HW2: Parallelizing a Particle Simulation

Charis Liao
MS in MSSE
`charisliao@berkeley.edu`

Apratim Banerjee
MEng in EECS
`apratimbanerjee@berkeley.edu`

Yulin Zhang
PhD in Computational Biology
`zhangyulin9806@berkeley.edu`

February 23, 2024

1 Introduction

Particle simulations are essential in scientific domains like mechanics, biology, and astronomy, where particles interact to exhibit various behaviors. Efficient computation becomes crucial, especially with increasing particle numbers. This report focuses on optimizing the naive serial code for particle simulation, and then paralleling it using Open MP. Our working steps involved optimizing performance from a naive $O(N^2)$ to $O(N)$ using a modified serial code with a bin structure. Then, we achieved further speedup by distributing workload across multiple processors, targeting an overall time complexity of $O(N/p)$ with p processors. By achieving speedup and scalability and demonstrating our results using appropriate graphs, we showcased the efficacy of parallel programming in optimizing particle simulations.

2 Serialization

- The naive code logic: For each particle i, the inner loop iterates through all other particles j and calculates the force between the two particles using the `apply_force` function. As a result, the total number of force calculations is proportional to N^2. This kind of pairwise interaction is computationally expensive as the number of particles increases, so optimizing the code is required.
- We first tried binning techniques to reduce the complexity of the pairwise force calculations from $O(N^2)$ to $O(N)$. The idea was to divide the simulation space into bins and only consider interactions between particles that are within the same or neighboring bins.
- Adjusting the Number of Bins: Take `num_bin_x = num_bin_y = floor(size / cutoff)` so that the bins are small enough but larger than `cutoff`. Increasing the number of bins can lead to more accurate spatial partitioning but may come with increased computational overhead for creating bins and updating bins, but since the number of bins is always smaller than N, increasing the number of bins would not largely impact the overall $O(N)$ performance.
- Optimizing the Spatial Binning Logic: We also optimized the logic within the nested loops, like using separate loops for particles within the same bin and those in neighboring bins to minimize redundant checks.
- The thing that worked well for us was to use a data structure, such as a vector of lists, to store particles within each bin. It avoids repeated iteration over the entire list of particles for each bin.
- The final version of our working code got our serial simulation time down to **0.08s** for **1000** particles, **1.04s** for **10k** particles, **12s** for **100k** particles, **300s** for **1 million** particles (without writing the data into a file).

3 Parallelization using openMP

Synchronization is crucial in maintaining data integrity in shared-memory implementations. In this case, two different implementations came to mind, critical and locks.

Locks operate by providing mutually exclusive access to shared resources, allowing only one thread to acquire the lock and access the protected section of code at a time. This ensures that conflicting operations on shared data do not occur concurrently, preventing data corruption and race conditions.

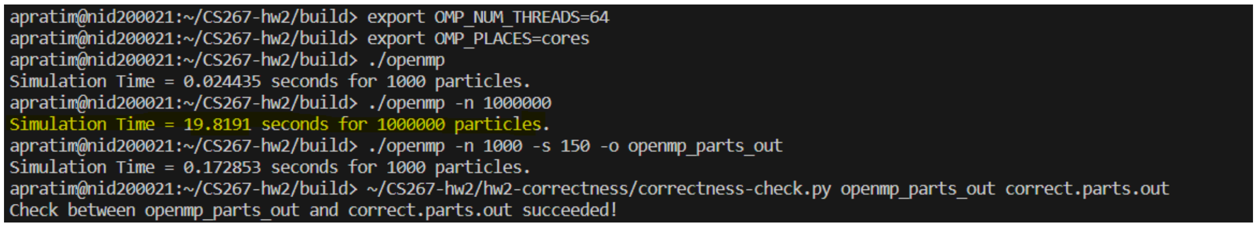
In contrast, `omp critical` only allows one thread to execute the critical section simultaneously. While `omp critical` ensures exclusive excess to shared resources, it may lead to performance bottlenecks, especially in our case assigning particles to bins, as it serializes execution.

