HW 2.2 Report

Emma Brugman Sabrina Temesghen Dulce Torres

Introduction

The purpose of this assignment is to implement a parallel version of the particle simulation using Message Passing Interface (MPI). The distributed memory implementation consisted of partitioning particles into a row-layout in order to avoid redundant calculations and communication. Additional considerations consisted of particle distribution and ghost particle interactions. While improvements in performance were achieved by distributing calculations across processes, additional modifications would need to be implemented in order to achieve optimal performance. The bottleneck in the final optimization was rooted in communication time, thus, future modifications would focus on implementing a 2-dimensional layout.

Methods

1D Row Layout

Domain Decomposition

In the init_simulation function, the simulation space was partitioned among the MPI ranks along the y-axis following a 1D Row layout where each row was assigned to a processor. Each rank is assigned particles whose y-coordinates fall within its subdomain to ensure spatial locality. After partitioning, the indices definite boundaries of each rank are stored in *mpi_start_index* and *mpi_end_index*. Which will be later used in ghost communication.

Once the domain decomposition is established, each rank initializes the local *my_particles* vector with the particles within its region. At the end of the init_simulation function, the *MPI_Barrier* is then applied to ensure synchronization before proceeding.

Ghost Particle Communication

Identification of Ghost Particles

MPI uses a distributed-memory parallel implementation; prior to exchanging data from memory, each processor is responsible for computing interactions between particles locally and within a cutoff boundary of its assigned neighboring processors. In order to determine the location of the particles that are within the boundaries of the domain, each processor is assigned a top and bottom neighbor to exchange particles within the cutoff distance (*p_cutoff = 0.01*) from the boundary. As such this communication method using MPI will be used to accurately calculate the force within a rank while minimizing communication overhead. Using a for loop to interact through the vector of Particles (*my_particles*) with y-coordinates less than the top boundary (*mpi_start_index[rank] + cutoff),* are designated to send to the lower neighbor while particles with y-coordinates greater than the bottom boundary(*mpi_end_index[rank] - cutoff)* were sent to the higher-ranked neighbor. Before communication begins, the identified ghost particles near the top and bottom boundaries are stored in separate buffers (*send_bottom_ghost_particles*) and (*send_top_ghost_particles)* respectively.

The end ranks found at the beginning and end of the layout are set to only send and receive from their single adjacent neighbor. The first rank will only be sending and receiving from the neighbor below and the last rank will only send and receive from the neighbor above.

SendRecv

After identifying and storing the ghost particles, the data is exchanged between the adjacent processors using MPI's point-to-point communication function *MPI_Sendrecv*. However, prior to exchanging the data. Each processor first exchanges the number of particles it sends to the neighboring particle to allocate memory for the incoming data. Once the buffer sizes have been exchanged (top neighbor receives particle count from below & bottom neighbor receives particle count from above), the processor sends and receives the particle data to and from its neighbors in parallel. Using *MPI_Sendrecv,* for ranks with a bottom neighbor rank, the rank sends the count of particles to be exchanged and then receives the number of particles it will get. The rank will then allocate memory for the received particles. At the end, the actual particle data is exchanged between the neighbor ranks. The same process occurs for particles with a top rank.

MPI_Sendrecv works to increase parallelization so that each process has access to the ghost particles from its neighbors so that it can accurately compute the forces. This ghost particle exchange supports the interactions that will be handled locally within each processor's subdomain.

Force Calculation & Move

Each processor computes the forces within its assigned domain by iterating through local particles and applying the force function to pairs within the cutoff distance. To account for the particles near the boundaries interacting with the particles in the neighboring processors, we use ghost particles. Once ghost particles have been received, the simulation proceeds with computing the force for local particles where each rank calculates the forces between all pairs of local particles within its own domain. The forces between the ghost particles and the local particles are calculated separately.

