CS267 HW2 Part 2 - 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 2024

1 Introduction

Homework 2.2 focuses on parallel programming within a distributed memory model, using MPI to parallelize a particle simulation. The simulation involves particles interacting via repulsive forces within a defined cutoff distance. Our aim is to exploit parallelism to achieve significant speedup over a serial implementation, capitalizing on the expected O(n) interactions due to the low particle density.

2 MPI

2.1 Overview

The Message Passing Interface (MPI) serves as a standardized communication protocol for parallel computing on distributed memory systems. It provides a framework for developing efficient and scalable parallel applications, enabling processes running on different processors or computing nodes to communicate, synchronize, and collaborate in performing complex computations. MPI is particularly well-suited for parallelizing scientific simulations, such as particle simulations, where the computational workload can be distributed across multiple processing units to accelerate overall computation.

Key Implementations:

1. **MPI Initialization**: The program initializes MPI using `MPI_Init` and retrieves the number of processes (`num_procs`) and the rank of the current process (`rank`) using `MPI_Comm_size` and `MPI_Comm_rank`, respectively.

2. **MPI Datatype Definition**: MPI provides mechanisms for defining custom datatypes to facilitate communication of complex data structures. In your implementation, a custom MPI datatype `PARTICLE` is defined to represent the `particle_t` struct, enabling efficient communication of particle data between processes.

3. **Parallel Particle Initialization**: Particle initialization is parallelized using MPI. While particle data is generated on the root process (`rank == 0`), it is broadcasted to all processes using `MPI_Bcast`, ensuring that each process has access to the same initial particle data.

4. **Simulation Execution**: The simulation is executed in parallel across multiple processes. Each process performs its portion of the simulation, computing particle interactions and advancing the simulation state. Synchronization and coordination between processes are managed through MPI communication operations.

5. **Output Handling**: Output handling is also parallelized using MPI. Simulation results are periodically saved to an output file, with the root process (`rank == 0`) responsible for gathering and saving the simulation state. This approach ensures that the output file is generated correctly even in parallel execution environments.

2.2 Methodology

- **Serial Optimization**: First we wrote a code that would handle serial implementation of $O(n)$ complexity versus the standard $O(n^2)$ complexity.

- **1D Spatial Decomposition**: The first attempt at writing the MPI parallelization code was dividing the spatial domain into rows and assigning each row to a different MPI process. This is a form of 1D MPI parallelization where the simulation space is divided along one dimension (rows in this case) among MPI processes. We ensured effective communication among rows and the exchange of particle data, including actual particles and ghost particles, to account for forces across process boundaries. We also ensured load balancing to ensure an equal amount of computational work for each processor.
  Our implementation fell short since we were constantly running into segmentation faults.

- **2D Spatial Decomposition**: Our next design approach was performing 2D spatial decomposition approach using MPI. The domain was divided into a grid of boxes, and each MPI process was responsible for managing the particles within its assigned boxes. This spatial decomposition helped distribute the computational load more effectively. After fixing the memory issues, we were able to successfully achieve a code that was fast even for high number of particles. Here's the breakdown of our code below:

2.3 Code Layout

This is how our code is segregrated:

2.3.1 Initialization and 2D Spatial Decomposition:

- The simulation domain was divided into buckets, forming rows along the y-axis and columns along the x-axis.

- MPI ranks were assigned ranges of rows, and each box in a row was associated with a specific column.

- Ghost particles were maintained for accurate force calculations and were exchanged during redistribution.

- Particles were initially assigned to boxes based on their positions, and each process handled particles in its assigned boxes.

2.3.2 Simulation Step and Force Calculation:

- The `simulate_one_step` function calculated forces between local particles and ghost particles received from neighboring processes.
- A 2D loop iterated over boxes and particles, applying forces within the same box and neighboring boxes.

2.3.3 Particle Movement and Redistribution:

- This particle simulation, given that the only force being considered is the attracting force among particles within a distance initialized as `cutoff` in the header file, we could easily see that communications are only necessary between neighboring bins that contain particles within the `cutoff` range of all particles in the focal bin. Therefore, by making sure the bin sizes are larger than `cutoff`, we confirm that communications needed in the simulation are between neighboring bins.
- After force calculation, particles were moved using Velocity Verlet integration.
- Particles moving to different boxes were temporarily stored in `move_bins` and then redistributed to their new boxes.
- Redistribution was done to maintain load balance and avoid deadlocks by splitting communication into even and odd ranks.
- Ghost particles were exchanged between neighboring processes.

