Using MPI to Parallelize a Particle Simulation

CS 267
Spring 2024
Giovanni Battista Alteri, Jeffy Jeffy, Elizabeth Gilson

Introduction:
The assignment is an extension of the particle simulation performed in Assignment 2.1. The previous assignment was a naive implementation of a looping algorithm which calculates the force on particles in a closed system. Based on the force and acceleration of each particle, the particle moves within the box, and interacts with each other. The number of particles is constant, and so is the volume of the system (defined by the variable “size”, given by the square root of the total number of particles). We have chosen a density sufficiently low so that we will have $O(n)$ interactions with n particles. This means that we could achieve the same level of accuracy of the naive code even if we consider only the interactions among particles that are sufficiently close to each other.

This is a snapshot of the simulated system:

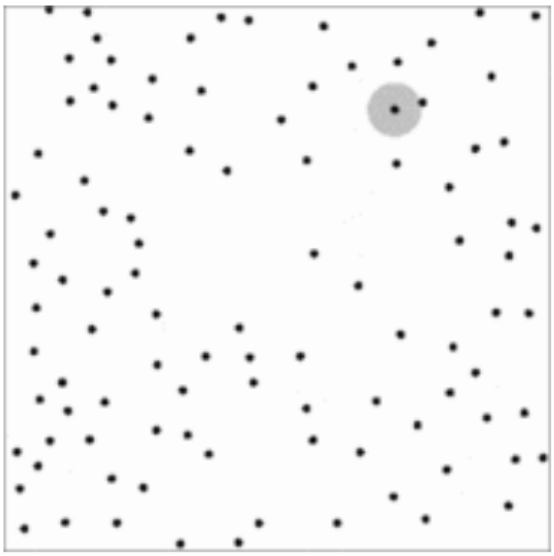

Figure 1. Example system

The starting naive implementation computes the forces on all the particles, by iterating through every pair, with a final asymptotic computational complexity of $O(n^2)$. This algorithm calls a nested for-loop that computes unnecessary interactions between pairs too far to interact, making this approach inefficient and slow.

Here you can see the naive code implementation:

```
void simulate_one_step(particle_t* parts, int num_parts, double size) {
    // Compute Forces
    for (int i = 0; i < num_parts; ++i) {
        parts[i].ax = parts[i].ay = 0;
        for (int j = 0; j < num_parts; ++j) {
            apply_force(parts[i], parts[j]);
        }
    }

    // Move Particles
    for (int i = 0; i < num_parts; ++i) {
        move(parts[i], size);
    }
}
```

Figure 2. Main loop for naive implementation of simulation

MPI Implementation

The communication in the distributed memory implementation primarily relies on MPI (Message Passing Interface) for inter-process communication. This implementation divides the simulation domain into multiple processes, with each process responsible for a portion of the domain. Processes exchange particle information with neighboring processes to calculate forces accurately and simulate particle interactions across process boundaries.

At the beginning of the simulation, each process initializes its local particle data and determines the distribution of particles across processes based on their positions. Particle data is exchanged between neighboring processes to ensure that each process has the necessary information about particles located in adjacent regions. This exchange is facilitated by the send_out and receive functions, which send and receive particle data using MPI communication primitives: MPI_Send and MPI_Recv.

```
82
83  void receive(std::list<particle_t>& in_parts, int incoming_rank)
84  {
85      MPI_Status status;
86      int num_particles;
87
88      MPI_Recv(&num_particles, 1, MPI_INT, incoming_rank, 0, MPI_COMM_WORLD, &status);
89
90      for(int i = 0; i < num_particles; i++)
91      {
92          particle_t received_particle;
93          MPI_Recv(&received_particle, 1, PARTICLE, incoming_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
94          in_parts.push_back(received_particle);
95      }
96  }
97
98
99  void send_out(std::list<particle_t>* outgoing_parts, int send_rank, int destination_rank)
100 {
101     int num_particles = outgoing_parts->size();
102
103     MPI_Send(&num_particles, 1, MPI_INT, destination_rank, 0, MPI_COMM_WORLD);
104
105     for (const auto& particle : *outgoing_parts) // foreach loop
106     {
107         MPI_Send(&particle, 1, PARTICLE, destination_rank, 0, MPI_COMM_WORLD);
108     }
109 }
110
```

