Particle Simulator

Nikola Gostovikj
University of Primorska
Koper, Slovenia

ABSTRACT

This document is a collection of concepts, ideas, and code that I have written while working on the Particle Simulator project in Java. The project can be run in three different modes: sequential, parallel, and distributed. The first part of this document focuses on the sequential part, followed by the parallel and distributed modes. I chose this project because of my interest in Mathematics and Physics, the physics being the particles and the branch of Linear Algebra.
"Ars Longa, Vita Brevis" - is the quote which gave me motivation and is a great encapsulation of my work. I wish to further improve this project and make it an optimized and efficient project.

1 INTRODUCTION

The goal of this project is to simulate a system of n particles, each having a positive or negative charge. The particles move in a 2D plane within a bounded rectangle, interacting with each other based on the forces generated by their charges and distances. The rectangle has its own charge, repelling the particles and keeping them within bounds. The simulation involves the particles moving according to the forces between them and their initial velocities. At the beginning of the simulation, the particles are assigned random positions and speeds.

2 BASIC RULES

The force F between two particles i and j is calculated as:

$$F = \frac{|c_i \cdot c_j|}{d^2}$$

where:

- c_i and c_j are the charges of particles i and j,
- d is the Euclidean distance between them.

The interaction between particles depends on the sign of their charges:

- Particles with the same charge attract each other.
- Particles with opposite charges repel each other.

Additionally, the boundary rectangle has a charge that repels the particles, ensuring they remain within the plane.

The program has a Menu class, which will be used to start the simulation. For the 3 versions we have a choice of running the simulator with visuals on or off. If we want to measure pure performance of our simulator, we turn off the visuals.

3 BACKGROUND

More information on the problem can be found in Chapter 11.2.1 of the book Foundations of Multithreaded, Parallel, and Distributed Programming by Gregory R. Andrews (available in the university library). Note that this problem is a variation of the problem described in the book. This is a classic N-body problem, where instead of simulating objects with forces of gravity, here we simulate using the electrostatic force. There

4 PARTICLE OBJECT

To ensure proper encapsulation of the project in my codebase, I made a class "Particle", which ensures that every particle is it's own object.
Every particle has it's own random charge, either negative or positive. To ensure proper values and calculations for the particles, so they don't behave erratic and too fast, I've limited the charge to be from -5 to 5 (I've also added 1 to the equation so there is no 0 value charge).
The particles have a value called radius, which is set to 50. This means that particles will collide with other particles, if they are in that given radius. The reasoning behind this decision, is that if particles are over this range, the force that is in their velocity vector is insignificant in the global calculation for every particle.
The position of the particles depends on the initial random positions that they are given when generating the particles. At first the particles have positions from the original size of the frame of 800 width * 600 height.

```java
public class Particle{
    double x;
    double y;
    public double radius;
    int id;
    public double charge;
    private double dx;
    private double dy;
    double mass;
```

Listing 1: Properties of Particle class

Red colored particles will have a negative charge and blue colored will be positively charged. This is for simulation purposes.

4.1 Simulation parameters

There are 2 important parameters for this simulator, the number of particles, represented by:

```java
int n;
```

and the number of cycles by:

```java
int cycles;
```

where one cycle is defined as all collision of every particle in one run.

```java
public static ArrayList<Particle> generate(int count) {
    ArrayList<Particle> particles = new ArrayList<>();
    Random r = new Random();

    for (int i = 0; i < count; i++) {
        int x = r.nextInt(750);
        int y = r.nextInt(550);
        double radius = 6;
        int charge = r.nextInt(-5, 6) + 1;
        Color color = charge >= 0 ? Color.BLUE : Color.RED;
        particles.add(new Particle(i, x, y, 3.0, radius, color, charge));
    }
    return particles;
}
```

Listing 2: Generate function to create particles

5 FORCE AND POSITION CALCULATION AND SEQUENTIAL VERSION

One of the most important parts of our simulator is how do we calculate forces, and how do we move the objects realistically. We want to create a proper realistic physics engine, which will have proper vector calculations, proper repulsion or attraction and proper collision overlay handling.
Each particle p_i has:

- position (x_i, y_i),
- velocity (v_{x_i}, v_{y_i}),
- charge c_i,
- radius r_i,
- mass $m_i = \pi r_i^2$.

