lecture42

Uppsala University

High Performance Programming

1TD062 62013 VT2025

Assignment 3: The Gravitational N-body Problem

Author:
Abhinav Ramalingam

February 13, 2025

Contents

1 The Problem

The task is to simulate the motion of N particles in a gravitational system over tiny, discrete time steps. Each particle has a position, velocity, and mass, and their interactions follow Newton's law of gravitation. The initial state is read from a binary file, and for each time step, the net gravitational force on each particle is computed, leading to updates in acceleration, velocity, and position. The final state is written to an output file after a specified number of steps. The simulation runs in two dimensions and has a computational complexity of $O(N^2)$ due to pairwise force calculations.

2 The Solution

The core data structure used is the `struct Particle` data structure, which represents each body in the simulation. It includes:

- **Position:** `x, y` (spatial coordinates of the particle)
- **Mass:** `mass` (Constant throughout the simulation)
- **Velocity:** `vx, vy` (updates at each time step based on acceleration)
- **Brightness:** `brightness` (A property stored but not actively used in calculations)

The Code Structure is as follows:

- **Initialization:** Reads the initial particle data from a binary file (`read_input`).
- **Main Loop:** Iterates for a specified number of timesteps (`num_steps`), computing gravitational forces and updating each particle's motion.
- **Force Calculation:** The base algorithm uses an *O(N²)* approach where each particle interacts with all others (`gravitational_force_net`, `gravitational_force`).
- **Update Equations:** Applies Newton's laws to update acceleration, velocity, and position.
- **Final Output:** Writes the updated state of all particles to a binary file (`write_output`).

The program follows a Brute-Force Approach ($O(N^2)$, as indicated in figure 1) which computes forces by iterating over all particle pairs. This is simple and accurate but inefficient for large N. Alternative approaches like Barnes-Hut Algorithm ($O(NlogN)$) can reduce time complexity by approximating distant particle interactions using a quadtree, but requires additional data structures and implementation complexity. However, Rather than implementing an optimized solution upfront, we started with a simple brute-force algorithm to ensure correctness first. This allows for a step-by-step approach to optimization, making it easier to observe speed-up effects as different optimizations are introduced progressively.

3 Performance and Discussion

The base program was executed without any optimizations. The parameters used for this run were:

- **Number of particles (N)**: 3000
- **Number of time steps (nsteps)**: 100
- **Time step size (delta_t)**: 0.00001
- **Graphics flag**: 0 (no graphical output)

With these settings, the simulation ran for **234.748453** seconds.

