CS267 HW2-1

Molly McGuire, Milad Shafaie, Yinuo Zhang

February 26, 2025

1 Introduction

This assignment focuses on parallelizing a particle simulation using a shared-memory programming model. The simulation models n particles interacting through repulsive forces, with interactions only occurring within a specified cutoff distance.

In the initial naive implementation, the force computation was $O(n^2)$, iterating over all pairs of particles. However, with an efficient binning approach, we aim to achieve an $O(n)$ time complexity by restricting force calculations to local neighbors only.

We first implemented an optimized serial solution using spatial binning to limit force evaluations. Then, we parallelized the simulation using OpenMP, distributing the workload among multiple threads while ensuring correct synchronization to avoid race conditions.

The **objectives** of this assignment were:

1. Implement an $O(n)$ serial algorithm for force computation.
2. Parallelize the simulation using OpenMP and analyze its performance.
3. Achieve strong and weak scaling on the Perlmutter supercomputer.

2 Member Contribution

- Yinuo Zhang implemented the serial solution and provides optimizations strategies such as neighbor precomputing, particle bin update for only cross-boundary particles and even-odd rows split, (which is originally used for maximum parallel performance while reducing data-race as much as possible but we decide to not use this method eventually) We will discuss some of those optimizations techniques in the later section in this report.
- Molly McGuire gathered timestep results for serial and parallel implementations, as well as strong and weak scaling data, all of which are plotted in the Results section. She attempted to debug serial and parallel solutions but did not produce code used in the final implementation.
- Milad Shafaie helped in designing the parallel implementation, including identifying which aspect of the serial implementation were embarrassingly parallel versus those which would take some further thought to parallelize. Then, he was responsible for implementing the agreed upon parallel approach, debugging it, and coming up with an altered parallel approach when he could not resolve bugs that appeared in the initial approach.

