CS267 HW2-2: Parallelizing a Particle Simulation

Grace Wei, Yinjun Zheng, Emin Burak Onat (equally contributed)

Spring 2023

1. Introduction

In this assignment, we implemented an algorithm to parallelize a particle simulation where particles interact through repulsive forces. The asymptotic complexity of the naive approach would be *O(n^2)*. We have improved the code to run in *O(n)* time on a single processor by the same way as in HW2-1. Then we further parallelize the code to reach to *O(n/p)* time complexity when using *p* processors through distributed memory parallelism (MPI).

2. Algorithm Description

2.1 Algorithm:

The algorithm we use for achieving *O(n)* time complexity is the same as in HW2-1. We divide the 2D space into square bins with dimensions BIN_SIZE and allocate particles to each bin. To compute forces on particles of one bin, we only consider particles in that bin and in the neighboring eight bins.

To parallelize the code to achieve *O(n/p)* time complexity, we further allocate the bins into different processors, with each processor having its own memory. Each processor receives a set of bins. In our final algorithm, the bin space is divided into rectangular chunks. This is to minimize the communication at the processor boundaries.

Initializing the Simulation:
To initialize the simulation, we first initialize the subdomain of each processor, and then initialize the local bins that belong to each subdomain. The local bins include the bins exclusive to each processor as well as ghost bins that overlap with other processors. We add particles to only the bins that are initialized in the domain. Additionally, we also tell the processor the ranks of its neighboring processors and the indices of the ghost bins that will be sent to each neighboring processor. We can assume that particles will move only one processor away, so the communication will only be performed between the current processor and its eight neighboring processors.

Simulating One Timestep:
First, we send the particles that belong to the ghost bins of neighboring processors to the neighboring processors via a non-blocking send. We first send the number of particles to be sent, and then the particles themselves. The neighboring processors then perform a blocking receive for the number of particles, followed by a non-blocking receive for the particles themselves. We then clear the current particles in the ghost bins, and bin the new ghost particles received from neighboring processes.

After obtaining the correct ghost particles, we compute the forces on the particles in our bins serially, taking into consideration each bin's interaction with its neighboring bins. After computing forces on all the particles for that processor, we iterate through each bin and move each particle. When we move the particle, we keep track of whether it is leaving the subdomain of the processor, and which neighboring processor rank it is leaving to.

Finally, we rebin all the particles that are staying in the processor and perform an Alltoall call to send the particles leaving the processor to all the other processors. Each processor then rebins the particles received.

2.2 Data Structure:
(1) bins: We divide the 2D space into square bins of size BIN_SIZE. We are using data structure *vector<vector<particle_t>>* to represent the bins. Vector bin[i] represents the i-th bin, and inside the vector bin[i] are the particles that are located in the i-th bin.

(2) send_bins, receive_bins: Send-bins are the bins each processor needs to send to the neighboring bins to update their ghost bins at each timestep. We use a map<int, vector<int>> to represent the send_bins. The key is the rank of the processor that communicates with the current processor, and the value is a vector containing the indices of bins that we want to send to the respective neighboring processor. Similarly, we use the same data structure for receive_bins to represent the bins that are received by the neighboring processors.

(3) send_parts, receive_parts: Send_parts are the particles each processor needs to send to the neighboring bins to update their ghost bins at each timestep. We are using map<int, vector<int>> to represent send_parts. The key is the rank of the neighboring processors, and the value is the vector of particles that need to be sent from the current processor. Similarly, we are using the same data structure to represent receive_parts which are the particles that are received from the neighboring processors.

(4) send_part_all, receive_part_all: Send-part_all and receive_part_all are vector<particle_t> structures that contain all the particles that are moving out of the processor and all the particles that are moving into the processor, respectively.

(5) send_part_num, receive_part_num: Send_part_num and receive_part_num are vector<int> structures of size num_procs that contain the number of particles to be sent to each processor in MPI_COMM_WORLD from the current processor and the number of particles to be received by the current processor from all other processors in MPI_COMM_WORLD. This data structure is primarily used in the MPI_Alltoall function to send and receive outgoing and incoming particles.

(6) send_req, receive_req: send_req and receive_req are map<int, array<MPI_Request, 2>> structures where the key is the rank of the target processor (to send to or receive from) and the value is an array of 2 MPI_Request structures, the first of which is the request for sending/receiving the number of particles, and the second of which is for sending/receiving the particles themselves. These structures are used when the ghost particles are being updated by the neighboring processors.

