CSC367 Project Phase 2 Report

MPI implementation

Our MPI implementation can be described in the following steps:

1. Horizontally partition the whole map based on the number of processes and size of the map.
2. Master process initializes all the particles and broadcasts them to the other processes in the array **all_particles**.
3. Each process manages three arrays: **local_particles** which represents the particles in this process's region. **above_ghost_particles** which represents the particles in the ghost zone above the process's region and **below_ghost_particles** which represents the particles in the ghost zone below the process's region. Every process iterates through the array **all_particles** and assigns them into the three arrays.
4. Each process initializes local grids including ghost zone.

   for each timestep:
   1. Each process allocates **local_particles**, **above_ghost_particles** and **below_ghost_particles** into the grids.
   2. do collide computation except the ones in the ghost zone grids.
   3. clean nodes in each grid (grid is a linked list)
   4. move the particles in **local_particles**.
   5. Exchange the information with neighbor processes. Update the **local_particles, above_ghost_particles** and **below_ghost_particles.**

Note for not fully implemented part:

For the mpi.txt: we did not maintain particles' locations at each step in this txt file since we did not have the time to fully implement this part. What important is that we have made sure that particles are interacting correctly. Our current MPI implementation has passed all correctness tests (by checking absmin and absavg) when running job-teach-mpi, auto-teach-mpi-small and auto-teach-mpi-large.

We also test the correctness of our code by checking each particles' velocity and acceleration at each step (by printing our these values and compared with the started code's output after setting a seed). Therefore, we confirmed that particles are interacting correctly.

Communication in the distributed memory implementation

In our distributed memory implementation. The whole simulation map is horizontally splitted into N regions where N is the number of processes. For every iteration of the simulation, each process needs to send the particles that move out of the region to above or below neighbors, and receive the particles that come into the region from the above or below neighbors; then send the particles in the ghost zone for above and below neighbors and receive the particles in ghost zone from above and below neighbors.

All of these communications are non-blocking communication (asynchronous), we use MPI_Wait operation to ensure all the communications are finished before running into the next iteration.

Design Choices

- Horizontal partition the whole map

  Regarding the method on partitioning the whole map, we had two choices: either horizontally partition the whole map or block partition the whole map (just like the two methods we talked about in lecture)

  We choose the horizontal partition instead of the block partition because we are doing particle simulation. And the goal of partitioning is to minimize the communicating data. In our case, each neighbor process only exchanges a very small amount of data every iteration despite the surface to volume ratio. Thus we want to minimize the amount of data exchanged by minimizing the number of neighboring processes. And this can be achieved by horizontal partition.

Experiments and Results for: MPI

For the **strong scaling**, we keep the particle size constant and vary the number of processes to be 1, 2, 4, 8 and 16. For the small dataset, we set this particle size to be 2000. For the large dataset, we set this particle size to be 10000.

For the **weak scaling**, we increase the problem size proportionally with the number of threads. For the small dataset, we set the particle size to be 2000, 4000, 8000, 16000 and 32000, while the corresponding number of processes are 1, 2, 4, 8 and 16. For the large dataset, we set the particle size to be 10000, 20000, 40000, 80000 and 160000, while the corresponding number of processes are 1, 2, 4, 8 and 16.

Below are the results we got:

