THE GRAVITATIONAL N-BODY PROBLEM
Assignment 4

Daniel Owen-Berghmark & Sijia Wang

February 13, 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

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

Instead of implementing the brute force solution for this problem the Barnes-Hut simulation is utilized. Barnes-Hut simulation approximates the values for the bodies differently depending if the stars is close by or far away from each other. If they are near each other the previously explained algorithm is used to calculate the force. Otherwise if they are multiple stars far away this algorithm treats them as one large object. The stars are considered far away and can bu summed together if $h/r << 1$ holds, where h is the maximum distance between the objects in the group and r is the distance from the object to the group. If it holds one can use the groups center of mass, and the center point to calculate the force invoking on the object.

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

$$\begin{aligned}&\Delta t = 10^{-5},\\ &G = 100/N,\\ &\epsilon_0 = 10^{-3}\\ &\Theta_{max} = 0.25.\end{aligned}$$

Our Solutions

For this assignment we tried to create one solution each. When there was one solution we tried to optimize it.

There are several data structures in this program. 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. This data structure is used to maintain the structure for when the program should read and write the values from a file

Then there is a data structure which contained forces in x and y direction. This is used for saving the force for a particular particle. Another similar structure to the previous one is used which contains the position of a star in x, y values, and the stars mass. This structure is called data_node.

To determine if some objects are near each other a quad tree is implemented where a data structure is used to explain one node and its connections. This contains pointers to its children, a pointer to a data_node, some variables to explain the region where this quadrant is placed, the center of mass, the point to the center of the quadrant and an index if it holds a star. This structure is called quad_node

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.
- Reading a configuration file to obtain the initial particle values.
- Calculating the simulation.
- Store the end result of the simulation into a file.

The simulation steps is where the most of workload are. Where one step consist of starting of by creating the data_nodes for each particle for later using this information to create the quad tree with N data_nodes. When the algorithm is creating the quad tree it is inserting each data_node recursively. Here we could have implemented a different solution where the algorithm inserted quad_nodes instead of data_nodes. The main reason for why we chose to not pre-allocate the quad_nodes is that our implementation can not determine how many quad_nodes it requires for a particular tree thus it depends on how close two points are. The tree would be deeper if two points are close by to each other, if two points is farther away from each other a smaller tree could been used. This is because our quad_node can only contain one star. Thus it has to make the tree deeper if two stars gets inserted to the same quadrant. After the insert part of the algorithm it updates the center of mass and and center point for each quadrant node in the tree. When this is done, all the necessary information for calculating the force has been set up. Our simulation calculates the force by iterating over each star object and recursively calculates the force from the quad tree. It traverse further down a branch if the distance criteria does not hold otherwise it calculates the force with the center of mass and the center point. If it reaches a leaf its doing the ordinary force calculation.

When the force invoking on each object is known we use the Euler symplectic to update the velocities and positions.

There are several source files for this solution, where the graphics driver is in the the graphics folder. The main function lies in the galaxy.c. The implementation of the quad tree is in the source file quad_tree.c.

Improvement methods

When the thought that our implementation worked, we tested an profiling tool to see where it would be most beneficial to make improvements on our code. The tool we used is called gprof which told us that the code spent most time in the a function where we calculate the forces invoking on an object. When we revisited the code we found out that it only calculated force if a node only is a leaf. Which is not the right implementation of Barns-Hut algorithm. Fixing this problem made the implementation go from an execution time of 50 seconds to around 20 when we used the configuration file ellipse_N_2000.gal with options time steps of 200 $\theta_{max} = 0.1$. Fixing the order of the if statement contributed also to the better execution time.

For finding the a reasonable θ_{max} value, we iterated over the values $0.05 - 0.5$ where we increased the value by 0.05. This done to acquire a good enough accuracy for our implementation.

The compiler can also contribute to a better performance. The flags used for testing if makes the execution time better are, -O0, -O2, -Ofast, -ftree-vectorize and -march=native.

The information from the profiler gprof, said that the algorithm spent most time in the function which traversed the tree and calculating the force. So we introduced some modifications to this function for example reordering the if statements and introduced some variables instead for recalculating the values.

In an improved version there is also one more data structures and there is some modifications to some other data structures. The new data structure separates the velocity from the galaxy. Removed the index from quad_node and introduced a pointer into the data_node which points to the quad_node which holds the data. In the original version, it traversed the tree for a quad_node to find a matching index. Now the improved version can access the right quad_node directly instead. The previous version also divided the mass with force to obtain the acceleration. This is reworked with a new array storing the masses as one over mass. This is done to reduce double precision division.

An attempt to not re create the quad tree for each time step, were made but the attempt failed. The idea was to see if a star still were in the quadrant, if it is leave it there. Otherwise see if it can fit in the quadrant of its parent. Doing this until it can find a quadrant where it is in the region. Then inserting the node into this region. If this worked there would be less allocation and deallocation of the memory.

Performance and discussion

In Table 1, configuration file ellipse_N_02000.gal with 200 time steps is used. It shows that as the θ becomes bigger, pos_maxdiff becomes bigger and the running time reduced. To limited the pos_maxdiff less than 10^{-3}, we choose $\theta_{max} = 0.25$ for our code.