Figure 3. Code for send_out and receive functions

During the simulation, each process computes forces acting on local particles by considering interactions with both local and neighboring particles. The apply_force function calculates the force between each particle and its neighbors within a certain cutoff distance. To account for particles located in neighboring regions, ghost particles are exchanged between processes to ensure accurate force calculations across process boundaries. This exchange is performed at each simulation step to update the ghost particle data.

Design choices

We found that using the original 2D vector that we used in homework 2-1 was not efficient for this project. Instead the choices in this implementation include the partitioning of the simulation domain into processes, the method for exchanging particle data between processes, and the strategy for handling particle interactions at process boundaries. These choices directly impact the performance of the simulation in terms of scalability, communication overhead, and computational efficiency. For example, the granularity of domain decomposition affects the balance of computational workload among processes, while the frequency and method of ghost particle exchange influence the overhead of inter-process communication. By optimizing these design choices, the performance of the distributed memory implementation can be improved, leading to better scalability and efficiency for large-scale simulations.

We also originally tried to use the MPI_Gather function. However, later on we decided it would be more efficient and easier to implement code that just uses the send and receive function. In this way we also had fewer issues like potential deadlocks and having to carefully place barriers.

Results

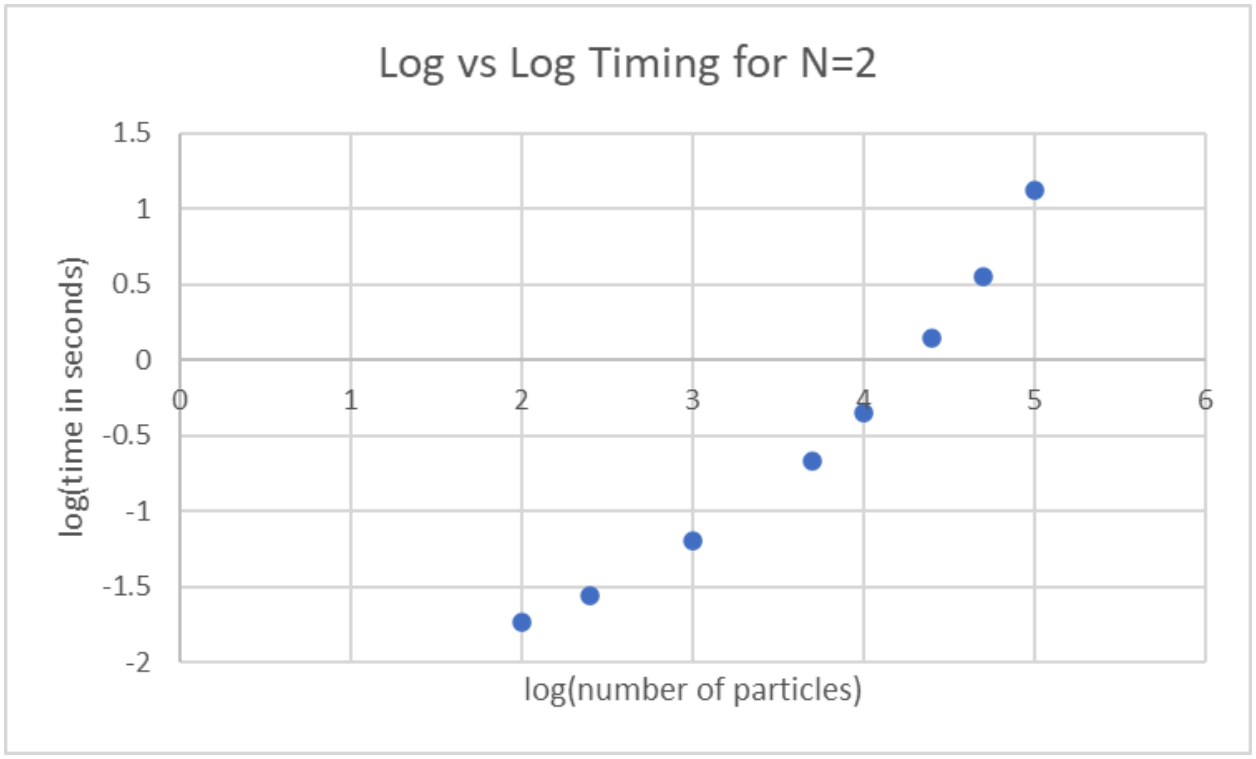

Figure 4. Log(Timing) vs Log(Number of Particles) Results for Two Nodes

This graph show the log of the number of particles vs the log of the time in seconds for a simulation run on 2 nodes, with the number of tasks per node set to 64. This shows that the correlation between log(time) and log(number of particles) is approximately logarithmic.

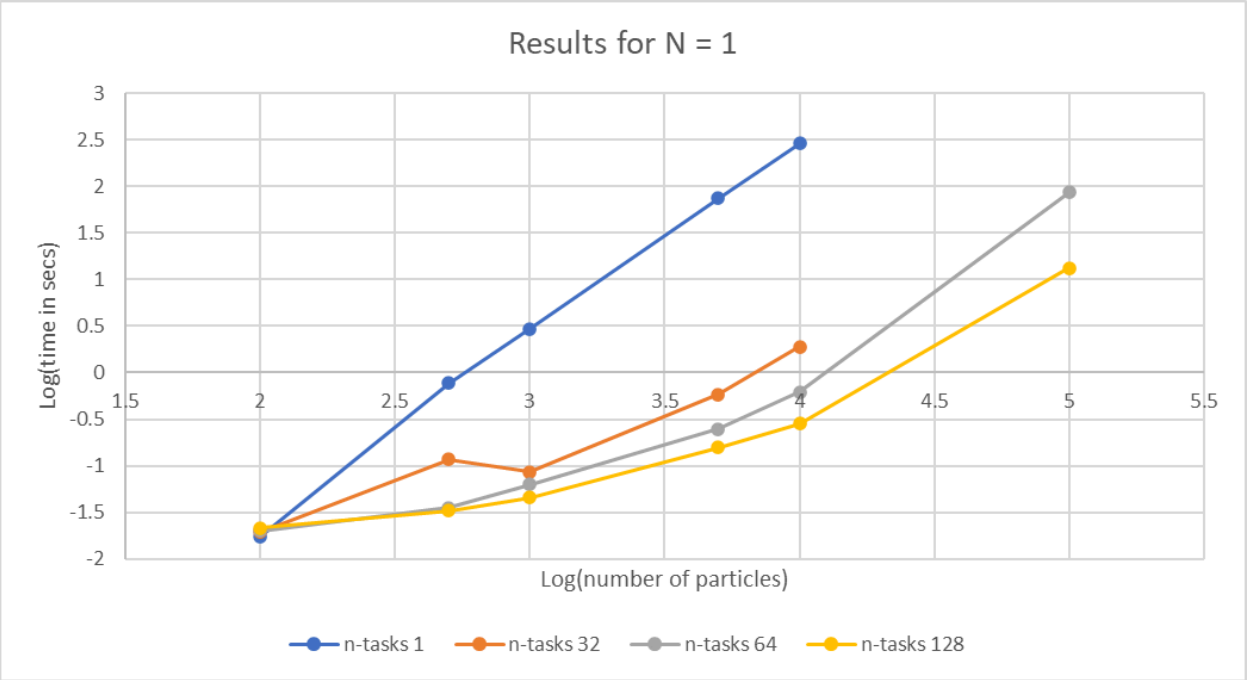

Figure 5. Log(Timing) vs Log(Number of Particles) Results for One Nodes and Various Task Numbers

This graph shows the results for scaling when using one node with varying number of tasks per node. The x-axis shows the log of the number of particles used in the simulation and the y-axis is the log of time it took the simulation to run. Each line uses a different numbers of tasks per node (1, 32, 64, and 128). You can see that as the number of tasks per node increases, the speed of the simulation increases throughout the different numbers of particles.