3. MPI Communication

We use MPI to speed up the particle simulation algorithm in multi-processing scenarios. Each distributed memory process communicates with its neighboring processes to exchange information about particles and bins. The communication between processors is performed using the non-blocking MPI_Send and non-blocking and blocking MPI_Recv functions, as well as MPI_Alltoall. The code first computes the bins that need to be sent and received by each processor to update the ghost bins. It then initializes the MPI requests using the MPI_Isend and MPI_Irecv functions. The code then waits for the communication to complete using the MPI_Waitall function for each non-blocking send and receive. Each process maintains a map of neighboring processes and the bins and particles that need to be sent and received. The information about each bin and particle is sent and received using MPI tags, which are used to identify the type of data being communicated.

3.1 MPI_Isend

We use a non-blocking send, MPI_Isend on two occasions: firstly, to send the number of particles to a neighboring processor, secondly, to send the particles of each processor to its neighboring processors. This is used prior to the force computation to send the particles of the current processor to its neighboring processors to update the ghost particles in each processor. We use a non-blocking send to improve performance.

3.2 MPI_Recv

We use a blocking receive to receive the number of particles being received from a neighboring processor. We cannot use a non-blocking receive here because we need our receives are order-dependent. We must first receive the number of particles before we can receive the particles themselves. This is because MPI_Recv/Irecv requires you to specify the number of particles to receive.

3.3 MPI_Irecv

We use MPI_Irecv to receive the particles themselves, after receiving the number of particles. Similarly to sending the particles, since we already get the particles to receive in the receive_parts, we can get the particles to send with r*eceive_parts[rank]* where rank is the source rank of neighboring processors. Then we can use *MPI_Irecv* to receive particles according to the source rank. We opt for a non-blocking receive here to improve performance.

3.4 MPI_Wait

Because we are using some nonblocking receives and sends, we have to use MPI_Wait() to ensure all the particles/numbers are properly sent and received. We use MPI_Wait() to first ensure all the particles in each bin have been sent prior to clearing out the ghost bins. Then, we use MPI_Wait() to wait on the receives from each neighboring rank prior to rebinning the particles collected from the receive.

3.5 *MPI_Alltoall*

Finally, we use MPI_Alltoall to communicate the particles that move to different processors. Finally, we need to move particles in the current bin. There are three different circumstances: (1) if the particle is still in the same bin, then do nothing (2) if the particle moves to a different bin that is in the same processor, then we move it to other bins accordingly (3) if the particle is moving to the another bin in a different processor, then we use the MPI_Alltoall function to communicate particles between these processors.

4. Design choices

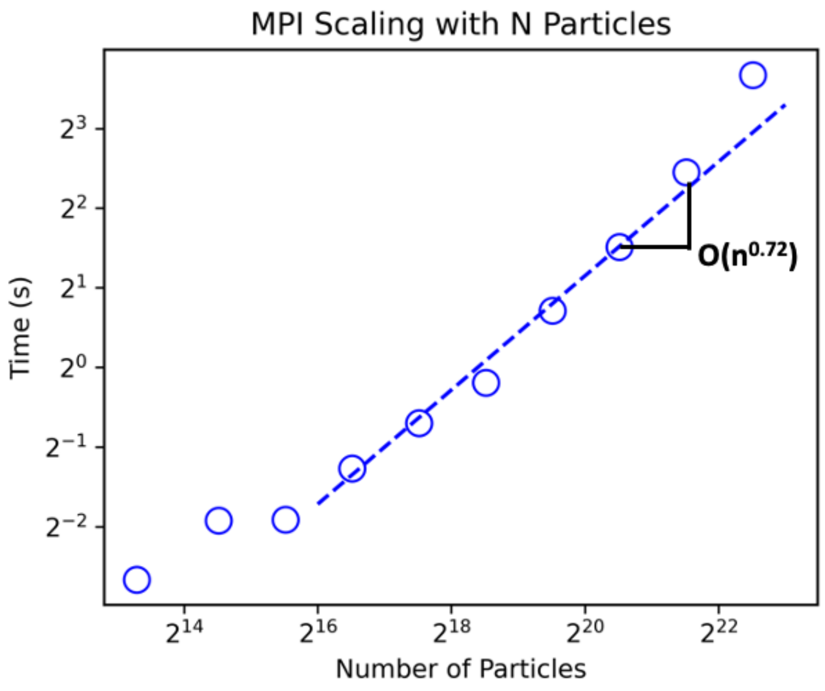
4.1 Domain Decomposition

There were two choices that we attempted for decomposition of the domain. The first was a 1D domain decomposition where each processor's subdomain consisted of rows of bins. Each processor would only have two neighboring processors to communicate with. The second was the 2D decomposition demonstrated in our final code. Upon testing of both domains, we found that the 2D decomposition provided higher parallelism and achieved faster performance for 128 MPI ranks and 6 million particles.