3 Results

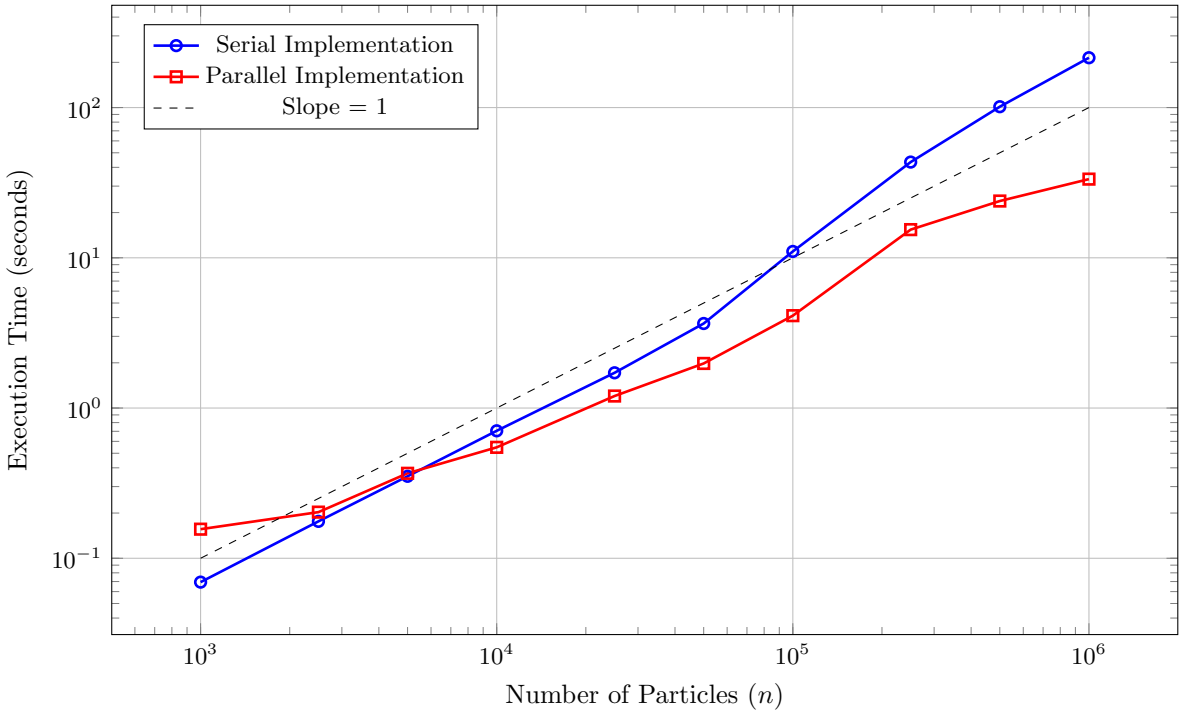

Figure 1: Serial vs. Parallel Execution Time

Figure 1 illustrates the execution time of serial and parallel implementations as a function of the number of particles (n). The serial implementation follows an $\mathcal{O}(n)$ trend, achieved through **spatial binning**, **neighbor search optimization**, and **Velocity Verlet integration**, which collectively reduce redundant force calculations.

The parallel implementation demonstrates improved execution times for larger problem sizes, but for small particle counts, synchronization and thread management costs result in slightly **higher execution times** than the serial version. As n increases, parallelization significantly reduces execution time by distributing the computational workload. However, for $n \geqslant 100,000$, performance begins to **diverge from ideal scaling**, likely due to **thread contention** and **memory bandwidth constraints**. While the parallel execution initially follows the ideal **slope = 1** reference line, diminishing returns emerge as synchronization overhead increases.

These results emphasize the scalability of parallelization while also highlighting **bottlenecks such as lock contention and memory access overhead**, which become increasingly significant at higher thread counts.

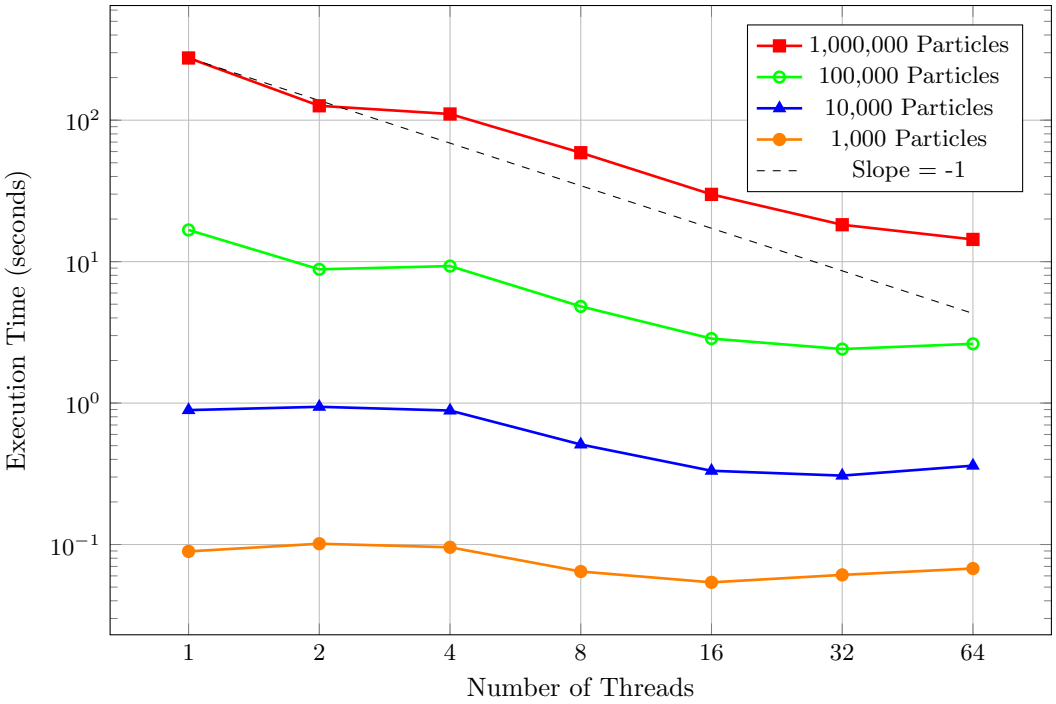

Figure 2: Strong Scaling Performance

illustrates strong scaling performance, where execution time decreases as the number of threads increases while keeping the problem size fixed.

