# CENG 478

Assignment 1 Report
Deadline: March 11, 23:59

Full Name: Yasin Fatih ALPUL
Id Number: 2098739

## Complexity

If we assume $n$ is the problem size i.e. the number of floats to find the maximum of and $p$ as the number of parallel processes, then the serial algorithm has a complexity of $O(n)$. If we use a parallel algorithm as described in the assignment text, then the complexity becomes,

$$O\left(\frac{n}{p} + log_2 p\right)$$

The first part comes from the first part of the algorithm. $p$ processes share the $n$ numbers and process them in parallel. The second term comes from the second part of the algorithm. All $p$ processes reduce the number of items by half in each step. This method make use of several processor resources in parallel instead of a single processor and using it in a sequential process.

## Design Choices

In the assignment, `MPI_Send` and `MPI_Recv` are mostly used as well as `MPI_Scatter`. Initially, the master process passes the file data using `MPI_Scatter` and the second part of the algorithm is handled by `MPI_Send` and `MPI_Recv`. The first part could also employ the same send and recv functions but `MPI_Scatter` is more appropriate since we do not explicitly send and receive and possibly allow for a more efficient implementation. Although the input size is 1 million and all of the experiment sizes (1,4,8,16) are evenly divisible by 1 million, I have implemented subroutines for the cases of unequal distribution anyway for completeness sake. In those cases, these routines first send how many floats will be sent to each process and then sends those numbers using `MPI_Send`.

## Experiments

Figure 1 shows the running time versus number of procesors. Initially, time increases as number of processors go from 1 to 4. Then it starts to decrease. This is most likely because of parallellization overhead as we make the algorithm parallel instead of sequential. Once we start to make the parallel algorithm run on even more processors, the running time starts to decrease.

Figure 2 shows the speed improvement versus number of processors. The speed improvement is calculated as,