Motion Update and Wall Collision

First, the position is updated as:

$$x_i \leftarrow x_i + v_{x_i}, \quad y_i \leftarrow y_i + v_{y_i}$$

Then, the boundary conditions are applied:

$$\text{If } x_i - r_i \le 0 \text{ or } x_i + r_i \ge W: \quad v_{x_i} \leftarrow -0.8 \cdot v_{x_i}$$

$$\text{If } y_i - r_i \le 0 \text{ or } y_i + r_i \ge H: \quad v_{y_i} \leftarrow -0.8 \cdot v_{y_i}$$

where W and H are the width and height of the simulation window.

Electrostatic Force Calculation

For each pair of particles (p_i, p_j), if the distance is within the influence radius:

$$d_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + \varepsilon$$

$$\vec{F}_{ij} = k \cdot \frac{c_i c_j}{d_{ij}^3} \cdot \begin{bmatrix} x_i - x_j \\ y_i - y_j \end{bmatrix}$$

with:

- $k = 200$ (Coulomb constant scaled for simulation),
- $\varepsilon = 2$ (softening to prevent infinite forces),
- $d_{ij} = \sqrt{d_{ij}^2}$.

Then, update velocities using Newton's second law:

$$\vec{v}_i \leftarrow \vec{v}_i + \frac{\vec{F}_{ij}}{m_i}, \quad \vec{v}_j \leftarrow \vec{v}_j - \frac{\vec{F}_{ij}}{m_j}$$

Collision Response

If the particles overlap:

$$d_{ij} < r_i + r_j$$

We resolve overlap by pushing them apart equally along the normal vector:

$$\vec{n} = \frac{\vec{p}_j - \vec{p}_i}{\|\vec{p}_j - \vec{p}_i\|}, \quad \text{overlap} = (r_i + r_j - d_{ij})$$

Update positions:

$$\vec{p}_i \leftarrow \vec{p}_i - \frac{\text{overlap}}{2} \cdot \vec{n}, \quad \vec{p}_j \leftarrow \vec{p}_j + \frac{\text{overlap}}{2} \cdot \vec{n}$$

Then, compute the relative velocity and impulse:

$$\vec{v}_{\text{rel}} = \vec{v}_j - \vec{v}_i$$

$$\text{impulse} = \frac{-(1+e)(\vec{v}_{\text{rel}} \cdot \vec{n})}{\frac{1}{m_i} + \frac{1}{m_j}}, \quad e = 0.8$$

Apply impulse:

$$\vec{v}_i \leftarrow \vec{v}_i - \frac{\text{impulse}}{m_i} \cdot \vec{n}, \quad \vec{v}_j \leftarrow \vec{v}_j + \frac{\text{impulse}}{m_j} \cdot \vec{n}$$

5.1 Sequential Java implementation

The pseudocode is provided for the sequential implementation of this algorithm. Mind you the worst case analysis is

$$T(n) = O(n^2)$$

since we calculate every particle with every other particle. There is a more efficient way of calculating the N-body problem, using the Barnes-Hut method, but for this simulation, the For-All algorithm is provided.

Algorithm 1 Update particle positions and handle collisions

procedure UPDATEPOSITION(particles)
  for all particle p in particles **do**
    $newX \leftarrow p.x + p.dx$
    $newY \leftarrow p.y + p.dy$
    if $newX - p.radius \leq 0$ or $newX + p.radius \geq$ width **then**
      $p.dx \leftarrow -0.8 \cdot p.dx$
      Clamp $newX$ within bounds
    if $newY - p.radius \leq 0$ or $newY + p.radius \geq$ height **then**
      $p.dy \leftarrow -0.8 \cdot p.dy$
      Clamp $newY$ within bounds
    $p.x \leftarrow newX; \quad p.y \leftarrow newY$
  for $i \leftarrow 0$ to $particles.size - 1$ **do**
    for $j \leftarrow i + 1$ to $particles.size - 1$ **do**
      $p_1 \leftarrow particles[i], \quad p_2 \leftarrow particles[j]$
      $dx \leftarrow p_1.x - p_2.x, \quad dy \leftarrow p_1.y - p_2.y$
      $dist^2 \leftarrow dx^2 + dy^2 + \varepsilon, \quad \varepsilon = 2$
      $dist \leftarrow \sqrt{dist^2}$
      if $dist < 200$ **then** // Skip distant particles
        $invDist^3 \leftarrow 1/(dist^2 \cdot dist)$
        $f \leftarrow k \cdot p_1.charge \cdot p_2.charge \cdot invDist^3$
        $(ux, uy) \leftarrow (dx/dist, dy/dist)$
        $(fx, fy) \leftarrow f \cdot (ux, uy)$
        Update velocities of p_1, p_2 using force and mass
        if $dist < p_1.radius + p_2.radius$ **then**
          Resolve overlap by shifting along (ux, uy)
          Compute relative velocity along normal
          if particles are moving toward each other **then**
            Apply impulse to both particles