The log-log plot in **Figure 2** demonstrates strong scaling, where execution time decreases as the number of threads increases for fixed problem sizes. The dashed black line represents the ideal **-1** Slope for scaling. Ideally, speedup should follow:

$$S(p) = \frac{T_1}{T_p} \tag{1}$$

where T_1 is the single-thread execution time and T_p is the time with p threads. Initially, the parallel implementation achieves near-ideal speedup. However, as threads increase beyond $p = 16$, **performance begins to plateau due to three primary factors**:

- **Synchronization overhead:** As more threads attempt to modify shared bins, **lock contention** increases, limiting parallel efficiency.
- **Memory bandwidth saturation:** High thread counts lead to increased **cache misses** and **non-uniform memory access (NUMA) penalties**, reducing speedup.
- **Computation vs. synchronization breakdown:** At $p = 64$, more time is spent waiting on locks than on actual force calculations.

For larger problem sizes ($n \geqslant 100,000$), the parallel implementation initially follows the expected trend but deviates from ideal scaling for $p \geqslant 16$ due to synchronization overhead, memory contention, and thread management costs. As threads compete for shared resources, contention increases, leading to diminishing returns.

For smaller problem sizes ($n = 1,000$), the overhead of parallelization outweighs computational benefits, causing execution time fluctuations instead of consistent speedup. This inefficiency arises from the workload per thread being too small relative to synchronization costs, highlighting the trade-off between computation and parallel execution overhead.

While the implementation achieves reasonable strong scaling, performance degrades at high thread counts due to hardware constraints and shared resource contention. Further optimizations, such as reducing lock contention, improving cache locality, and using non-blocking synchronization, could enhance scalability, particularly for larger thread counts.

