lecture42

Uppsala University

High Performance Programming

1TD062 62013 VT2025

Assignment 3: The Gravitational N-body Problem

Author:
Abhinav Ramalingam

March 7, 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

2.1 Data Structure

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)

2.2 Algorithm

Algorithm 1 N-Body Simulation

```
Input: Initial particle data from input file
Output: Updated particle data written to output file
Initialize particles from input file
for each timestep t from 1 to num_steps do
  for each particle i do
    Set net_force_i to zero
    for each particle j do
      if i ≠ j then
        Calculate gravitational force between particle i and j
        Add force to net_force_i
      end if
    end for
    Update particle i's acceleration, velocity, and position based on
    net_force_i
  end for
end for
Write updated particle data to output file
```

The solution is presented in Algorithm 1. 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.

2.3 Testing and Benchmarking the Solution

The mathematical correctness of our solution was tested with the given `compare_gal_files.c` program which takes in two gal files and a size value

as arguments and compares whether all the physical values (mass, velocities, positions, brightness) match between the two files. A max difference of 0.0000 indicated *SUCCESS*.

For measuring the performance of our program, we used the `clock()` function to get the current time at 4 instants - start of the program, after input file is copied into a Data Structure in the program, after the processing has been done for `timesteps` time intervals, and after the processed Data Structure is written back into an output file created called `results.gal`.

Thus, the results would be presented in a `[2 x (C + 1)]` dimension table, where `C` is the number of optimization steps applied (extra 1 is the initial run without optimizations), and `2` denotes - runtime without and with compiler flags. For each serial optimization attempted, `compare_gal_files.c` was tested on the program again, and it was ensured that the new changes comply with the old changes.

Here, only the middle processing part (between the 2nd and 3rd instance of recording clock value) is taken in account since it was observed that reading and writing contributes to very little runtime (in the order of 10^{-3} seconds).

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)

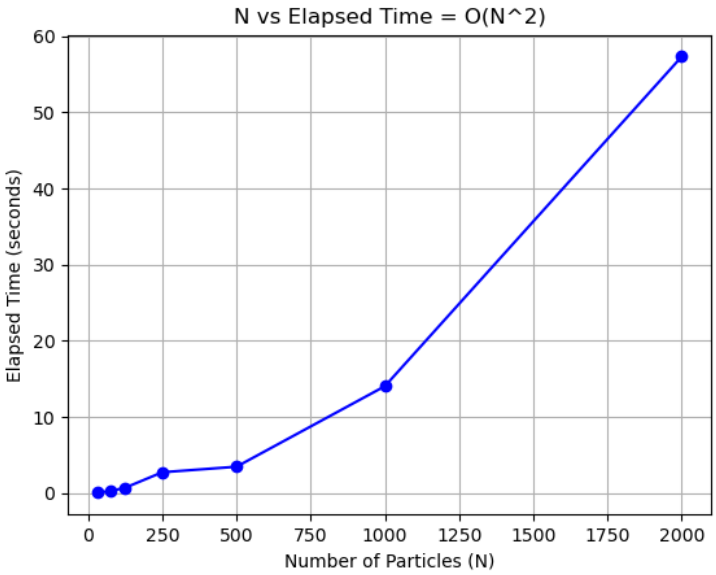

Figure 1: Base program performance graph

3.1 Optimization 1: Removing Function calls

A profiler known as `gprof` user used, and the observed results show that:

- **72.56%** time taken by `gravitational_force` – **29.30 seconds**
- **22.39%** time taken by `gravitational_force_net` – **9.04 seconds**

Also, examining the call graph gives us an information that:

- `gravitational_force_net` – called **300000** times
- `gravitational_force` – called **899700000** times

This shows that there is high overhead involved in a function call. Since the program is a stand-alone one, removing those function calls (while still maintaining readability through proper comments, indentation, dividing into section, etc.) gives a speed-up in code execution.

3.2 Optimization 2: Reducing Expensive Computations

Our program can be further optimized by reducing various computations or replacing them by their lighter counterparts. The followed optimizations were implemented:

1. Replacing the `pow` function call without multiplication.
2. Replacing (* 2) with $\ll 1$, (* 6) with $(\ll 1) + (\ll 2)$
3. Removing redundantly calculated values
4. Declaring unchanging variables as constant for efficiency
5. Moving Loop invariant computations outside loop, for e.g., moving i-dependent values outside j-loop.

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.

This approach leads to increased dynamic heap memory allocations because, when calculating Force-on-i-by-j, we need to remember Force-on-j-by-i too to avoid recomputation. This induces overhead due to the need of creating **auxilliary arrays**. The program still speed up, although not exactly by half.

Another benefit of this optimization is that after calculating all the j-loop values of a particular i-th particle, it is not necessary to revisit computation for that same particle as everything is already precomputed by then. Because of that, an extra loop for updating the values separately s not required.

3.4 Optimization 4: Mass cancellations

In the force computation, mass was multiplied and the same mass was divided while computing acceleration. This redundancy was removed. Along with this optimization, the part of acceleration computation which was loop invariant, was moved outside the loop.

3.5 Optimization 5: Structure-of-Arrays

Additional optimizations were applied to improve performance. First, the **Structure of Arrays (SoA)** format was used instead of the Pure Array format which was used before. This improves **cache utilization**.

Additionally in this optimization step, using auxiliary arrays (as in Optimization 3) were avoided by storing and directly updating the particle properties (such as velocity, etc.) in the same arrays during each timestep, rather than calculating and storing intermediate results in separate arrays.

3.6 Results comparison

In The following GCC compiler flags are used for optimization:

```
gcc -O3 -march=native -ffast-math -o galsim galsim.c -lm
```

- `-O3` : Enables aggressive optimizations for better performance.
- `-march=native` : Optimizes the code for the specific CPU architecture of the compiling machine.
- `-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.

As discussed in Section 2.3, Table 1 represents the results obtained after every optimization step.

Optimization	Runtime without Flags	Runtime with Flags
Initial Run	234.75	22.05
Optimization 1	210.82	21.90
Optimization 2	36.90	21.87
Optimization 3	26.52	10.97
Optimization 4	22.27	7.55
Optimization 5	19.55	5.60

Table 1: Optimization vs Runtime

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)