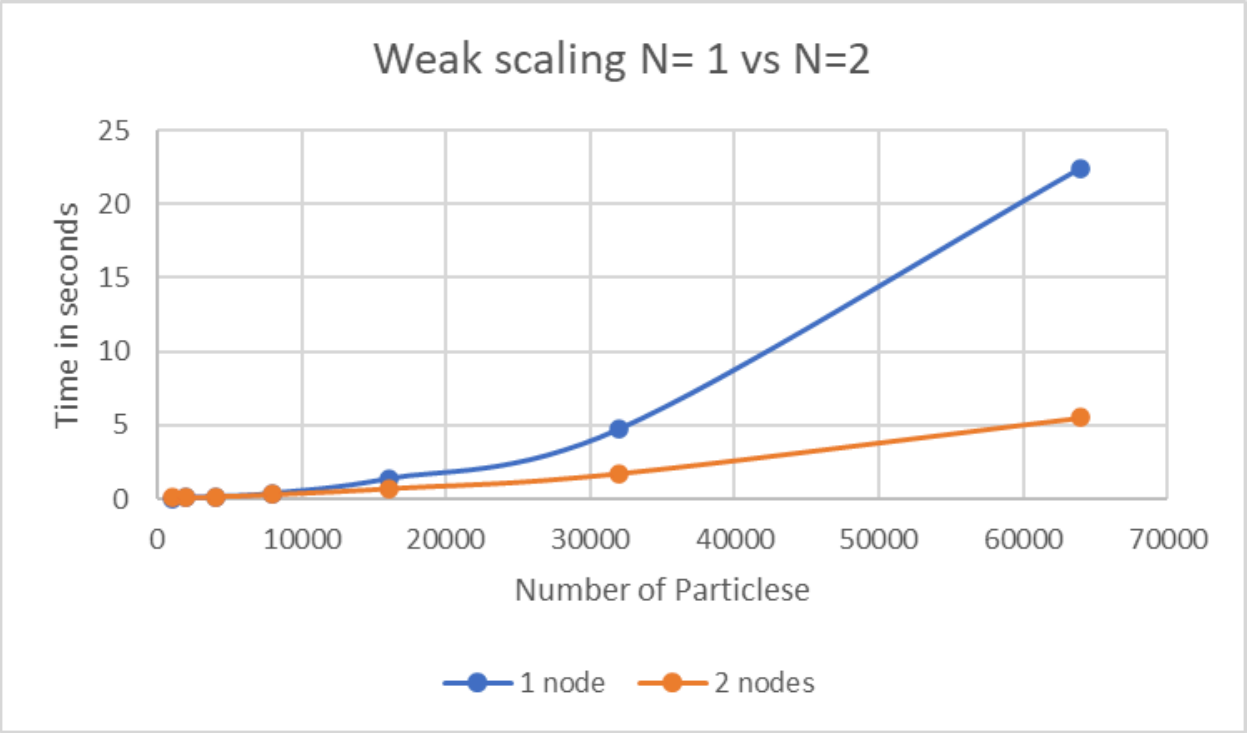

Figure 6. Timing vs Number of Particles Results for One vs Two Nodes

For weak scaling we used varying numbers of particles for one node and two nodes. For this trial we set the number of tasks per node to 64. The weak scaling for particle numbers less than 16000 is not good, however at and above that point the weak scaling is very good. As the number of particles doubles (i.e. from 32000 to 64000) we would assume that double the number of nodes being used would result in a similar runtime. We do see this in our results (when using big enough numbers of particles).

