Uppsala Universitet

High Performance Programming

1TD062

Assignment 3

By:
Jacob K. Malmenstedt Ture
Hassler

February 17, 2023

1 Introduction

The N-body problem is a classic simulation task, where the gravitational forces from N different objects are calculated for each step, and from there the path of the objects can be simulated.

To compensate for the numerical instability when the radius goes to zero a factor ϵ_0 was added, giving the so called Plummer spheres. The acceleration and force on each object is given by eq. 1.

$$\boldsymbol{a_i} m_i = \boldsymbol{F_i} = -G m_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij} + \epsilon_0)^3} \boldsymbol{r_{ij}} \tag{1}$$

2 Solution

During the development of the solution, several different iterations and methods were tried, both different implementations and slightly different algorithms.

The data that needed to be saved for each particle was the position, mass, velocity and brightness, furthermore all values were given in doubles. This was implemented using the following struct:

```c
struct Particle
{
    double x;
    double y;
    double mass;
    double vx;
    double vy;
    double brightness;
};
```

To keep track of all the particles, an array was created that pointed to each individual particle's struct.

As the performance timings is run without graphics the code checks if the user wants to display graphics or not. If the user wants graphics the code runs a copy of the normal code with the difference that it also updates the graphics in each timestep. The reason we have a entire copy of the code for the graphics part of the code instead of for example just calling an update-function is that when the code is run in ”performance-mode” unnecessary calls to a separate update-function might slow down the program rather than just having the entire code written out. This might increase the size of the file but since we do not care about size and only time it increases performance slightly.

The actual part of the program that is used to update positions of all particles in the timed non graphics part of the code uses a rather simple algorithm (in pseudocode):

```
for all timesteps
    for all particles j
```

```
        set current particle to p1
        for all particles k=0, k<j
            calc. acceleration on p1 and save in a sum
        for all particles k=j+1, k<N
            calc. acceleration on p1 and save in sum
        update velocity using calculated acceleration
        reset acceleration sum to 0
    for all particles j=0, j<N
        update pos using velocity
```

Our implementation of this algorithm in code can be seen below:

```
for (int i = 0; i < nsteps; i++) // for all timesteps
{
    for (int j = 0; j < N; j++) // for all particles update acc and vel
    {

        p1 = particles[j]; // current particle
        for (k = 0; k < j; k++) // calculations of acc for all particles
    before current particle
        {

            p2 = particles[k];
            rij = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y -
     p2.y));
            acc_k = p2.mass / ((rij + e0) * (rij + e0) * (rij + e0));
            acc_x += acc_k * (p1.x - p2.x);
            acc_y += acc_k * (p1.y - p2.y);
        }

        for (k = j + 1; k < N; k++) // calc for all particles after current
    particle
        {
            p2 = particles[k];
            rij = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y -
     p2.y));
            acc_k = p2.mass / ((rij + e0) * (rij + e0) * (rij + e0));
            acc_x += acc_k * (p1.x - p2.x);
            acc_y += acc_k * (p1.y - p2.y);
        }
        acc_x *= -G;
        acc_y *= -G;

        // update velocity
        particles[j].vx += acc_x * delta_t;
        particles[j].vy += acc_y * delta_t;

        // reset acceleration
        acc_x = 0;
        acc_y = 0;
    }

    for (int j = 0; j < N; j++) // for all particles update pos
    {
        // update position
        particles[j].x += particles[j].vx * delta_t;
        particles[j].y += particles[j].vy * delta_t;
    }
}
```

One alternative algorithm that could theoretically be an improvement on our algorithm is the following:

```
for all timesteps
    for all particles j
        set current particle to p1
        for particles k=j+1, k<N
            set particle[j] to p2
            calc. force between p1 and p2
            add to acceleration sum of p1
            add to acceleration sum of p2
        update velocity using calculated acceleration
    for all particles j=0, j<N
        update pos using velocity
        reset acceleration sum to 0
```

This algorithm would give an improvement over the first one because we know that the force between two particles are equal and opposite and therefore we can compute the acceleration of particle 1 with respect to particle 2 at the same time that we calculate the acceleration of particle 2 with respect to particle 1. In this implementation we would need to store the acceleration sums for all particles but we can drastically decrease the number of computations needed as we basically get two accelerations computed for the price of one.

We implemented this second algorithm in code but found that it was actually a bit slower than our original algorithm. We were a bit confused by this as we thought that the decrease in the number of calculations would result in a lower run-time. One explanation we thought of was that the storage of the additional values for acceleration for each particle would mean that the particle would take up more space in memory and therefore when data is read into cache we would not read in as much data that can be used in the following calculations and thus increase our cache-misses. Since the computer would then have to go "further" to get data for the next calculation this would cause our code to run slower and even decrease performance to such a level that even the decrease in calculations would not make up for it.

After our code has calculated and updated the last position we write the data to an output file and free the memory allocation for all particles.

A large part of our optimization was made to minimize the time in the innermost for-loop, since this is the one where the program will spend most of it's time.

The innermost loops are divided into two separate loops to avoid having an if-statement inside a single loop that checks that we do not calculate the acceleration with respect to the current particle. The current implementation first calculates the acceleration for all particles before the current and then for all particles after and thus does not need an if statement.

