. When this happens, the remainder of the array is then sorted locally and then merged with the respective responses from child processes with the other halves of the array.

This method creates a hierarchy while dividing the initial array, therefore, when properly illustrated it will look like a binary tree for the initial “divide” part.

For testing, the same data inputs were used in order to be consistent and have a comparable baseline between the methods. Multiple tests were performed on the application in order to see the actual performance gain related to the number of processes used. Given the binary tree structure of the solution, the parallel implementation was run with 2, 8, 16, 32 and 64 processes. For testing purposes and to illustrate there is almost no gain when using a number of processes that is not equal to a power of 2, the application was also run with 5 and 10 processes.

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



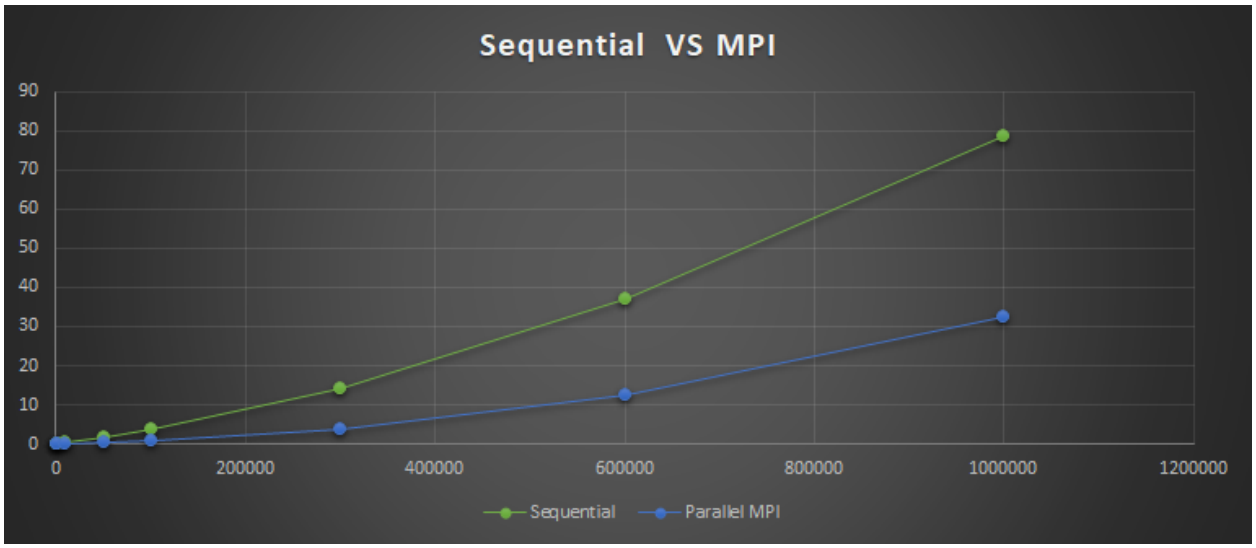
As we can see, the initial run with 2 processes is very slow compared to the other runs. As we increase the processor count to 5 and 8 we can see the algorithm progress to a more stable state regarding the runtime. Going up to 32and 64 processes, the runtime on smaller tests increases while the runtime 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 run the program with more than 32 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 16 processes seems to make full use of the parallelization and offer the best runtime. A bigger number of processes brings instability regarding the runtime, and a lower count of processes is not worth it compared to the best time offered by the run with 16 processes.

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.



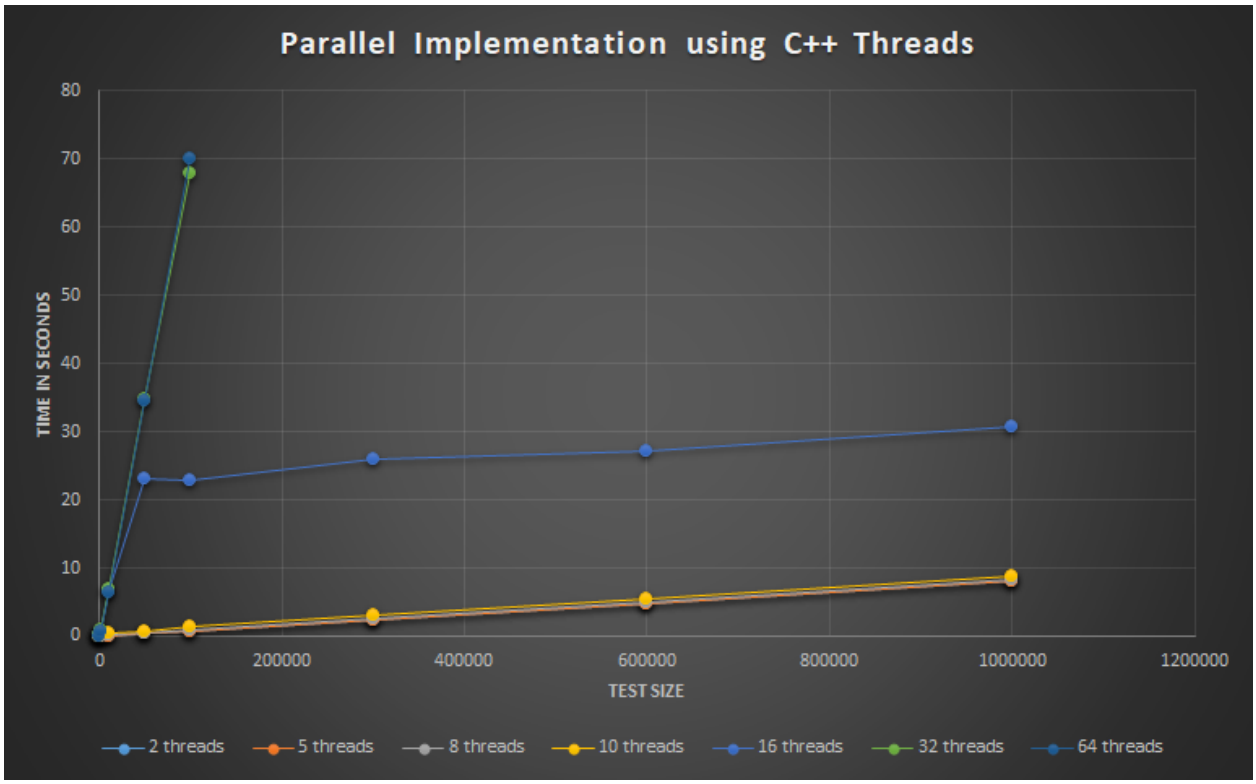
What is noticeable right away from this table is the fact that the two implementations are worlds apart. The parallel implementation manages to significantly outperform the sequential implementation due to the fact that most of the sorting is done in parallel, therefore the speed is almost cut in half.