6 PARALLEL VERSION

To solve the N-body problem efficiently, I implemented a shared-memory parallel algorithm using the Java `ExecutorService` and a fixed thread pool. The core idea is to divide the particles evenly among the available CPU threads using a technique called **striped partitioning**.

Let:

- N be the total number of particles,
- T be the number of available threads (based on processor count),
- $C = \lceil \frac{N}{T} \rceil$ be the chunk size per thread.

Each thread $t \in \{0, 1, \ldots, T-1\}$ processes particles from index:

$$\text{start}_t = t \cdot C, \quad \text{end}_t = \min((t+1) \cdot C, N)$$

Phase 1: Force Calculation (No Locks)

Each thread independently computes the Coulomb force acting on its assigned particles. To avoid race conditions, I allocate a 2D array of local forces:

$$F^x[t][i], \quad F^y[t][i]$$

This way, each thread accumulates partial forces in its own array without touching shared memory.

The total force on particle i is computed after all threads finish:

$$F_i^x = \sum_{t=0}^{T-1} F_x[t][i], \quad F_i^y = \sum_{t=0}^{T-1} F_y[t][i]$$

Then, each particle's velocity and position is updated:

$$v_{x_i} \leftarrow v_{x_i} + \frac{F_i^x}{m_i}, \quad x_i \leftarrow x_i + v_{x_i}$$

Wall collision handling is also performed in this step, by checking if the particle crosses simulation bounds and reflecting its velocity with damping if necessary.

Phase 2: Collision Resolution (Synchronized)

Once the positions are updated, particles may overlap. Each thread again works on a chunk of particles and compares them to all others. If an overlap is detected:

$$\text{distance}(i, j) < r_i + r_j$$

Then a thread-safe resolution is required. To do this, I apply a technique called **ordered locking**. For each overlapping pair (i, j), we acquire locks in a fixed order based on particle IDs:

$$\texttt{lock}(\min(i, j)) \rightarrow \texttt{lock}(\max(i, j))$$

This guarantees that no two threads will deadlock while trying to update shared positions or velocities.

Benefits and Performance

This approach ensures:

- Work is evenly distributed among threads.
- Force calculations avoid shared data writes and synchronization.
- Collision resolution is thread-safe via minimal locking.

To show and test the 2 versions, I used a AMD Ryzen 7 7730U with Radeon Graphics, 2000 Mhz, 8 Core(s), 16 Logical Processor(s) processor. The following simulation is 16 cores vs 1 core. Efficiency is measured as the speed up, divided by number of processors.

7 PARALLEL EFFICIENCY ANALYSIS

While the parallel version shows consistent improvement over the sequential version, it does not scale linearly with the number of cores. This is expected due to Amdahl's Law, which states that the speedup of a program is limited by its sequential parts. On a 16-core machine, we observed an average speedup of approximately 2×. This is a result of overheads such as thread synchronization, false sharing, cache contention, and locking during collision resolution. Additionally, hyperthreading may inflate logical core count, but physical limitations remain.

8 MEMORY EFFICIENCY AND OPTIMIZATION

In the parallel version, each thread maintains its own local force matrix to avoid race conditions. While this ensures correctness, it increases memory usage to $O(T \cdot N)$. Some improvements may be:

- Use a shared force matrix with atomic operations.
- Apply symmetry ($F_{ij} = -F_{ji}$) to reduce total computations by half.

