CS267 HW2-1

Molly McGuire, Milad Shafaie, Yinuo Zhang

February 25, 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. Additionally, she contributed to the write-up, including the Introduction, Optimization, and Results sections.

3 Results

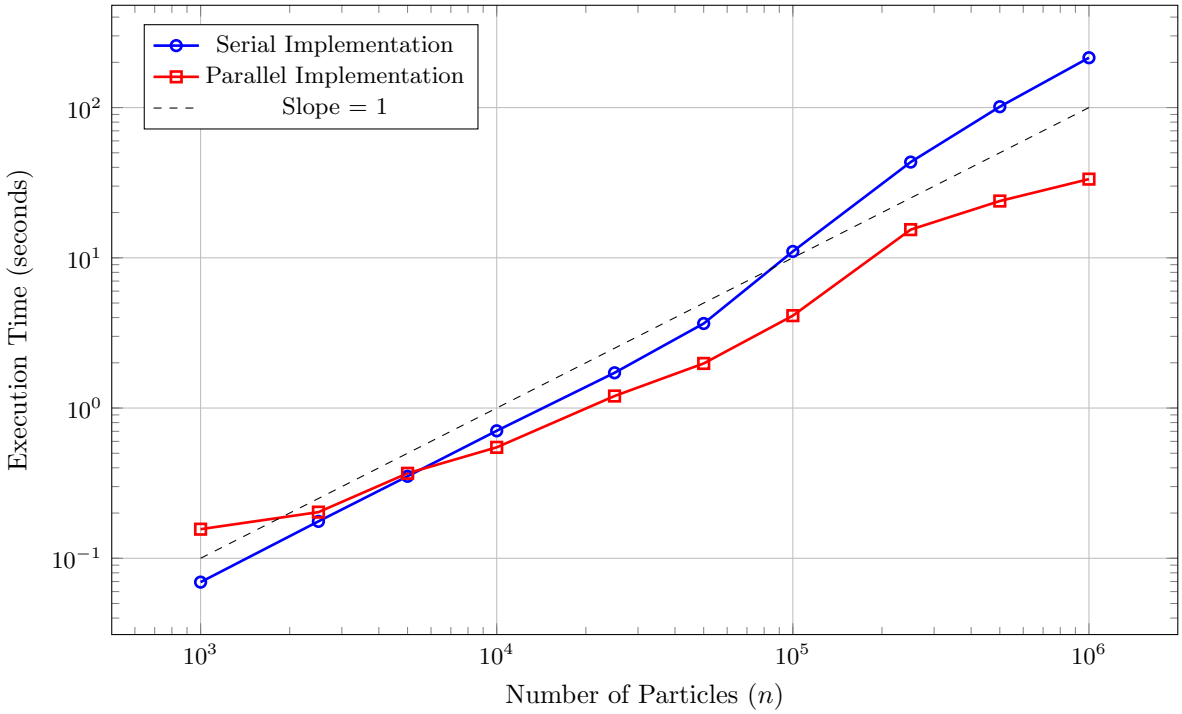

Figure 1: Serial vs. Parallel Execution Time

Figure 1: 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.