Table 1: Performance on different θ

θ	time	pos_maxdiff
0.05	22.375270	0.000012595186
0.1	13.531210	0.000112272450
0.15	8.599072	0.000247182503
0.2	6.354487	0.000412324340
0.25	4.004953	0.000929409861
0.3	3.388624	0.002250150910
0.35	2.655392	0.002867293511
0.4	2.221484	0.004753390608
0.45	1.943632	0.004920552462
0.5	1.728933	0.005668081183

Then we used the input values of $\triangle t = 10^{-5}$, time steps 100, $\theta_{max} = 0.25$, configuration file ellipse_N_05000.gal to do some time measurements with different versions on a laptop with Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, GCC 7.3, Arch Linux, kernel release 4.14.15-1-ARCH and on the lab computer arrhenius.it.uu.se with Intel(R) Xeon(R) CPU E5520 @ 2.27GHz, GCC 5.4.0, Linux, kernel release 4.4.0-112-generic.

Table 2: Performance with different flags

Computer\ Version	Original	Improved
arrhenius	17.565837	15.588552
laptop	6.531464	5.682541

In Figure 1 we can compare the execution time for Barnes-Hut algorithm to simple $O(N^2)$ algorithm. When N is a small number, simple $O(N^2)$ algorithm is faster than Barnes-Hut algorithm. When N becomes a larger number (approximately larger than 3000), Barnes-Hut algorithm is more efficient.From this phenomenon, we infer that the complexity of Barnes-Hut algorithm is less than $O(N^2)$.

In order to find the complexity of Barnes-Hut algorithm, different functions have been tried to fit it's execution time. In Figure 2, we found that the function $C * Nlog(N)$,$C = 0.00045$ fits the execution time well. So we guess the complexity of Barnes-Hut algorithm is $O(Nlog(N))$.

In Table 2 on can study the overall result for the different versions. Where the improved versions tries to utilize the catch memory in better way and do less computations overall. It is interesting to see that the combination of the mentioned improvements contributed to 1-2 seconds improvement.

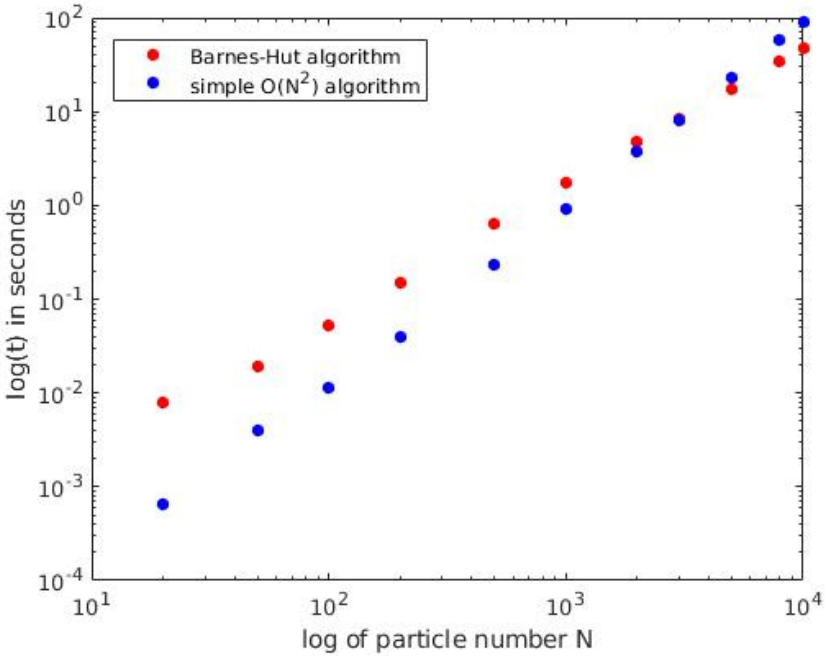

Figure 1: A runtime plot for Barnes-Hut and simple O(N^2) algorithm

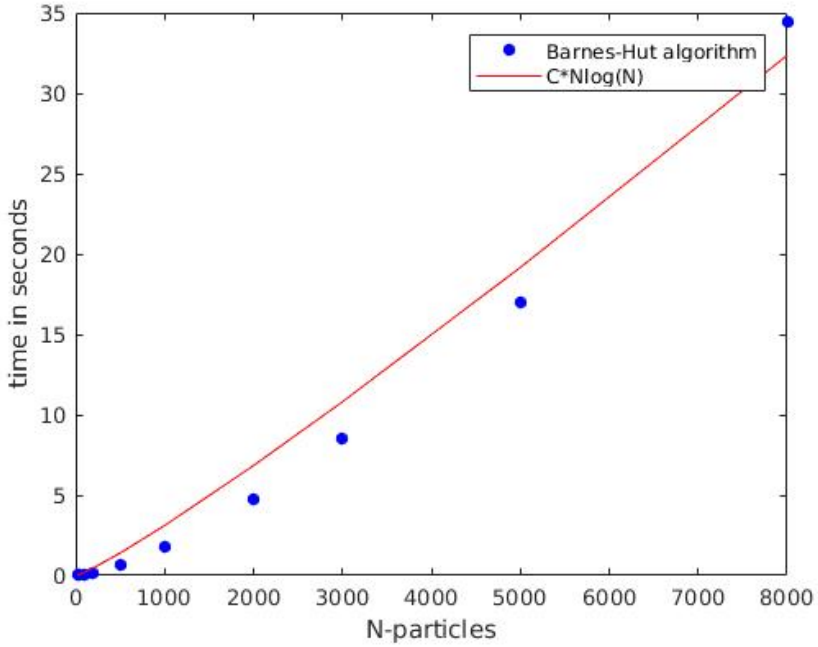

Figure 2: A runtime plot for Barnes-Hut and function CNlog(N)

Contributions

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