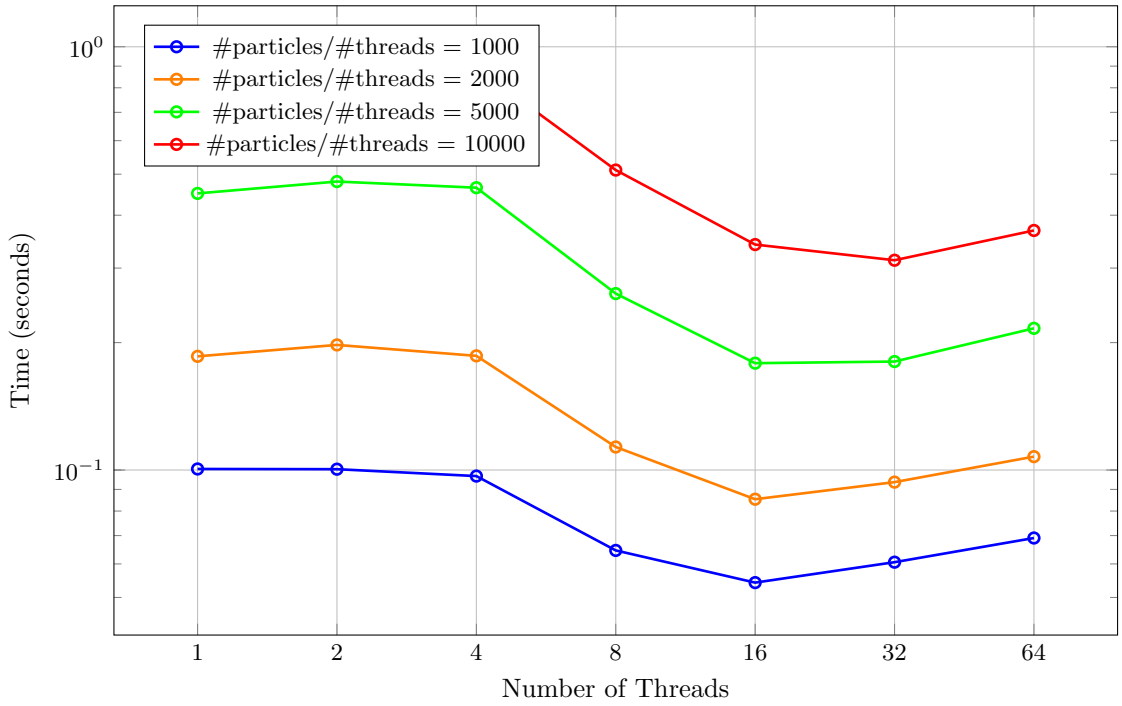

Figure 3: Weak Scaling Performance

The log-log plot in Figure 3 illustrates the weak scaling performance of our parallel implementation. Weak scaling measures how execution time changes as both the problem size and the number of threads increase proportionally. Ideally, the execution time should remain constant if the workload per thread is perfectly balanced and there are no significant synchronization or memory bandwidth constraints.

Each curve in the plot represents a different number of particles per thread, ranging from 1000 to 10000. At lower thread counts, execution time remains relatively stable, indicating efficient workload distribution. However, as the number of threads increases beyond 16, performance begins to degrade slightly.

This behavior can be better understood using the Roofline model, which considers computational intensity: the ratio of floating-point operations to memory accesses. Our implementation is bounded by memory bandwidth rather than compute throughput, meaning performance is constrained by how quickly data can be moved rather than raw arithmetic operations. As the number of threads increases, memory traffic grows due to concurrent accesses to particle data and bin structures, which limits potential speedup. Without better use of cache and memory access patterns, the implementation cannot take full advantage of the available computing power.

Despite these challenges, the parallel implementation demonstrates reasonable weak scaling behavior up to 16 threads. However, beyond this point, the overhead of parallel execution outweighs the benefits, leading to increased execution time. These results suggest that further optimizations—such as reducing lock contention, improving data locality, or employing alternative parallelization strategies—may be necessary to sustain weak scaling efficiency at higher thread counts.

4 Implementation Details

In this section, we detail the structure of our code, the key variables and lists used, and the rationale behind the serialization and precomputation techniques.

4.1 Code Layout and Key Variables

The code is organized into several functions that manage particle interactions, movement, and bin reassignment. The primary global variables include:

- **Binning Parameters:**
  - `bin_count`: The number of bins per row/column.
  - `bin_size`: The size of each bin, chosen to match the cutoff distance.
- **Particle Bins:**
  - `bins`: A vector of vectors holding indices of particles residing in each bin.
- **Neighbor Lists:**
  - `neighbors`: For each bin, this list holds the indices of adjacent bins that need to be checked for interactions. We will discuss more about this in neighbour precomputing section.
- **Synchronization:**
  - `bin_locks`: An array of locks used to ensure thread-safe operations when updating bins during particle movement.

4.2 Functions Overview

- `get_bin_index(x, y)`: Converts a particle's (x, y) coordinates to a corresponding bin index, facilitating fast bin assignment.
- `apply_force(particle, neighbor)` [Serial Version]: Computes the repulsive force between two particles and updates their accelerations based on Newton's Third Law. This approach avoids redundant computations by updating both particles simultaneously.
- `apply_force(particle, neighbor)` [Parallel Version]: Computes the repulsive force between two particles and updates their accelerations based on Newton's Third Law. In the parallel implementation, this function only updates accelerations for one of the particles to allow parallelization of force computation across bins while avoiding race conditions. This is elaborated on further later.
- `move(p, size, old_bin)`: Updates a particle's position and velocity using the Velocity Verlet integration method. It checks for boundary crossings—returning a flag when a particle moves from one bin to another so that bin reassignments are minimized.
- `init_simulation(parts, num_parts, size)`: Initializes the simulation by setting up the bins, precomputing neighbor relationships, and assigning particles to their initial bins. Here, the precomputation of neighbor lists is key: instead of repeatedly checking for valid adjacent bins during each simulation step, the neighbor indices are computed once at initialization. This results in significant performance gains during the simulation loop.
- `simulate_one_step(parts, num_parts, size)`: Encapsulates one simulation step, handling the force computation, particle movement, and bin reassignment. The force computation phase is divided based on even and odd row processing to minimize synchronization overhead, while the bin update phase only reassigns particles that have actually crossed bin boundaries.