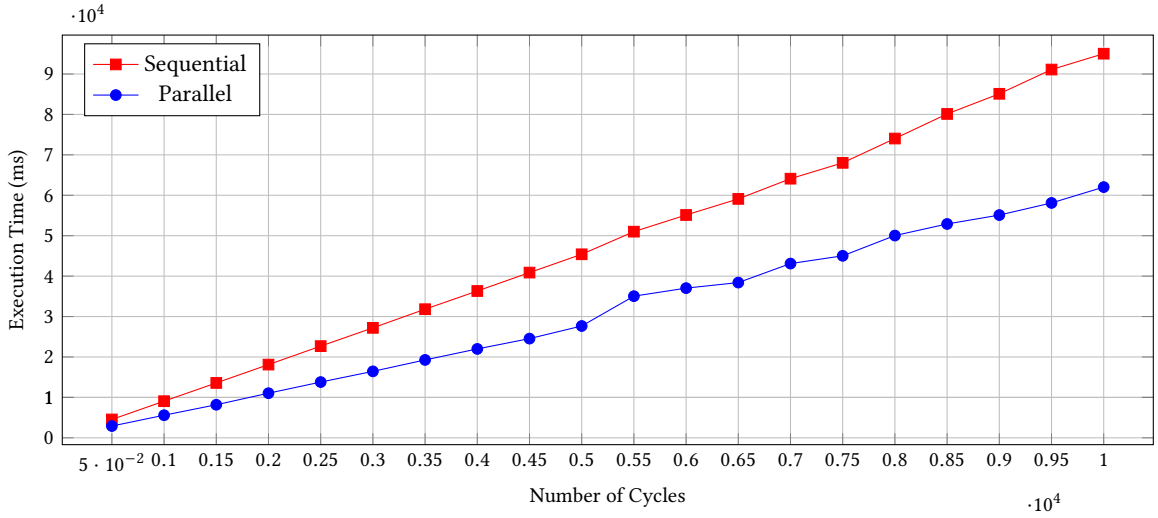

Figure 1: Execution time comparison of sequential and parallel implementations for increasing cycle counts.

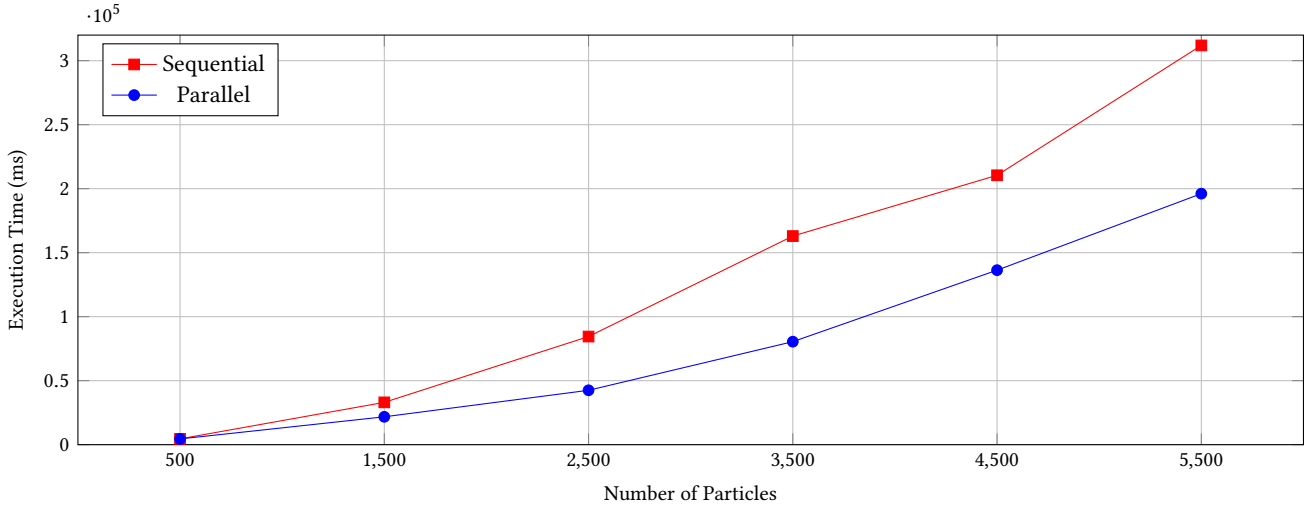

Figure 2: Execution time comparison of sequential and parallel implementations as a function of the number of particles (10 000 cycles).

