CS 267 HW 2 Report: Parallelizing Particle Simulation

Kathryn Wang, Milad Shafaie (Group 65)

March 10, 2025

1 Member Contributions

- Milad
  - Contributed to architecture of our approach, and implemented the key skeleton and data structures needed for the approach.
  - Attempted some of the further (unsuccessful) optimizations listed later in this report.
  - Wrote section 2 and 3.4 of this report.
- Kathryn
  - Contributed to design of architecture of approach, and implemented message passing scheme.
  - Created all performance plots, including log-log plot of performance, weak and strong scaling plots, and plots capturing distribution of runtime across different sections of code.
  - Wrote section 3 of this report.

2 Method

2.1 Communication in the Distributed Memory Implementation

Our implementation uses a distributed memory model based on MPI. We decompose the simulation domain into square grids and assign each processor (MPI rank) to a contiguous subset of the rows of the grid. Communication between ranks is handled primarily through `MPI_Sendrecv calls.` The use of `MPI_Sendrecv` first involves the sender sharing an integer representing the number of particles it is going to send so that the receiver can allocate space to receive that amount of particles. There are two major communication phases:

2.1.1 Ghost Cell Exchange:

In order to accurately compute the forces on the particles within the rows of the grid it controls, each processor must be aware of the positions of particles in the row directly above and directly below, if they exist, of the rows the processor controls, since these particles may be close enough to those the processor controls to exert a force onto them. We refer to these regions as "ghost" regions.

Each rank must exchange "ghost" particles with its immediate neighbors at the beginning of each step simulation. For ranks that have a neighbor above, the ghost region along the top row is sent to the rank above, while that rank sends its bottom row back. A similar exchange occurs with the lower neighbor. This ensures that each rank has the necessary boundary data to compute forces accurately.

2.1.2 Migrated Particle Exchange:

When a particle moves from a row under the control of one processor into a row under the control of another processor, the processor which controlled the particle prior to the move must communicate the particle to the new processor. Again, we use `MPI_Sendrecv` to simultaneously send the number of migrating particles and then exchange the actual particle data between adjacent ranks.

2.2 Design Choices and Their Impact on Performance:

2.2.1 Binning Structure:

We first divide the domain into square grids based upon the distance within which particles can exert forces on one another. By doing this, we reduce force computation calculation for particles in a bin to just involve the particles in the bin itself and its neighboring bins. Since the size of individual bins remains constant as the number of particles increases, the amount of work per bin is constant if you assume that there is a constant number of particles per bin. Thus, the computational complexity of the problem scales linearly with the number of bins, which is linear in the number of particles. All in all, this means that dividing the domain into grids enables our algorithm to achieve $O(n)$ performance for the particle simulation.

2.2.2 Domain Decomposition

We partition the domain by rows so that each rank is responsible for a contiguous subset of rows of the grid. In this way, processors can compute forces on their particles and move them in parallel. The only communication that needs to occur is between neighboring processors, as described in section 1.1. This minimizes the number of neighbors (typically only two) that each rank must communicate with. As a result, communication is limited to adjacent ranks, which helps reduce overhead at moderate scale. A downside of this approach, however, is that if there are fewer rows than processors, some processors may not be utilized. This could be overcome by assigning processors to more flexible subsets of the grids in the domain, but we do not explore this in this homework.

2.2.3 Ghost Regions

For each processor, we store an array of particles in the ghost region above the rows controlled by the processor, and a separate array for the particles in the ghost region below the processor's rows. The particles in these arrays are then further subdivided by the column they are in. By sending only exchanging ghost particles (those in the boundary bins), we limit the communication volume compared to sending all particles of one processor to another. Further binning the ghost particles into columns minimizes force computation on the particles within a processor, since only ghost particles in columns including or adjacent to that of a particle controlled by the processor need to be considered.

2.2.4 Use of MPI_Sendrecv

Our choice to use blocking send/receive operations via `MPI_Sendrecv` simplifies the code and guarantees that ghost data is available before force computation begins. However, this synchronization can contribute to overhead, especially as the number of ranks increases.

