THE GRAVITATIONAL N-BODY PROBLEM
Assignment 3

Daniel Owen-Berghmark & Sijia Wang

February 5, 2018

Introduction

In this assignment, the C code for simulate the evolution of a galaxy in 2D is implemented. To approximate the evolution, we calculated the motion of an initial set of particles using symplectic Euler time integration method. This method updated the velocity u_i and position x_i of particle i with

$$\begin{aligned}\mathbf{a_i^n} &= \frac{\mathbf{F_i^n}}{m_i},\\ \mathbf{u_i^{n+1}} &= \mathbf{u_i^n} + \Delta t \mathbf{a_i^n},\\ \mathbf{x_i^{n+1}} &= \mathbf{x_i^n} + \Delta t \mathbf{u_i^{n+1}},\end{aligned}$$

where $\mathbf{a_i}$ is the acceleration, Δt is the time step and $\mathbf{F_i^n}$ is the force on particle i exerted by the other particles. It could be calculated using Newton's law of gravitation, which is

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

where G is the gravitational constant, m_i and m_j are the masses of particle i and j, ϵ_0 is a small number and $\mathbf{r_{ij}}$, r_{ij} are

$$\begin{aligned}\mathbf{r_{ij}} &= (x_i - x_j)\mathbf{e_x} + (y_i - y_j)\mathbf{e_y},\\ r_{ij} &= \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.\end{aligned}$$

In our C code, the parameters in the method above are

$$\begin{aligned}\Delta t &= 10^{-5},\\ G &= 100/N,\\ \epsilon_0 &= 10^{-3}.\end{aligned}$$

Our Solutions

For this assignment we created one solution each, then we created one more solution for improving the execution time. The main structure is the same for the different versions. There is a data structure containing suitable names for the data variables associated to one particle, which are the the x and y positions of the particle, the mass, the velocity in x and y directions and the brightness of the star.

The execution flow of the program consists of,

- Allocating memory onto the heap. Which consists of N number of elements for the current time step and one for the next time step.
- Reading a configuration file to obtain the initial particle values.
- Calculating the simulation.

- Store the end result of the simulation into a file.

The different versions are mostly different in the simulation stage. The first two versions are similar to each other. They only differ with some auxiliary function calls in the math library. Otherwise they are similar in the steps taken to calculate the force. They start of by calculating how much force each individual particle are invoking onto one particle. When the force is known for one particle, it is updating its velocity and position according to symplectic Euler in the next time step array. Then it does the same procedure for all the stars. Before the next time step the algorithms switch the pointers' location. Our improved version takes into account that the force is symmetric (Newton's third law), by updating the force for both the objects the algorithm can avoid recalculating some forces. This lead to a changed nested loop, where the inner loop has its iterations reduced, but it updates the force twice. Example,

```
// First version
for (i = 0; i < N ;i++)
   for (j = 0; j < N ;j++)
      if(i!=j)
         force[i] += ...;
// Optimized
for (i = 0; i < N ;i++){
   for (j = i + 1; j < N ;j++) {
       force[i] += ...;
      force[j] += ...;
    }
}
```

The optimized version had an additional data structure which contained forces in x and y direction. This is used for saving the force for a particular particle. Which was not required by the other versions.
For improving our solutions we experimented with several methods, where the methods consists of:
Method 1: Compiling flags, -O0 flag to -Ofast.
Method 2: Change if(graphic==1) to if(graphic)
Because it is easier to check if graphic is nonzero than compare it with 1, use if(graphic) is expected to be more efficient. However, it might cause some problems in our program. For example, if the value of graphic equals to 5, it will turn on the graphic. To solve this problem, we changed the line for giving values to graphic to $graphics = (atoi(argv[5]) == 1)$, which would limit the value of graphics to $0, 1$.
Method 3: Add const to variables that does not change through the program.
Method 4: Reuse calculated forces, $f_i^j = f_j^i$
Method 5: Calculate $\frac{1}{(r_{ij}+\epsilon_0)^3}$ once and reusing it twice.
Method 6: Allocating memory once instead of twice.

