Project: Part 2

Xingku Yin
Alexandru Marcu

Background:

As in Part 1, we are still assuming the same task, which is to parallelize a toy particle simulator. In this part of the report, we are tasked with creating one distributed memory implementation, using MPI (Message Passing Interface), that runs in O(n) time, with as close as possible to O(n/p) scaling (where p is the number of processors).

We are given a naive MPI implementation. It is naive for the reasons that after initializing and distributing the particles among the processors, at every time stamp of the simulation all processors gather all the particle data globally, and then apply the naive O(n^2) run time computation algorithm. These two areas are where we will try to make the most improvement on.

To run our MPI implementation, first run the script ./setup-project.sh . Then, to run the code, it is identical to the method outlined in the hand out. Use the command sbatch auto-teach-mpi-* , where auto-teach-mpi-* are the provided scripts for the small and large datasets.

MPI Implementation And Discussion:

Our first implementation to simply replace naive apply force loop with the one we wrote for the O(n) serial implementation in part 1. The result is similar to the original code from source file. This is documented in mpi_plan0.cpp.

In the second implementation, we divided rows of bins to processors horizontally. For initialization, we let rank 0 processor to distribute the initial particles. Particles are send to processors. For each step, processors will remove particles that move away from the rows it is assigned and send them to the corresponding processor. Then, each processor will send their particles at edge as particles in the “ghost” zone to their neighbouring processors. Processors will apply force to their local particles and move them.Then the next step begins. If the “-no” flag

is off, we will gather all particles at rank 0 at the save step. The result for 160000 particles is around 12s, so we tried to improve it.This is documented in mpi.cpp as well as mpi_plan1.cpp

The improvement we tried is to send the particles to move to other processors and the particles at edges at once. We have half the number of send and receive. However, the time spend for 160000 particles is still around 12s. This documented in mpi_plan2.cpp.

In our final two MPI implementations attempt, we unfortunately fall short of the benchmark given in the project handout. The given benchmark is ~2.6 seconds for 160,000 particles, and our implementation does ~12 seconds for 160,000 particles, which is about ~4.6x slower. We notice that our MPI implementation is similar to our OpenMP implementation in terms of timings.

Another reason might be we are using MPI_Probe to detect the receiving message size which is basically a MPI_Resv but not actually receiving the message. This would add another weight to the communication overhead. There should be better ways to eliminate this weight but for now, this the one we are working with.

After some thought, we hypothesize that we lose a lot of time due to the fact that our implementation uses multiple vector containers. With these, and due to our strategy of communication, we need to use the vector functions erase(), insert(), resize() too often, which are all O(n) operations. This makes our implementation computationally costly.

Along with that point, we think that our method of communication in MPI incurs too much overhead. In our implementation we make use of MPI_Probe() to anticipate the number neighboring particles a processor will receive. This usage of MPI_Probe() adds another layer of communication overhead. We broke up the run time into computational time and communication time to take a look at how long of the run time our communication strategy uses. For this, we mainly concern ourselves with the weak scaling, large dataset, as we feel it gives us the best insight.

