# CS C267 Homework 2.1 Report

Maya Lemmon-Kishi, Zach Van Hyfte, and Minan Wu

Homework Group 63

## 1 Overview

In this report, we discuss the performance optimizations we made to a C/C++ program that simulates particle simulation. In Section 2, we discuss the design of the optimizations we made to the serial version of the algorithm program, which reduce its time complexity from $O(n^2)$ to $O(n)$, resulting in a final runtime of 691 seconds on a simulation containing $1,000,000$ particles. In Section 3, we discuss our implementation of an parallel version of the algorithm using OpenMP; on a single one of Cori's Knights Landing nodes, with `OMP_NUM_THREADS` fixed at `68`, this optimized parallel algorithm completes the same $1,000,000$-particle simulation in 17.9 seconds. Finally, in Section 4, we analyze the performance of both the serial algorithm and the parallel algorithm in greater detail.

## 2 Serial Algorithm Optimizations

We restructured the provided naïve two-nested-loops implementation of the function `simulate_one_step`, and introduced a number of optimizations throughout `serial.cpp` that dramatically improved the performance of the serial version of the simulation.

### 2.1 Particle Binning

The provided naïve serial implementation of the serial code was an $O(n^2)$ algorithm — it individually examined every possible permutation (not combination) $(i, j)$ of two particle IDs to determine whether particle $j$ was exerting a force on particle $i$. To reduce the number of operations performed and turn the algorithm into an $O(n)$ one, we adopted the particle binning strategy suggested in the recitation. Our optimized serial algorithm divides the simulation space into a grid of square blocks, with the length and width of each block equal to the cutoff radius ($r$). We selected $r$ as our block width because it minimizes the area of the neighboring blocks that must be examined to find all of the particles that could be exerting a force on a particle within a given block. Figure 1 illustrates this point — no matter where a particle $i$ may be located within its containing block $B$, all of the other particles that are close enough to $i$ to exert a force on it must reside within the eight blocks surrounding $B$.

Let $d$ be the particle density of the simulation, and let $n$ be equal to the number of particles in the simulation. The width and height of the simulation space, $s$, is thus equal to $\sqrt{dn}$. In a square grid with area $s$, comprised of blocks of width $r$, each row $l$ of the grid contains $\lceil \frac{s}{r} \rceil$ blocks, for a total of $l^2$ blocks in the entire grid.

We modified the function `init_simulation` to first calculate the size of each block and the number of blocks needed to cover the whole simulation space, based on the size and density of the simulation. Once it has calculated these grid dimensions, our `init_simulation` implementation calculates the indices of each block's eight surrounding "neighbor blocks," storing the list of neighbors for each block in a two-dimensional vector for later use. It then iterates through all of the particles in the provided array `parts` and computes the block number of each particle, assembling for each block a vector containing the indices of all of the particles within that block.

Specifically, our algorithm represents the grid of blocks as a two-dimensional vector `blocks`, a data structure of type `std::vector<std::vector<int>>`. The outer vector contains $l^2$ vectors, one for each

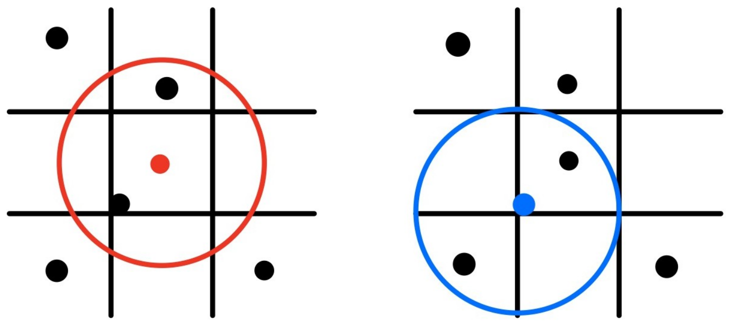

Figure 1: When the block width is $r$, No matter where a particle may be located within is containing block, a total of at most nine blocks need to be searched to find all other particles that may be exerting a force on that particle.

block. Each of the $l^2$ inner vectors stores the $ID$s of the particles currently located within that block. (The $ID$ of a particle is its index in the provided array `parts`.) The block number in which a particle represented by a `particle_t` object named `part` is located is equal $(\lfloor \frac{\texttt{part.x}}{r} \rfloor * l) + \lfloor \frac{\texttt{part.y}}{r} \rfloor$. A similar two-dimensional vector, whose outer vector also contains one vector corresponding each block, is is used to store the lists of each block's "neighbors."