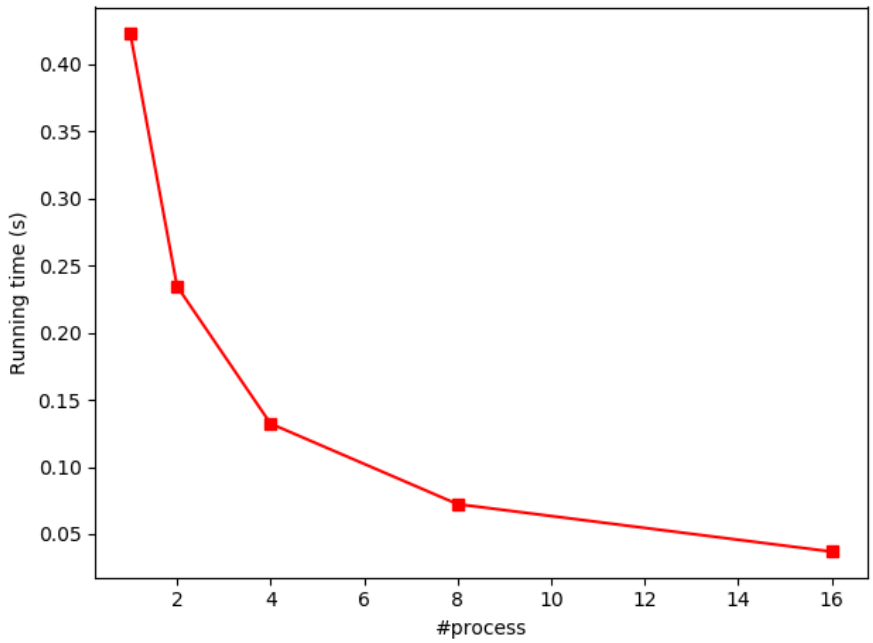
Our MPI implementation can finish the simulation for 160000 particles **around 2.9 seconds** with 16 processes, which is about the same as the reference performance provided in the handout.

- A table that shows the average weak and strong scaling efficiency of our optimized MPI implementation.

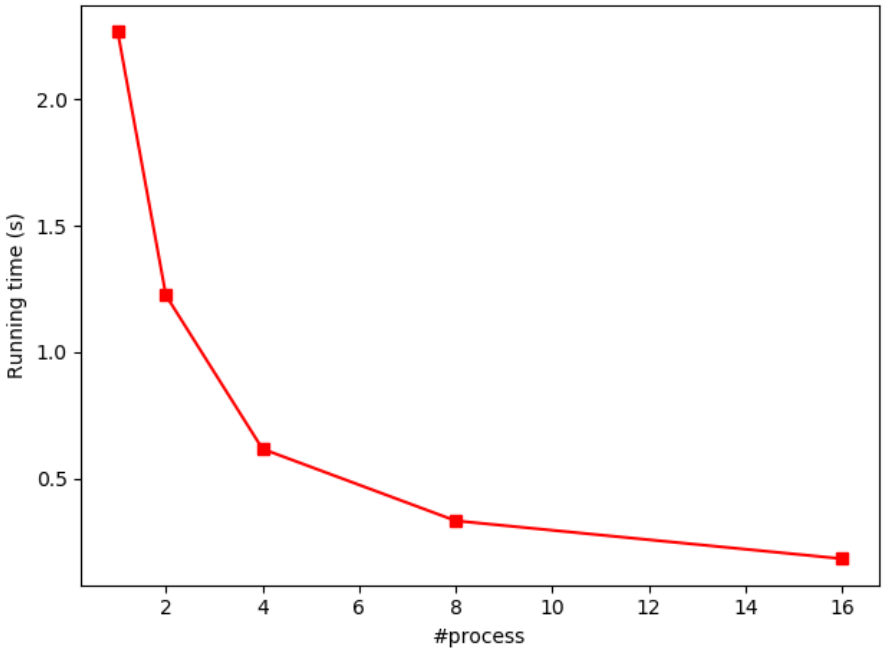
	Average strong scaling efficiency	Average weak scaling efficiency
Small Dataset	0.71	0.75
Large Dataset	0.74	0.76

- The strong and weak scaling plots that show the running time of our optimized MPI implementation.