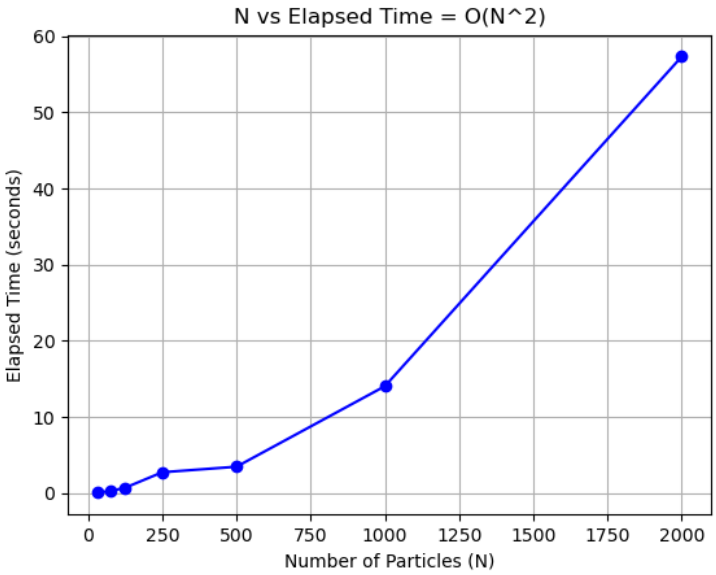

Figure 1: Base program performance graph

3.1 Optimization 1: Removing Function calls

A profiler known as `gprof` user used, and the observed results show that **72.56%** of the execution time is taken by `gravitational_force`, accounting for **29.30 seconds** and **22.39%** of the time is taken by `gravitational_force_net`, contributing **9.04 seconds**. Also, examining the call graph gives us an information that `gravitational_force_net` is called **300000** times, and `gravitational_force` is called **899700000** times. This shows that the runtime involved in a function call over a loop can get pretty long, since the overheads of copying the inputs and results from one location to another multiples based on the number of bodies. Since the program is a standalone one, removing those function calls (while still maintaining readability through proper comments, indentation, dividing into section, etc.) can give a speed-up in code execution.

After applying this optimization, the simulation ran for **210.823595** seconds.

3.2 Optimization 2: Replacing Expensive /Redundant Operations

Our program uses the `pow` operation to calculate squares and cubes. Since the `pow` operation is optimized to calculate `pow` over a large exponent, it can add unnecessary overhead for smaller exponents like 2 (square) and 3 (cube). So, we replace `pow(dx, 2) + pow(dy, 2)` with `dx*dx + dy*dy` and `pow((r + EPSILON_0), 3)` with `(r + EPSILON_0) * (r + EPSILON_0) * (r + EPSILON_0)`. Furthermore, `(r + EPSILON_0)` itself can be stored in a separate variable to avoid redundant calculations. In addition, the inverse denominator can be computed separately using `denom_inv = 1 / denominator`. We write `const double G_const = G(N)` to store the gravitational constant once, avoiding unnecessary recalculation of `G(N)` within each iteration. Furthermore, the mass inverse calculation `double inv_mass = 1.0 / particles[i].mass;` is pushed out of the loop, and `ax = net_fx * inv_mass;` and `ay = net_fy * inv_mass;` are calculated within the loop using the precomputed value of `inv_mass`. The expression `double Gm_i = -G_const * particles[i].mass;` is also moved out of the loop. This avoids recalculating the gravitational force for each particle within the nested loop.

After applying these optimizations, the simulation ran for **36.900264** seconds, which is a significant speed-up.

3.3 Optimization 3: Using Newton's Third Law

According to Newton's Third Law, the force exerted on particle j by particle i is equal in magnitude but opposite in direction to the force exerted on particle i by particle j. Leveraging this property, we can represent the forces as an i-j matrix and compute only the upper triangular portion. The corresponding lower triangular values can then be obtained by simply negating the computed forces, effectively reducing the number of force calculations by half. While this approach may lead to increased dynamic heap memory allocations, introducing some overhead, the resulting speed-up outweighs the additional cost in this scenario.

After applying this optimization, the simulation ran for **22.467780** seconds.

3.4 Final Optimization: Compiler flags

The following GCC compiler flags are used for optimization:

```
gcc -O3 -ftree-vectorize -march=native -funroll-loops -ffast-math
galsim.c -o galsim -lm
```

- `-O3` : Enables aggressive optimizations for better performance.
- `-ftree-vectorize` : Enables automatic loop vectorization to improve execution speed.
- `-march=native` : Optimizes the code for the specific CPU architecture of the compiling machine.
- `-funroll-loops` : Unrolls loops to enhance instruction-level parallelism.
- `-ffast-math` : Enables aggressive floating-point optimizations, allowing the compiler to break strict IEEE floating-point rules for better performance.

These flags collectively enhance execution speed by optimizing memory access, instruction scheduling, and floating-point operations.

After applying the above flags, the simulation ran for **11.143078** seconds.

References

[1] OpenAI, *ChatGPT*, `https://chat.openai.com`, accessed: 2025-02-13. (Used for automating trivial calculations and code reformatting, but not for code generation.)

[2] Uppsala University, *Lecture 4: Serial Code Optimization*, `https://uppsala.instructure.com/courses/95091/pages/lecture-notes?module_item_id=1285027`, accessed: 2025-02-13. (Used for concepts of Serial Optimization)