## 2.2 Loop Structure

The naïve implementation of `simulate_one_step` iterates through every possible permutation $(i, j)$ of two particle IDs, calling `apply_force` on every such permutation. We replaced this implementation with one that iterates through each row of blocks in the grid from top to bottom, moving from left to right within each row before moving onto the next row. Within the loop iteration corresponding to a block $B$, an inner loop iterates through all of the particles $i$ within $B$. For each particle $i$ in $B$, the algorithm iterates through all of the particles in $B$ and in the eight blocks surrounding $B$, one block at a time. For each particle $j$ within these blocks, it calls `apply_force(parts[i], parts[j])`.

We chose this loop ordering to maximize (to the extent possible) the spatial locality of our algorithm. Because we iterate through particles block-by-block, information like the list of indices of the neighbors of block $B$ are more likely to be loaded into the cache once and kept there for the duration of the loop iteration corresponding to block $B$. An early version of our code iterated through every particle in `parts`, calculating its block index on the fly, but we quickly shifted to iterating through particles block-by-block because it allowed us to eliminate these redundant block number calculations, and because information about each block's neighbors could be kept in the cache rather than being recalculated or reloaded from memory every time the algorithm started processing a particle from that block.

In our first binning-optimized implementation of `simulate_one_step`, the loop iteration corresponding to a particle $i$ in block $B$ examined all of the eight blocks surrounding $B$ for other particles within the cutoff radius. As we optimized our algorithm further, we found a way to reduce the number of neighboring blocks examined during the outer loop iteration corresponding to $B$ to just four. We were able to make this optimization because of the changes we made to the function `apply_force` as part of the implementation of the symmetric force calculation optimization described in Section 2.3. Figure 2 shows how this works: we iterate through the blocks of the grid starting from Block 0, in the top left corner of the grid. By searching the current, bottom center, top right, middle right, and bottom right blocks, we are able to compute all possible force interactions.

## 2.3 Applying Symmetric Forces

For any pair of particles $(i, j)$, the repulsive force particles $(i, j)$ exerted on particle $j$ by particle $i$ has the same magnitude as the repulsive force exerted on particle $i$ by particle $j$, but acts in the opposite direction. Thus, we modified the vanilla `apply_force` to update the acceleration values of both of the input particles at a time, rather than just the first of the input particles. We made a corresponding modification to our implementation of `simulate_one_step` such that it calls `apply_force` once for every *combination* of two particles, rather than once for every *permutation* of two particles. In the serial implementation, updating the acceleration values of both of the input particles within a single call to `apply_force` does not affect the correctness of our algorithm. However, it does introduce a potential data race that requires additional synchronization logic when two threads executing `apply_force` could potentially be updating the same particle's acceleration values. We addressed this potential data race in the synchronization logic that we added to our parallel implementation, as described in Section 3.2.

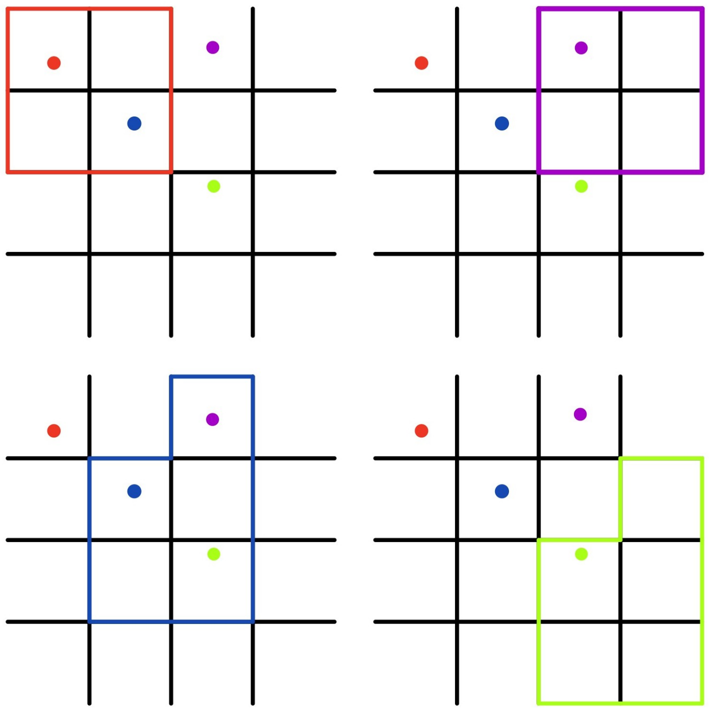

Figure 2: When symmetric forces on two particles are applied during a single call to `apply_force`, a maximum of five blocks must be searched to find all other particles that may be exerting a force on any given particle. The colored blocks represent the blocks searched to find all other particles that may be exerting a force on the particle with the matching color. This implementation saves the assertion of whether a certain pair of particles are already computed.