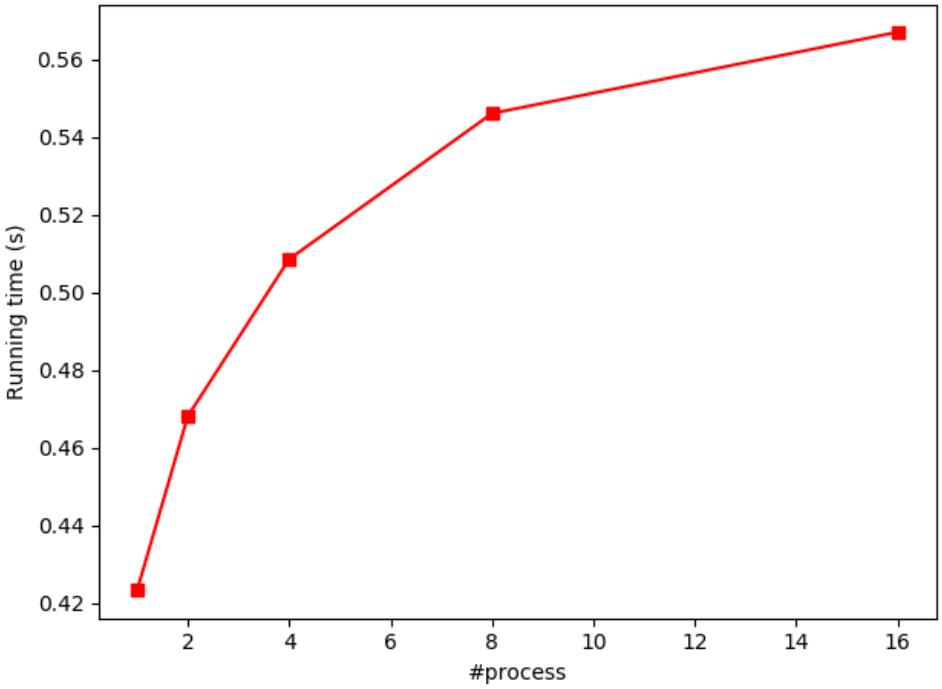
Strong scaling plot of running time for the small dataset



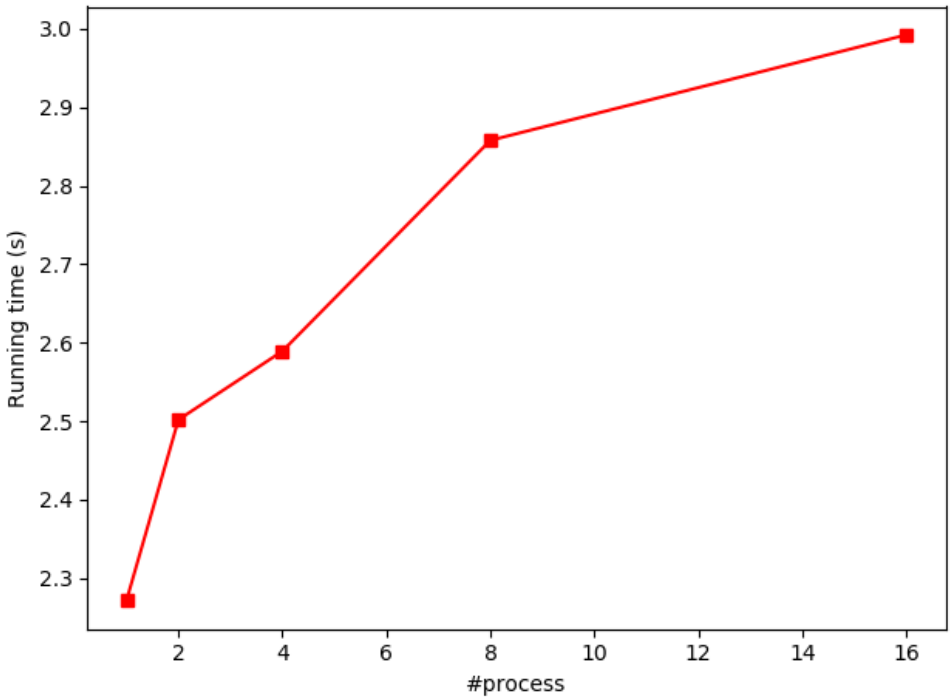
Strong scaling plot of running time for the large dataset