We attempted both `critical` and `locks`, and found that locks were indeed faster because of the reasons above. Moreover, using locks instead of `omp critical` also guarded against overhead as they allow for finer-grained control over synchronization, enabling thread to access different bins concurrently without imposing unnecessary restrictions on parallelism. We chose only to apply locks when assigning particles into bins to minimize contention and avoid unnecessary serialization which helped reduce synchronization overhead, leading to improved overall performance in our parallelized code.

Additionally, we parallelized three `for loops` for bins assigning, force applying and moving particles. This parallelization strategy accelerates key computational tasks within the simulation. By distributing the workload across multiple threads, we optimized the efficiency of bin assigning, force calculation, and particle movement, leading to significant performance enhancements in particle simulation applications. To achieve better performance and reduce the cost of `locks` in parallelizing the code, we add additional data structures `movein` and `moveout` to record only particles that move to another bin in every new iteration, which avoid potential simultaneous $O(N)$ writes and reduce the cost of locks.

3.1 OpemMP Result

- The last version of our parallel code gave us pretty impressive results. For a fixed number of processors, i.e., 64 threads (single core), we got a simulation time for 1×10^6 particles in under 20 s (19.8 s to be precise).



```
apratim@nid200021:~/CS267-hw2/build> export OMP_NUM_THREADS=64
apratim@nid200021:~/CS267-hw2/build> export OMP_PLACES=cores
apratim@nid200021:~/CS267-hw2/build> ./openmp
Simulation Time = 0.024435 seconds for 1000 particles.
apratim@nid200021:~/CS267-hw2/build> ./openmp -n 1000000
Simulation Time = 19.8191 seconds for 1000000 particles.
apratim@nid200021:~/CS267-hw2/build> ./openmp -n 1000 -s 150 -o openmp_parts_out
Simulation Time = 0.172853 seconds for 1000 particles.
apratim@nid200021:~/CS267-hw2/build> ~/CS267-hw2/hw2-correctness/correctness-check.py openmp_parts_out correct.parts.out
Check between openmp_parts_out and correct.parts.out succeeded!
```

Figure 1: Simulation Time shown for 1×10^6 particles

- **Evaluating Parallel Code**
  To evaluate our parallel code's performance, we performed two tests, strong scaling and weak scaling.

- **Strong Scaling**: Keeping the number of particles (N) constant, we plotted the number of processors (threads) and the simulation time. Ideally, as the number of threads increased 2x, the simulation time decreased 2x. When we ran the code, we got a near-linear graph (as expected). The results are shown in Figure 2.

- Ideally, the simulation time should decrease linearly as the number of threads increases. However, in reality, there is a slight deviation as we witness diminishing reductions as the number of threads increases. This is probably due to the presence of non-parallelized parts in the program and multiple synchronization points in the code. Nevertheless, there is a close relationship between the ideal scenario (a curve with slope $= -1$) when the thread count is low. The deviation becomes noticeable only as the thread counts start to increase.

- **Weak Scaling**: Now increasing the number of particles (N) as we increased the number of threads by the same amount, we plotted the number of processors (threads) and the simulation time. Ideally, as the number of threads increased 2x and the number of particles increased 2x as well, the simulation time is supposed to stay constant. When we ran the code, we got a slight deviation as the number of threads increased, as expected. The results are shown in Figure 3.
- Several factors contribute to why it is difficult to achieve the ideal curves. One of the most prominent reasons is due to Amdahl's Law. Amdahl's Law assumes that there is a fixed portion of the computation that cannot be parallelized. If the parallelizable portion is smaller than anticipated or if the problem size is not large enough, the impact of parallelization may be limited, leading to deviations. Algorithmic efficiency may also play a role in this.

