# High-Performance Programming: Assignment 3 Report

Edvin Bruce, Marcus Nyfjäll, Viktor Kangasniemi

23rd February 2025

## 1 N-Body Problem

The N-body problem is about predicting the motion of $N$ interacting particles under mutual gravitational forces. In this assignment, the task at hand was specified as to *write a program that solves the given equations of motion for a galaxy with the given initial conditions.*

## 2 The Solution

For the solution of this problem the set of data structures seen in Table 1 was used. The listing 1 shows a for loop which performs the calculations behind the particle-simulation. The function `calculate_forces_over_mass()`, listing 2, calculates the $\mathbf{a}_n^i$ variable (as defined in the assignment instructions) for each particle, utilising Newtons third law, and stores them in a preallocated buffer for memory efficiency. This function has time complexity $O(n^2)$ as it contains a nested for loop where both loops iterate over $N$-particles.

After the buffer has had its values set, the function `update_particles()`, listing 3, utilises the buffer of $\mathbf{a}_n^i$-variables to update the position and velocities of all particles. This function has time complexity $O(n)$ as it simply iterates over all the particles once. Thus the time complexity of these two functions is bounded by $O(n^2 + n)$, which simplifies to $O(n^2)$.

When we measure the execution time of the program, this part of the program is what is actually being measured (and not the time from process creation to termination, or something else).

| **Name** | **Type** | **Description** |
|---|---|---|
| `N` | `int` | Number of particles |
| `P_pos_x`, `P_pos_y` | `double*` | Positions of particles |
| `P_vel_x`, `P_vel_y` | `double*` | Velocities of particles |
| `P_mass` | `double*` | Masses of particles |
| `P_brightness` | `double*` | Brightness values (for rendering) |
| `force_buf` | `double*` | Stores computed force/mass values |
| `buffer` | `double*` | Temporary array for file reading |

Table 1: Key Data Structures in the N-Body Simulation

## 3 Performance and Discussion

In this section, we will go over what optimisations were made and what effect they had on the execution time of the program using various configurations (i.e. running with different

input-files and parameters).

### 3.1 Execution Time Analysis

| file | $N$ | *nsteps* | $\Delta t$ | comment | **mean wall time (s)** |
|---|---|---|---|---|---|
| ellipse_N_03000.gal | 1000 | 100 | 0.00001 | with AoS implemented | 0.736966 |
| ellipse_N_03000.gal | 3000 | 100 | 0.00001 | with AoS implemented | 6.652267781 |
| ellipse_N_10000.gal | 10000 | 100 | 0.00001 | with AoS implemented | 41.693164 |
| ellipse_N_03000.gal | 1000 | 100 | 0.00001 | with SoA implemented | 0.658110 |
| ellipse_N_03000.gal | 3000 | 100 | 0.00001 | with SoA implemented | 5.980954623 |
| ellipse_N_10000.gal | 10000 | 100 | 0.00001 | with SoA implemented | 37.646316 |

Table 2: Measured times for the *galsim* program, executed on the following CPU: *AMD Ryzen 7 5700U with Radeon Graphics*, and compiled with *gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0*. From this table, it is evident that the *SoA*-implementation (as described in Section 3.2 has a clear advantage over the initial *AoS*-implementation, which was expected.

All measured times shown in this report were run on an *Acer Aspire A315-44P* machine, using a *AMD Ryzen™ 7 5700U with Radeon™ Graphics × 16* CPU. The compiler used was *gcc (Ubuntu 13.3.0-6ubuntu2-24.04*, and the program was compiled using the following flags (if nothing else was specified):

- **-O3** This is a high-level optimisation flag that enables aggressive optimisations in GCC. It includes all optimisations from *-O2* and additional ones that may increase performance but could also lead to larger code size. Specifically, it enables optimisations such as function inlining, vectorization, and loop unrolling.
- **-funroll-loops** This flag instructs the compiler to unroll loops where it determines that doing so would improve performance. This flag is largely redundant if the *-O3* flag is already being used, but it provides even more aggressive unrolling if used in combination with *-O3* as compared to just using *-O3*.