After computing the forces for both local and ghost particles, each processor updates the positions of its assigned particles independently to ensure correct motion. Because particles move across partition boundaries redistribution is utilized by detecting particles that have exited a rank's assigned region and sending them to their respective neighboring processor. This communication is handled by MPI_Sendrecv. This implementation ensures accuracy of force calculations within boundary interactions.

Particle Redistribution

After the forces have been calculated, the move function updates the particle positions and each rank updates its particle positions. Since some particles may have crossed domain boundaries, these must be reassigned to the correct rank.Without this step, particles will migrate out of a rank's boundaries in the simulation and could lead to inaccurate results.

To do so, particles that exceeded the domain's boundaries were identified by comparing the y-coordinate to the *mpi_start_index[rank].* Those with smaller y-coordinates were sent to the *send_bottom_particles*, while larger y-coordinates were added to the *send_top_particles*. In order to prevent duplication of particles, the identified particles were then removed from the *my_particles* vector and stored in the *remove_indices*.

Utilization of *MPI_Sendrecv* during particle redistribution allowed for the exchange between ranks. As previously mentioned, it does this by counting the number of particles to allocate memory efficiently before transferring the data.

At the end of the simulation one-step, we applied the MPI_Barrier to allow for all ranks to complete redistribution before proceeding to the next simulation step.

Gather & GatherV

Once the simulation is complete, the particles are gathered at rank 0 for output. MPI_Gatherv is utilized to collect particle data and accommodate different numbers of particles per rank. After the particles are gathered at rank 0, they are ordered by their ID before writing to the output file.

Both *MPI_Gather* and *MPI_Gatherv* use all to one communication for the purpose of this assignment, *MPI_Gather* is used to collect the number of particles from each rank and send its particle count *num_parts* to rank 0. *Send_counts* stores the number of particles in each rank allowing for rank 0 to allocate memory for collecting all the particles (similar steps to *MPI_Sendrecv).*   Similarly, *MPI_Gatherv* is used to collect data from all particles, however it is used because different ranks may have different numbers of particles.

Finally, after gathering all the data, rank 0 sorts the particles by ID to maintain correct ordering.