5. Log-Log Scale Plots

5.1 Log-Log Plot (running time vs number of particle)

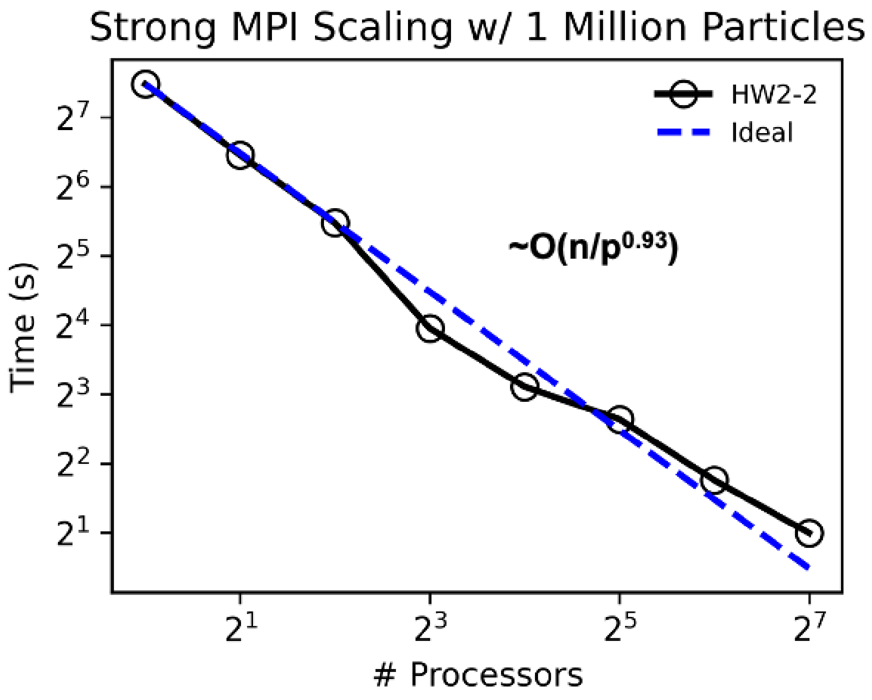
We increase the number of particles and compare the running time. Calculations were performed on 128 MPI ranks on 2 nodes. We can see the MPI code runs with scaling better than O(n), at around O($n^{0.72}$) time. At smaller particle counts, this scaling is even better. The MPI scaling performs the best out of the three implementations done thus far (serial, OpenMP, and MPI).



5.2 Strong Scaling

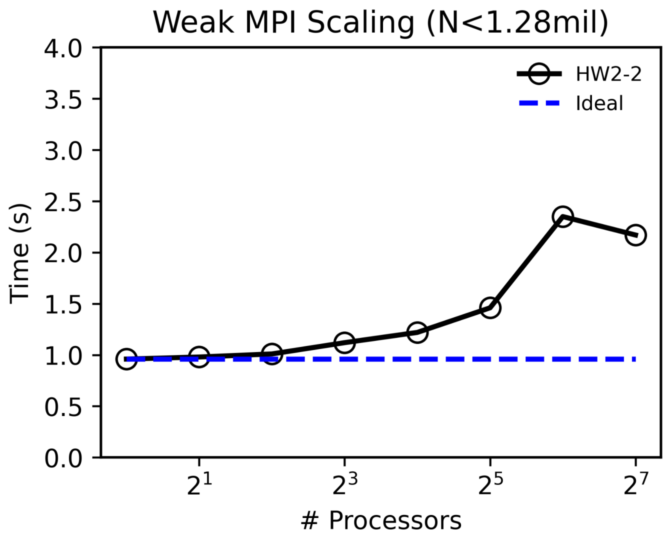
We run the code for 1 million particles with 1 to 128 MPI ranks and plot the strong scaling below. As we can see from the graph, the running time is close to $O(n/p^{0.93})$, which is very close to O(n/p) as expected. Our program thus has a strong scaling efficiency of 93%, and closely approaches the idealized p-times speedup. This efficiency is computed by taking the slope of the linear fit through the log-log plot. The MPI implementation has much better strong scaling than our OpenMP implementation, which reached a strong scaling efficiency of 75%.

