# CS 267 HW 2-2

### Parallelizing a Particle Simulation with MPI

**Abstract:** In this assignment, we implemented the particle simulation system in MPI. Based on HW 2-1, we used the binning method to reach T = O(n) time complexity on a single process. Using MPI, the simulation will run parallelly on p processors and therefore runs close to T/p.

## Member & contribution

- **Bo Li**: Implement particle movement within and across processors
- **Jingran Zhou**: Implement force computation within and across processors, as well as "gather and save"

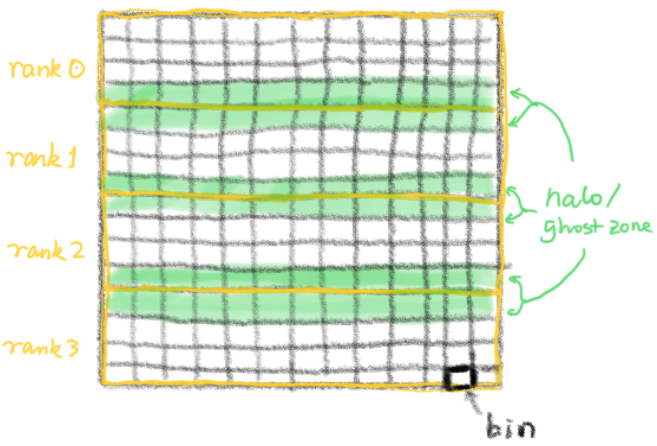
## Data structures: `unordered_set` and `vector`

In HW2-1, we represented each bin as an `unordered_set` of particle pointers. Here, we chose to stick with it because the calculation is largely similar: during initialization, particles that belong to the current processor are inserted into the corresponding bin based on its initial position. When applying forces, we iterate over particles in the bins to simulate interactions. Finally, when we move the particles, if a particle should "jump" to a new bin, we delete the particle from the old bin and either insert it into a new local bin or send it to another processor.

Secondly, as we will demonstrate below, we need buffers to store particle structures for communication purposes. The ideal candidate here is `vector` for its dynamic and continuous memory allocation. Also, its well-written API makes it a better choice than the troublesome standard array.

## Halo, buffer, and padding

The "world map" for particle collision is shown on the right, where each black square denotes a bin. We run the program on multiple processors. Instead of dealing with *all* the particles, each processor now only handles particles in a certain region. In the figure, each yellow slab is handled by a different processor. For particles to affect each other and move across processor regions, we need to communicate with other processors when we apply forces, update the particles' positions and collect

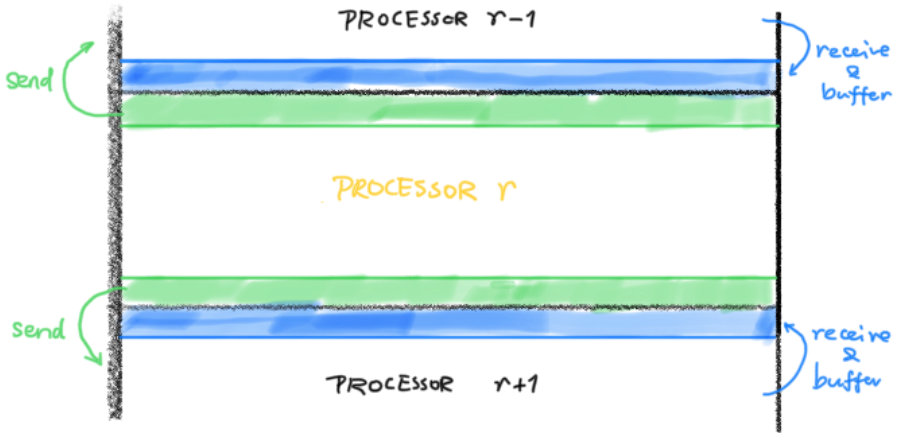


particles from different processors. Such communication takes place on the boundaries between processors, (e.g., the green regions) and is achieved by using MPI.

