UPPSALA
UNIVERSITET

HIGH PERFORMANCE PROGRAMMING

Assignment 3

Sy-Hung Doan, Marius Nitzsche

February 15, 2019

1 The problem

The goal of this Assignment was to simulate the evolution of possible galaxies. Based on Newton's law of gravitation in two dimensions the following equation will help compute new positions and velocities of stars after a certain time step.

$$\mathbf{F}_i = -Gm_i \sum_{j=0,j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij} \tag{1}$$

m_i is the mass of a particle i, d_{ij} is the distance between two particles and $\mathbf{r}_{ij}$ is the two dimensional vector, which describes the positions of the particles dependent on each other. With the help of the symplectic Euler time integration method and the result of 2 new positions and velocities can be updated.

Initial positions and velocities are given in a file; our program is supposed to read them and save the updated information after computing a certain amount of time steps. The constants have the following values:

- $\epsilon_0 = 10^{-3}$
- Gravity $G = 100/N$ (N is the amount of particles)
- timestep $\Delta t = 10^{-5}$

The program is able to run with and without graphics, which demonstrate the current state of the simulated galaxy.

2 The solution

2.1 Data structure

We have two versions of the code. The first one we use `struct`

```
typedef struct point {
    double px; // x position
    double py; // y position
    double m;  // mass
    double vx; // x velocity
    double vy; // y velocity
    double b;  // brightness
} point;
```

and the second one we use two `array`'s.

```
// 0 is x position
// 1 is y position
```

```
    // 2 is mass
    // 3 is x velocity
    // 4 is y velocity
    // N is the provided number of points
    double params[N][5];
    double brightness[N];
```

2.2 Code structure

Our code is divided into 4 main parts:

The first part contains `#include`'s and some coefficients and constants that are used later on.

Next part is the definitions of some helper functions like reading data from the input file, executing with or without graphics, calculating the data at the next time step and writing data to the corresponding output file.

What follows is the `main` function. The last part is helper functions' bodies.

2.3 Algorithm implementation

First, we have two arrays of `point`'s. `prev_points` is for the data of the **current** state and `next_points` is for the data of the **next** state.

```
    point prev_points[N]; // N is the number of particles/stars/points.
    point next_points[N];

    for(int i = 0; i < N; i++)
    {
        double fx = 0.0, fy = 0.0;

        for(int j = 0; j < N; j++)
        {
            if(i == j)
                continue;

            // distance between 2 particles along x direction.
            double rx = prev_points[i].px - prev_points[j].px;

            // distance between 2 particles along y direction.
            double ry = prev_points[i].py - prev_points[j].py;

            // distance between 2 particles.
            double r = sqrt((rx * rx) + (ry * ry));

            // double r1 = pow((r+e0), 3);
            double r1 = (r + e0) * (r + e0) * (r + e0);
```

```
24
25              double tempx = prev_points[j].m * rx / r1;
26              double tempy = prev_points[j].m * ry / r1;
27
28              fx = fx + tempx;
29              fy = fy + tempy;
30          }
31
32          //accelerations
33          double ax = - fx * G;
34          double ay = - fy * G;
35
36          // calculate new velocities in each direction.
37          next_points[i].vx = prev_points[i].vx + delta * ax;
38          next_points[i].vy = prev_points[i].vy + delta * ay;
39
40          // calculate new positions in each direction.
41          next_points[i].px = prev_points[i].px + delta * next_points[i].vx;
42          next_points[i].py = prev_points[i].py + delta * next_points[i].vy;
43      }
```

Here we have the formula:

$$\mathbf{F}_i = -Gm_i \sum_{j=0,j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}_{ij} \tag{2}$$

In line 6,

$$\mathbf{f}^x = \sum_{j=0,j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}^x_{ij} \tag{3}$$

$$\mathbf{f}^y = \sum_{j=0,j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\mathbf{r}^y_{ij} \tag{4}$$

We calculate r^x_{ij} and r^y_{ij} in line 13 and 14. Line 15 shows the calculation of d^x_{ij}.