Weak scaling plot of running time for the small dataset

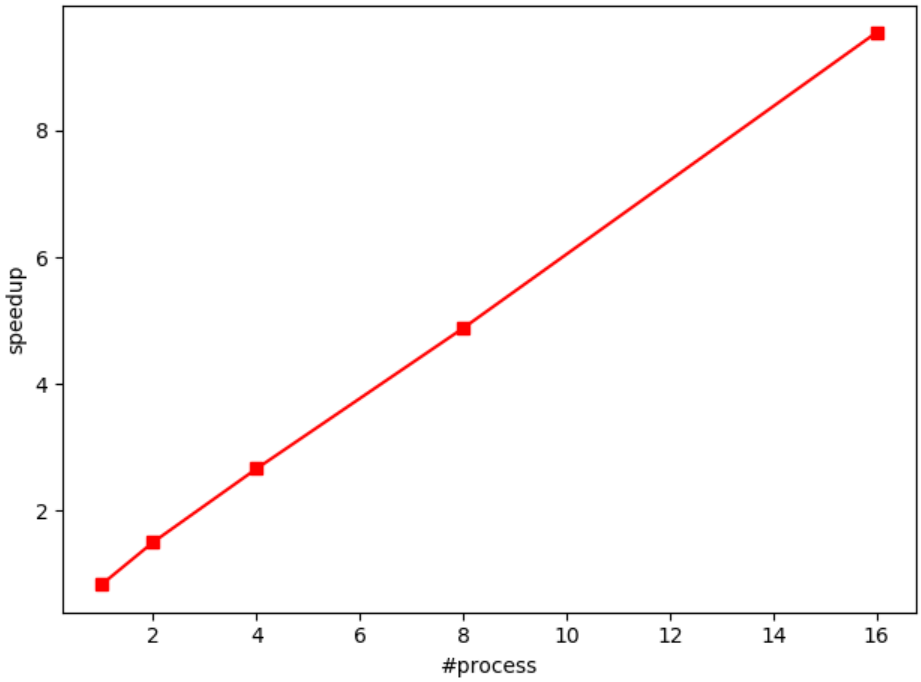


Weak scaling plot of running time for the large dataset

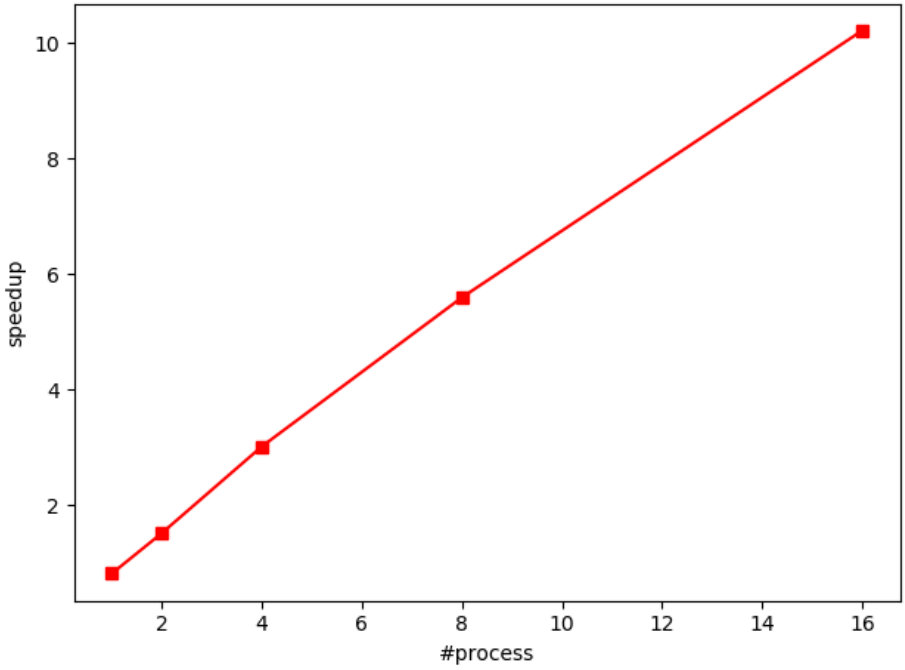


- The strong scaling plots that show the speedup of our optimized MPI implementation.