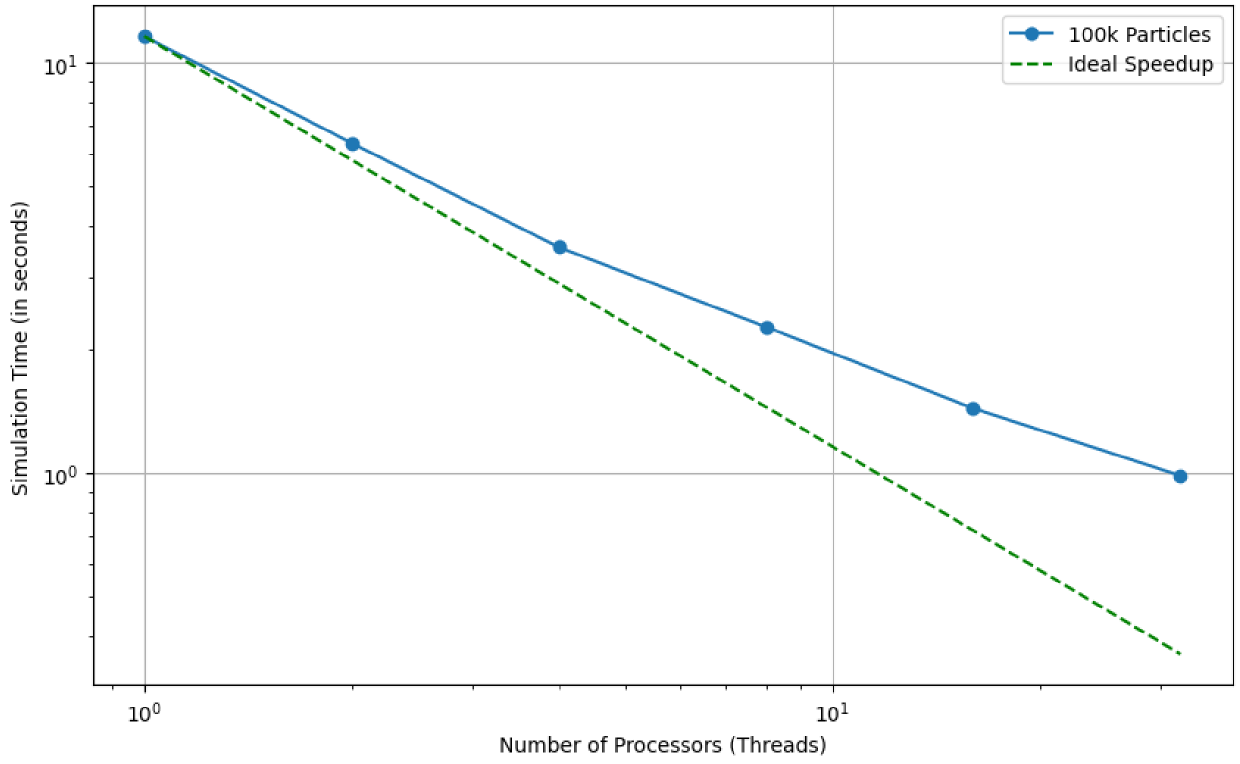


Figure 2: Strong Scaling, P vs T for fixed N

4 Conclusion

In conclusion, our efforts in optimizing and parallelizing the naive serial code for particle simulation have yielded significant performance improvements as shown in Figure 1. By transitioning from a naive $O(N^2)$ complexity to an optimized $O(N)$ complexity using spatial binning techniques, we managed to reduce computational overhead and enhance efficiency. Furthermore, the parallelization of critical sections using OpenMP, particularly employing locks instead of `omp critical`, facilitated concurrent execution across multiple threads, leading to substantial speedups in assigning particles into bins.