Performance and discussion

The tests were made on several computers with different CPUs.
CPU 1: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, GCC 7.3
CPU 2: Intel(R)Core(TM)i5-3320M CPU @ 2.60GHz, GCC
CPU 3: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz, GCC 5.4

Table 1: Performance on different machines

CPU	Original	Method 1	Method 2	Method 3	Method 4	Method 5
CPU 1	15.121868	4.158810	4.152268	4.139221	2.681618	2.848261
CPU 2	24.306268	11.686961	11.697528	11.677528	5.782733	4.053890
CPU 3	45.009430	30.492752	30.345752	30.453184	15.284745	10.941924

Our results in Table 1 may have been more interesting if it had been done individually and then with the combinations of them. Due to time constraints we choose the later. The Table 1 shows us the methods combined from left to right. The measurements used the input values of $\triangle t = 10^{-5}$, time steps 100, with the configuration file ellipse_N_03000.gal. We did not take any notes for measurements on method 6.

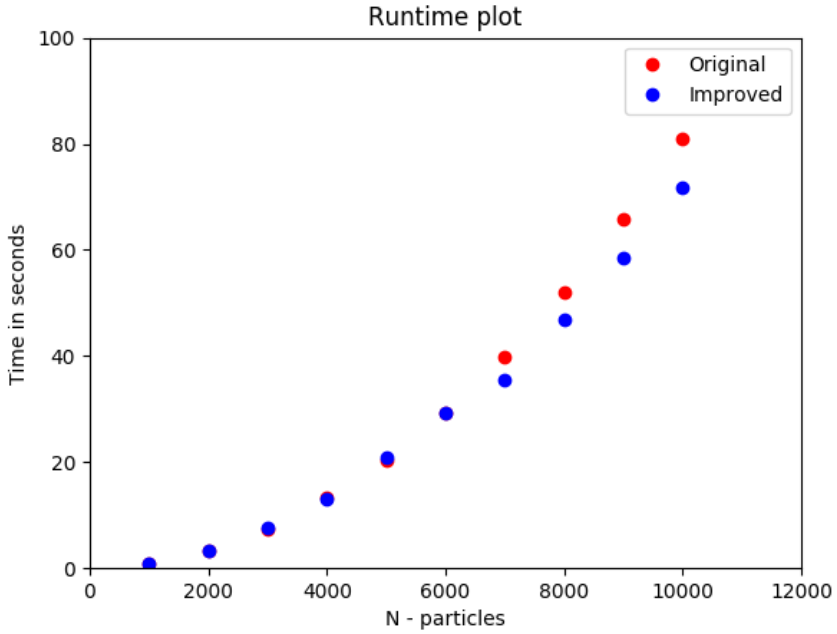

Figure 1: A runtime plot as a function of N. This benchmark was done on CPU 1.

In the Figure 1 one can study the execution time for algorithms as a function of N. Where the improved version is slightly faster than the original one. This

confirms that the algorithms is in $O(N^2)$.

The compiler made the greatest impact on the performance. This lead us to believe that we should always try those before moving onto some other optimizations options. This had also impacted some other optimizations options, which we did not have to consider like making some variables to constants. This did not have any impact on the execution time.

For the methods 2 and 3 might have been more interesting to see the effects without the compilation optimization.

Reducing the number of calculations had a great performance improvement. Method 5 had some varying results, which could potentially depend how fast CPU/compiler handles floating-point division. As an example in our code, one division plus two multiplications has to be faster than two divisions. Depending one how many cycles it takes for one division this might not be an improvement. Method 6 did not help for a better performance in our solution for CPU 1, this might also indicate that the compiler does something behind the scenes.

The optimized version is only optimized for the execution time, because it allocates more memory than the other version.

Contributions

In this assignment both has contributed to the code and the report.