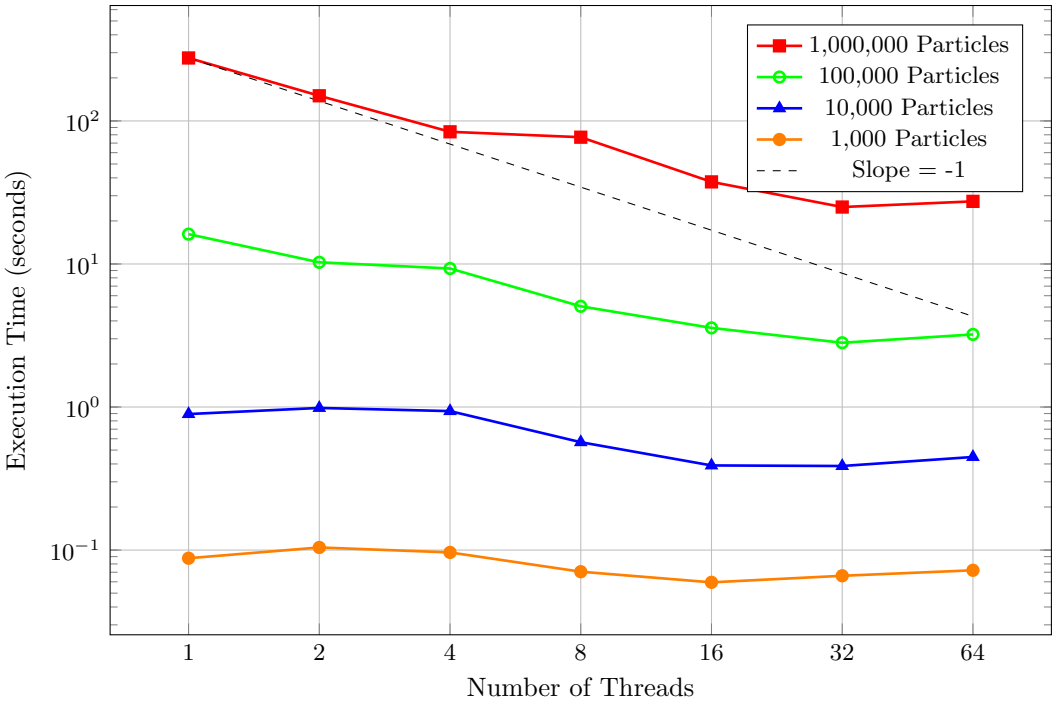

Figure 2: Strong Scaling Performance

The log-log plot in Figure 2 demonstrates strong scaling performance, where the execution time decreases as the number of threads increases for fixed problem sizes. The ideal scaling (dashed black line) represents perfect $\frac{1}{p}$ scaling, where p is the number of threads.

For larger problem sizes ($n \geqslant 100,000$), the parallel implementation follows the expected trend but begins to diverge from ideal scaling at higher thread counts. This is likely due to synchronization overhead, memory contention, and thread management costs, which limit further speedup.

For smaller problem sizes ($n = 1,000$), the impact of parallelization is minimal, and the execution time fluctuates slightly due to parallelization overhead outweighing computational benefits. These results highlight the trade-offs in parallel efficiency, where performance gains depend on both problem size and hardware constraints.

Speedup measures how closely our OpenMP implementation approaches the idealized p-times speedup, where doubling the number of threads ideally halves execution time. The speedup $S(p)$ is defined as:

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

where T_1 is the execution time using a single thread, and T_p is the execution time using p threads.

Figure 2 shows the speedup of our parallel implementation for different problem sizes. Initially, as the number of threads increases, speedup follows the ideal trend ($S(p) = p$), indicating efficient parallelization. However, for higher thread counts ($p \geqslant 16$), the speedup begins to deviate from the ideal case due to:

- **Synchronization overhead:** Increased thread contention for bin locks reduces efficiency.
- **Memory bandwidth limits:** As more threads access shared data, memory access latency increases.
- **Imperfect load balancing:** Despite even problem size distribution, small variations in thread execution introduce inefficiencies.

For small problem sizes ($n = 1000$), speedup is limited by parallelization overhead, as the computational workload per thread is too small to outweigh synchronization costs.

While our implementation achieves reasonable strong scaling, further optimizations—such as reducing lock contention, improving cache locality, or leveraging non-blocking synchronization—could enhance parallel efficiency, especially at higher thread counts.