2.3.4 Data Gathering for Output:

- The `gather_for_save` function collected the final particle states from all processes to the root process (rank 0) using MPI communication functions.

2.3.5 Explanation of `MPI_Gather` and `MPI_Gatherv`

Both `MPI_Gather` and `MPI_Gatherv` are MPI collective communication operations used for gathering data from all MPI processes to a single process. However, `MPI_Gatherv` offers more flexibility by allowing each process to contribute a different amount of data. In this implementation, `MPI_Gatherv` is chosen for the `gather_for_save` function to accommodate variable-sized data contributions from each process, facilitating coordinated output generation on the root process while maintaining efficiency and scalability.

2.4 MPI Parallelization results

- **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 (ranks) and the simulation time. Ideally, as the number of ranks increased 2x, the simulation time decreased 2x. When we ran the code and tested out the results for varying ranks keeping N constant, we got a near-linear graph (as expected). The results are shown in Figure 1.

- As you can see from the plot, there is a slight deviation as we witness diminishing reductions as the number of ranks increase. This is probably due to the presence of some 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 ranks are in the lower end. The deviation becomes noticeable only as the number of ranks crosses 64.
- **Weak Scaling**: In this plot we increased the number of particles (N) as we increased the number of ranks by the same amount, and 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 2.
- 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.

