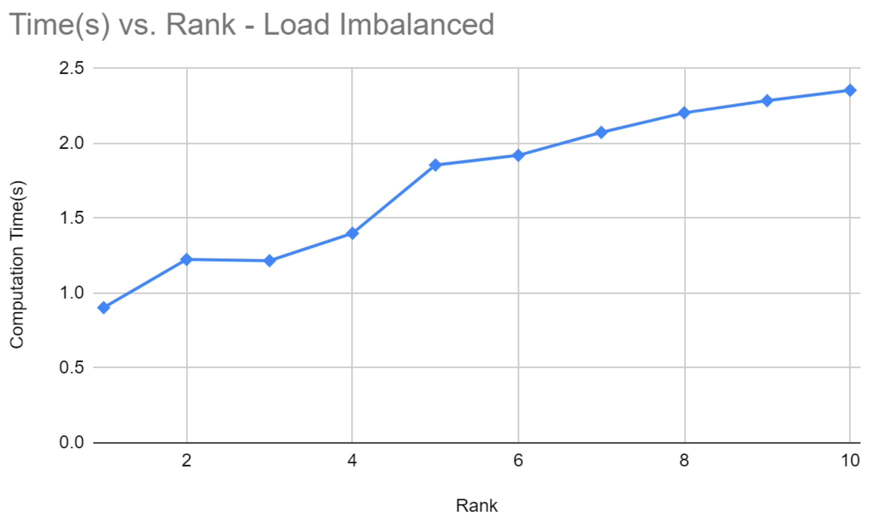
LUIS GJURAJ 7/19/2023

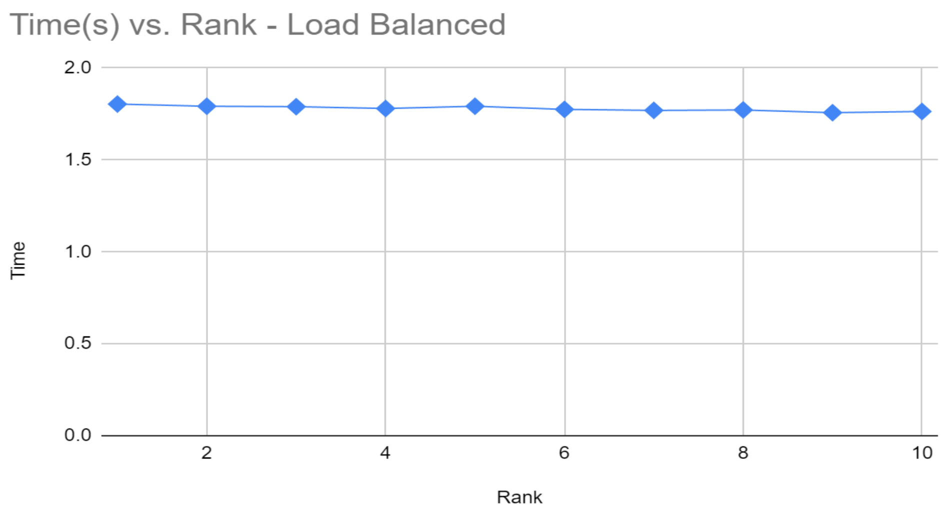
After running the initial program without parallelizing it, I found out that it took 15.647029 to compute the number of primes from 1 to 10 million.

Then, I parallelized it using the master-worker approach and dividing the same amount of work we had in the initial program in 10 equal portions (for 10 worker ranks) between them.



As we can see from the plot, each rank completes the task with an increasingly longer time the later they come in the loop for receiving values. That is because the initial ranks take in values that are much smaller than the values that the ranks at the end of the loop are given. Because of that, the initial ranks complete their operations faster than the latter ones.

After balancing the load of work that each rank needed to do by utilizing a buffer with a size of 5000 elements (chosen semi-randomly) and leaving everything else the same, we can see that now all the ranks take an almost equal amount of time to finish their work as shown in the plot below.