Strong scaling plot of speedup for the small dataset

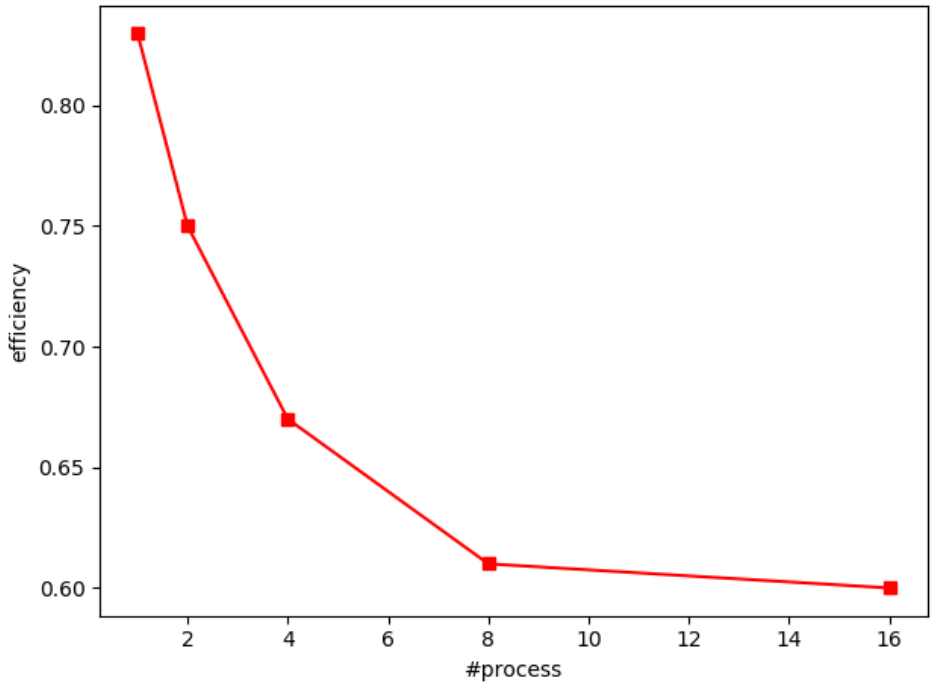


Strong scaling plot of speedup for the large dataset

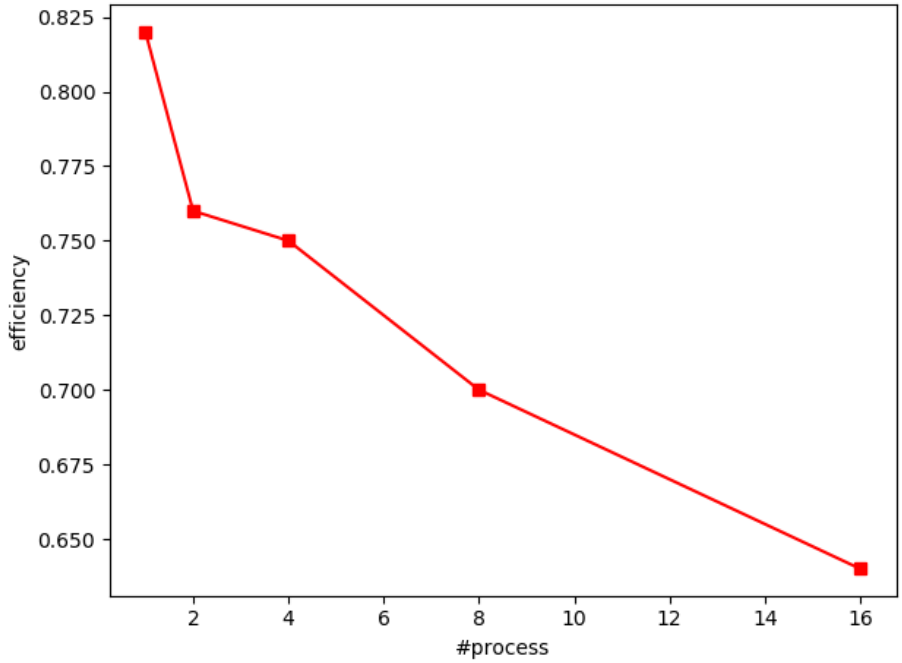


- The strong and weak scaling plots that show the efficiency of our optimized MPI implementation