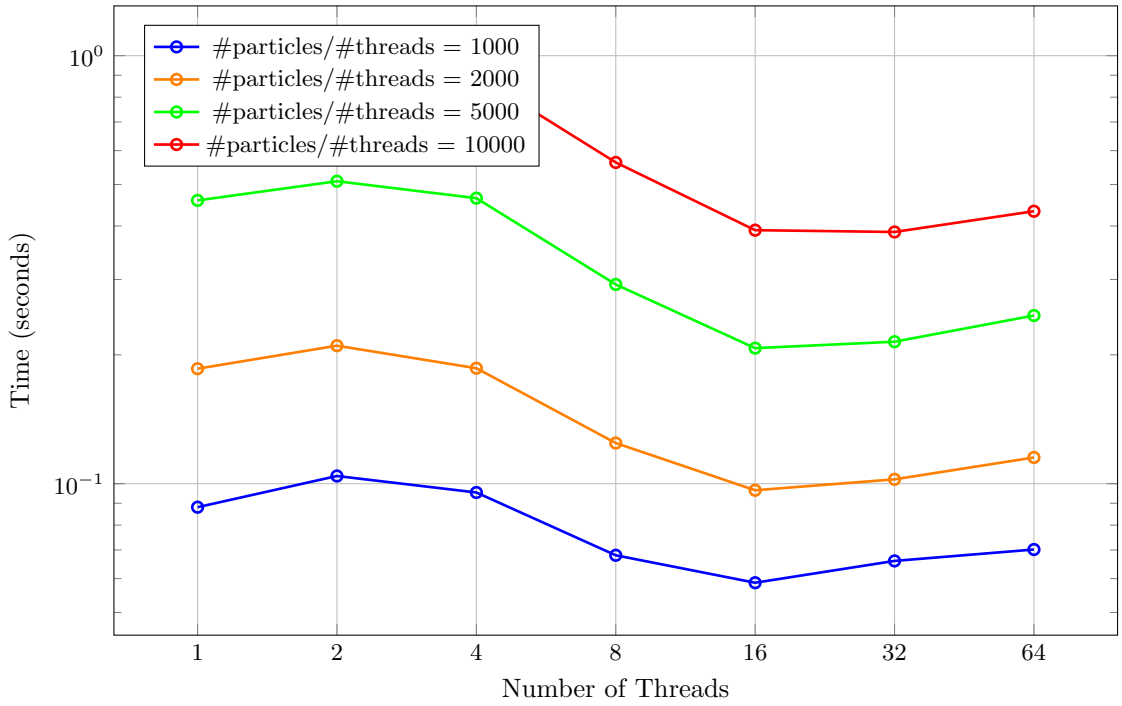

Figure 3: Weak Scaling Performance

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 due to factors such as:

- **Synchronization Overhead:** As the number of threads increases, contention for shared resources (e.g., locks on bins) can introduce delays.
- **Memory Bandwidth Bottlenecks:** With more threads accessing shared data, limited memory bandwidth can slow down performance, preventing ideal scaling.
- **Load Imbalance:** Although work is distributed evenly, slight variations in execution time between threads can lead to inefficiencies.

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.
- **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)`: 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.
- `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 Serialization and Parallel Considerations

Even though the simulation is designed for parallel execution using OpenMP, the code is structured in a way that it can be understood in a serial context. The main stages are:

1. **Initialization:** All necessary data structures, such as bins, neighbor lists, and locks, are established. Precomputation of neighbor lists avoids costly boundary checks in every time step.

2. **Force Computation:** The simulation calculates forces by only considering particles in the same bin or in one of the precomputed neighboring bins, thus reducing the potential $O(n^2)$ interactions to $O(n)$. The even/odd row strategy further minimizes race conditions by ensuring that adjacent bins are processed in separate iterations, preventing simultaneous updates to neighboring bins. Bins in even-numbered rows are processed first, followed by bins in odd-numbered rows in a separate parallel loop. This strategy ensures that any two concurrently processed bins are not neighbors, eliminating potential race conditions without requiring expensive synchronization mechanisms such as locks.

   However, certain optimizations from the serial implementation had to be reverted to enable efficient parallel execution. In particular, modifying the **apply_force** function to compute forces for both particles at once led to race conditions, making it necessary to revert to a version where each bin checks all of its neighboring bins individually. Additionally, bin reassignment had to be performed for all particles at each step instead of just moved particles because ensuring safe concurrent updates across multiple threads was difficult due to locking overhead. While this approach introduces some redundant computation, it ensures data consistency and prevents memory corruption in a multi-threaded environment.

   Despite these modifications, overall efficiency is maintained. Since only particles that have changed bins are reassigned rather than all particles, binning operations remain at $O(n)$, ensuring that the computational cost does not scale quadratically with the number of particles.

