

UPPSALA UNIVERSITET



Assignment 3

*N-Body Problem in High Performance Programming course
Spring 2025*

Xiong LUO
Rasmus LILLRANK
Joakim JOHANSSON

1 Introduction

The gravitational N-body problem is a classical physics problem in which one computes the motions of particles under the influence of gravitational forces. The challenge lies in solving the equations of motion for all particles at each timestep, where the computational complexity grows rapidly with increasing N , the number of particles. This assignment focuses on solving the problem using Newton's law of gravitation and the symplectic Euler integration method.

2 The Problem

2.1 Governing Equations

The force between particles i and j in a 2D space is given by Newton's law of gravitation:

$$f_{ij} = G \frac{m_i m_j}{r_{ij}^3} r_{ij}$$

Where G is the gravitational constant, m_i and m_j are the particle masses, and r_{ij} is the distance vector between particles i and j . To prevent numerical instabilities, a modified formula is used:

$$f_{ij} = G \frac{m_i m_j}{(r_{ij} + \epsilon)^3} r_{ij}$$

Here, $\epsilon = 10^{-3}$ is a small number that smooths the force calculation.

The symplectic Euler integration method is used to update particle velocities and positions:

$$\begin{aligned} \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + a_i \Delta t \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1} \Delta t \end{aligned}$$

3 Implementation

3.1 Data Structures

The simulation uses arrays to store particle data such as positions, velocities, and masses. Each particle has a mass, position (x, y) , velocity (v_x, v_y) , and brightness which are stored in separate arrays in a Structure of Arrays (SoA) setup but when loading and saving the file we use a Array of Structures (AoS) approach because it's more readable.

3.2 Algorithm

The algorithm computes the gravitational force for every particle pair using the $O(N^2)$ method, updates the velocities and positions of all particles, and outputs the final state to a file.

To improve performance and reduce redundant calculations, we can take advantage of the fact that each force is split into the X-axis and Y-axis components. Since the force exerted by one particle on another has an equal and opposite counterpart (Newton’s Third Law), we can store the computed force and reuse it instead of recalculating it. By simply inverting the direction of the stored force, we eliminate redundant computations while maintaining numerical accuracy.

3.3 Code Example

Here’s a brief snippet of the main code implementing the force calculation:

Listing 1: Gravitational Force Calculation

```

1 for (int i = 0; i < N-1; i++) {
2     double accx = 0;
3     double accy = 0;
4     for (int j = i+1; j < N; j++) {
5         // Compute force between particle i and j
6         double dx = particles[i].x - particles[j].x;
7         double dy = particles[i].y - particles[j].y;
8         double r2 = dx * dx + dy * dy + epsilon;
9         double r = sqrt(r2);
10        double force = G * particles[i].mass *
11                particles[j].mass / (r2 * r);
12        // Update velocities for j particles
13        velX[j] -= dt * force * dx / mass[j];
14        velY[j] -= dt * force * dy / mass[j];
15        accx += dt * force * dx / mass[i];
16        accy += dt * force * dy / mass[i];
17    }
18    // Update velocity for i particle
19    velX[i] += dt * accx;
20    velY[i] += dt * accy;
21 }
```

4 Performance and Optimization

We tested the performance of the algorithm by varying N and measuring the execution time. The results confirm the expected $O(N^2)$ complexity. Below is a graph showing the relationship between the number of particles and the execution time.

Number of Particles	Execution Time (s)
10	0.000025
30	0.000230
50	0.000382
70	0.000828
100	0.001329
200	0.009035
300	0.015859
500	0.040732
700	0.082559
1000	0.193930
2000	0.791252
3000	1.903994
4000	2.270771
5000	3.459048
6000	5.176397
7000	7.733947
8000	9.877742
9000	13.564442
10000	19.079474