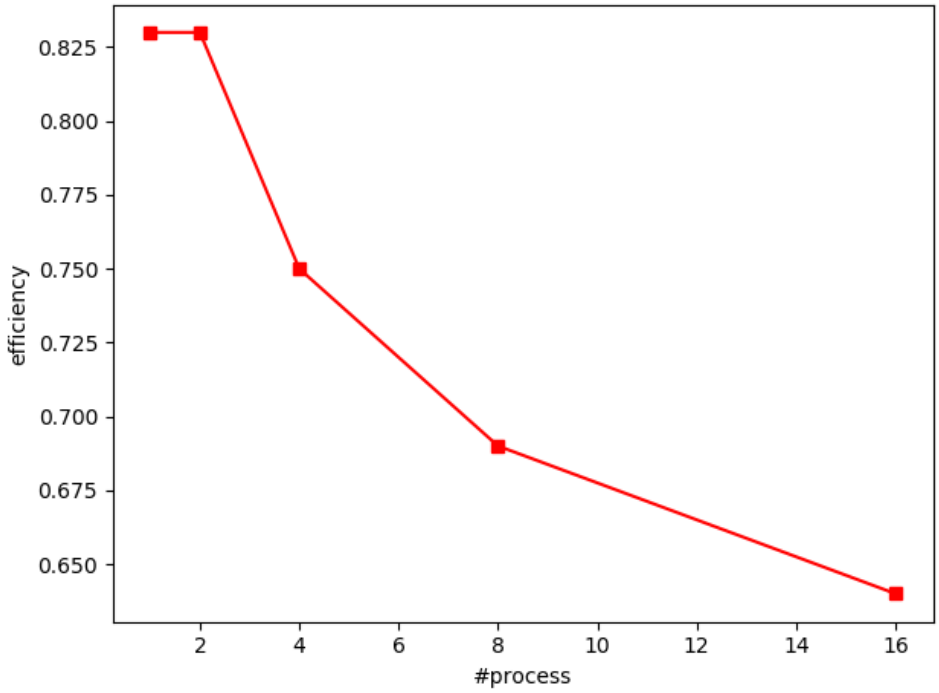
Strong scaling plot of efficiency for the small dataset



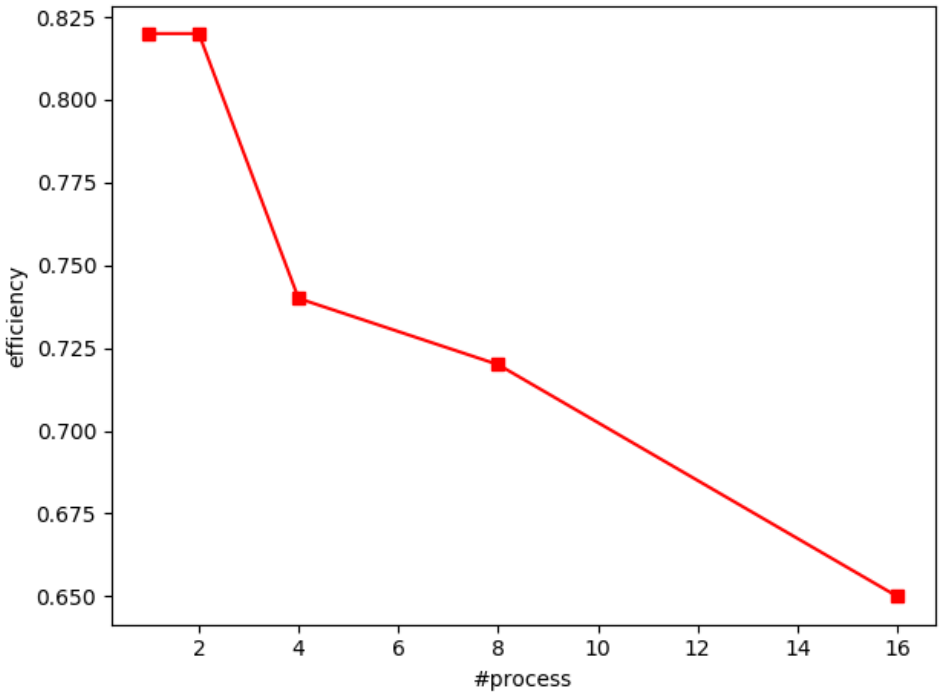
Strong scaling plot of efficiency for the large dataset



