# CS267 HW2-2 Write Up

**Contributions:**

- Write-Up: All
- Brandon: MPI troubleshooting; computation and communication breakdown.
- Kevin: MPI implementation and debugging, runtime optimization, and benchmarking

**Introduction:**

Homework 2.2 focuses on optimizing a two-dimensional particle simulation that models repulsive particle forces using a distributed memory parallelism framework with MPI. In Homework 2.1, we were given a naïve serial solution that computed forces between every pair of particles, resulting in an $O(n^2)$ complexity. By leveraging spatial binning, we optimized this approach to $O(n)$ complexity. Additionally, we utilized OpenMP, a shared-memory parallelism framework, to further improve performance by distributing workloads across CPU cores on a single machine system.

This assignment extends that work by transitioning from a shared-memory parallelism model to distributed memory parallelism using MPI. Unlike OpenMP, MPI allows execution across multiple nodes, where processes communicate explicitly through message passing. The objective was to partition the simulation space across multiple processes, ensuring that each process manages a subset of the particles while maintaining correct force calculations across partition boundaries.

In this report, we discuss our MPI communication implementation and design choices, including what strategies worked well and what challenges we encountered. Additionally, we analyze how effectively our implementation approaches T/p scaling, where T is execution time and p is the number of processes.

**Data Structures and Optimization Strategies**

As a reminder, in order to optimize force calculations we implemented spatial binning that divided the simulation space into a two-dimensional grid of bins. Each bin stored a vector of pointers to particles contained within that bin. This significantly reduced the number of force calculations, as particles only interacted with others within their own bin or neighboring bins, rather than iterating over every particle in the system.

In our MPI implementation, we partition the simulation space with a 1D row layout, assigning a row of height size/num_procs to each process. We extend the top and bottom of each process's domain by a halo region of height cutoff to account for ghost particles. Each process only stores and operates over particles that fall within its extended domain. We introduce a helper function, build_bins, that fills the spatial binning data structures from the serial implementation with a process's local particles. Each process manages its own local copy of the bins that overlap with the process's extended domain. This was a key optimization we implemented that removed a significant portion of loop overhead caused by iterating over empty bins outside of a process's domain.

During the init_simulation() phase, each MPI rank computed its domain and halo region beyond its domain boundary. Since each rank only processes a partition of the simulation space in the y-dimension, it required a halo region that extended beyond its domain boundary to account for interactions with neighboring ranks. The size of this halo region was determined by the interaction cutoff distance, ensuring that particles near the domain boundaries received repulsive forces from neighboring ranks. Without these halo regions, particles near domain edges would be missing forces from their actual neighbors in another rank's partition, leading to incorrect forces.

To achieve this partitioning, the simulate_one_step() function includes a partitioning phase after force computation that first determines whether a particle belongs to the rank's domain. If so, the particle is moved to its new position. This filtering prevents ghost particles from being considered for subsequent operations within this simulation step. For correctness, a process should not move or communicate its ghost particles, as those will be handled by neighboring processes. After movement, the function decides whether the particle remains in the rank's domain and if it should be sent to a neighboring rank.

During this partitioning phase, local particles are evaluated for potential exchange with neighboring ranks. If a particle moves beyond the halo region boundary of a neighboring process, it is placed in a send buffer for transfer to the corresponding neighboring rank. This accounts for both particles that move into a neighboring process's domain as well as particles that will become ghost particles for a neighboring process in the next step, ensuring that each rank's particles are updated in sync with its neighbors. Particles that are still within the process's local domain after movement will remain in the process's local memory for the next simulation step.

Once particles are categorized as remaining, moving up, or moving down, simulate_one_step() proceeds to the communication phase, where ranks exchange updated positions with their top and bottom neighbors. This structured approach ensures that each rank correctly updates its own domain, while also maintaining accurate ghost regions for cross-rank force calculations. With this approach, we only need one communication phase per simulation step to update a process's local particle copies with both its owned and ghost particles.

**MPI Communication and Distributed Memory Implementation:**

Since each MPI rank processes a subset of the simulation domain, ranks must exchange data with their neighboring ranks to ensure correct force calculations across partition boundaries. To accomplish this, we implemented a particle exchange mechanism using MPI point-to-point communication (MPI_Send and MPI_Recv).

First, all processes except the process handling the top-most strip of the simulation space will send data to its above neighboring process. Next, all processes except the bottom-most process will receive data from below. Because the top-most process only receives, this setup avoids deadlock. This logic is then repeated in the downward direction to complete the communication phase.

