High Performance Programming Report

Assignment 5: Solving the N-body problem
using Barnes-Hut method with parallelization

Group 55: *Peralta Alguacil, Francisco Jose*
Dao, Tuan Anh

February 20, 2018

1 Introduction

In this assignment, we study the N-body problem under the view of Newton's law of gravitation using Barnes-Hut method with parallelization in the most time-consuming part of the program.

The report is organized as follows: Section 2 briefly describes the problem; Section 3 discusses the final data structures and algorithms; the choice of $\theta_{\max}$ for the given target accuracy, complexity, performance and related experiments are included in Section 4; Section 5 is the conclusion.

Regarding group collaboration, following previous works, we continued to do programming separately at first. Then we sat together to choose the best implementation on each part. We also based this report on those selections and improvements. We have been using a private GITHUB.COM repository to manage the code and report files.

Table 1: CPUs used for performance testing in this report

#	**Name**	**CPU information**	**Operating System**	**Compiler**
1	Macbook 2016	Intel(R) Core(TM) i5-6360U CPU @ 2.00GHz	MacOS 10.13.1	gcc 7.2.0_1
2	Berzelius	Intel(R) Xeon(R) CPU E5520 @ 2.27GHz	Ubuntu 16.04.3 LTS	gcc 5.4.0
3	Tussilago	AMD Opteron(TM) Processor 6282 SE @ 2.60GHz	Scientific Linux 6.9	2.60GHz, gcc 4.4.7

In the purpose of reproducibility, we include the information of CPUs in Table 1. From this point on, those machines are well-defined and to avoid duplication, we only cite the names whenever needed.

2 The N-body problem

We shortly recall the N-body problem in this section.

Let $P = \{p_i\}_{i=1}^{N}$ be the set of N particles. The following variables are given at time $t_0 = 0$.

- Let $x_i \in \mathbb{R}^2, i = 1,\ldots,N$ be the two-dimensional coordination of particle p_i;
- Let $u_i \in \mathbb{R}^2, i = 1,\ldots,N$ be the two-dimensional velocity of particle p_i;
- Let $m_i \in \mathbb{R}, i = 1,\ldots,N$ be the mass of particle p_i.

The objective of this problem is to determine x_i, u_i at some desired future time point $t_{k+1} > t_0$.

We can solve this problem using *symplectic Euler* – a finite difference method. By considering an axe of discrete equidistant time points $\{t_k\}_{k=0}^{\infty}$ with time step Δt, we continuously update the values of $x_i, u_i, i = 1,\ldots,N$ at time t_{k+1} based on the known information at time t_k. For this purpose, we define some additional variables at some given time point.

- Let $F_i \in \mathbb{R}^2, i = 1,\ldots,N$ be the two-dimensional force on particle p_i;
- Let $a_i \in \mathbb{R}^2, i = 1,\ldots,N$ be the two-dimensional acceleration of particle p_i.

The new values at time point t_{k+1} are calculated for each particle i as follows.

1. $F_i^k = -Gm_i \sum_{j=1, j\neq i}^{N} \frac{m_j}{(||x_i^k - x_j^k|| + \varepsilon_0)^3}(x_i^k - x_j^k)$;
2. $a_i^k = \frac{F_i^k}{m_i}$;
3. $u_i^{k+1} = u_i^k + \Delta t a_i^k$;
4. $x_i^{k+1} = x_i^k + \Delta t u_i^{k+1}$

where $||.||$ is the normal Euclid norm, G is the gravitational constant and ε_0 is some small real number added to avoid the case when $||x_i - x_j||$ is too small. Throughout this assignment, we use fixed values of constants: $\varepsilon_0 = 10^{-3}, \Delta t = 10^{-5}$ and $G = 100/N$.

However, in the program implementation, by observation, we can avoid redundant calculation in some steps. This point is discussed more in the next section.

3 Data Structures and Algorithms

The code is implemented on C language. This section discusses the structure of our program regarding both data and algorithms.

3.1 Data Structures

We formed some structs for the ease of managing quadrangles and particles.