4.3 Binning Strategy

Precomputing Neighbor Bins: To reduce lookup time during force calculations, each bin precomputes its valid neighboring bins once during initialization. Unlike the serial version, which considers only four neighbors (left, top-left, top, and top-right), the parallel implementation stores all eight adjacent bins per bin to facilitate force computations.

Initial Particle Assignment: Each particle is placed into its appropriate bin based on its (x, y) coordinates. This assignment is parallelized using OpenMP with locks ensuring thread safety.

Bin Reassignment Strategy: In the **serial** implementation, only particles that crossed bin boundaries were reassigned. In the **parallel** implementation, we initially attempted to update only changed bins, but this approach led to race conditions. To avoid synchronization issues, all bins are cleared and repopulated from scratch each step, with parallelized bin assignment ensuring efficient updates.

4.4 Force Computation

Reducing Redundant Interactions: In the **serial** implementation, forces were computed using a neighbor list that ensured each particle pair was considered only once. In the **parallel** implementation, this approach was abandoned due to potential race conditions when updating forces on both particles. Instead, forces are computed separately for each particle-bin pair, reintroducing some redundant calculations but ensuring correctness.

Parallelizing Force Computation: A naive parallelization would lead to data races when updating particle forces. Initially, we considered processing even-row bins first and odd-row bins in a subsequent step, ensuring no two concurrently processed bins were neighbors. However, reverting to the original force computation method yielded better performance due to increased parallelism. The force computation is parallelized by iterating over bins with OpenMP.

4.5 Synchronization

Bin Locks for Safe Parallel Writes: Each bin has an associated lock to prevent race conditions when particles are assigned. When a thread writes a particle index into a bin, it must first acquire the bin's lock, perform the write, and then release the lock.

Avoiding Data Races in Force Computation: Because forces are updated for individual particles rather than both particles in a pair, there is no need for additional synchronization during force computation.

Parallel Bin Reassignment: Our initial approach of updating only changed bins with locks led to segmentation faults. Instead, bins are cleared and repopulated from scratch in a parallelized manner, avoiding the complexities of handling concurrent modifications.

Balancing Synchronization Overhead: While locks ensure correctness, they introduce contention, particularly at high thread counts. Future optimizations could explore finer-grained locking strategies or alternative approaches such as lock-free data structures or task-based parallelism.

4.6 Code Alignment with Optimizations (Parallel Code)

The following code snapshot highlights key portions of the implementation that align with our optimization strategies, namely the `init_simulation` and `simulate_one_step` methods. The helper methods used in these functions are defined earlier in this document.

Algorithm 1 Initialize Simulation

Require: Array of particles `parts`, number of particles `num_parts`, simulation size `size`

1: $bin\_size \leftarrow cutoff$
2: $bin\_count \leftarrow \lceil size / bin\_size \rceil$
3: Resize $bins$, $neighbors$, and bin_locks to bin_count^2

▷ Initialize Locks

4: **for** $i \leftarrow 0$ to $bin\_count^2 - 1$ **do**
5: INIT_LOCK($bin\_locks[i]$)
6: **end for**