### 3.2 Optimisation Techniques

We tested to inline both functions shown in Listing 1, which actually gave us worse performance on average which we did not expect (2% slower on average, not much but we still found it strange). When only compiling with the *-O3* flag and letting the compiler handle inlining on its own, the `calculate_forces_over_mass()` was not inlined, but `update_particles()` was. This was verified by also compiling with *-fopt-info-inline* to see exactly what function-calls were inlined by the compiler.

Another technique we utilised was to store the information regarding particles more efficient in memory (i.e. Array of Structure vs Structure of Arrays). Initially, each particle was represented by its own instance of a struct that held all its properties in its fields. This is suboptimal as in this simulation we need to access the position of all particles frequently, and for this it is better if the data is adjacent in memory in order to not get as many cache-misses when running the program. It would be better if all data regarding the particles position was adjacent in memory such that reading one cache line would imply fetching data on many particles (and not just one). For this purpose, we substituted the previously mentioned particle-struct representation and opted for storing all the data regarding the particles in 6 different arrays (one array for each property). Thus we ended up with one array for all particles x-positions, one array for their y-positions, one array for their masses (and so on). When we compared the performance of this implementation to the previous, we found that a performance gain of 10% was yielded on average. Table 2 shows measured times for a variety of configurations, comparing the *Array of Structures* implementation to the *Structure of Arrays* implementation. Additionally, incorporating Newton's third law into the `calculate_forces_over_mass()` function significantly enhanced performance, reducing the execution time by 50%. This is because for each reaction that a particle experiences an opposite reaction will affect another particle. This in short halves the amount of computations allowing for a faster execution time.

The optimisation technique that yielded the highest performance gain was simply by compiling with the *-O3* flag. As explained previously in section 3.1, the *-O3* flag performs some aggressive optimisations that can lead to increased size of the executable, and enables optimisations such as function inlining, vectorization, and loop unrolling. Table 3 shows measured times for a variety of configurations, comparing when the executable was compiled with the *-O3* flag and when it was not.

### 3.3 Complexity Confirmation

As seen in figure 1 the execution time with and without the O3 flag is directly correlated to $O(N^2)$ time complexity.

| file | $N$ | *nsteps* | $\Delta t$ | comment | **mean wall time (s)** |
|---|---|---|---|---|---|
| ellipse_N_00010.gal | 10 | 200 | 0.00001 | with 03 | 0.006849061 |
| ellipse_N_00010.gal | 10 | 200 | 0.00001 | without 03 | 0.007159936 |
| ellipse_N_00100.gal | 100 | 200 | 0.00001 | with 03 | 0.019940662 |
| ellipse_N_00100.gal | 100 | 200 | 0.00001 | without 03 | 0.049100119 |
| ellipse_N_00500.gal | 500 | 200 | 0.00001 | with 03 | 0.337506030 |
| ellipse_N_00500.gal | 500 | 200 | 0.00001 | without 03 | 1.047757877 |
| ellipse_N_01000.gal | 1000 | 200 | 0.00001 | with 03 | 1.334661014 |
| ellipse_N_01000.gal | 1000 | 200 | 0.00001 | without 03 | 4.120340623 |
| ellipse_N_02000.gal | 2000 | 200 | 0.00001 | with 03 | 5.331434915 |
| ellipse_N_02000.gal | 2000 | 200 | 0.00001 | without 03 | 10.917401354 |
| ellipse_N_03000.gal | 3000 | 100 | 0.00001 | with 03 | 4.824927238 |
| ellipse_N_03000.gal | 3000 | 100 | 0.00001 | without 03 | 9.122930265 |

Table 3: Measured execution times for a variety of configurations, comparing when the executable was compile with and without the *-O3* flag. For all the measured times above, the *SoA* implementation (as described in previously in this section) was used.