2.3 Log-Log plot of Parallel Performance and Data Structures Used

- `int start_row`: The index of the first row that the processor controls (inclusive).
- `int end_row`: The index of the final row the processor controls (exclusive)
- `vector<particle_t> my_parts`: a vector of the particles that a particular processor controls.
- `vector<vector<int>> bins`: a 2D vector where the ith nested vector contains the indices of particles in the ith-indexed bin controlled by the processor. Note that the 0 index of this vector refers to the bottom leftmost bin that the processor controls.
- `vector<particle_t> ghost_above_recv`: a vector of the ghost particles to be received by the processor from the processor above it (with rank 1 greater)
- `vector<particle_t> ghost_above_send`: a vector of the ghost particles to be sent to the processor above it (with rank 1 greater)
- `vector<particle_t> ghost_below_send`: a vector of the ghost particles to be sent to the processor below (with rank 1 smaller)

- `vector<particle_t> to_move_above`: a vector of the particles that have moved to the processor above and need to be sent to it
- `vector<particle_t> to_move_below`: a vector of the particles that have moved to the processor below and need to be sent to it

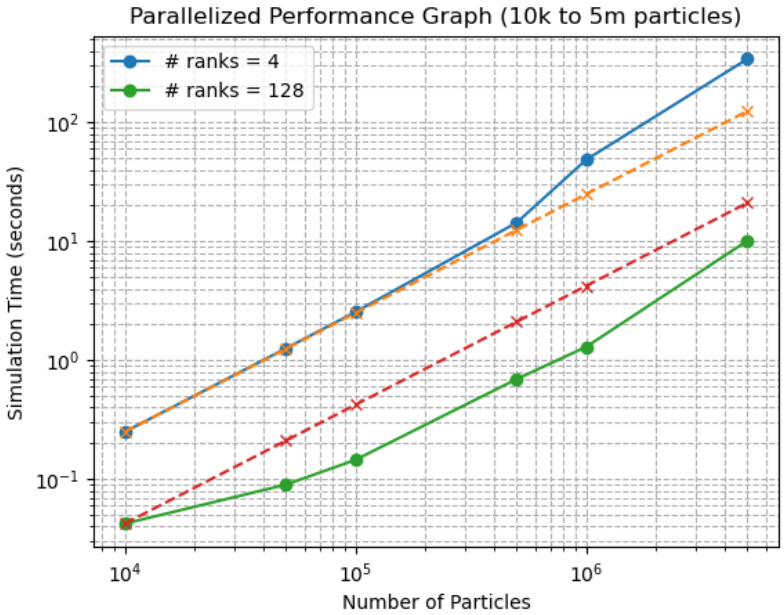


Figure 1: Performance plot, Dashed line is ideal linear performance

3 Results and Discussion

3.1 Weak Scaling

When we scale the problem size proportional to the number of ranks, our runtime remains near constant for low to moderate p. Each rank handles a similar workload, and communication overhead is manageable. After a certain point (around 32-64 ranks in our tests), the runtime gradually increases. This shows the rising cost of communication. We approximate the ideal weak-scaling at first, but communication overhead becomes a bottleneck at higher ranks.