Figure 1 {}: Gatherv implementation

Results

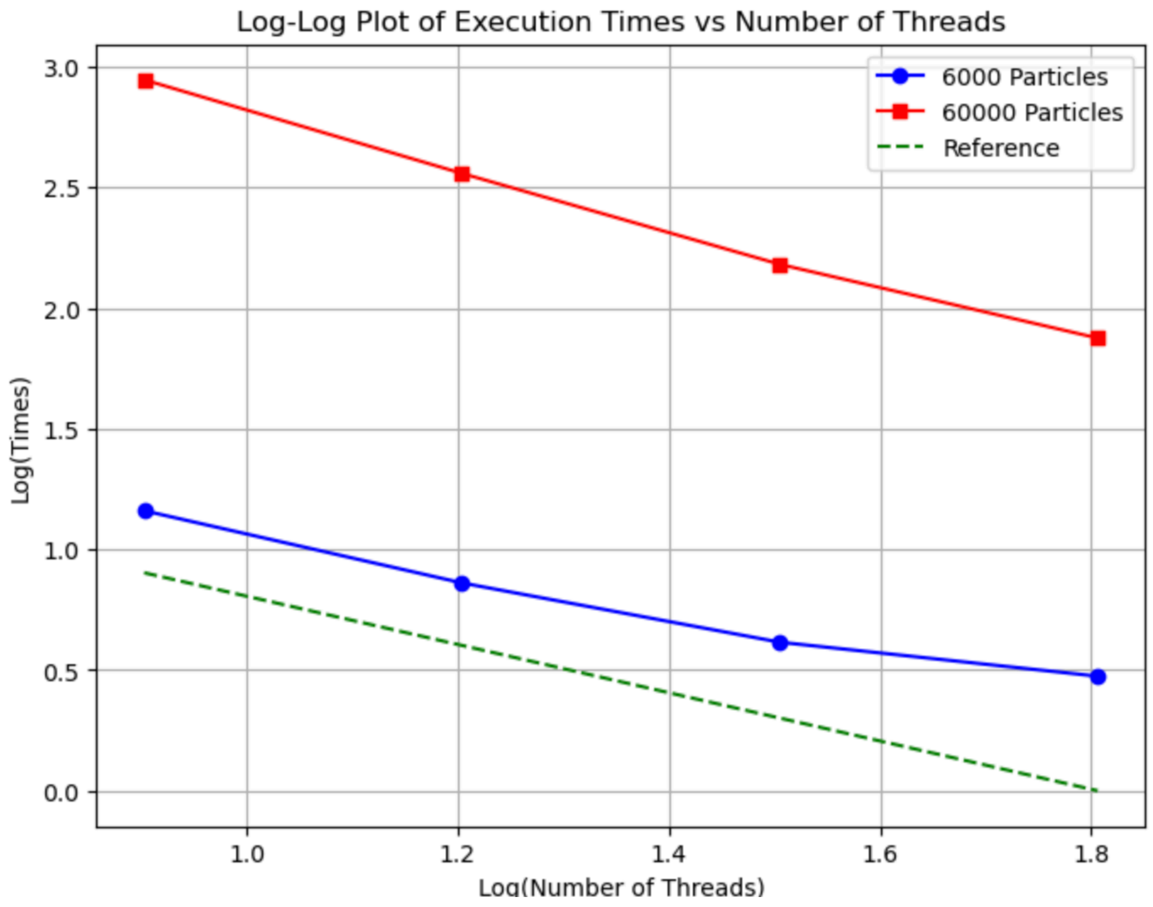

Figure 2 Strong scaling benchmark plot showing the logarithmic scale relationship between the number of threads and the corresponding times.