We perform the symplectic Euler 1 time integration method in line 29-36. Following is formulas in direction-i.

$$\mathbf{a}^n_i = \frac{\mathbf{F}^n_i}{m_i} \tag{5}$$

$$\mathbf{v}^{n+1}_i = \mathbf{v}^n_i + \Delta t \mathbf{a}^n_i \tag{6}$$

$$\mathbf{x}^{n+1}_i = \mathbf{x}^n_i + \Delta t \mathbf{v}^{n+1}_i \tag{7}$$

3 Performance and Discussion

All of our test were made on our own computer, it has the following specs:

```
Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
Ubuntu 18.04 LTS

L1d cache:  32K
L1i cache:  32K
L2 cache:   256K
L3 cache:   6144K

gcc (Ubuntu 7.3.0-16ubuntu3) 7.3.0
```

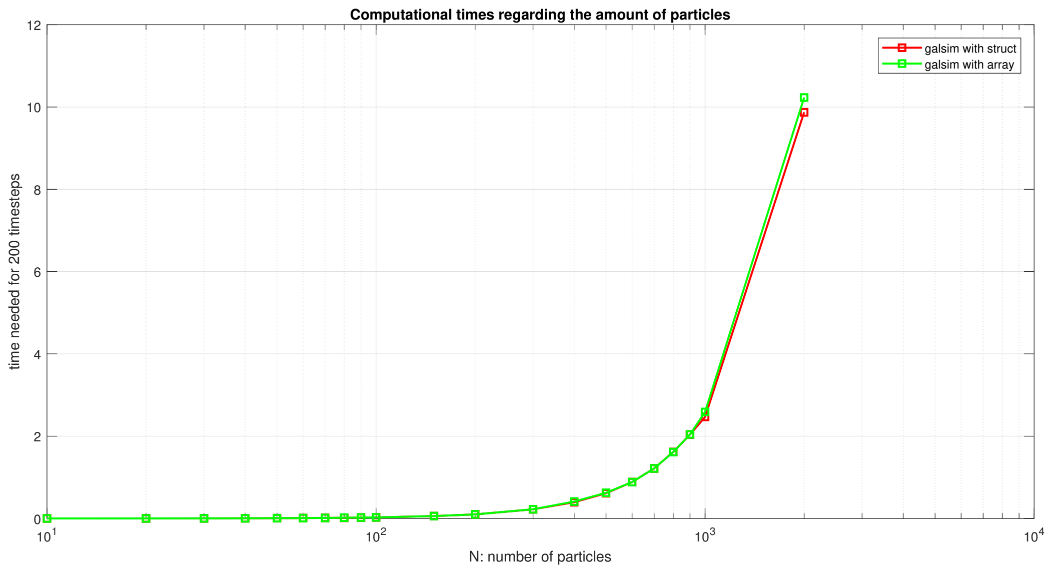
We have gone through different steps to optimize. However, to keep it succinct and brief, we first tried to get as many independent instructions as possible. We divided the calculation into 2 main parts corresponding to 2 directions, `x` and `y`. Since those calculations of each direction are uncorrelated, they would be executed in parallel.

The most impressive optimization that we made was the calculation of $r + \epsilon_0$. (in line 22 and 23 in the Algorithm implementation section - page 3).

Line 22 is the slower way and Line 23 makes the code run faster.