▷ Precompute neighbor bins

7: **for** $bx \leftarrow 0$ to $bin\_count - 1$ **do**
8: **for** $by \leftarrow 0$ to $bin\_count - 1$ **do**
9: $bin\_index \leftarrow bx \times bin\_count + by$
10: **if** $bx > 0$ **then**
11: Add $(bx - 1) \times bin\_count + by$ to $neighbors[bin\_index]$ ▷ Left
12: **end if**
13: **if** $by < bin\_count - 1$ **then**
14: Add $bx \times bin\_count + (by + 1)$ to $neighbors[bin\_index]$ ▷ Top
15: **end if**
16: **if** $bx > 0$ **and** $by < bin\_count - 1$ **then**
17: Add $(bx - 1) \times bin\_count + (by + 1)$ to $neighbors[bin\_index]$ ▷ Top-left
18: **end if**
19: **if** $bx < bin\_count - 1$ **and** $by < bin\_count - 1$ **then**
20: Add $(bx + 1) \times bin\_count + (by + 1)$ to $neighbors[bin\_index]$ ▷ Top-right
21: **end if**
22: **if** $bx < bin\_count - 1$ **then**
23: Add $(bx + 1) \times bin\_count$ to $neighbors[bin\_index]$ ▷ Right
24: **end if**
25: **if** $by > 0$ **then**
26: Add $bx \times bin\_count + (by - 1)$ to $neighbors[bin\_index]$ ▷ Bottom
27: **end if**
28: **if** $bx > 0$ **and** $by > 0$ **then**
29: Add $(bx - 1) \times bin\_count + (by - 1)$ to $neighbors[bin\_index]$ ▷ Bottom-left
30: **end if**
31: **if** $bx < bin\_count - 1$ **and** $by > 0$ **then**
32: Add $(bx + 1) \times bin\_count + (by - 1)$ to $neighbors[bin\_index]$ ▷ Bottom-right
33: **end if**
34: **end for**
35: **end for**

▷ Assign each particle to a bin

36: **for** $i \leftarrow 0$ to $num\_parts - 1$ **do**
37: $bin\_index \leftarrow$ GET_BIN_INDEX($parts[i].x, parts[i].y$)
38: LOCK($bin\_locks[bin\_index]$)
39: Add i to $bins[bin\_index]$
40: UNLOCK($bin\_locks[bin\_index]$)
41: **end for**

Algorithm 2 Parallelized Simulate One Step

Require: Array of particles `parts`, number of particles `num_parts`, simulation size `size`

▷ Step 1: Compute forces using precomputed neighbors

1: **for** $bx \leftarrow 0$ to $bin\_count - 1$ **do**
2: **for** $by \leftarrow 0$ to $bin\_count - 1$ **do**
3: $bin\_index \leftarrow bx \times bin\_count + by$
4: **for all** $i \in bins[bin\_index]$ **do**
5: $parts[i].ax \leftarrow 0, \quad parts[i].ay \leftarrow 0$ ▷ Reset acceleration
6: **end for**
7: **for all** $i \in bins[bin\_index]$ **do**
8: **for all** $j \in bins[bin\_index]$ **do**
9: APPLY_FORCE$(parts[i], parts[j])$
10: **end for**
11: **for all** $neighbor\_bin \in neighbors[bin\_index]$ **do**
12: **for all** $j \in bins[neighbor\_bin]$ **do**
13: APPLY_FORCE$(parts[i], parts[j])$
14: **end for**
15: **end for**
16: **end for**
17: **end for**
18: **end for**

▷ Step 2: Move particles

19: **for** $i \leftarrow 0$ to $num\_parts - 1$ **do**
20: MOVE$(parts[i], size, old_bin)$
21: **end for**

▷ Step 3: Reassign particles to bins

22: **for** $i \leftarrow 0$ to $bin\_count \times bin\_count - 1$ **do**
23: $bins[i] \leftarrow \emptyset$
24: **end for**
25: **for** $i \leftarrow 0$ to $num\_parts - 1$ **do**
26: $bin\_index \leftarrow$ GET_BIN_INDEX$(parts[i].x, parts[i].y)$
27: LOCK$(bin_locks[bin_index])$
28: $bins[bin\_index] \leftarrow bins[bin\_index] \cup \{i\}$
29: UNLOCK$(bin_locks[bin_index])$
30: **end for**