In conclusion, the MPI parallelization method offers a considerable improvement over the sequential method, showing a trend of reducing the sequential runtime by half.

V. Parallel Implementation II (C++ Threads)

For the second parallel implementation, the same logic presented at the first implementation was used. The only difference being the way the threads are allocated.
In this implementation, there is a maximum number of threads that will be allocated at a given time. If there are enough available threads to be allocated, one will be delegated to merge a part of a given array from a parent thread, otherwise the sorting will be done locally, contained in the current running thread.
When the thread finishes sorting it's current chunk of the array, it will exit and the thread pool size will go up.

The same tests were run on this implementation as the previous two. The following chart shows the C++ Threads performance:

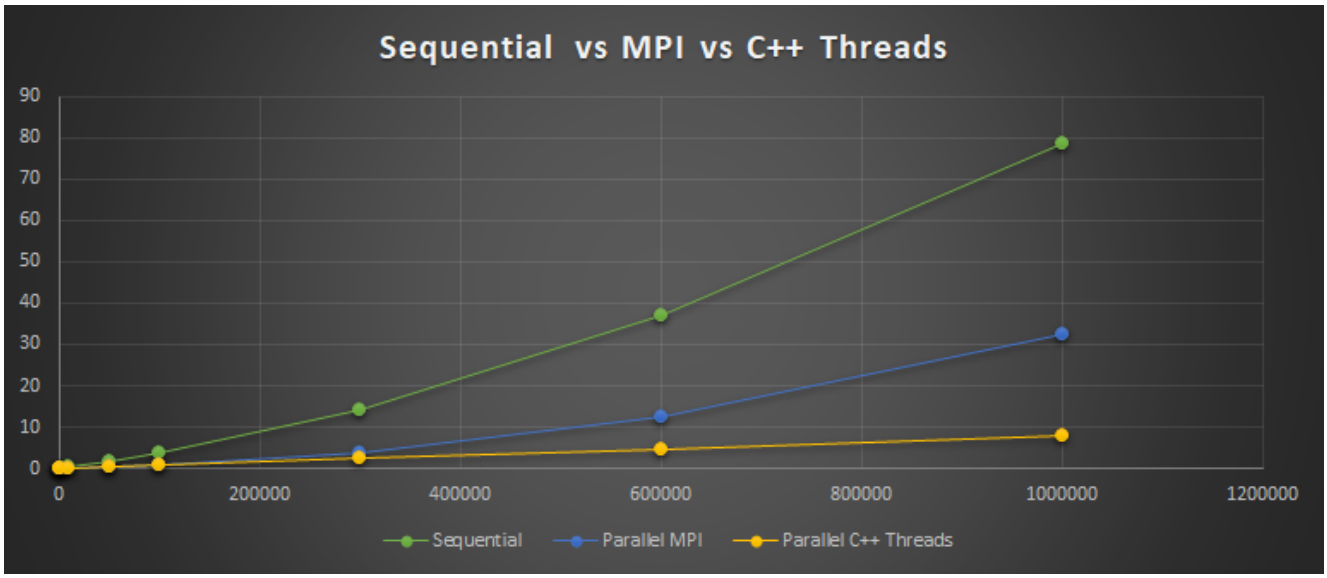


As we can see, the results are not quite as expected. Starting with the obvious, the runs with 32 and 64 threads performed poorly, incredibly worse even compared to the sequential implementation. A test with 300.000 numbers on those two configurations took approximately 200 seconds, and the tests on 600.000 numbers did not even finish in a 10 minute time window, so those tests and bigger have been omitted from this chart in order to keep the data shown relevant.

By analyzing the relevant tests, we can actually see that the best time achieved overall on those tests is when running the program with only 2 threads. Adding additional threads seems just to slow down the algorithm gradually.

VI. Comparisons

Below, a chart is shown with the average execution timings for each of the three implementations on the same predefined tests. As we can see, the sequential implementation of the algorithm takes the longest time to complete each test. The MPI implementation is twice as fast in comparison to the sequential execution in finishing the tests. The C++ Threads implementation is 4 times faster than the MPI implementation, and, therefore 8 times faster than sequential implementation.



To conclude, we can say with certainty that the best implementation of the Merge Sort Algorithm is the one using C++ Threads.

VII. References

GitHub Repository
Sequential MergeSort
MPI Tutorials
Pthreads Information
Parallel STL methods