- Use spatial partitioning (e.g., uniform grid or quadtree) to limit pairwise checks.

9 DISTRIBUTED VERSION

To extend the simulator across multiple processors in a distributed memory environment, I implemented a full MPI-based version using the MPJ Express library. The simulation logic is encapsulated in the `DistributedGui` class.

Architecture Overview

In the distributed implementation, the simulation is executed across multiple MPI processes. Each process:

- Receives the full list of particles using `MPI_Bcast` from the root (rank 0),
- Computes updates for a subset of particles assigned to it,
- Shares updates with other processes using `MPI_Allgatherv`,
- Renders the simulation only on rank 0 (if enabled).

To facilitate fast communication, all particle information is packed into a flattened `double[]` array containing six fields:

$$[x, y, dx, dy, \text{mass}, \text{charge}]$$

This layout allows efficient broadcasting and gathering across processes and supports SIMD-friendly memory access.

Simulation Phases

Each simulation cycle includes the following phases:

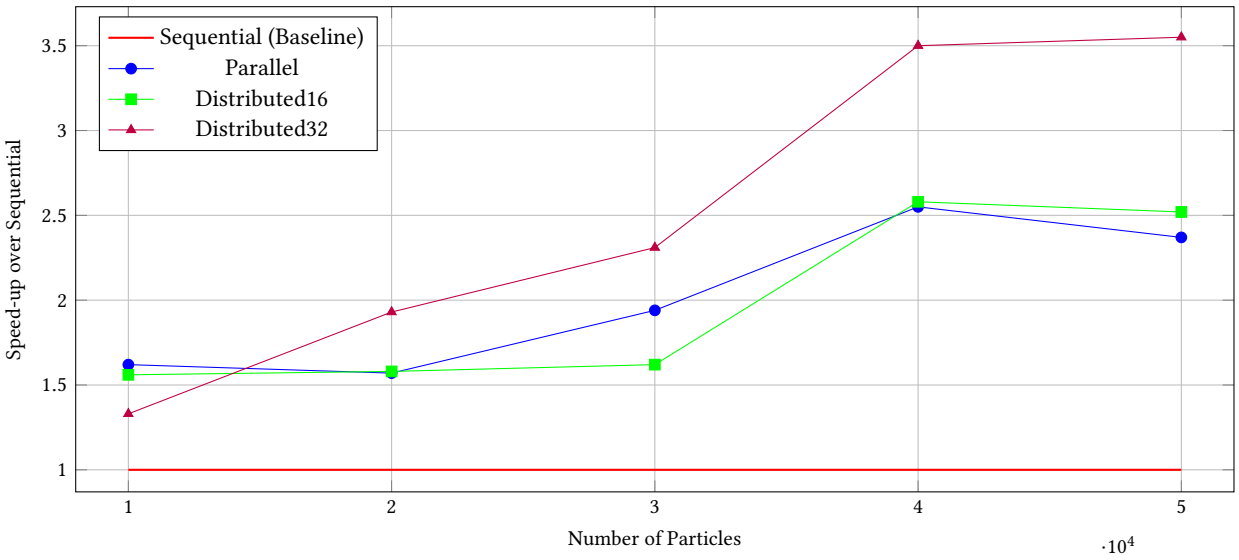

Figure 3: Speed-up of parallel and distributed implementations compared to the sequential baseline over 500 cycles.

1. Broadcast and Initialization. The root process (rank 0) initializes the particle list and broadcasts the flattened particle array `flatCurrent` to all MPI ranks.

2. Force and Collision Computation. Each rank computes electrostatic forces and elastic collisions for its assigned subset of particles using the For-All (All-Pairs) algorithm. This includes:

- Coulomb force: $F = \frac{k \cdot q_i \cdot q_j}{(d^2+\varepsilon)^{3/2}}$
- Collision response for particles within distance < 12 using overlap resolution and reflection based on a restitution coefficient.