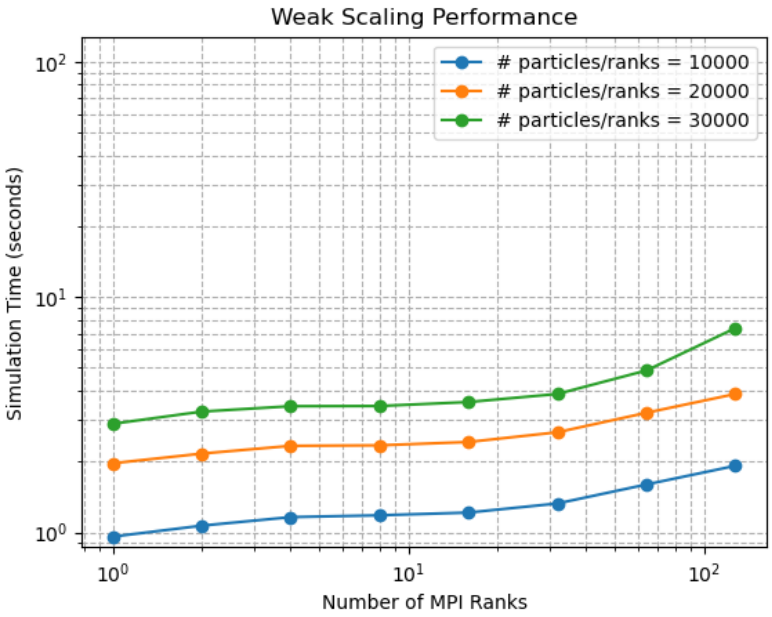
3.2 Strong Scaling

Our strong scaling experiments track the ideal p-times speed up line at moderate ranks. One thing that was interesting was for 1M particles, the measured runtime goes below the ideal line which suggests superlinear speedup. This can be caused by:

- **Cache/Memory Effects:** Distributing a large data set among more ranks reduces the working set per rank. This can improve cache locality and lead to faster performance than expected from simple linear speedup.
- **Baseline Inefficiencies:** The single-rank baseline may be suboptimal, making multi-rank runs appear superlinear by comparison.

However, superlinear speedup is not sustainable. As the number of ranks increases, communication and synchronization overhead will dominate which prevents improvements better than linear speedup. So even though it appears we "exceed" p-times speedup, it doesn't violate theoretical limits. We are just benefiting from memory/cache advantages and probably an inflated baseline.

Figure 2: Weak Scaling Plot, N=2, Ranks 1 to 128



Parallel scaling principles tell us we can't surpass linear speedup indefinitely. The superlinear regions typically occur for moderate p due to the factors mentioned above.

3.3 Where the Time Goes

- *Force computation* dominates, even more so when the number of ranks is small when each rank handles a large part of the particle interactions.
- *Communication* (ghost exchange and migration) become a lot more significant as p increases. It becomes the principal factor limiting speedup in some cases and with more ranks.
- *Binning* overhead is less costly but still a contributing factor for the runtime. However, it also benefits from smaller local workloads at higher p.

Therefore, at large scale, the bottleneck shifts from computation to communication. This matches our strong and weak-scaling results too!

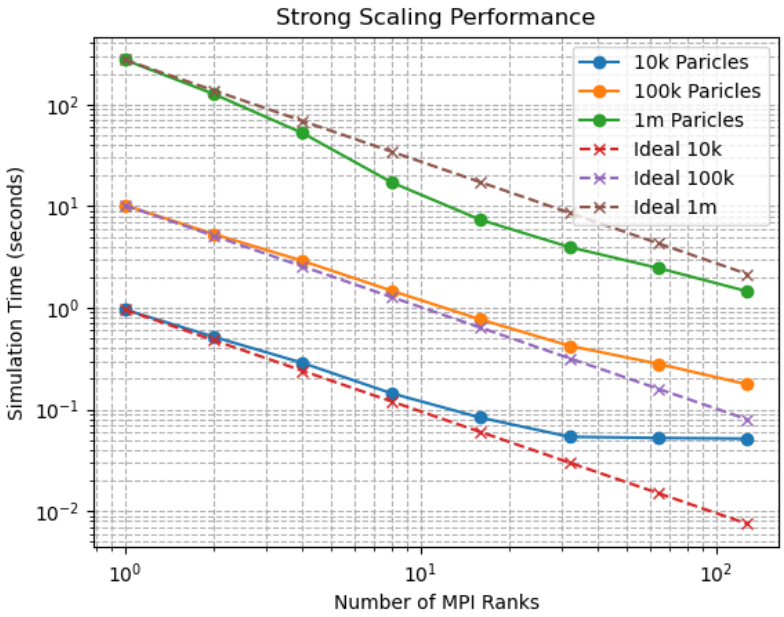
3.4 Future Optimizations

Below are a few optimizations we attempted but were unable to successfully implement. These ideas could address some of the weak points of the approach we have detailed in this report.