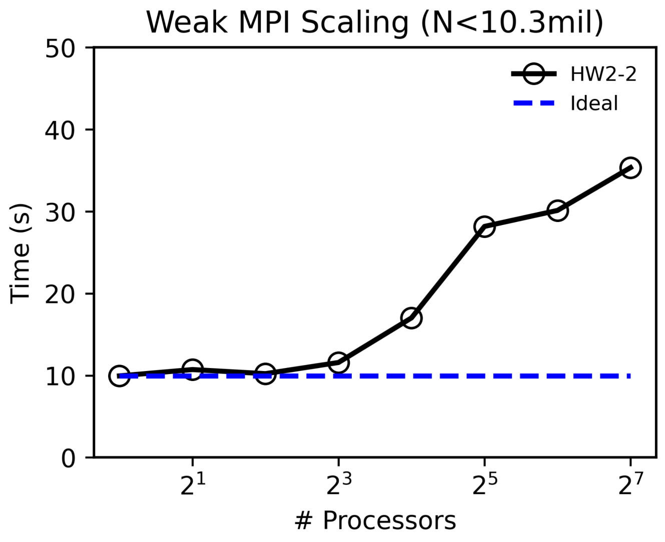
It is definitely possible to improve the strong scaling of our code, especially at a lower number of processors. Currently, a large chunk of our code is still serial. For example, although each bin is accessed in parallel, each neighboring bin and each particle in the bin and neighboring bins are all accessed in serial. Amdahl's law tells us that strong scaling efficiency is dictated by the amount of serial code in the program. Because our program has components that could be parallelized, we should be able to improve our strong scaling.

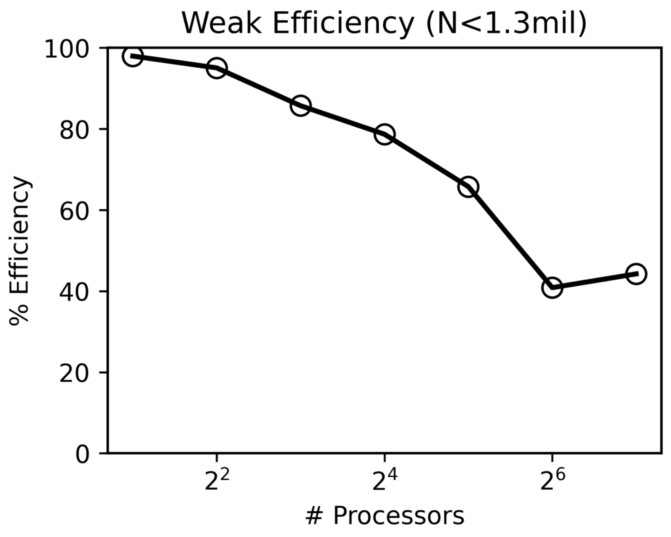


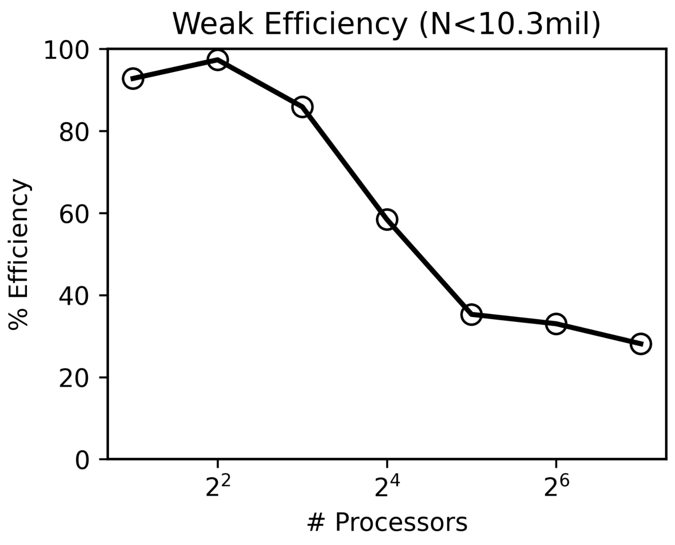
5.2 Weak Scaling

We increase the number of particles proportionally to the number of processors so the work/processor stays the same, and here is the plot we get. In the plot on the left, we start with 10,000 particles. In the plot on the right, we start with 100,000 particles. The trajectory of the MPI scaling is similar for both numbers of particles. We observe good scaling for less than 8 processors. After 8 processors, the scaling becomes worse, but improves at a high number of processors (>64 processors).