3. Position and Wall Update. Each particle is updated using velocity integration. Wall collisions are resolved with reflective damping:

- $dx = -dx \cdot 0.8$ when crossing horizontal bounds
- $dy = -dy \cdot 0.8$ when crossing vertical bounds

4. State Synchronization with Allgatherv. The updated local particle data is gathered from all processes using `MPI_Allgatherv`, forming the new global `flatCurrent` array for the next cycle.

9.1 Testing Distributed with Sequential

I previously tested the parallel with sequential, where we have a fixed size of 3000 particles and increasing cycles by 500, and a test where we have 10 000 cycles and increase particles by 1000. The following graph of the speed up is below.

The result of this graph, shows that the sequential work is faster than distributed work on 8 processors, while it is slower than distributed work on 16 processors. The speed up for the distributed work will increase as we increase the number of particles and the number of cycles. This will increase until we reach a breaking point, where it will eventually stay at a steady speed up.
Below there are 2 graphs for 2 given tests, the first one, we show the speed up compared to the sequential version. The speed up is on the y-axis, while the number of cycles on the x-axis. I tested the distributed version using 16 processes and 8 processes separately. The second graph is the graph where we test it using fixed cycles and increasing particles as I have mentioned.

Table 1: Execution times (in ms) for Sequential, Distributed (16), and Distributed (8) with Increasing Cycles by 500 and Particles fixed at 3000 and the time is measured in ms

Cycles	Sequential	Distributed (16)	Distributed (8)
3500	31817	23745	31817
4000	36319	26647	36319
4500	40878	29001	40878
5000	45401	32972	45401
5500	51102	35243	47053
6000	55000	37671	53000
6500	59718	38021	58034
7000	64124	43397	62012
7500	68867	46418	65901
8000	74035	50023	70012
8500	80313	52836	79102
9000	85123	55209	82012
9500	91002	57265	88023
10000	95302	58467	89643

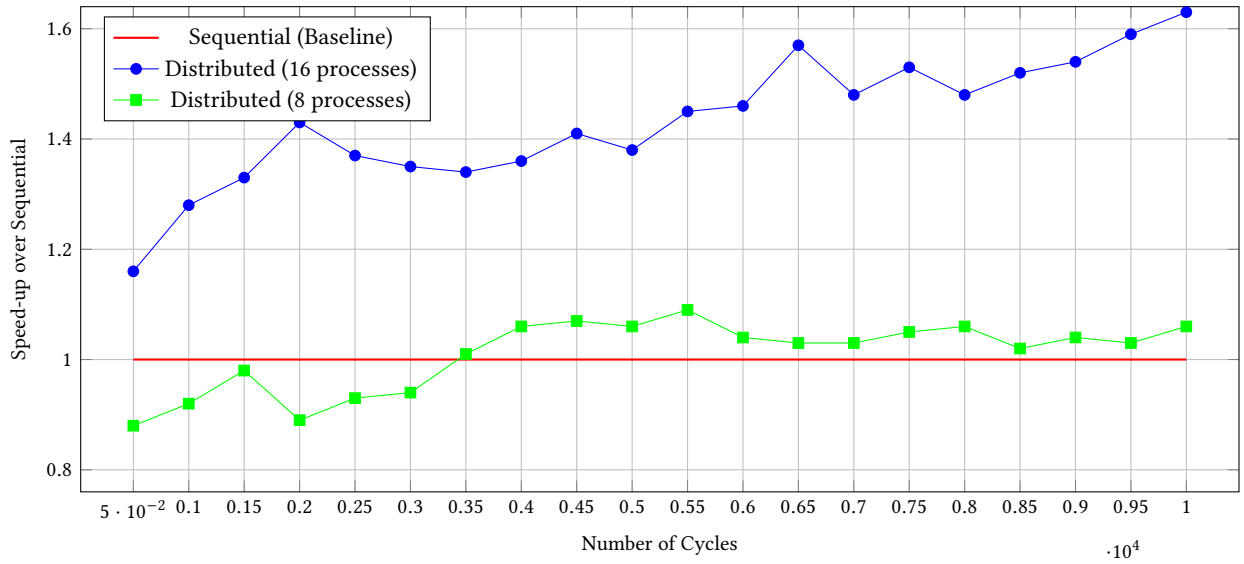

Figure 4: Speed-up of Distributed implementations over Sequential baseline for 3000 particles and increasing cycles.