3.4.1 Pre-Computation of Neighbors and Two-Way Force Computation

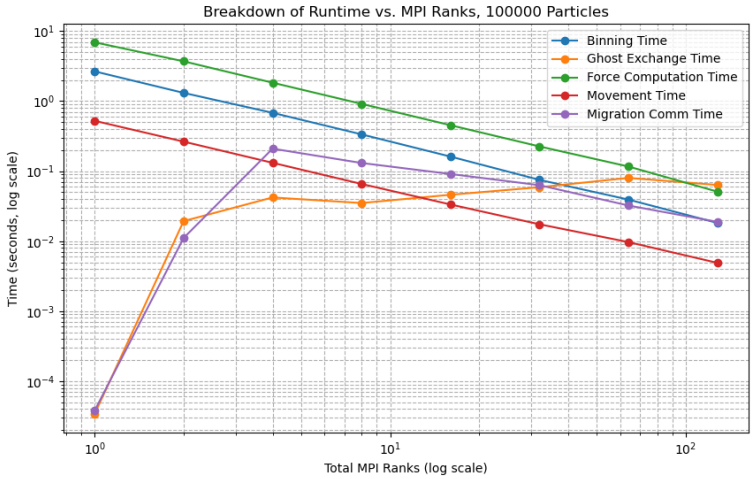
As shows in the previous subsection, the computation of forces dominates the runtime of our approach. Although we have taken some steps to make force computation more efficient, such as by diving the domain into grids and organizing ghost particles by column, there is still redundant force computation because `apply_force` has to be called twice for each pair of particles that exert forces on one another. Because the force exerted on one particle is the opposite of the force exerted by the other particle on the first, `apply_force` could be modified to stored forces computed on both particles passed to it at the same time. By doing this, the computational cost for force computation could be theoretically havled.

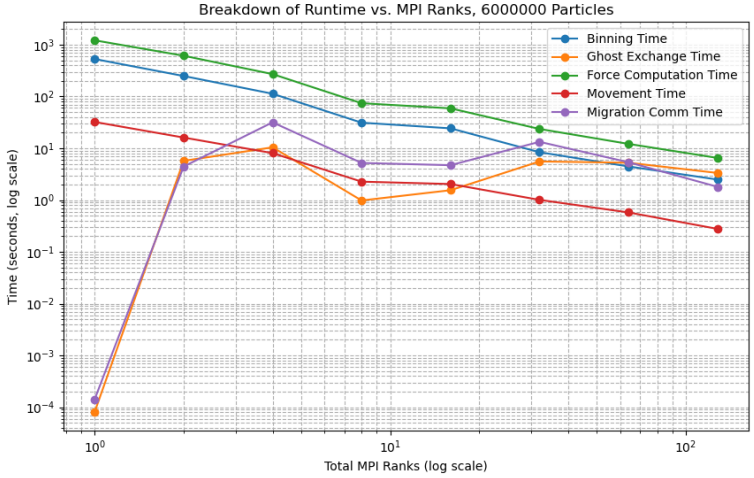
Figure 3: Strong Scaling Plot, N=2, Ranks 1 to 128, Dashed lines are ideal p-times speedup



3.4.2 Communicating Ghost Particles and Migrated Particles Together

Another weak point of our approach is that ghost exchange time and migration communication time each grow large as the problem size scales. The contribution of these parts of the code to the overall runtime could be minimized by communicating ghost particles and migrated particles at the same time, for example by appending them to a single array, with some buffer element separating the two segments. In this way, the `MPI\_Sendrecv` scheme seen in our code would only need to be used once.


Breakdown of Runtime vs. MPI Ranks, 100000 Particles
Binning Time
Ghost Exchange Time
Force Computation Time
Movement Time
Migration Comm Time
Time (seconds, log scale)
Total MPI Ranks (log scale)


Breakdown of Runtime vs. MPI Ranks, 6000000 Particles
Binning Time
Ghost Exchange Time
Force Computation Time
Movement Time
Migration Comm Time
Time (seconds, log scale)
Total MPI Ranks (log scale)