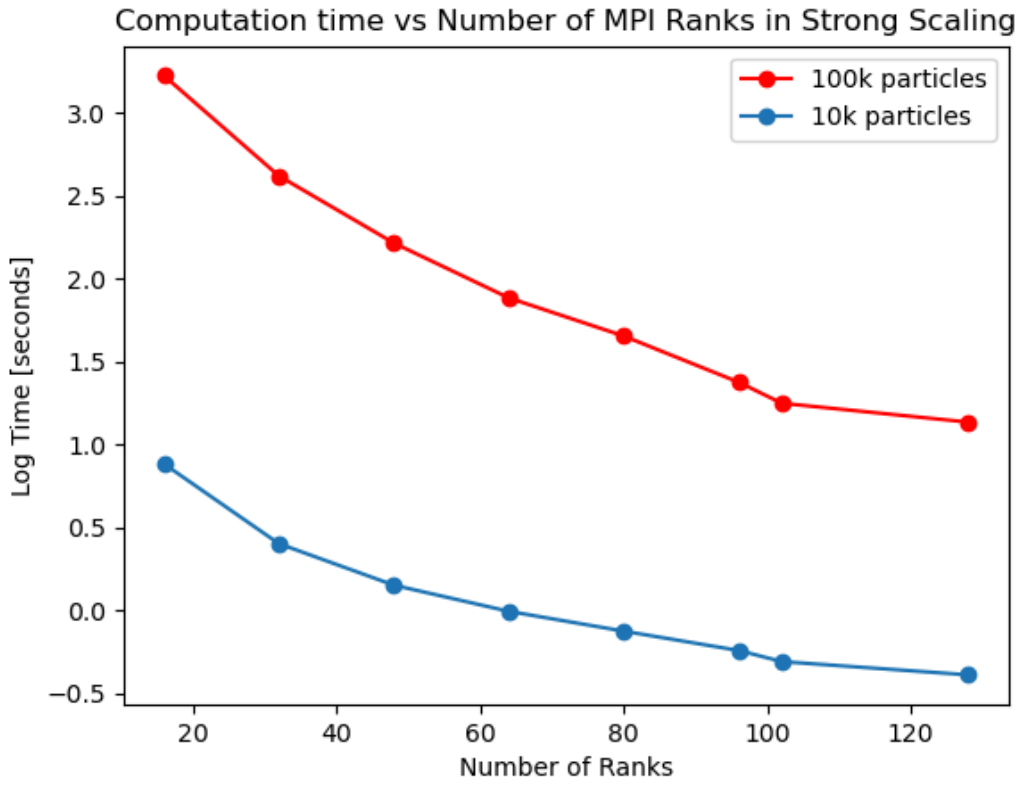

Figure 7. Log(Timing) vs Number of MPI Ranks Results for Varying Particle Numbers

For the strong scaling we tried to increase the number of MPI ranks from 16 to 128 (with an increase of +8 MPI ranks for each test), keeping the size of the system constant. We did it with a system of 100000 particles and another of 10000 particles, using two nodes in both cases (N =2). The results are similar in both systems: the strong scaling shows that our code can drastically reduce the computational time of the simulation when we increase the number of total ranks until ~100; above that number, the computational time is clearly unaffected by a further increase in the number of ranks.