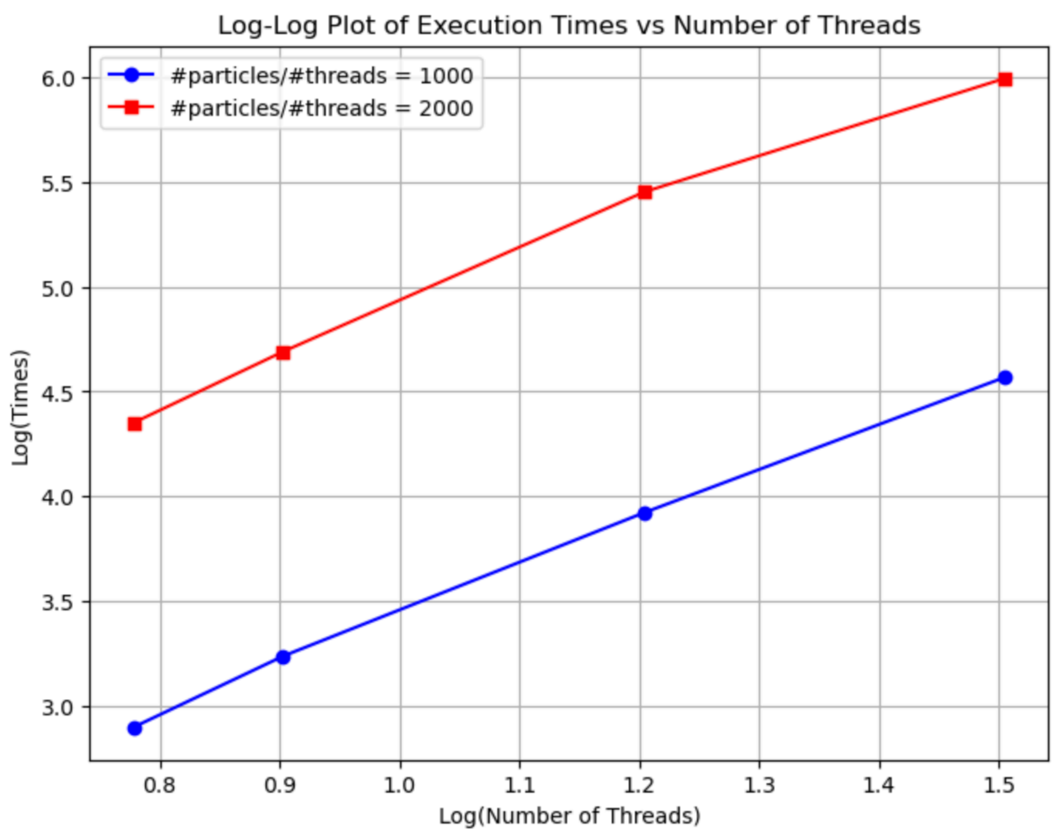

Figure 3 Weak scaling benchmark plot showing the logarithmic scale relationship between the number of threads and the corresponding times.

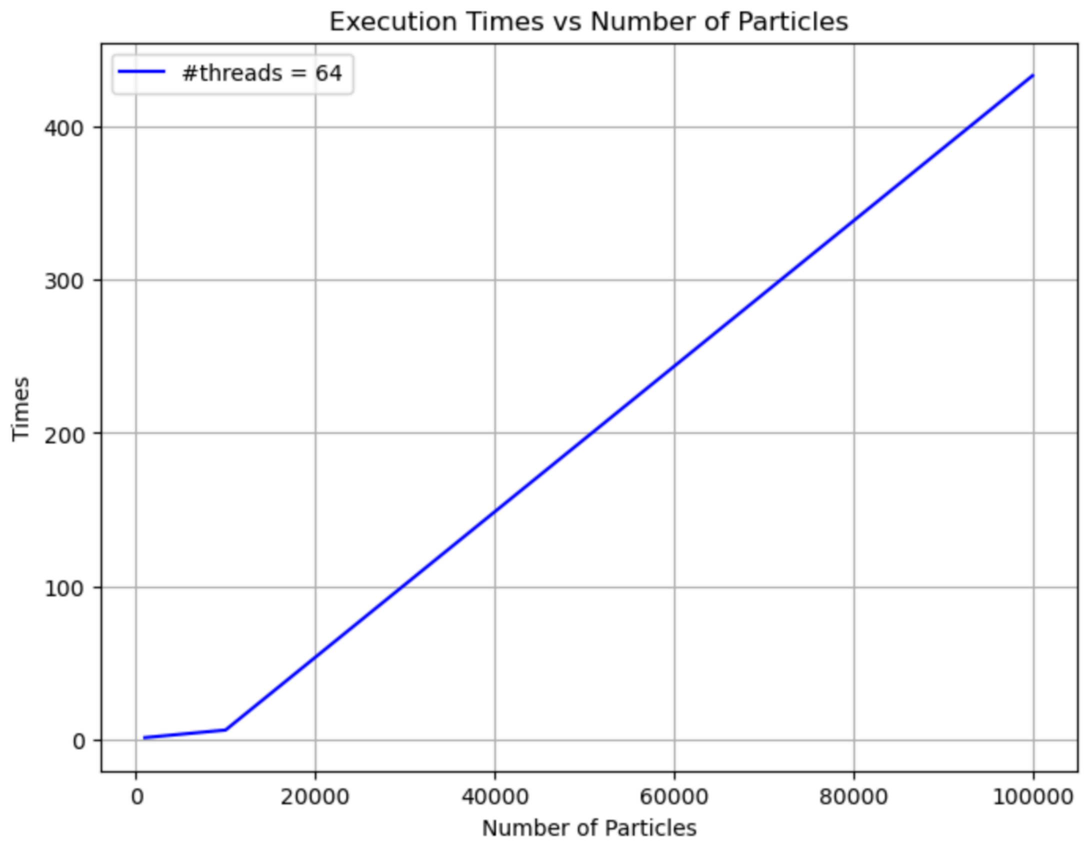

Figure Time complexity plot showing the relationship between the number of particles and the corresponding times for 64 threads.