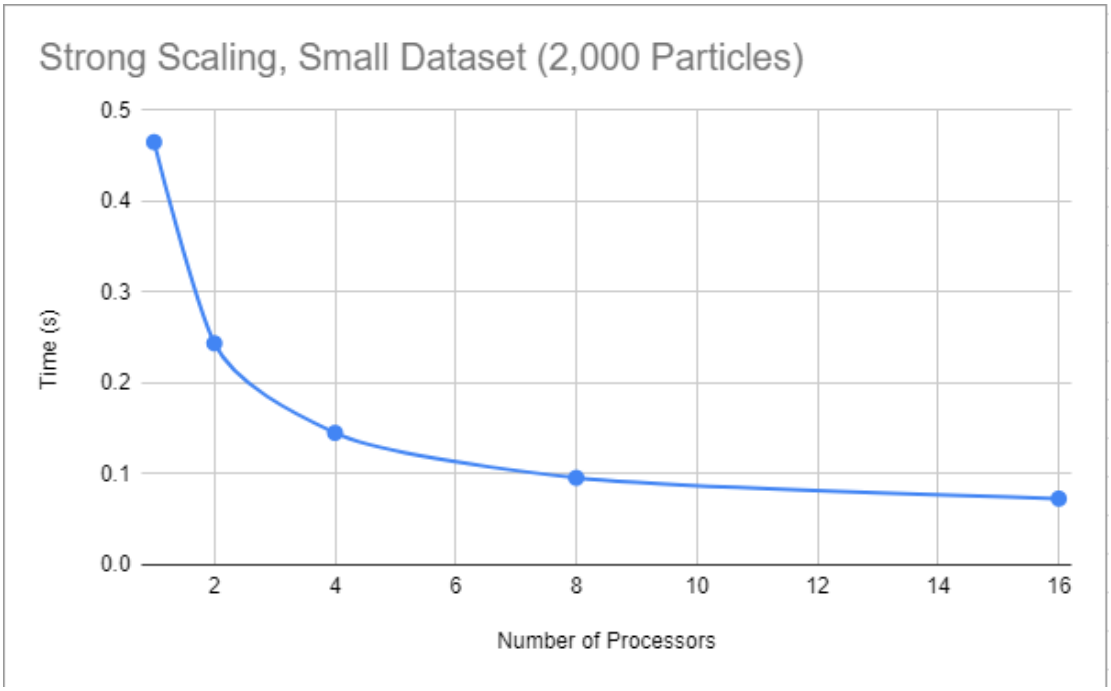
# of Processors	# of Particles	Simulation Time (s)	Communication time (s)
1	10,000	2.45402	0.209512
2	20,000	2.64236	0.454289
4	40,000	2.82684	0.95080
8	80,000	6.16428	2.295457
16	160,000	12.141	5.17985

Looking at the data, the communication overhead becomes very apparent. Everytime the # of processors are doubled, there is roughly a 44% increase in communication time. On the largest dataset of 160,000 particles, the communication time takes roughly 43% of the run time!

This seems to us much too high, and further investigation into optimizations and communication strategy should be explored.

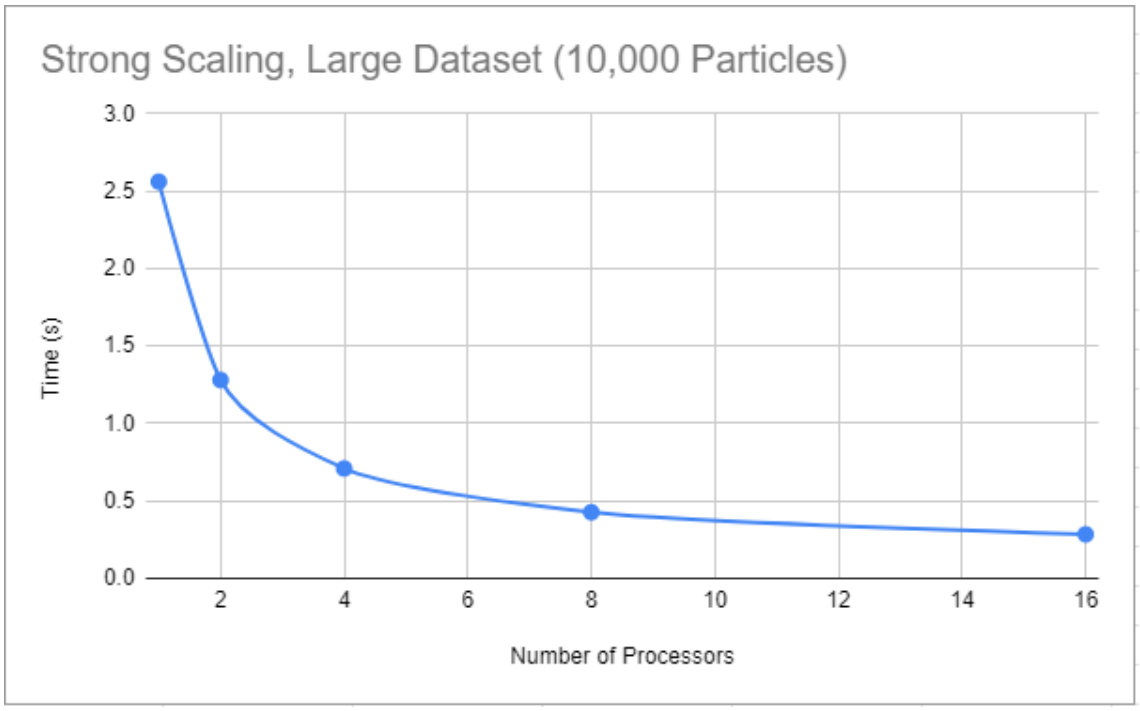
MPI Data & Analysis:

Strong Scaling Run Time Performance:



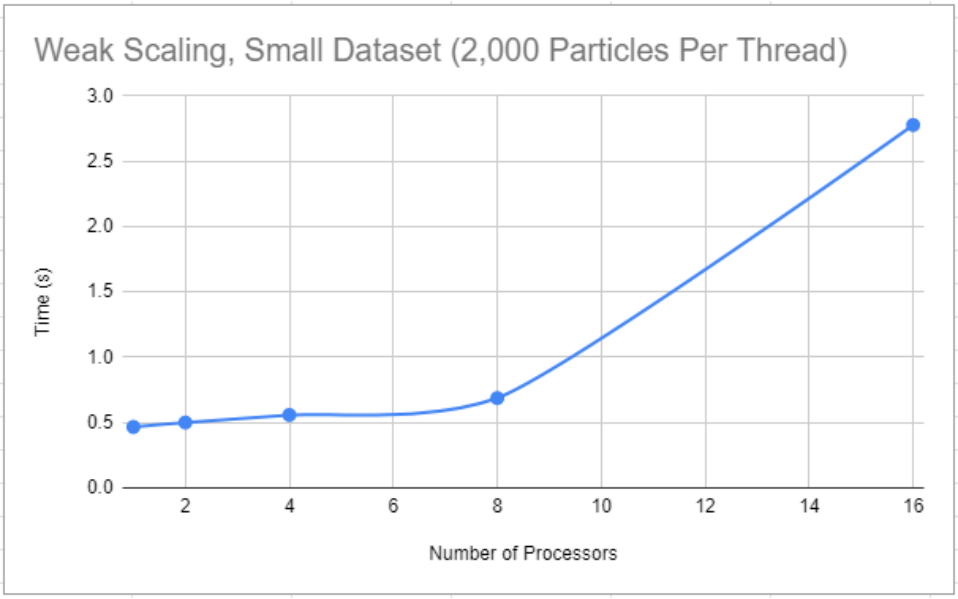
As discussed previously in part 1, the method of Strong Scaling is to keep the input size constant (in this case we keep the number of particles constant to 2,000), while increasing the number of processors (in powers of two, from 1 to 16).

Looking at the performance of our MPI implementation on the small dataset (using auto-teach-mpi-small), we note that we see a desirable trend - the more processors, the faster the code runs. For a singular thread, we get a run time of 0.465085 seconds, slightly slower than the serial implementation with a singular thread, but this is due to overhead of initiating MPI. Compared to the singular thread of the naive MPI implementation which was 15.1972 seconds, we see a drastic improvement. We continue to see this drastic improvement at the other end of the spectrum as well, with the 16 thread run time from the naive implementation of 1.127931 seconds to our improved 0.072804 seconds.