## 2.4 Re-Binning Particles

During each call to `simulate_one_step`, after the forces on each particle have been calculated and the particles have been moved (via calls to `move`) to reflect the actions of the relevant forces, the block numbers for all of the particles need to be recomputed to determine whether any particles have moved to different blocks (and thus whether those changes need to be reflected in updates to our data structure). We tested two different re-binning strategies as we implemented our serial and parallel algorithms:

- **Re-Bin All**: Completely clear out all of the block vectors in the `blocks` data structure. Then, iterate through the entire list of particles; for each particle, calculate its current block number, and then append the particle's ID to the inner vector in `blocks` that corresponds to that block number.
- **Re-Bin As Needed**: Leave the block vectors in `blocks` intact. Then, iterate through the entire list of particles; for each particle, calculate its current block number. Then, append the particle's ID to

the corresponding inner vector of `moved_particles` — a two-dimensional vector whose outer vector contains $l^2$ vectors, one for each block, such that the inner vector `moved_particles[i]` contains all of the particles that have moved into block $i$ during the current simulation step. Then, for each inner vector `moved_particles[i]`, appending all of the particle IDs in `moved_particles[i]` to `blocks[i]`.

**Re-Bin All** yields better performance for the serial algorithm, while **Re-Bin as Needed** yields better performance in the parallel setting. We believe that this is because, while **Re-Bin as Needed** is computationally more expensive overall, it allows for greater parallelization because threads spend less time waiting to acquire locks as threads can acquire all locks for particles in the same bin without waiting.

## 3 Parallel Algorithm Implementation with OpenMP

### 3.1 Parallelization

As suggested in the recitation, we divided the simulation step into three main phase such that each phrase is dependent on the result of the previous phrase. We used OpenMP to parallelize the three phases:

- **Applying Forces**: We evaluated two different methods of parallelizing the calls to `apply_force`: parallelizing at block-level granularity and parallelizing at particle-level granularity. The superior parallelization strategy for this phase of the simulation was parallelizing at block-level granularity. The loop ordering and data structures used for this phase of the parallel algorithm are identical to the ones used in the corresponding phase of our serial algorithm. Each thread is assigned a group of blocks. For each block it is assigned, the thread calls `apply_force` on every combination of two particles from inside that block. It then calls `apply_force` on every combination consisting of a particle from inside the block and a particle from one of the block's four "neighbors." Block-level parallelization prevents as much lock-waiting in `apply_force` as possible, as it keeps all of the processing related to a particular particle as local to a specific processor as possible.

  Particle-level parallelization is sub-optimal, as the order of the particles is not fully predictable. Based on our tests, it appears that it is more likely under this strategy that two threads will try to update the same `particle_t` object simultaneously. This can introduce a significant slowdown, as threads are stalled waiting for a lock more often, and keeping the processor busy requires more context-switch overhead.

- **Moving Particles**: At this phrase, we faced similar design choices as phrase one. Since we already chose to log the moved away particles to implement `Re-Bin As Needed`, we hence need to compute the new bin index for each particle. So we chose parallelization across bins such that we can automatically obtain the old bin index and compare it against the new bin index to assert whether a particle moved away.

  We integrated moving and computing the new bin index in a single clause. If a particle moved away, we can log its new bin index and pop it out from its old bin.

- **Re-Binning Particles after Motion**: At the final phrase, we only need to push the moved away particles to its new bin as we already popped it out from its old bin. We can pop the particle form our log and push it to the new bin right after. Since the log is also divided across bins, so each thread can process each bin and its associated log which prevents data race at the beginning.