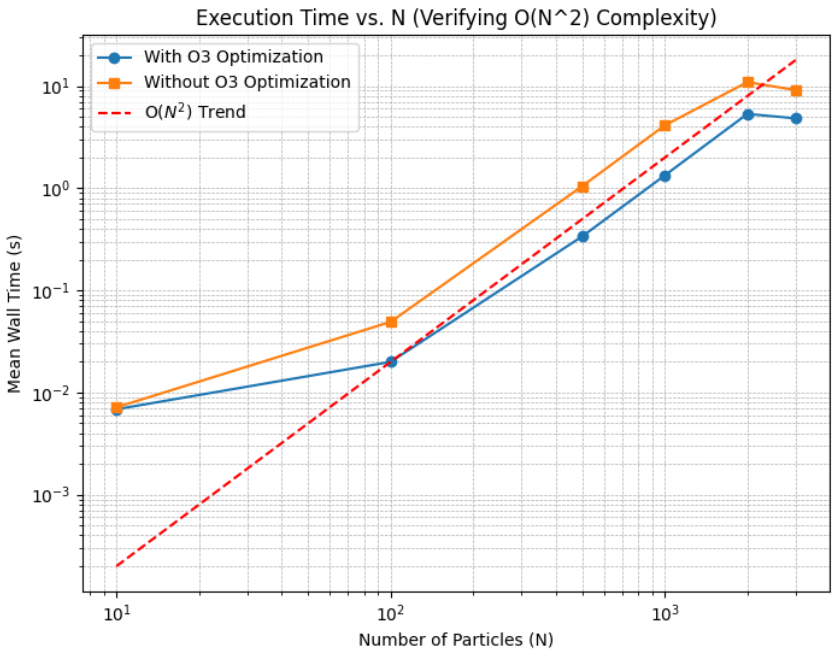


Figure 1: A graph showing execution time as a function of $N$ proving $O(N^2)$ complexity.

# 4 References

# References

# 5 Appendix

```
203 static inline void no_graphics_loop()
204 {
205     for (int i = 0; i < nsteps; i++)
206     {
207         calculate_forces_over_mass(N, force_buf); // Step 1: Compute accelerations
208         update_particles(force_buf, N, delta_t);  // Step 2: Update positions &
    velocities
209     }
210 }
```

Listing 1: The two functions that cover the "main logic" behind the simulation. This portion of the program is what is being timed (wall time).

```
158 static inline void calculate_forces_over_mass(int N, double *buf)
159 {
160     double G_over_N = 100.0 / N; // Given in the assignment
161     double epsilon0 = 1e-3;      // Plummer softening
162 
163     // reset for each computation so previous forces are not included
164     memset(buf, 0, 2 * N * sizeof(double));
165 
166     for (int i = 0; i < N; i++)
167     {
168         for (int j = i + 1; j < N; j++)
169         {
170             double dx = P_pos_x[j] - P_pos_x[i];
171             double dy = P_pos_y[j] - P_pos_y[i];
172             double r2 = dx * dx + dy * dy;
173             double r = sqrt(r2) + epsilon0;
174             double F = G_over_N / (r * r * r);
175 
176             double Fx = F * dx;
177             double Fy = F * dy;
178 
179             // Apply force to particle i
180             buf[2 * i] += Fx * P_mass[j];
181             buf[2 * i + 1] += Fy * P_mass[j];
182 
183             // Apply equal & opposite force to j
184             buf[2 * j] -= Fx * P_mass[i];
185             buf[2 * j + 1] -= Fy * P_mass[i];
186         }
187     }
188 }
```

Listing 2: The function calculate_force_over_mass

```
189 static inline void update_particles(double *restrict forces_over_mass, int N,
        double delta_t)
190 {
191     for (int i = 0; i < N; i++)
192     {
193         P_vel_x[i] += forces_over_mass[2 * i] * delta_t;
194         P_vel_y[i] += forces_over_mass[2 * i + 1] * delta_t;
195 
196         P_pos_x[i] += P_vel_x[i] * delta_t;
197         P_pos_y[i] += P_vel_y[i] * delta_t;
198     }
199 }
```

Listing 3: The function update_particles