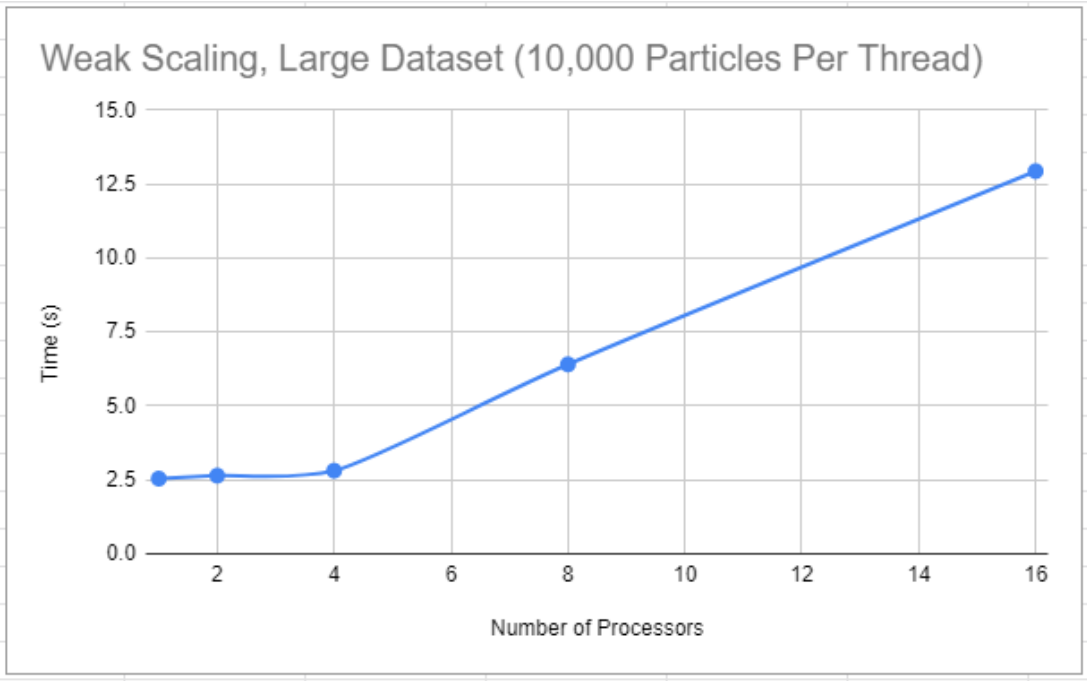


Now we turn our attention to the Strong Scaling performance using the Large Dataset (10,000 particles). The large dataset being tested using auto-teach-mpi-large. Again, we see a desirable curve. Considering that the naive implementation does not seem to complete in any sort of reasonable time, we feel as though we have made a good improvement for this dataset, as for 16 processors we achieve a run time of 0.283835 seconds.

Weak Scaling Run Time Performance:

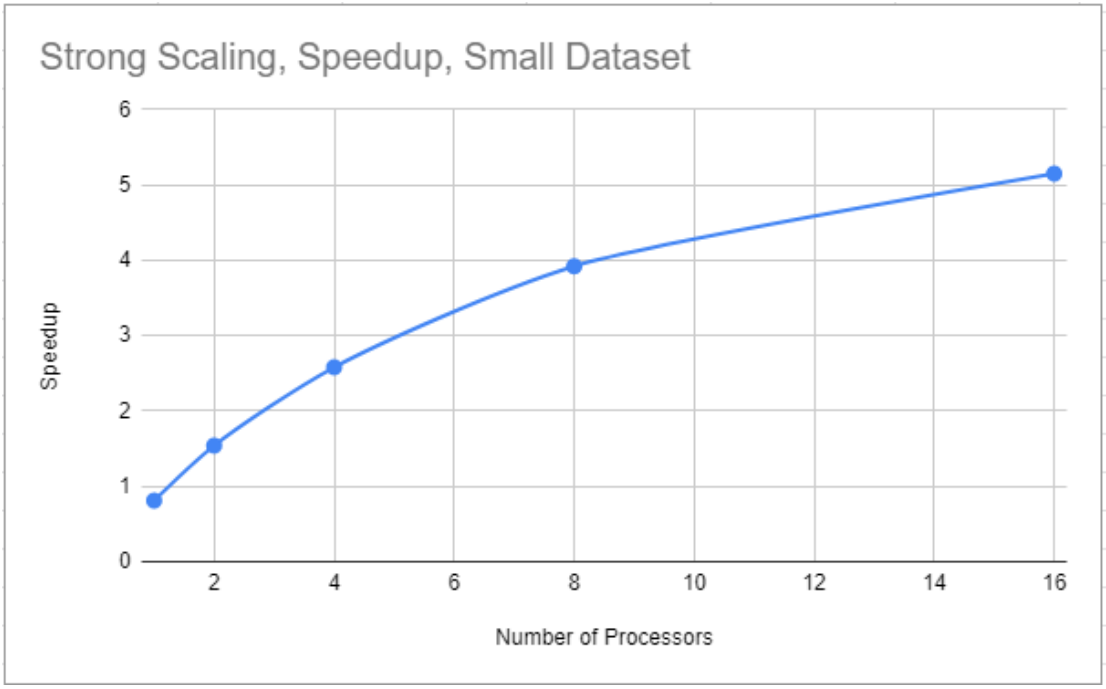


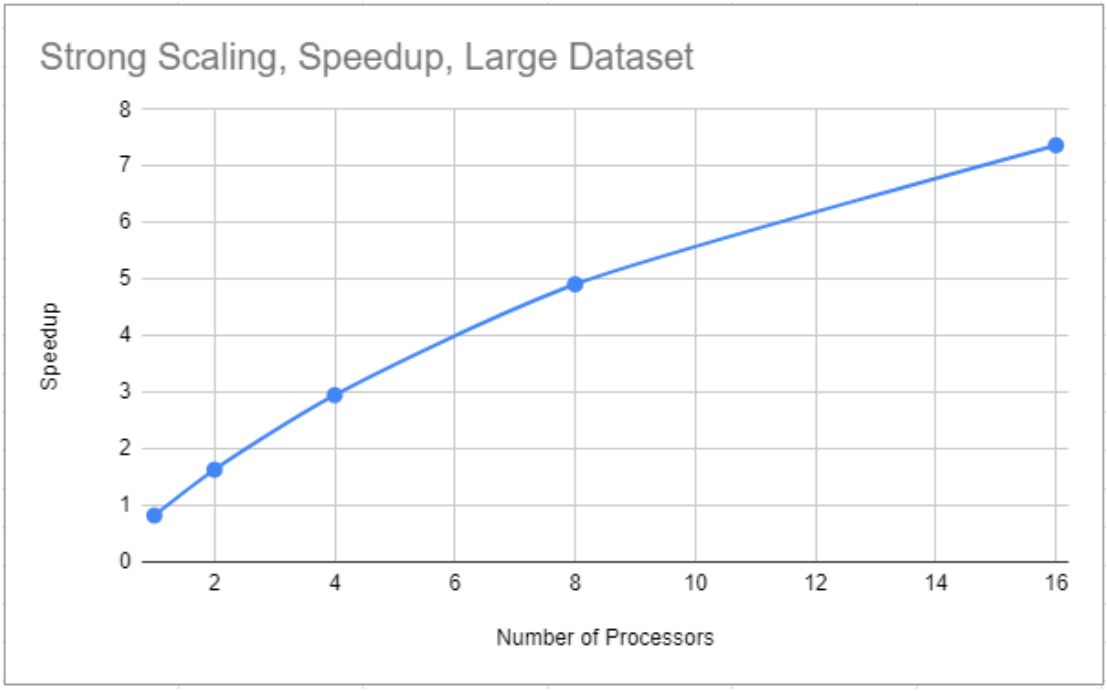
Weak Scaling involves testing the performance by proportionally increasing the work per processor, as the number of processors increases. For the small dataset (auto-teach-mpi-small), this is 2,000 particles per thread. That is, with only a singular processor, we are running with a small number of particles, only 2,000 of them. For 16 processors, this will then be increased to 32,000 particles. We would expect to see a more linear slope for this graph, but it seems as though for processor counts of 1, 2, 4, 8, there is not much variance in run-time, until we have a sharp increase for a processor count of 16.