### 3.2 Synchronization

As described in Section 2.3, we modified the implementation of the function `apply_force` to update the acceleration values of both of the input particles, rather than just the first of the input particles, to eliminate redundant arithmetic operations. However, this introduces a potential data race — two threads might call `apply_force` at the same time on two pairs of particles that have a particle in common. Thus, any of the `particle_t` objects in the provided array `parts` might be subject to multiple simultaneous updates. To serialize the accesses to each `particle_t` object, we modified the `init_simulation` function to set up a lock table for use throughout the simulation. The lock table contains one `omp_lock_t` for each particle in `parts` — the lock `lock_table_parts[i]` is used to serialize accesses to `parts[i]`. To avoid deadlock, we carefully structured our `apply_force` implementation such that each thread holds at most one lock at a time — a thread always releases the lock it currently holds before acquiring another lock.

Our parallel algorithm uses a two-dimensional vector `moved_particles` to log which particles have moved out of their current bin after a given step. `moved_particles`' outer vector contains $l^2$ vectors, one for each block, such that the inner vector `moved_particles[i]` contains all of the particles that have moved into block $i$ during the current simulation step. Since the updated block numbers for particles are calculated in parallel, two threads may try to write to the same inner vector if they happen to simultaneously encounter particles that have moved to the same block. To serialize accesses, we introduced another lock table, `lock_table_moved_particles`. This lock table contains one `omp_lock_t` for each block — the lock `lock_table_moved_particles[i]` is used to serialize accesses to `moved_particles[i]`. When a thread encounters a particle that has moved from its current block to a new block $B$, it first the lock from `lock_table_moved_particles` that corresponds to block $B$ before it attempts to modify the inner vector of `moved_particles` that corresponds to block $B$.

### 3.3 Block Size Increase

As detailed in Section 2.1, our algorithm divides the space of the simulation into square blocks with width and height equal to the cutoff distance $r$. In our testing, $r$ proved to be the optimal block size for the serial implementation — it minimizes the area of the neighboring blocks that must be examined to find all of the particles that could be exerting a force on a particle within a given block, which minimizes the total amount of work done per particle. However, the small sizes of these blocks means that many blocks usually don't contain any particles at all, and those that do usually contain just one or two. This is somewhat inefficient in the parallel setting, where work is divided among processors in units of blocks; the small size of the blocks means that too much parallelism is exposed, and that the distribution of work across different processors could be uneven. We attempted solve this load balance problem by switching to dynamic load balancing (using OpenMP's `schedule(dynamic)` clause), but this introduced more overhead than it was able to recoup in the form of efficiency gains. So, we instead turned to increasing the block size. In our tests, we found that the optimal block size for our algorithm in a parallel setting is $4r$. Increasing the block size reduced the running time of our parallel implementation on $1,000,000$ particles by 1–2 seconds.

## 4 Performance Analysis

### 4.1 Serial Slowdown

The running time of our serial algorithm on a simulation containing $1,000,000$ particles (with a random seed) is 683.062 seconds. The serial slowdown of our algorithm is 1.01 (Fig 3), indicating its $O(n)$ time complexity. Fig 4 shows that the OpenMP implementation of the particle simulation also has complexity of $O(n)$.

### 4.2 Strong Scaling

On a single one of Cori's Knights Landing nodes, with `OMP_NUM_THREADS` fixed at `68`, the running time of our parallel algorithm on a simulation containing $1,000,000$ particles (with a random seed) is 17.9 seconds, representing a $38.5\times$ speedup over the serial algorithm.

The running time of our algorithm on a $1,000,000$-particle simulation using 1 processor is 691 seconds, while the running time using 68 processors is 17.9 seconds. The algorithm is $38.6\times$ faster when using $68\times$ as many processors, so its strong scaling efficiency is $\sim 56.6\%$.

### 4.3 Weak Scaling

The running time of our algorithm on a $1,000$-particle simulation using 1 processor is 0.47 seconds, while the running time of our algorithm on a $68,000$-particle simulation using 68 processors is 0.81 seconds. Its

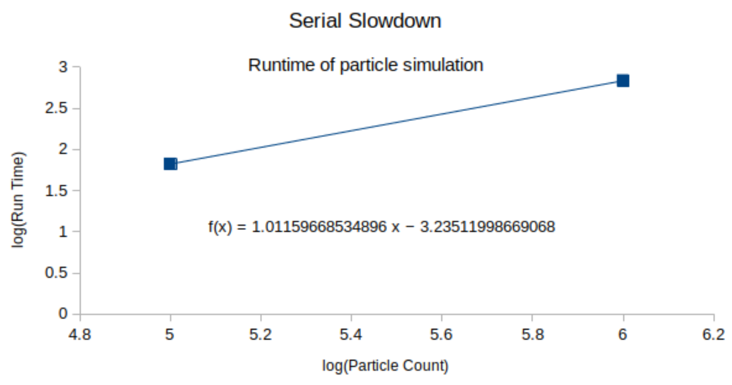

Figure 3: Serial slowdown plot of the $O(n)$ serial implementation of the particle simulation in log-log scale with a slope of 1.01.