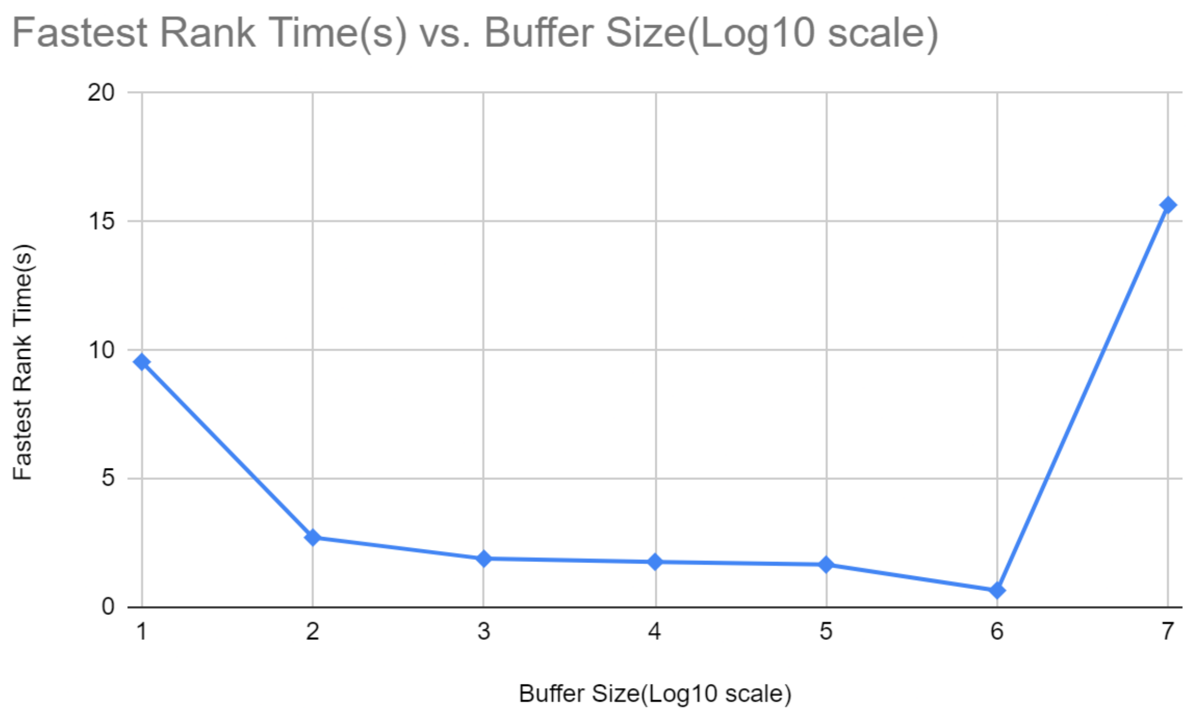
LUIS GJURAJ 7/19/2023

Lastly we were asked to observe the fastest rank compilation time in comparison to the buffer size of the MPI messages. By keeping everything the same and only changing the buffer size again, we were able to see a drastic difference between the compilation times of the fastest ranks for each run of the program.

We can see that with a very small buffer size of 10 elements per message, the time spent for the fastest rank to finish is around 9.5 seconds, which is much slower than almost all of the other runs. The slower performance must be due to the sheer amount of overhead being created by having these much smaller message sizes, as it leads to a much bigger number of messages being sent back and forth between master and worker.

The closer the buffer size becomes to an equally distributed workload, we notice that the fastest rank's time becomes increasingly smaller, and the difference between it and other ranks in the same run becomes bigger just like in the load imbalanced version of the program.

Any bigger than the equal distribution, we notice that gradually less ranks will be utilized even if they are available, as the workload now shifts toward the initial ranks which take most of the workload before it can be distributed anywhere else. This leads to an increase in time as the buffer gets bigger.



Does that mean that an equally distributed workload is the better choice? No. Although we see the fastest worker in the plot above, we do not see the times of the other workers. If we want to find the optimal buffer size, we have to switch the values that come from the fastest ranks every run to the slowest rank in every run, since we know that the program does not terminate until the last rank is finished with its task.