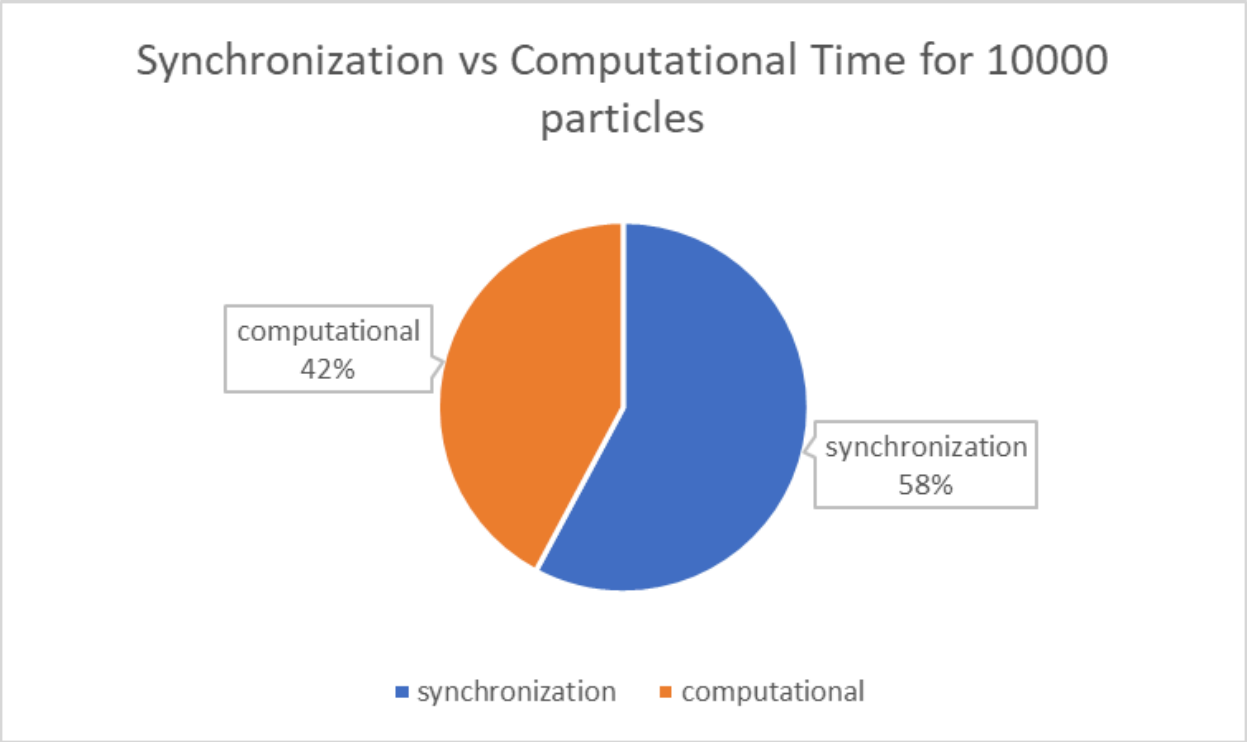

Figure 8. Breakdown of Computational vs Synchronization Time

We also measured the computational and synchronization timings for 10000 particles. We found that a large proportion of the runtime was used for synchronization compared to just computation. This makes sense as the MPI implementation relies heavily on communication and the send and receive commands. However as the number of particles increases the relative cost of the communication will decrease.

```
void simulate_one_step(particle_t* parts, int num_parts, double size, int rank, int num_procs) { // giovino

    comp_start_time = MPI_Wtime();
    double row_width = size / num_procs;
    for( particle_t& particle : actual_parts)   // try diff for loop
    {
        for (particle_t& neighbor : actual_parts)
        {
            apply_force(particle, neighbor);
        }
        for (particle_t& neighbor : ghost_parts)
        {
            apply_force(particle, neighbor);
        }
    }
    comp_end_time = MPI_Wtime();
    local_comp_time = comp_end_time - comp_start_time;
```

Figure 9. Example code for timing

This is an example of the code used to time the computational portion of the code, a similar method is used for the synchronization time.

Future Improvements

We are utilizing a 1D layout where each processor_i represents all the “bins” that are present in row_i. Although this is a faster and efficient way of representing MPI implementation, it can be shown that mathematically this takes much longer for each processor to compute the force in a given row. Thus, applying a 2D layout could have improved our speed up.