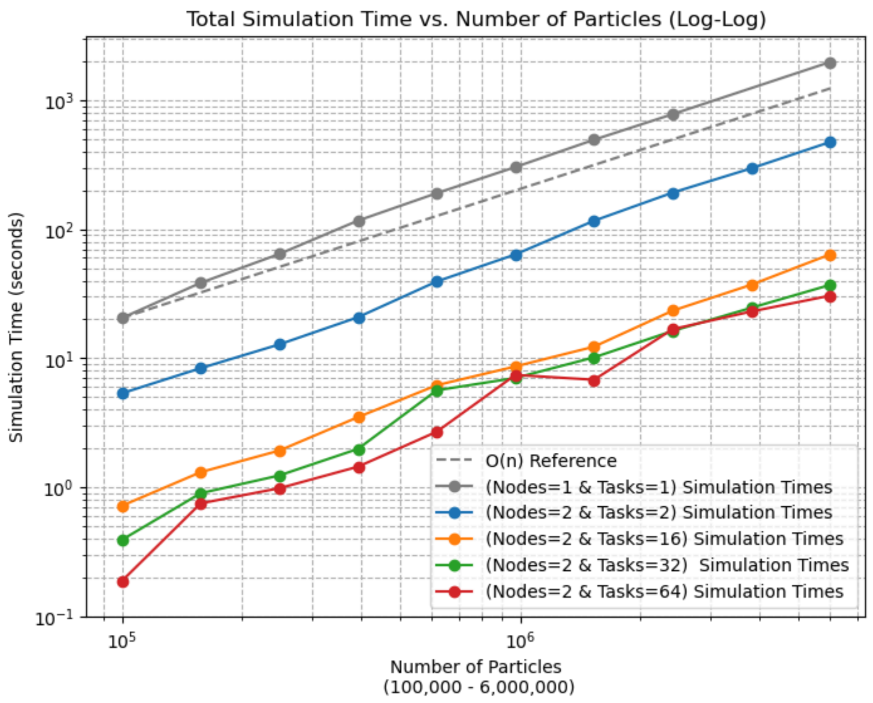
Our communication strategy followed a two-stage approach for both upward and downward communication:

1. Each rank sends the number of particles it will transfer to its neighboring rank.
2. Once the receiving rank knows the expected data size, the actual particle data is sent.

This two-stage approach provided a flexible design, allowing each rank to exchange a variable number of particles dynamically. This flexibility was essential, as the number of exchanged particles could change at each time step, depending on particle movement.

By exchanging only particles near domain boundaries instead of entire particle datasets, we minimized communication overhead while maintaining accurate force calculations. This approach optimized inter-process communication efficiency, ensuring that the simulation remained scalable as the number of ranks increased.