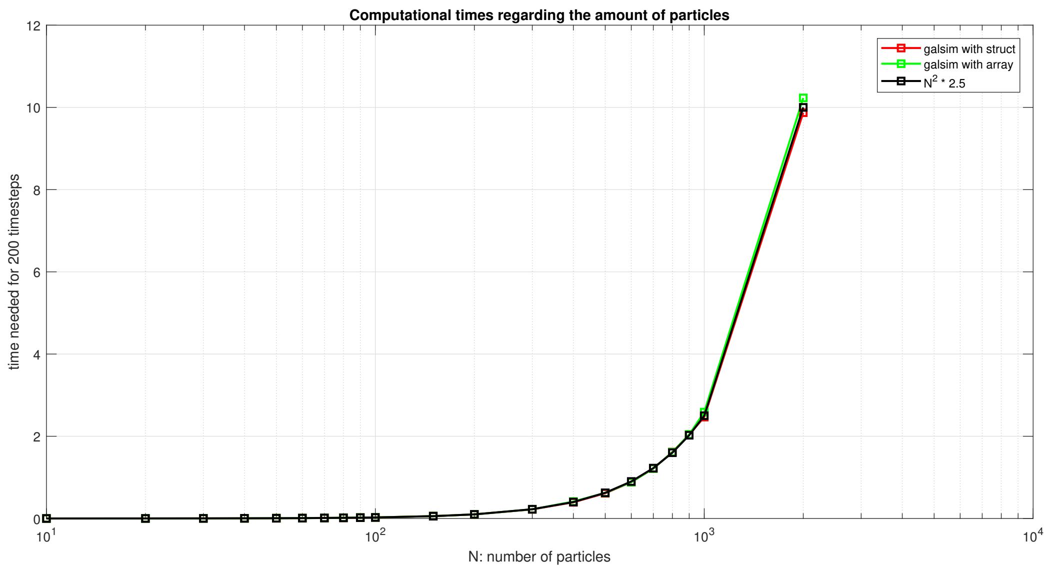
In the following test two things are looked at. First of all the computing time for different values of N is being shown in Figure 1. For less or equal than 100 particles the program is very fast; it is able to calculate 200 time steps in less than a tenth of a second. If N is increased the time of simulating is increased too. However, the computation time of the forces of gravity on 2000 particles after 200 time steps is still relative small at around 10 seconds.

Figure 1: Test for different values of N up until 2000



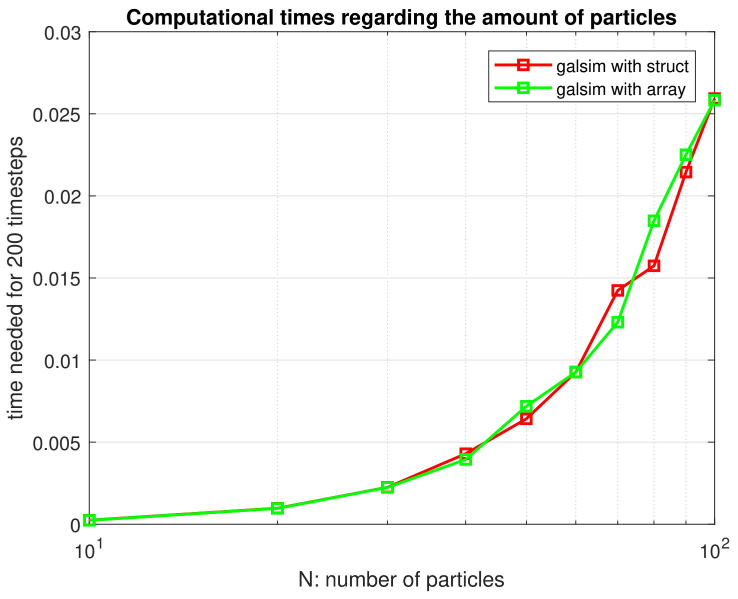
In Figure 2 the same curves are compared to an approximation function t, which is given by t(N)=N$^2 \cdot 1.5$. One can see clearly, that this is a good approximation. The computational cost is therefor of order $O(N^2)$.

Figure 2: Test for different values of N up until 2000 including an approximation



Furthermore, those graphics show a comparison of our two versions of the code. The green graph shows the code which uses arrays and the red one illustrates the one which is build around structures. Figure 3 shows the same test as Figures 1 and 2 but only for values of N up until 100 to highlight the differences of the separate approaches.

Figure 3: Test for different values of N up until 100



It seems that there is no winner between the different approaches of programming. In Figure

1 one might point to the fact that the 'struct' method wins out in the long run. However it is only by a very small margin.

4 Work contribution

We paired programming in order to create fewer bugs and make sure both understanding the codes.