Specifically, when communicating with neighbor processors, we need to both send our **halo** particles and *receive* boundary particles from our neighbors. Naturally, we need to allocate **buffers** for storing the received particles. Lastly, for the ease of iteration, we decided to pad a layer of outer bins to each processor, "pretending" that we have access to our neighbor processors' boundary particles. This makes the computation of `apply_force()` easier. Our design choices are further illustrated and justified below.

# Implementation details

## Computing forces




The figure above illustrates the setting for calculating interaction forces. Each processor is responsible for calculating the forces applied to the particles in its region. Note that particles in the inner bins of a processor will not interact with particles in *other* processors, so we can safely calculate them as in HW 2-1. However, the particles in the **halo** (i.e., the bins shaded in green) will possibly interact with particles in other neighboring processors (i.e., the bins shaded in blue). To capture this cross-processor interaction, we need to communicate with our neighbors.

Each processor has at most two neighbors: top and bottom. We use `MPI_Sendrecv()` to communicate with both neighbors to avoid deadlock. For a neighbor processor, such as the top one, we send all of our halo particles (e.g., all the top-side green particles), and we receive and buffer the neighbor's corresponding boundary points. After the communication is done, we put the received particles in their corresponding bins, which should be the outer padded ones (i.e., the blue region). Now that we have all the available information to calculate interaction forces for our particles, we iterate over all the bins that belong to the current processor and apply the forces accordingly.

### Moving particles

In moving particles, each processor has two buffers. `Imcoming_Pts_Buffer` receives the particles from other processors, and `Outgoing_Pts_Buffer` temporarily stores the particles to be sent to other processors. In each iteration, we will compute the new position of each particle. Based on the new position, we compute its corresponding rank and bin index. If the new rank is different from the current processor's rank, then it will be added to `Outgoing_Pts_Buffer`. Otherwise, it will be updated within this processor and moved from the current bin to the new bin. *Note that we assume that particles can only jump to neighboring processors (i.e., top and bottom) but not any further.* We used `MPI_Sendrecv()` to communicate with other processors because it can prevent deadlock. For new incoming particles, we added them to the corresponding bins based on their positions.

### Gathering for save

In gathering particles, the master processor with rank 0 will collect particles from all other processors, and each processor will send a buffer containing all the particles in its inner bins to the master processor. Because we know the particle id, we can directly use the particle ID as the index and place the particle to its supposed position.

## Synchronization with `MPI_Barrier()`

To synchronize all the processors during the communication. We utilize the MPI function `MPI_Barrier()`. The function blocks the caller until all the processors in the communicator have called it. It is thread-safe and makes sure that no processor can access the shared resource until all the reading or writing operations have been finished.

## Other design decisions

### `MPI_Gather()` vs. `MPI_Recv()`

When implementing `gather_for_save()`, one of the first MPI functions that come to mind is `MPI_Gather()`, which seems to fit exactly what we are trying to achieve. However, once we started to write the code, we realized that `MPI_Gather()` is only suitable when we have fixed-size messages to receive from all processors. Unfortunately, in this case, the total number of particles in each processor varies, so we cannot naively use `MPI_Gather()`. After further research, we have discovered a more powerful variant: `MPI_Gatherv()`, which allows messages of different sizes from different processors. Nonetheless, we have decided not to use this seemingly ideal function because of two reasons: 1) `MPI_Gatherv()` requires us to specify displacement information of *every* received message, which is too troublesome; and 2) `gather_for_save()` is not a performance-critical function since it is *only* called when the `-o` flag is specified.

That is, in the actual benchmark, the time for `gather_for_save()` will not be included in our total simulation time. Considering all the factors above, we decided to stick with the simple `MPI_Send()` and `MPI_Recv()`.