**Performance Analysis:**

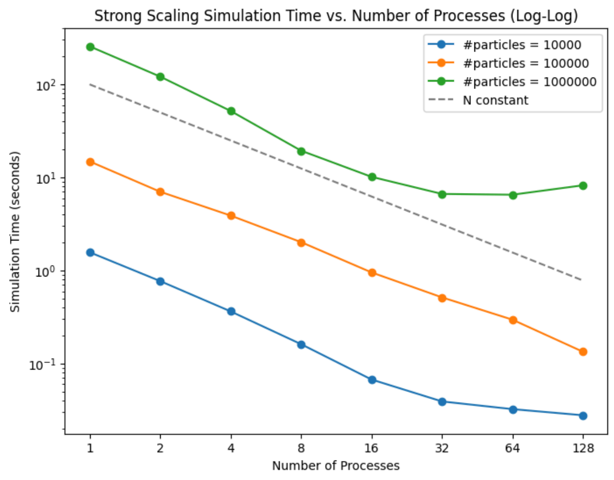


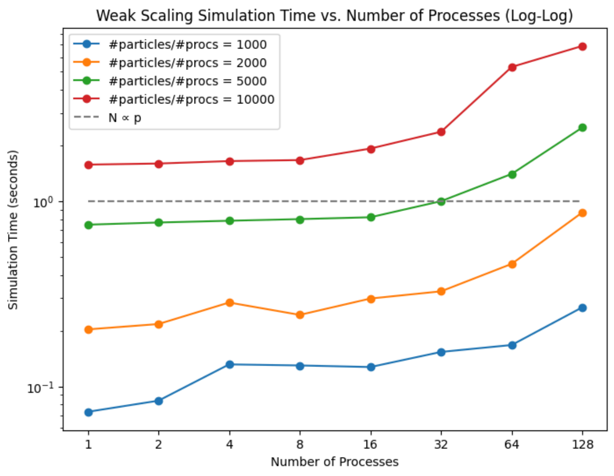
As a reminder of findings of homework 2.1, the implementation of spatial binning had a significant impact on the performance and scalability of our simulation. By limiting force calculations to neighboring bins, we reduced the time complexity from O($n^2$) to O(n). Looking at the MPI implementation of this same method run serially with 1 task on 1 node (solid grey line), there is some added overhead, presumably due to MPI overhead, particle domain partitioning, and communication operations that are not used. However, even with this unused overhead the scaling continues to be near linear.

More notably, the log-log plot in the first figure below demonstrates the performance improvements when running the MPI implementation with 4, 32, 64, and 128 ranks distributed across two nodes on Perlmutter. The most substantial speedup occurs when transitioning from serial (1 rank) to 4 ranks and then from 4 ranks to 32 ranks. However, beyond 32 ranks, the benefits of adding more ranks diminish. The performance gain from 64 and 128 ranks is less pronounced, suggesting a bottleneck that prevents perfect scaling.

This diminishing speedup is primarily due to the trade-offs between computation, partitioning, and communication overhead. While increasing the number of ranks allows for better load balancing in the force computation, it also introduces additional costs. Firstly, the added partitioning overhead where each rank must track and maintain particle movement across domains, increases the memory and computational costs. Secondly, communication overheard where the ranks must send explicit updates to each other updating their ghost regions. Both of these are added to all simulations with multiple ranks, but it appears that these added costs begin to outweigh the benefits of force computation load balancing with more than 32 ranks.

**Strong and Weak Scaling:**





The above two plots show the strong and weak scaling behaviors of our distributed memory implementation. In both plots, "number of processes" refers to the total number of processes (not number of processes per node). All process counts are equally split between two nodes except for 1.

To benchmark strong scaling, we hold the number of particles constant and plot the speedup as we increase the number of processes. On a log-log scale, we would ideally like to see the curves match a line with a slope of -1. Taking a look at the strong scaling plot, we see that the ideal trend is largely followed. For the range of processes tested, it appears that a problem size of 100,000 particles is a sweet spot where our code achieves ideal speedup. For the smaller and larger problem sizes, parallelization overhead begins to overtake parallel speedup after 32 total processes are exceeded.

To benchmark weak scaling, we increase processor count proportionally with the problem size. On a log-log scale, the ideal reference is a line with slope 0. In the weak scaling plot, we see that aside from some erratic behavior when the problem size and processor count are small, the ideal trend is followed until the number of processes exceeds 32, with some slight deviation after 8 total processes. After this point, parallelization overhead begins to overtake parallel speedup.

The deviation from the ideal trend is more severe in the weak scaling plot than the strong scaling plot. This indicates that communication bottlenecks account for most of the parallelization overhead as the number of processes grows. This analysis indicates that we can improve communication overhead with a new communication scheme. As explained earlier, the current scheme involves all processes sending data upwards before receiving data from below. This means that a process must wait until the process above it is done sending its data before it can receive, resulting in a cascade of waiting that grows linearly with the number of processes. To reduce the communication time, a possible improvement is to group neighboring processes into pairs, where one process sends first while the other process receives before sending. This new scheme may remove the linear growth of communication time as the number of processes increases.