$$\text{Speedup} = \frac{T_s}{T_p}$$

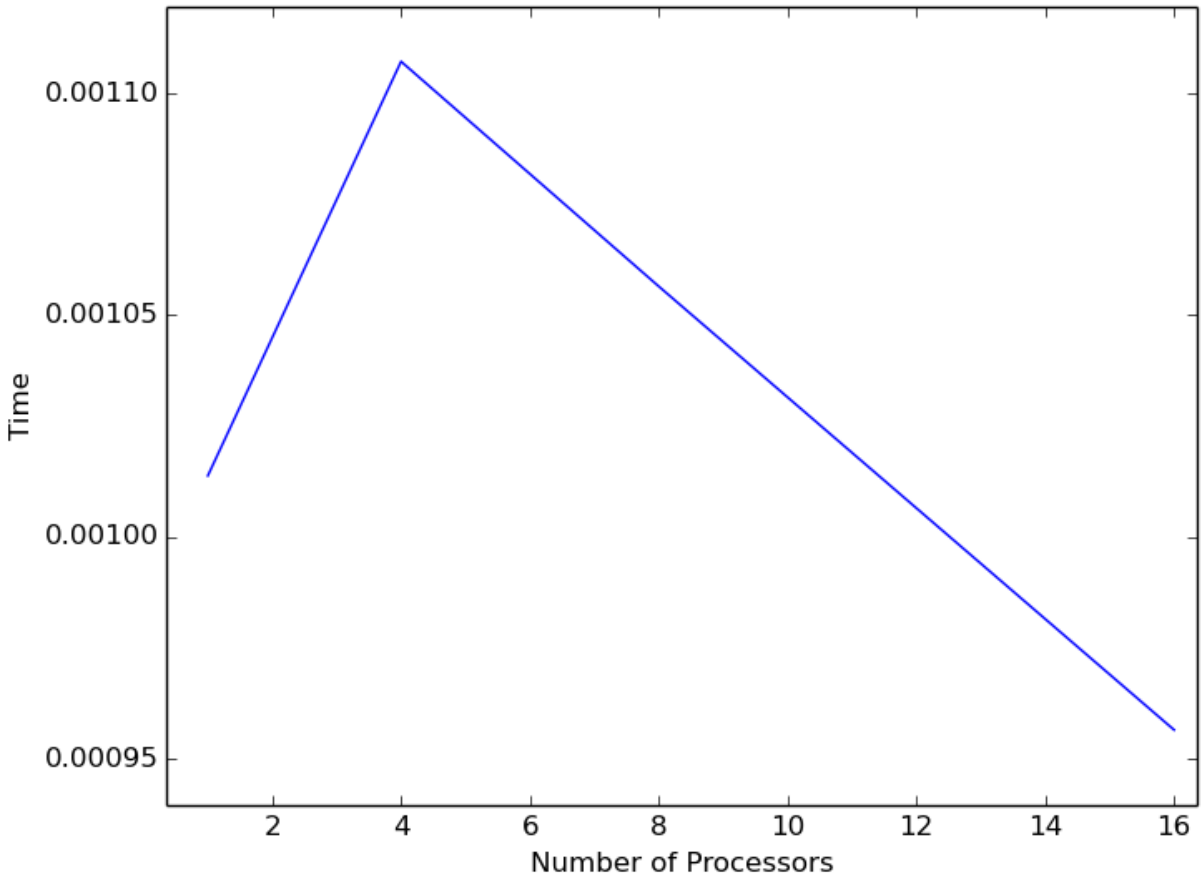


Figure 1: Time vs Number of Processors

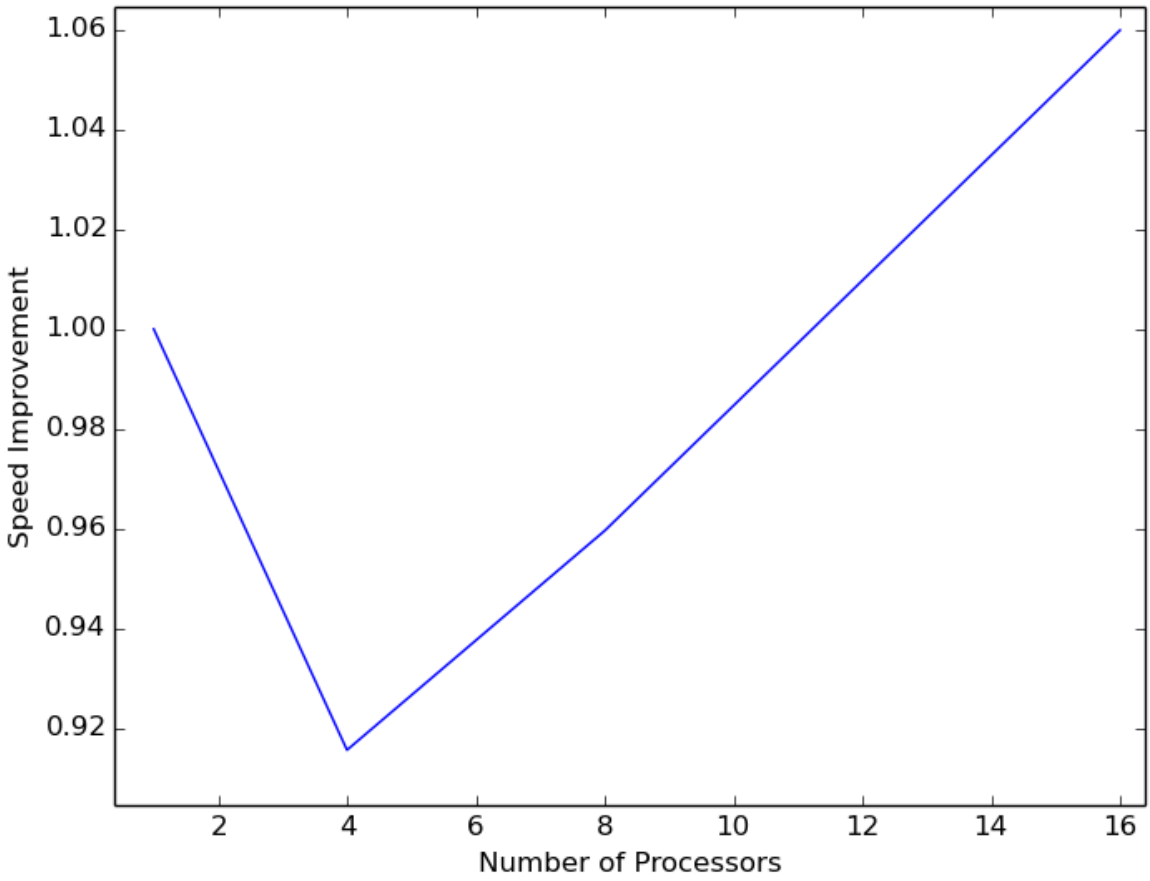


Figure 2: Speed Improvement vs Number of Processors

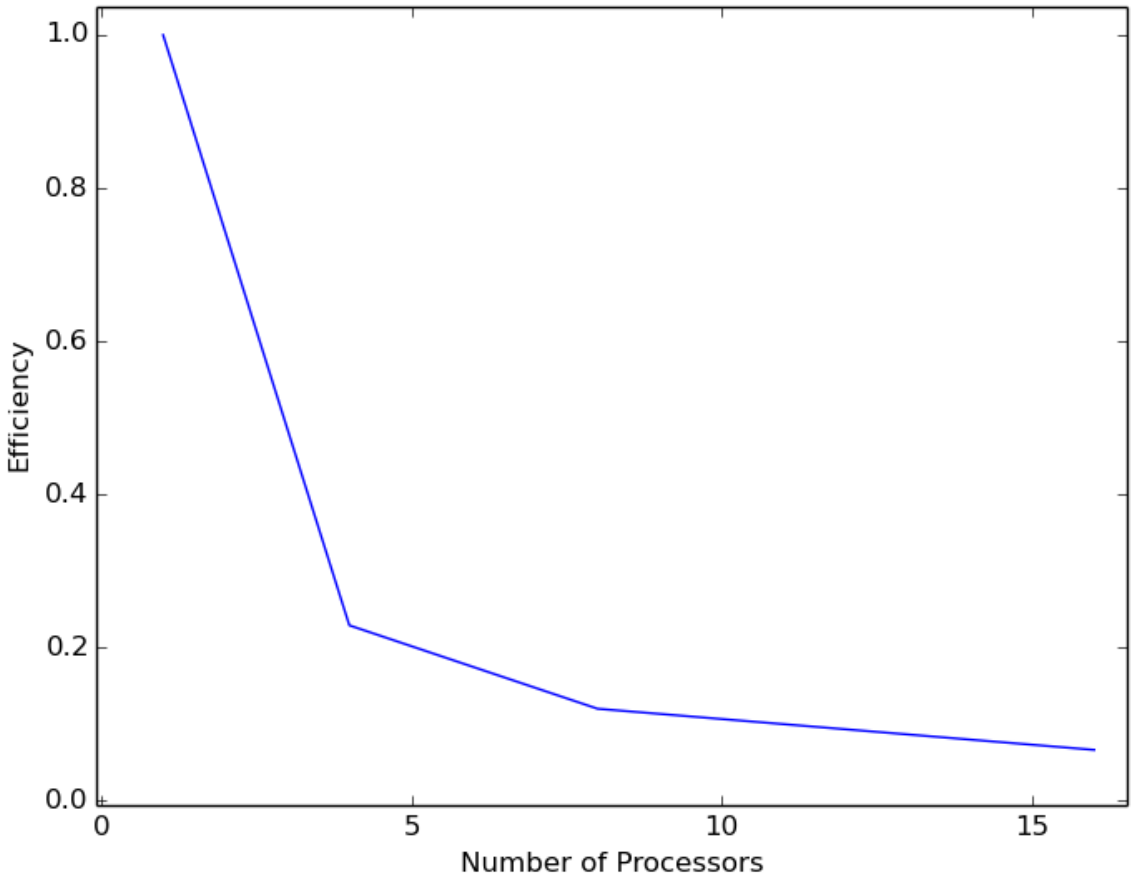


Figure 3: Efficiency vs Number of Processors

where $T_s$ is the running time of the sequential algorithm and $T_p$ is the running time of the parallel algorithm.

The efficiency is defined as Speedup divided by number of processors. As can be seen in Figure 3, we observe a drop as the number of processors increase. The parallel overhead increases and the total performance becomes less than the sum of the individual contribution of each processor.

To increase the performance, we can add more processors to reduce the running time. We can also try to improve the algorithm. If we remove the MPI functions for just comparing the numbers in two different processes and try to aggregate them, we could achieve a more efficient program. That is, if we have a small number of processors such as 8, we can just use `MPI_Reduce` to send all of the local maximum values to a single processor. If we had several hundred processors, we could again use `MPI_Reduce` and send the numbers from 8 processors to a single processor, and the algorithm will contine until only one processor remains. This will lead to a better complexity. For instance, in the second term of the complexity value I have written in the beginning of the report, $log_2 p$ becomes $log_8 p$ (assuming 8-fold is employed).

A note about the values in submitted `.txt` files is that I have run the experiments 10 times each and calculated the average of the running times. Then, I put the average value to the lines with `Max_Value`. I have also managed to pick a time slot where no one is working on `slurm` machine to minimize interference.