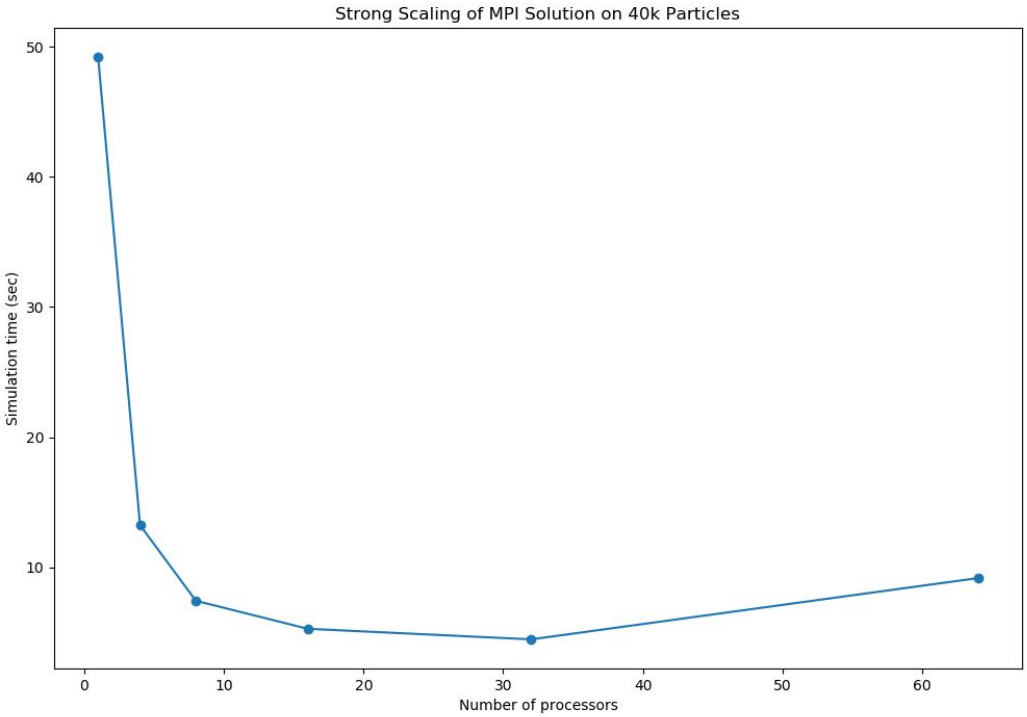
# Scaling performances

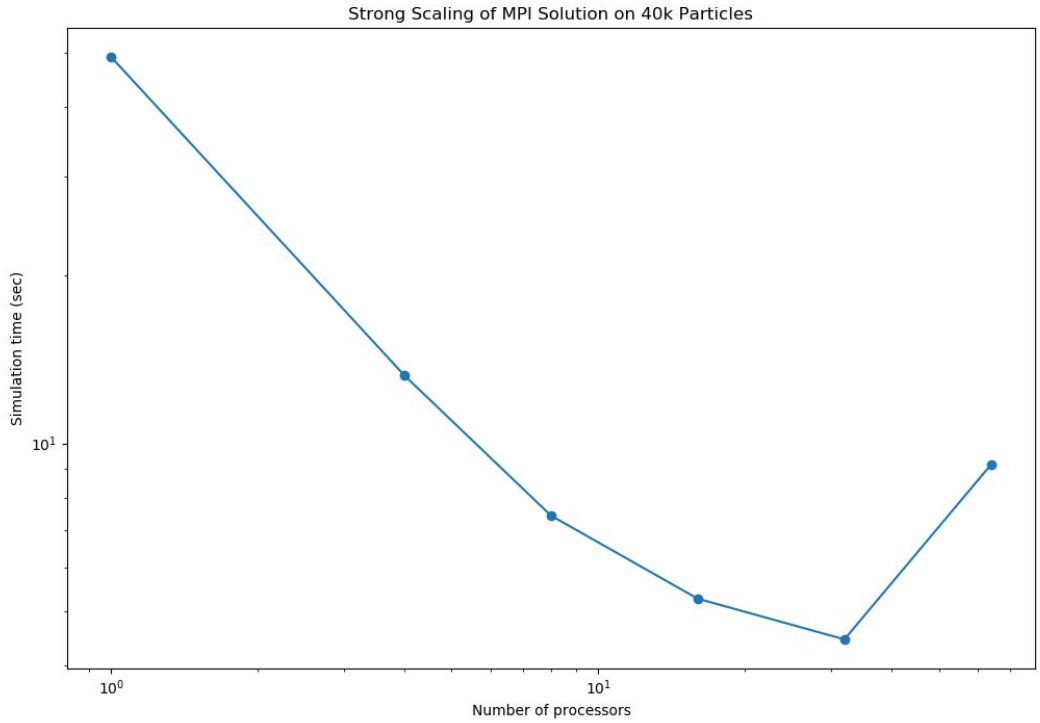
## Strong scaling

The below table demonstrates the strong scaling of our program on simulating 40,000 particles.

| **Number of Processors** | **Runtime (sec)** |
|---|---|
| 1 | 49.209 |
| 4 | 13.2324 |
| 8 | 7.4219 |
| 16 | 5.27019 |
| 32 | 4.45688 |
| 64 | 9.16603 |

Below are two plots of the above table. The first is on linear axes, while the second is a log-log plot.