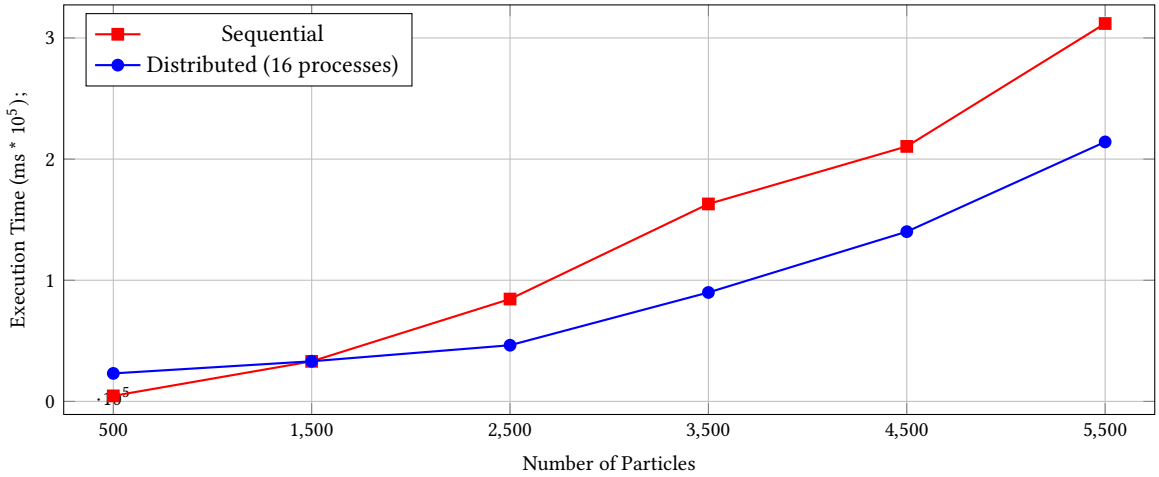

Figure 5: Execution time comparison of sequential and distributed implementations for increasing particle counts with fixed 10000 cycles.

10 CONCLUSIONS AND FUTURE WORK

The N-body problem benefits significantly from parallel and distributed computation, but only when the simulation parameters (particles) are large enough to offset the overhead of thread and process communication. This trend is clearly demonstrated in Figure 3 and Figure 4, where the distributed and parallel versions outperform the sequential one only after certain thresholds are reached.

The current implementation uses the For-All algorithm, where every particle interacts with every other particle. While simple and accurate, it is computationally expensive with a time complexity of:

$$O(n^2)$$

A more efficient method would be to use the Barnes-Hut algorithm, which approximates distant interactions to reduce the number of force calculations. This algorithm has a significantly better worst-case time complexity of:

$$O(n \log n)$$

For future work, I plan to implement Barnes-Hut spatial partitioning using quadtrees in 2D, and potentially extend the simulation into 3D. Another direction is improving distributed communication by reducing the number of synchronization points and optimizing memory layout for better cache coherence and lower communication overhead. If we have the parallel version of the Barnes-Hut and the distributed version of it also, speed ups will definitely will be even greater.

REFERENCES

[1] Alexander Brandt. *On Distributed Gravitational N-Body Simulations.* University of Western Ontario, 2022. Available at: https://arxiv.org/abs/2203.08966

[2] Abhyudaya Mourya. *N-Body Simulation using MPI.* CSE 633 – Parallel Algorithms, University at Buffalo, 2021. Mentor: Dr. Russ Miller.

[3] Tushaar Gangarapu, Himadri Pal, Pratyush Prakash, Suraj Hegde, and Dr. Geetha V. *The Parallelization and Optimization of the N-Body Problem using OpenMP and CUDA.* National Institute of Technology Karnataka, India, 2022.

[4] Wikipedia contributors. *N-body problem — Wikipedia, The Free Encyclopedia.* Available at: https://en.wikipedia.org/wiki/N-body_problem Accessed: August 3, 2025.