Table 1: Execution Time vs. Number of Particles

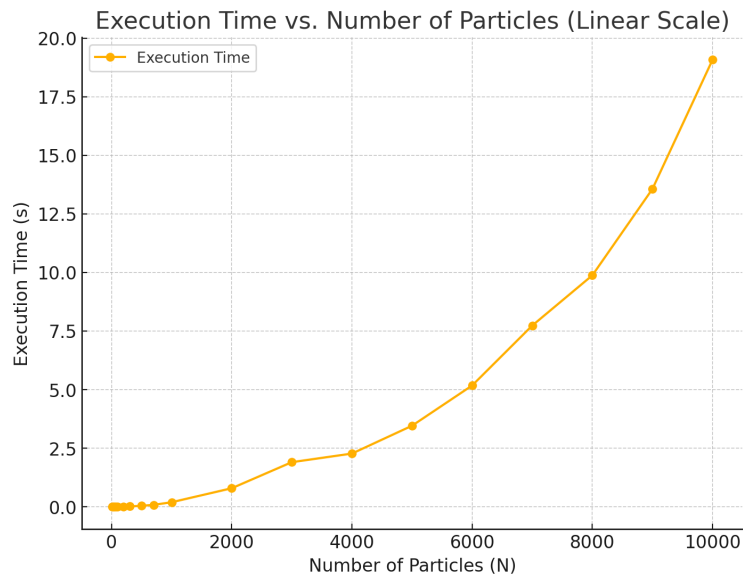


Figure 1: Execution Time vs. Number of Particles

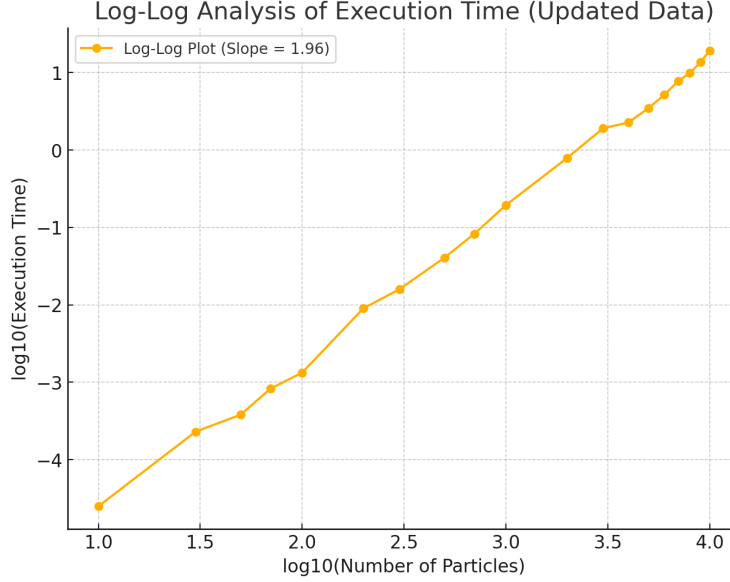


Figure 2: Execution Time vs. Number of Particles(log-log)

The data follows a nearly linear trend on the log-log scale with a slope of nearly 2, suggesting a polynomial relationship between execution time and number of particles. This is in line with the expected growth rate of $O(N^2)$.

4.1 Optimizations

We also applied compiler optimizations such as ‘-O3’ to improve performance, reducing the runtime significantly.

During some initial testing using the `ellipse_N_03000.gal` file using the -O3 flag which ran best a wall-time of 2.738719.

Within our initial naive implementation we broke out several steps as individual functions to keep the program readable during development. By combining these functions into one function that runs the entire simulation at each step we instead got a time of 2.675374.

Another thing we did in our naive implementation is to use a struct for our bodies where the values for each property of each body. This made it easy to follow the code but it was not possible for the compiler to auto-vectorize the loops used to calculate the next state.

To introduce auto-vectorization we create individual arrays of doubles for every property besides the brightness since it’s not used, saving us roughly 14% of data, going from Array of Structures to Structures of Arrays. We also changed some other minor things in the code and added the flags ‘-fast-math’ and ‘-march=native’ which allowed GCC to auto-vectorize our code and brought our time down to 1.762600.

We initially calculated the force of each particle on every other particle but we were able

to modify the loop to add the reverse force so that we only have to loop over half of the particles. This cut down the execution time to 1.132983.

We also tried implementing cache blocking but in the naive approach that creates four for loops which cannot be auto-vectorized, since GCC only supports one nested for loop, this lead to worse performance, probably because of the added overhead and the fact that 3000 bodies should fit into L1 data cache. Each body requires $5 \times 8 = 40$ bytes and we have 3000 bodies, giving us $3000 \times 40 = 120KB$ which is less than our 128*KiB* L1 data cache.

5 Conclusion

This assignment involved solving the N-body problem using a straightforward $O(N^2)$ algorithm. We successfully implemented the solution and optimized the code using compiler flags, auto-vectorization and algorithmic improvements. The performance of the algorithm was measured, and the results confirm the expected complexity. Future improvements could include exploring more sophisticated algorithms for better scalability.

6 References

1. Wikipedia. "N-body problem". Retrieved from https://en.wikipedia.org/wiki/N-body_problem.
2. Other relevant references.