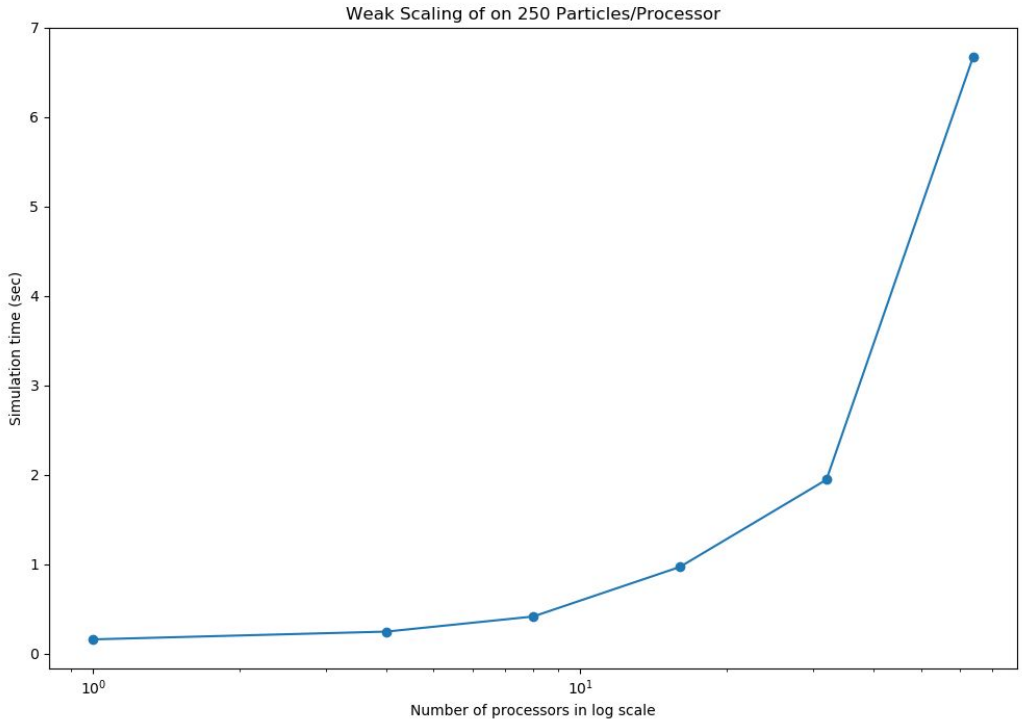

Strong Scaling of MPI Solution on 40k Particles
Simulation time (sec)
Number of processors
50
40
30
20
10
0
10
20
30
40
50
60




The effect of strong scaling is evident when the number of processors is relatively small. However, when the number of processors exceeds 64, we can see that the benefits of strong scaling vanish and the simulation time is even increasing. We hypothesize that this is because the cost from communication among processors increases more than the time saved by the processor parallelization.

## Weak scaling

| Number of Processors | Number of particles | Time (sec) |
|---|---|---|
| 1 | 250 | 0.157619 |
| 4 | 1000 | 0.244439 |
| 8 | 2000 | 0.412841 |
| 16 | 4000 | 0.966862 |
| 32 | 8000 | 1.94274 |
| 64 | 16000 | 6.66891 |




As shown in the above figure, when the number of processors increases with the number of particles, each processor has 250 particles on average. The weak scaling performs reasonably on the smaller number of processors. However, as the number of processors becomes larger, the time will increase. This is due to the reason that each processor doesn't get a balanced amount of jobs. Some processors will get more jobs and some processors will get fewer. The waiting time at the synchronization due to the barrier will increase the total time cost.