For the Large Dataset (auto-teach-mpi-large), we are testing the performance with 10,000 particles per thread. Here, we see a trend that we are more expectating to see, a linear trend between processor counts of 4,8, and 16. This might suggest that our implementation scales quite poorly. As well, we note that we do not achieve the benchmark given in the handout for a decent optimization (~2.6s for 160,000 particles), and tells us that although compared to the naive implementation we have made a good optimization, in reality our optimization is still poor (it is 6x slower than the benchmark). That being said, the timing is on par with our OpenMP implementation discussed in part 1 of the report.

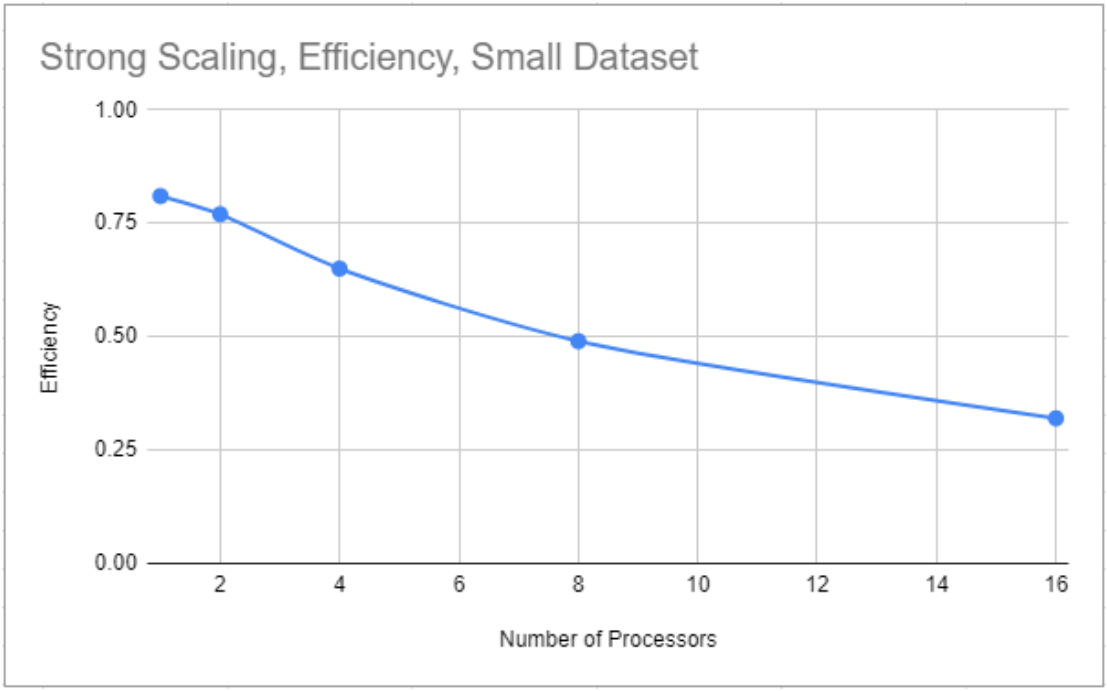
Strong Scaling Speedup Analysis:

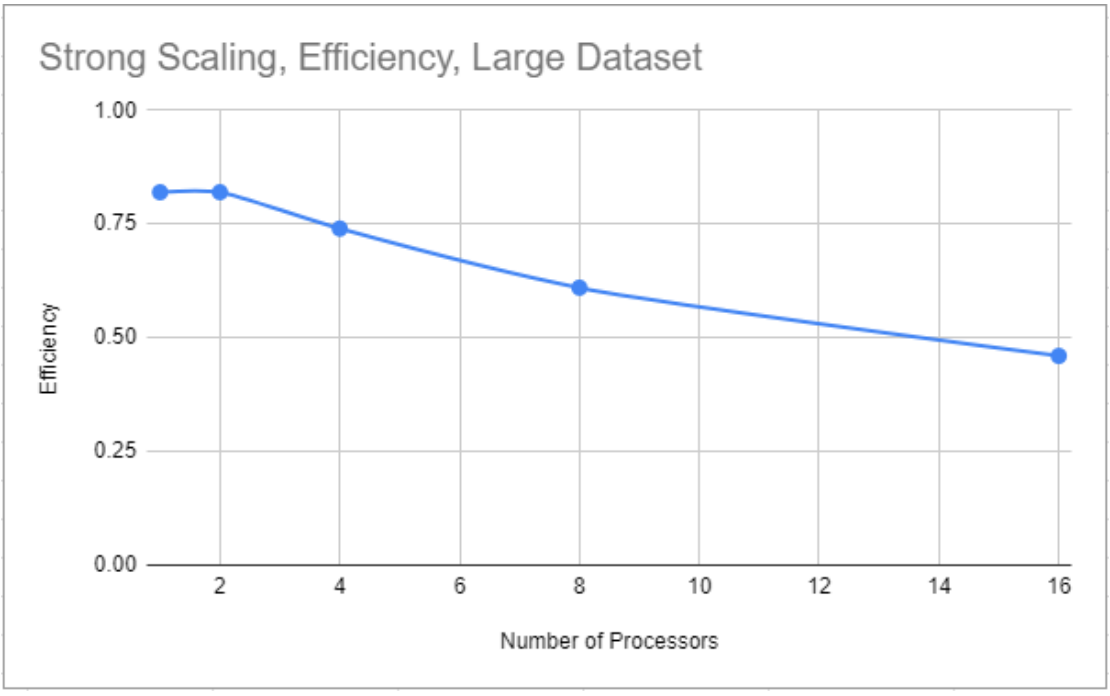




Looking at the data for Strong Scaling Speedup, we can see that the algorithm scales decently well for these data sets. This is evident because as the number of processors increases, the speedup is increasing at a similar rate, which is precisely the trend we wish to see.

Strong Scaling Efficiency Analysis:

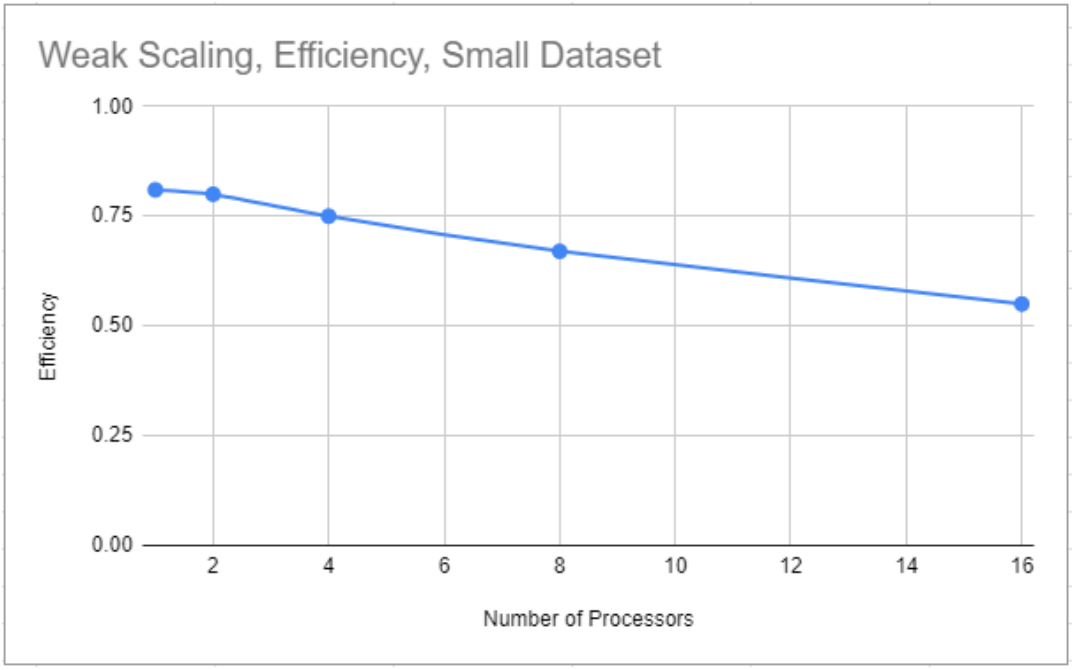


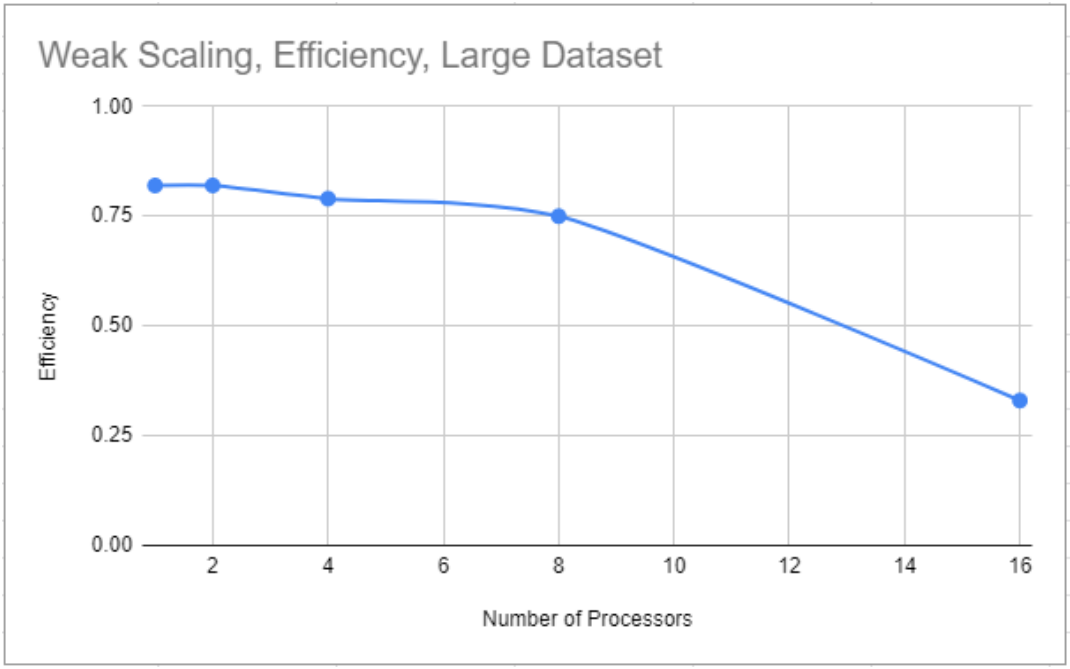


We observe that for the large dataset we get overall better efficiency across all processor counts. We think this is due to the fact that as the dataset gets larger, our implementation is better utilized. With the smaller dataset, the amount of communication

outweighs the benefit of using the algorithm on all cores, and we think that with a larger dataset, the communication overhead and MPI overhead becomes more negligible, so we see better efficiency.

Weak Scaling Efficiency Analysis:





In comparison with Strong Scaling, we see, overall, better efficiency with Weak Scaling. This tells us that as our dataset increases in size (in Strong Scaling the problem size remains constant) our algorithm better utilizes the resources of the system. However, we do have a significant outlier. For 16 processors, that is, 160,000 particles, we have very low efficiency, with a value of .33. This is less than half for almost all other particle sizes and processor counts. This suggests that our processors are spending a lot of time idle as the domain of the problem increases.