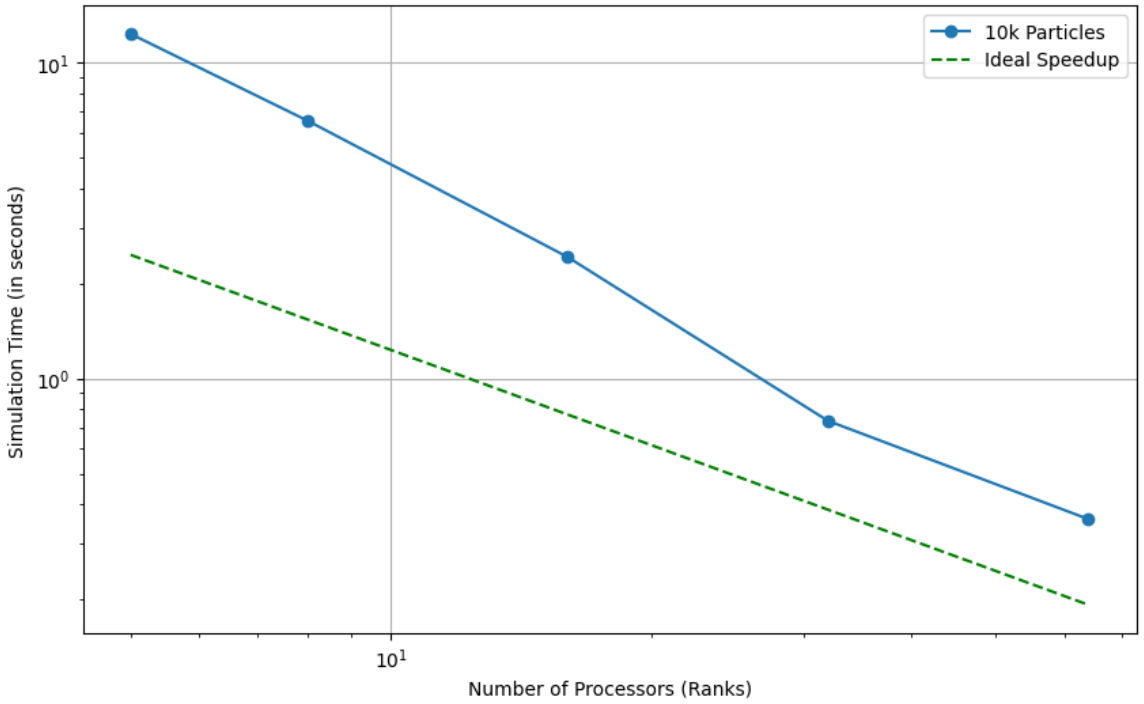

Figure 1: Strong Scaling, Ranks vs Time for fixed N

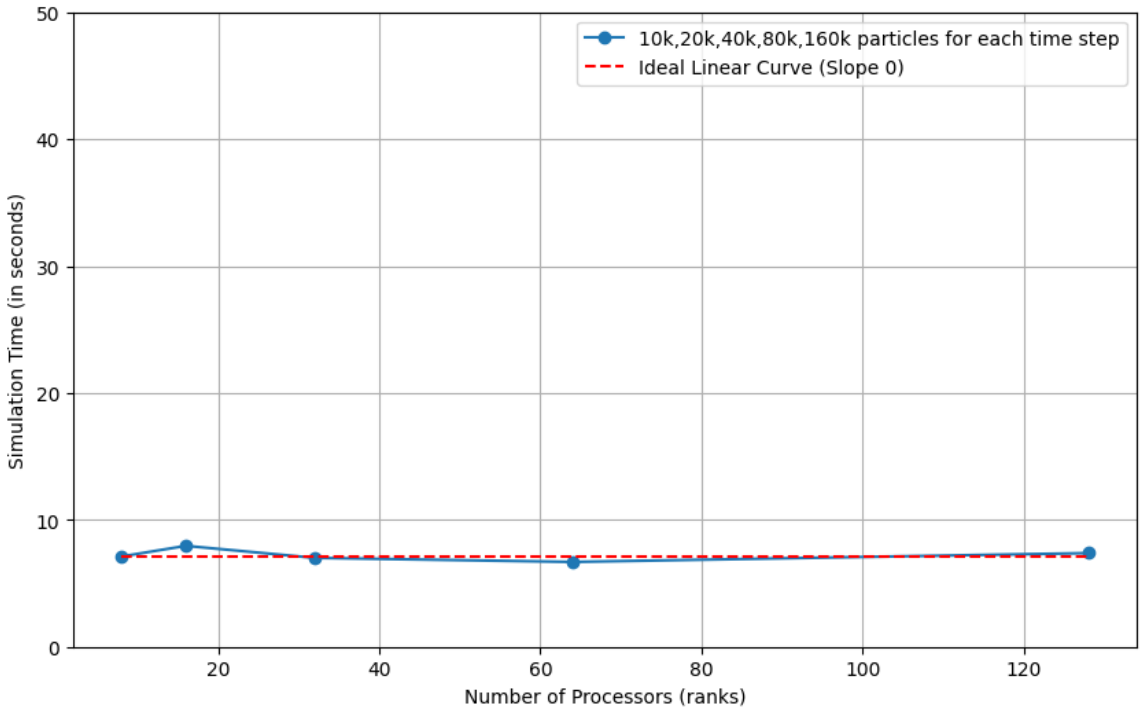

Figure 2: Weak Scaling, Ranks vs Time for varying N

3 Conclusion

In homework 2.2, we explored the parallelization of a particle simulation using MPI within a distributed memory model. Our primary objective was to achieve significant speed-up over a serial implementation by leveraging parallelism, particularly targeting the expected $O(N)$ interactions due to low particle density.

Through our implementation and optimization efforts, we successfully parallelized the simulation using MPI, employing both 1D and 2D spatial decomposition techniques. We overcame challenges such as segmentation faults and load balancing issues, ultimately achieving a scalable and efficient parallel code.

Evaluation of our parallel code through strong and weak scaling tests achieved near linear results. In strong scaling tests, where the number of particles was held constant, we observed near-linear speed-up with increasing ranks, indicative of efficient parallelization. Weak scaling tests, where the number of particles increased proportionally with the number of ranks, exhibited reasonable stability in simulation time despite deviations, attributed to factors such as Amdahl's Law and algorithmic efficiency.

Overall, our MPI parallelization approach demonstrated notable performance improvements and scalability, albeit with slight deviations from idealized speed-up curves. Future work could focus on further optimizing the code and addressing potential bottlenecks to achieve even better performance.

4 Contribution

Both Apratim and Yulin contributed to the implementation of MPI. Additionally, Yulin focused on optimizing the code from $O(N^2)$ to $O(N)$. Charis played a crucial role in identifying problematic functions that were causing segmentation faults. She also attended office hours to seek clarification and guidance. Together, we collaborated on writing the report.