We chose to compute the values for r_{ij} and store it in a variable as this uses an expensive square-root calculation that we wanted to avoid doing multiple times. In this way we save one square-root calculation everytime the innermost loop executes. We also chose to save a variable called acc_k which is a common constant for both the acceleration in y- and x-directions. This saves us from having to recompute the division and cube a second time. We observed that the multiplication with $-G * m_i$ in eq. 1 is applied to all components of the acceleration sum so we decided to move this out of the inner loop and apply it afterwards and therefore saving some computations inside the inner loop. Also the m_i cancels out in our implementation and we could ignore it completely.

3 Performance and discussion

We chose to optimize the performance of our code towards our personal hardware, in this case a M2 Macbook Air. All timings were on this machine unless stated otherwise and were performed on the $ellipse\_N\_03000.gal$ file with 100 steps and 0.00001 timestep. The compiler used was Apple clang version 14.0.0, but when comparing with gcc-12 we saw no noticeable difference, using the -O3 -ffast-math flags as default. All timing were done using the built in unix time function, measuring the total wall time for running the entire file.

We tried two different approaches for saving the data for each particle, In the first iteration of the code, the first part read the data and placed it in three arrays; one for the position and mass of each particle, one for the velocity and one for the brightness. The reason we placed the data in these three arrays were because these data values are needed in different frequency in the calculation and we do not want to load in unnecessary data to the cache. The thought was that this takes up space that might otherwise be filled with data that we need in the calculations. Particularly the innermost loop that calculates the total acceleration of a particle with respect to all other particles uses only data about the position and mass and we therefore thought that having less unused data in the cache might speed up this part of the code. However, due to the large size of the list we instead got a lot of cache misses.

We then tried changing to having the data saved in a struct instead. This turned out to be slightly faster and significantly reduced the amount of L1 cache misses. For comparison, for one specific run the time was reduced from 0.804s to 0.675s. We chose to continue with this implementation. When checking with valgrind on the linux computers, we achieved a 1.6% L1 cache miss rate and a very few last level cache misses.

One interesting thing we noticed when changing between the array implementation and the struct implementation, was that it became faster on our computers but slightly slower on the school linux computers. This could perhaps be because of the different ratio between the memory and cpu speed of the school computers compared to ours, but we could not make any conclusions.

We also tried different versions of reordering the innermost for-loop in an attempt to further increase the performance. One thing we did, to avoid any if statements in the innermost loop was split it into two loops, one for all particles before the current and one for all particles after. This was to avoid having to check if the current iteration through the loop was about to calculate the effect of gravity on itself. For comparison, the code using an if statement was around 0.5s slower. We also tried minimizing the amount of operations needed whenever possible, especially with respect to division and the sqrt call, since they need significantly more flops. Instead we chose to save these in temporary variables. Much of this had very minor effects on our measured time, which we believe could be due to the compiler already performing several of these changes when using the -O3 flag.

We tried several different compiler optimization flags. Our final version of the code took, for example 5.903s to run without any optimization flags, 1.014s with -O2 and 0.659s with -O3 -ffast-math. We ended up using the -ffast-math even though it can increase the numerical error, since the measured error when compared to the reference output data, was still very low.

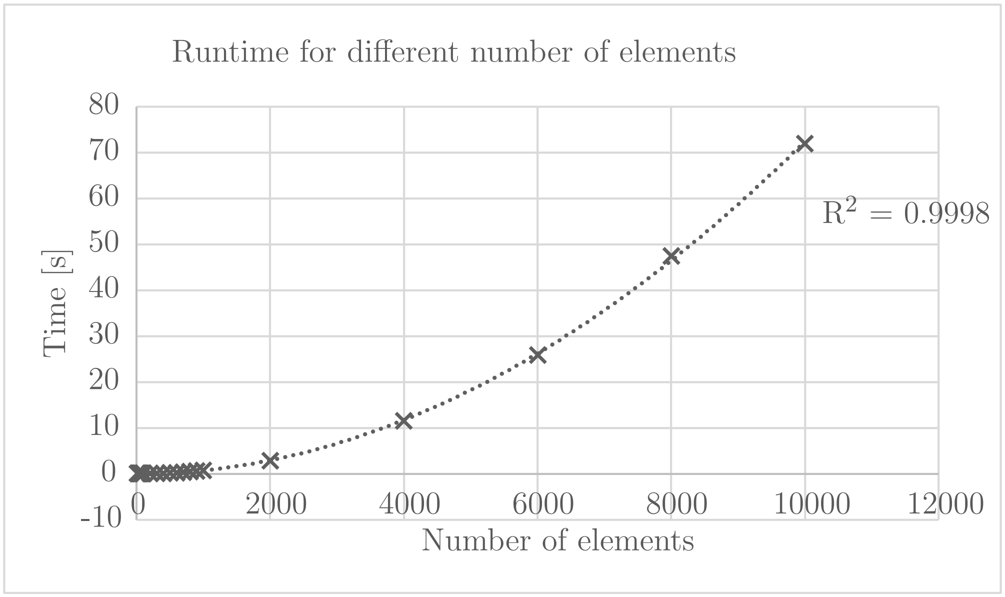

Figure 1: Plot of the different runtimes for a different number of elements

The time was measured for different N using the *ellipse_N_10000.gal* file, but only loading up to a given amount of particles. Each program was run for 1000 steps with 0.00001 timestep. The time clearly follows the $O(N^2)$ time complexity we expect from this straightforward implementation.