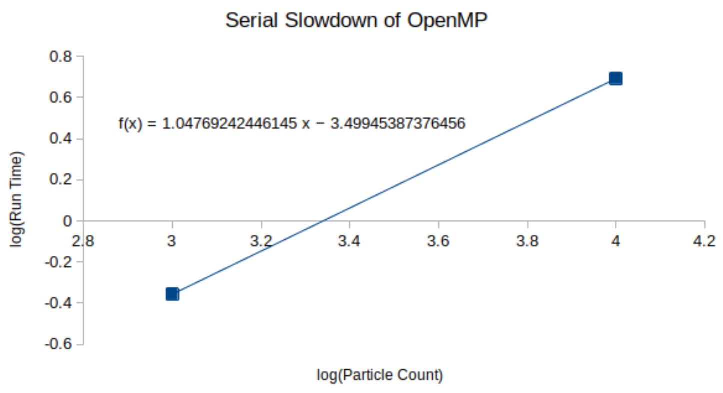

Figure 4: Serial slowdown plot of the OpenMP implementation of the particle simulation using a single thread, with slope of 1.04.

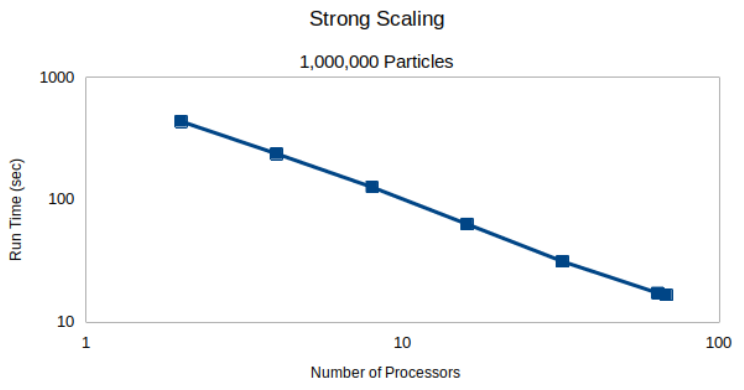

Figure 5:

weak scaling efficiency for 68 processors is thus equal to $\frac{0.47}{0.81} = \ \sim 58.0\%$.

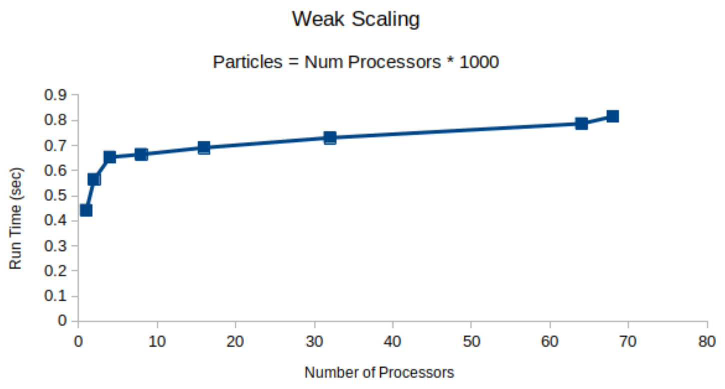

Figure 6:

## 5 Team Member Contributions

The members of our team contributed equally to this report and to discussions of different optimization strategies for the serial and parallel algorithms. The individual contributions of each team member are as follows:

- **Maya Lemmon-Kishi** implemented the initial versions of the particle binning and re-binning optimizations, the block neighbor index calculations, and the symmetric force calculations.
- **Zach Van Hyfte** optimized the loop structure and calculations of the initial version of the particle-binning serial algorithm to minimize its running time. He also tested alternative parallelization strategies and the developed the block size increase optimization for the parallel algorithm.
- **Minan Wu** Four Blocks Neighbors, Rebin as Needed, Three Clauses Parallelization, Lock Tables Synchronization