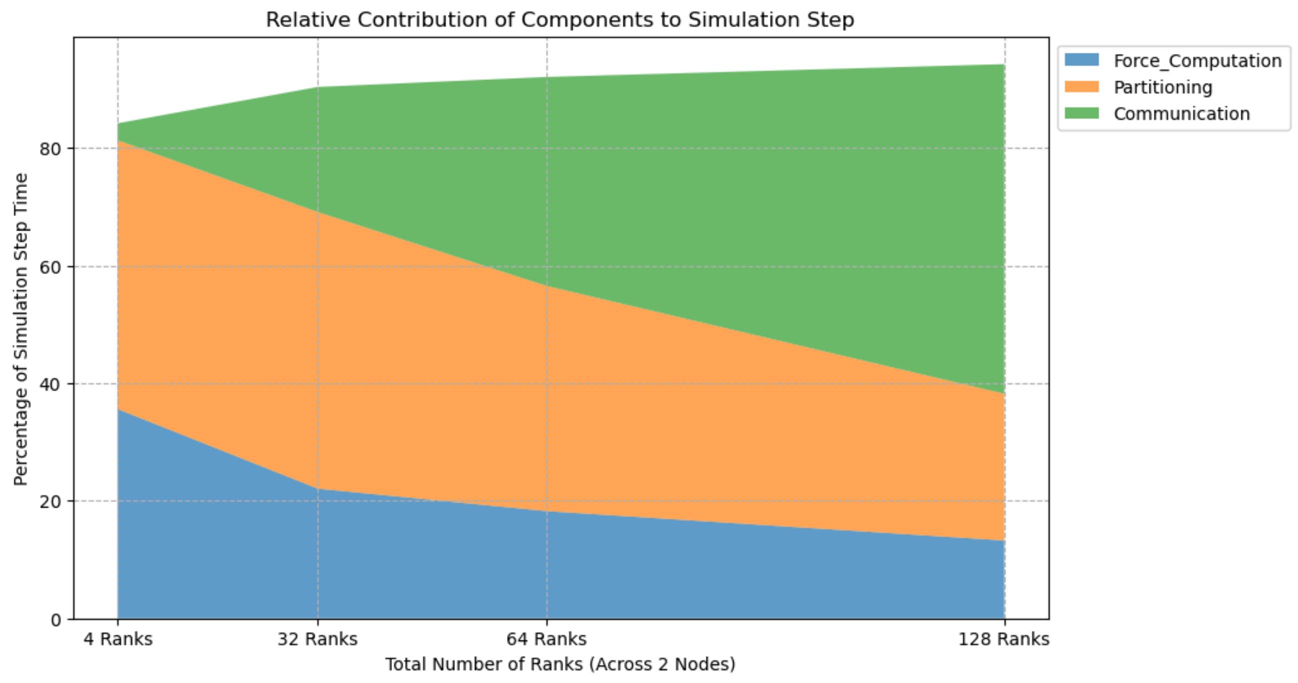
**Computation/Communication Breakdown:**

The figure below provides insight into how the total simulation time is distributed across different computational components as the number of ranks increases. After breaking the particle simulation down into initialization, bin building, one simulation step, force computation, partitioning/movement, and communication, it was apparent that the three major components that made more the 80% of the total computation time were force computation, partitioning overhead, and communication overhead. Excluding initialization as a small cost, we broke down these three major components as their relative contributions of the total time spent in the simulate_one_step() that was the source of 99% of the simulation time. These breakdowns can be seen in the stacked area graph below.

Initially, force computation is the dominant factor when using fewer ranks, particularly at 4 ranks. However, as the number of ranks increases, force computation time decreases significantly, as the workload is efficiently distributed across multiple ranks. This trend demonstrates the effectiveness of parallelization in reducing the computational burden of force calculations.

Meanwhile, partitioning overhead remains a significant factor throughout the scaling process, though it gradually decreases as more ranks are added. The decrease is due to the fact that each processor handles a smaller subset of particles, thereby reducing the complexity of partitioning operations and the particle movement in them. However, this improvement is not linear, as diminishing returns begin to appear when scaling beyond 32 ranks, where additional ranks contribute less significantly to reducing partitioning costs, potentially evidence as an area of further optimization.

In contrast, communication overhead increases substantially as the number of ranks grows, becoming the dominant cost at 128 ranks. While adding more ranks distributes the computational workload, it also increases the frequency of inter-rank communication. This growth in communication time indicates that at a high number of ranks, the benefits of parallelization are offset by the increasing cost of exchanging ghost region data between ranks. The graph below confirms that beyond a certain threshold, adding more ranks does not proportionally improve performance for this implementation although this could be an area of further optimization. As currently implemented, time spent on communication can outweigh the time saved in force computation and partitioning.



This analysis highlights a fundamental tradeoff in parallel MPI simulations. While increasing the number of ranks initially enhances performance by balancing force computations, it also introduces additional communication overhead. To improve scalability beyond 64 ranks, future optimizations could focus on reducing communication costs through techniques such as asynchronous communication, overlapping computation with data exchange, or reducing the frequency of ghost region updates. These optimizations could allow for more efficient parallel execution and ensure that the simulation scales closer to ideal speedup at higher rank counts.