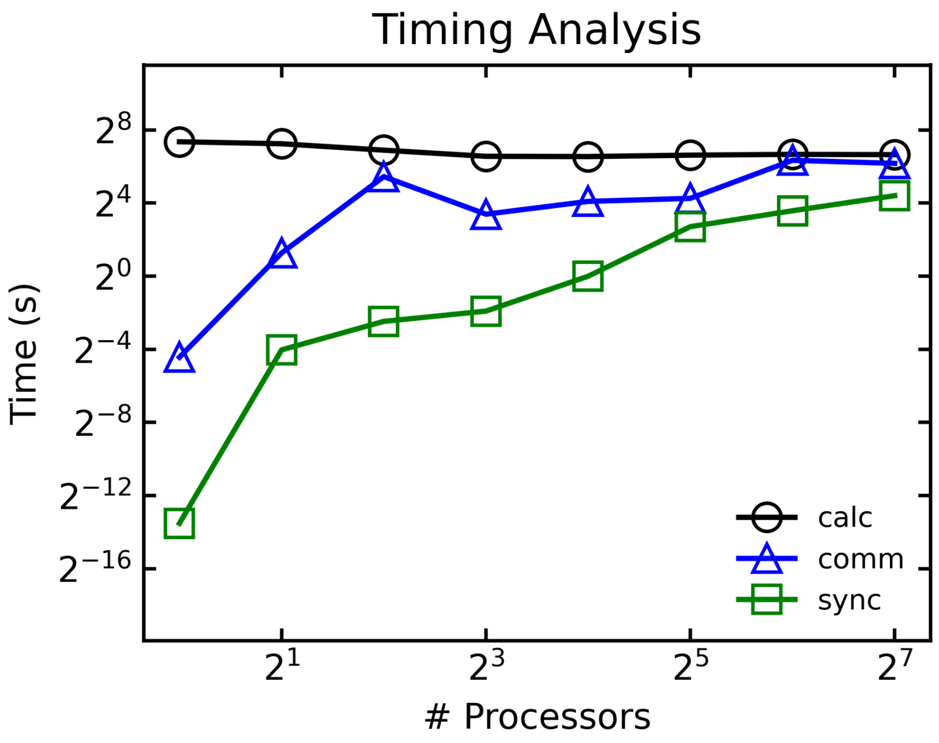




Additionally, we plot this efficiency as a function of the number of threads on a semilog scale to show how the weak efficiency drops exponentially with the number of threads and particles. We can also use these plots to observe how weak efficiency improves at a higher number of processes.

6. Time Analysis

We provide a rough idea of the timing spent in computation, communication, and synchronization by comparing the time spent updating the data structures (computation) with the time spent sending and receiving between processors (communication) and the time spent in MPI_Wait() and MPI_Barrier() (synchronization). To calculate the time spent in each of these chunks, we use MPI_Wtime() to record the start and end duration of each communication, synchronization, and calculation chunk of code and sum the time spent over all processors into global variables *calc_time, sync_time, and comm_time*. We then perform a reduction to obtain the total calculation, synchronization, and communication time spent across all processors. We record the time spent for a varying number of threads for one million particles and plot them below.



Our results show that the time spent in each category is dictated by the number of processors. The calculation time is practically the same despite an increase in the number of processors, which makes sense as the calculation time is computed serially and summed across all processors. The communication time varies directly with the number of threads; plotted on a $\log_2$-$\log_2$ scale we observe that it increases linearly with the number of processes until at 8 processors. From then on the communication time is roughly constant. This makes sense, because the majority of our communication is only with 8 neighboring processors, so having more than eight processors does not increase the communication time. Finally, we see that the synchronization time is practically insignificant until at a high number of processors, where it approaches the communication time and calculation time. This makes sense, because synchronization becomes increasingly difficult with more processors.