Weak scaling plot of efficiency for the small dataset



Weak scaling plot of efficiency for the large dataset



Analysis:

For the strong scaling plots for running time, we can see that the running time keeps decreasing as the number of processes increases. We see the same pattern with both the small dataset and the large dataset.

For the weak scaling plots for running time, we see that the running time increases a bit as the particle size grows proportionally with the number of processes. We see the same pattern with both the small dataset and the large dataset.

The strong scaling speedup plots show a linear pattern. The speedup increases linearly as the number of processes increases. The pattern is the same with both the small dataset and the large dataset.

For the strong scaling efficiency plots, we see that the efficiency decreases as the number of processes go up. The pattern is the same with both the small dataset and the large dataset.

For the weak scaling efficiency plots, we see that the efficiency is about the same when the number of processes is 1 or two. The efficiency decreases afterward as the number of processes goes up. The pattern is the same with both the small dataset and the large dataset.

There are two major observations we made in our experiments.

In the strong scaling plots, especially the speedup plots, we see a performance increase as the number of processes increases.

For the weak scaling plots, even though work/processor stays the same for each data point in the plots, we still see performance drops as the number of processes increases. This can be explained by the overhead of opening extra processes, extra communication since we have an increasing number of processes and the overhead caused by the synchronization with "MPI_Wait".

Suppose p is the number of processors and idealized p-time speedup to be the speedup equalled to the current number of threads. If we compare with the idealized p-time speedup, our MPI implementation has a speedup of about 85% of the idealized p-time speedup when p = 1, around 75% - 80% of the idealized p-time speedup when p is 2 and 4, around 70% of the idealized p-time speedup when p increases to 8, and less than 70% of the idealized p-time speedup when p is 16. Basically, even though our implementation's speedup is increasing as the number of processes increase, when we compare with the idealized p-time speedup, the ratio of this speedup and the idealized p-time speedup is decreasing as p increases.

This difference between our speedup and the idealized p-time speedup can be explained by context switching overhead, communication overhead between processes, some synchronization overhead because of the use of "MPI_Wait" and possible load imbalance (each partition contains different number of particles). It is possible to achieve a better speedup. One possible approach that might help the performance is fixing the load imbalance issue that may occur sometimes. We may dynamically partition the map based on the distribution of particles. In

this way, each process will have almost even workloads. This approach will mitigate the load imbalance and increase the speedup.

Our plots, especially the speedup plots, also show better performance with bigger datasets. This is expected as the **Amdahl effect**, for the same number of processors, the speedup is usually an increasing function of the problem size.

Computation time and communication time

In our MPI implementation, if we break down the runtime into computation time and communication time, a larger portion of time is spent on the communication of different processes. As we can see in the strong scaling plots of the efficiency, the efficiency keeps decreasing as the number of processors (we call it "p") increases. Since we know that, for the strong scaling, the particle size is constant. Then the total amount of time we need to spend on computation time theoretically will be roughly the same. Given that, the efficiency keeps decreasing when p goes up, we must have spent more and more time on the extra communication between processes. Hence we conclude that we have spent more on data communication than on computation. We have already discussed the partitioning strategy to reduce the communication cost in the "design choice" section. As we discussed in the "analysis" section, another possible way to improve the communication cost is to dynamically partition the map based on the distribution of particles. Please refer to the previous section for the detailed discussion.