The evaluation of our parallel code through strong and weak scaling analyses showcased its scalability and efficiency. In **Strong Scaling**, where the number of particles (N) was kept constant, we observed a near-linear decrease in simulation time with an increasing number of threads, as expected. However,

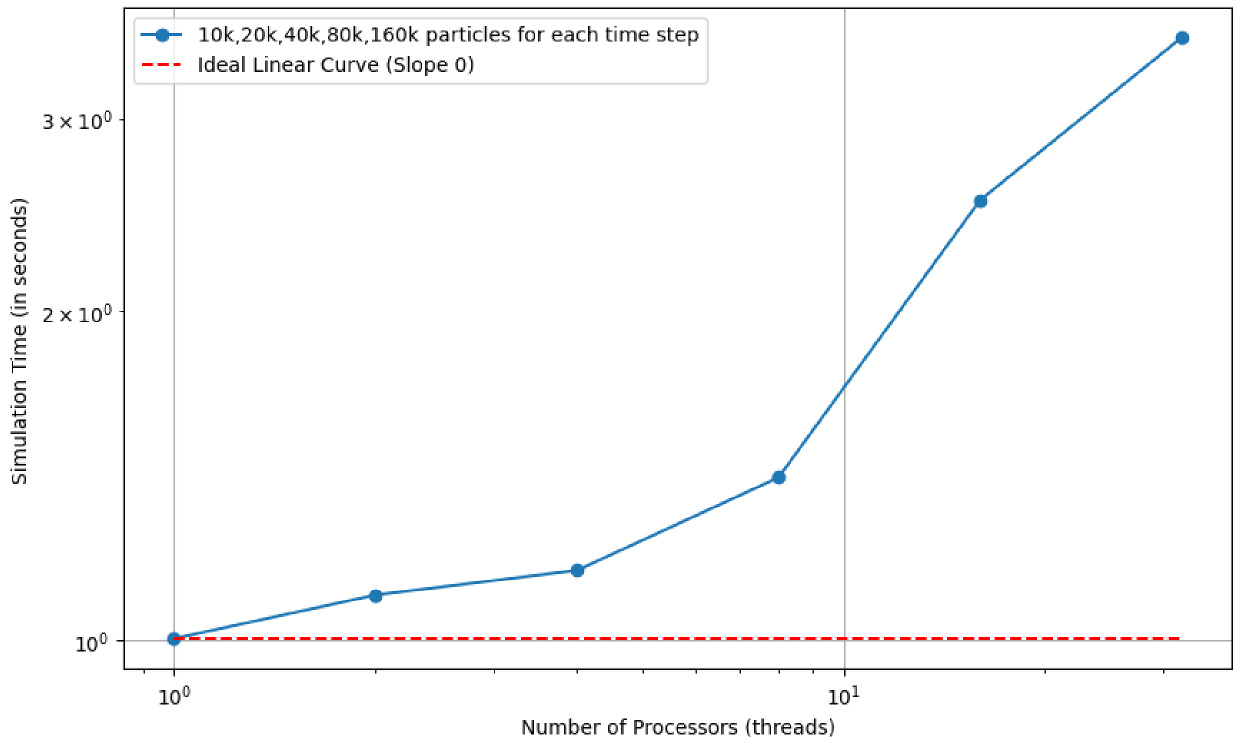


Figure 3: Weak Scaling, P vs T for varying N

slight deviations from ideal behavior were observed due to non-parallelized sections and synchronization overhead. Similarly, in **Weak Scaling**, where both the number of particles (N) and threads were increased proportionally, we expected the simulation time to remain constant. While there was a slight deviation as the number of threads increased, the results overall demonstrated good scalability. These deviations can be attributed to factors such as Amdahl's Law, which accounts for the presence of non-parallelizable portions in the code, and algorithmic efficiency. Overall, our optimized and parallelized particle simulation code showcases the effectiveness of parallel programming techniques in improving performance and scalability for scientific simulations.

5 Contribution

There was a good understanding between the team and we all worked together to collaborate effectively. More specifically, Apratim worked to optimize the serial code and worked on the final results to achieve strong and weak scaling. Yulin helped in fixing some bugs in the serial code, and worked on the OpenMP code to parallelize the optimized serial code. Charis worked on the OpenMP code as well and further optimized it to reduce the simulation time. Everyone contributed to the report writing.