<u>Discussion</u>

Strong scale benchmarking was implemented using 6,000 and 60,000 particles. This consisted of recording the execution time for different ranks while maintaining the size of the simulation system. These times were recorded for 8, 16, 32, and 64 threads. While it was evident in **Figure 2** that increasing the number of processes resulted in a reduced execution time, the rate of reduction is not ideal. An optimal performance would have resulted in a negative slope of 1, where the execution time was inversely proportional to the number of processes. However, the rate at which the time is reduced with increasing process counts decreases with increasing thread count. This was especially evident between processes 32 and 64 where the slope of the line begins to flatten out. This could be a result of the chosen data structure implemented in the serial optimization. Since communication is proportional to the length of a row border, execution time could have been improved through a 2D layout as it would have reduced communication time. Thus, it is believed communication was a prominent contributor to the bottleneck of the final implementation.

Weak scaling was implemented for two different particle to process ratios of 1000 and 2000. This is shown in **Figure 3**. Since the workload across ranks is maintained constant , a constant time across the different processes would have displayed optimal performance. However, a positive linear relationship between rank count and execution time is observed. Since the size of the system is increased proportional to the number of ranks, the increased time observed during weak scaling could have been a result of increased communication. This is especially plausible considering the 1-dimensional layout of the system. Since more interactions are expected with increasing particle count given the layout, computation time could also be a contributor to the observed bottleneck.

A closer look at time complexity reveals that there is much room for improvement in order to achieve O(n) from the parallel implementation. **Figure 4** shows the corresponding times for 1000, 10,000, and 100,000 sized systems across 64 ranks. The rapid growth in time across increasing systems reveals a quadratic time complexity. This was especially evident when testing higher number systems– increasing the system to 1,000,000 particles would exceed the allocated time. Since a linear time complexity was observed for the serial code, the quadratic time complexity is believed to be a result of communication overhead resulting from ineffective particle distribution. Further improvements would focus on implementing a grid layout in order to observe improvements in parallel performance.

Although we have minimized communication by using point-to-point opposed to all to all, and only communicating between ranks when necessary by explicitly applying communication between neighboring ranks. Communication still appears to be a significant bottleneck within our approach as expected. In addition, the *MPI_Sendrecv* works synchronously for each rank to wait until all neighboring ranks have been sent. Synchronization also increases with p, additional overhead can be attributed to Barrier Synchronization using MPI. While some ranks may finish earlier than others, they must wait at the *MPI_Barrier* for all other ranks to complete before proceeding. If workloads are unevenly distributed, the latency is reflected in the runtime.

In future attempts to overcome these bottlenecks, we can consider using *MPI_Isend* and *MPI_Irecv* to allow ranks to send and receive messages asynchronously so that they can continue with local computations. Similarly, we can also remove unnecessary *MPI_Barrier*'s so that force calculations between local particles can begin before waiting for the ghost particles. *MPI_Communication* can be also improved by dynamically adjusting the domain partitioning. Considering an adaptive load balancing of adjusting rank boundaries based on the number of particles in each rank will also simultaneously balance the workload distribution between ranks. *MPI_Allreduce* can also be applied to perform this balancing operation.

Such implementations aim to reduce idle time by overlapping communication with computation and improve parallelism for larger processor counts.

<u>**Contributions**</u>

Sabrina Temesghen:

- Implemented several parallelization methods.
- Debugging
- Contributed to write-up.

Dulce Torres:

- Implemented several parallelization methods
- Implemented performance analysis
- Benchmarking

Emma Brugman:

- Implemented several parallelization methods
- Testing different implementations
- Contributed to write up