3. **Particle Movement:** Particles are updated using the Velocity Verlet method. Only particles that change bins trigger bin reassignment, which keeps synchronization overhead low. Instead of clearing and repopulating all bins every step, we track which particles moved and only update those, significantly reducing redundant bin operations.

4. **Bin Reassignment:** After particle movement, bins are cleared and rebuilt. This step uses locks to ensure that multiple threads do not interfere with one another.

4.4 Rationale for Precomputing Neighbors

The precomputation of neighbor bins is a central optimization:

- **Efficiency:** By calculating valid neighbor indices once during initialization, the simulation avoids repeated conditional checks during each time step, reducing computational overhead. Additionally, the neighbor list for each bin is assigned in a one-way manner, where for each pair of adjacent bins, only one considers the other as a neighbor. This avoids redundant force computations, as the same particle interactions are not re-evaluated in multiple bins.

- **Simplified Loop Structure:** With neighbor indices readily available, the inner loops for force computation are simpler and faster, as they directly iterate over a precomputed list.

- **Synchronization Benefits:** Fewer computations per simulation step mean that threads hold locks for shorter durations, thus lowering the overall synchronization cost. To ensure correct bin assignments in a multi-threaded environment, each bin is protected by a lock, which must be acquired before a thread modifies its contents.

  Locks prevent simultaneous modifications by different threads, but they introduce some overhead, which is why bin reassignment is fully parallelized despite the slight computational redundancy. Future optimizations could explore lock-free approaches or atomic operations for improved efficiency.

4.5 Code Alignment with Optimizations

The following code snapshot highlights key portions of the implementation that align with our optimization strategies. Our parallel implementation was designed with the following considerations:

- **Precomputation of Neighbor Bins:** Instead of checking boundary conditions and determining neighbor indices during each simulation step, we compute and store the indices of the four selected neighboring bins (left, top, top-left, and top-right) during initialization. This process is parallelized in **init_simulation** to reduce redundant checks during force calculations.
- **Particle Assignment:** After initializing the neighbor lists, each particle is assigned to its appropriate bin with proper synchronization using locks, ensuring that concurrent updates are handled safely. This is performed in parallel to improve efficiency while avoiding race conditions.
- **Parallelizing Force Computation:** In **simulate_one_step**, each bin computes forces for its particles concurrently, avoiding redundant force calculations and ensuring balanced workload distribution.
- **Parallelizing Particle Movement:** Particle updates in **simulate_one_step** are executed independently across multiple threads, minimizing dependencies and maximizing concurrency.
- **Parallelizing Bin Reassignment:** Instead of modifying only changed bins, all bins are cleared and repopulated each step. While this introduces some redundant bin operations, it ensures correctness in a multi-threaded setting by preventing inconsistencies that could arise from concurrent updates.

Algorithm 1 Precomputation of Neighbor Bins and Particle Assignment

for $bx = 0$ to bin_count **do**
  for $by = 0$ to bin_count **do**
    $bin\_index \leftarrow bx \times bin\_count + by$
    if $bx > 0$ **then**
      $neighbors[bin\_index].push\_back((bx - 1) \times bin\_count + by)$ ▷ Left
    end if
    if $by < bin\_count - 1$ **then**
      $neighbors[bin\_index].push\_back(bx \times bin\_count + (by + 1))$ ▷ Top
    end if
    if $bx > 0$ and $by < bin\_count - 1$ **then**
      $neighbors[bin\_index].push\_back((bx - 1) \times bin\_count + (by + 1))$ ▷ Top-left
    end if
    if $bx < bin\_count - 1$ and $by < bin\_count - 1$ **then**
      $neighbors[bin\_index].push\_back((bx + 1) \times bin\_count + (by + 1))$ ▷ Top-right
    end if
  end for
end for
▷ Particle assignment: Each particle is placed into its corresponding bin.
for $i = 0$ to num_parts **do**
  $bin\_index \leftarrow get\_bin\_index(parts[i].x, parts[i].y)$
  $omp\_set\_lock(\&bin\_locks[bin\_index])$
  $bins[bin\_index].push\_back(i)$
  $omp\_unset\_lock(\&bin\_locks[bin\_index])$
end for