Table 2: Struct **vector** - common 2D Vector

DATA TYPE	NAME	DESCRIPTION
double	x	first coordination of the vector
double	y	second coordination of the vector

Table 3: Struct **particle** - coordinations and mass of a particle

DATA TYPE	NAME	DESCRIPTION
double	x	first coordination of the particle
double	y	second coordination of the particle
double	m	mass of the particle

Table 4: Struct **quadrangle** - a quadrangle object

DATA TYPE	NAME	DESCRIPTION
double	boundary_x	x-coordination of the lowest leftmost point
double	boundary_y	y-coordination of the lowest leftmost point
double	boundary_length	length of an edge
double	mass	total mass of particles inside it
double	mass_x	x–coordination of center of mass
double	mass_y	y–coordination of center of mass
struct particle*	particle	the pointer of particle if it contains exactly one (leaf node)
struct quadrangle*	sw	South West child quadrangle
struct quadrangle*	se	South East child quadrangle
struct quadrangle*	ne	North East child quadrangle
struct quadrangle*	nw	North West child quadrangle

As can be seen on the Table 5, we only load the coordinations and mass of each particle into the struct. We avoid adding velocity and brightness by storing them in separate arrays of built-in doubles since brightness is not used at

all in the iterations, and we only have to update the velocity once after *nsteps* loops. This decreases the size of the struct, and chance is the cache will contains frequently-accessed data for improving the performance.

We could simply use some arrays of doubles instead if we do not need a struct of quadrangle of which properties are included in Table 4. This struct is slightly more complicated and is designed to make the idea of quadtree work. Basically, we store some information to determine the position, size and mass distribution of each quadrangle, with four pointers to its four children.

3.2 Algorithms

Since the most problematic part of this assignment is to implement the quadtree correctly, with some operators: insert a new particle, calculate the masses and centers of mass for each nodes of the quadtree, and calculate force using such information of the quadtree, with a parameter of desired accuracy $\theta_{\max}$. We first describe the idea of the main program in Algorithm 1, then the details of related functions are later explained.

Algorithm 1 Main

```
1:  procedure MAIN
2:      Load particles ← INPUT
3:      for each time step do
4:          Initialize the quadtree
5:          for i = 1 to N do
6:              INSERT(i^th particle) into the quadtree
7:          end for
8:          CALCULATEMASS(quadrangles) for all nodes of the quadtree
9:          for i = 1 to N do
10:             CALCULATEFORCE(i^th particle)
11:         end for
12:         Update the new values of coordinations for all particles
13:         Deallocate the quadtree
14:     end for
15:     OUTPUT ← particles
16: end procedure
```

For the purpose of experimenting, we developed two versions of the same function to insert a new particle into the quadtree. The first one is the recursive version, which described in Algorithm 2. Algorithm 3 is the iterative version.

Both result same output. Aspects concerning performance and memory usage is more discussed in Section 4. Here we note that only the quadrangles which contain at least one particle is stored. Thus, every leaf node (which does not have children) contains a particle, and a particle is always stored in a leaf node.

Algorithm 2 Insert (Recursive version)

```
1:  procedure INSERT(quadrangle n, particle p)
2:      if n is NULL then
3:          Create a new node at n and store p in n
4:      else if n has children then
5:          Determine the child m ∈ {sw, se, ne, nw} to which p belongs
6:          INSERT(m, p)
7:      else
8:          np ← (pointer) the particle which n contains
9:          Determine the child m1 ∈ {sw, se, ne, nw} to which np belongs
10:         Insert a new node at m1 and move np to m1
11:         Determine the child m2 ∈ {sw, se, ne, nw} to which p belongs
12:         INSERT(m2, p)
13:     end if
14: end procedure
```

Algorithm 3 Insert (Iterative version)

```
1:  procedure INSERT(quadrangle n, particle p)
2:      if n is NULL then
3:          Create a new node at n and store p in n
4:      else
5:          Find current ← the smallest (can be null) quadrangle containing p
6:          (this quadrangle contains at most one particle other than p)
7:          while current contains another quadrangle np do
8:              Determine and add the children m1, m2 ∈ {sw, se, ne, nw}
9:              to which np, p belongs, respectively;
10:             current ← m2
11:         end while
12:     end if
13: end procedure
```

Next algorithm recursively calculates the mass of each node in the quadtree.

Algorithm 5 recursively calculates the force on each particle caused by all particles inside a quadrangle n in the quadtree.

Algorithm 4 Calculate Mass

```
1:  procedure CalculateMass(quadrangle n)
2:      if n is a leaf node then
3:          n surely contains a particle p
4:          n.mass ← p.mass;
5:          center of mass of n ← coordinates of p;
6:      else
7:          mass ← 0; center_of_mass = 0;
8:          for each child m of n do
9:              mass_m ← CalculateMass(m);
10:             mass ← mass + mass_m;
11:             center_of_mass ← center_of_mass + mass * m.center_of_mass;
12:         end for
13:         center_of_mass ← center_of_mass / mass;
14:     end if
15: end procedure
```

Algorithm 5 Calculate Force

```
1:  procedure CalculateForce(quadrangle n, particle p)
2:      θ = n.boundary_width / ||n.center_of_mass - p.coordinates||_Euclid;
3:      if n is a leaf node OR θ < θ_max then
4:          Calculate the force between n and p as in Section 2
5:      else
6:          total_force ← 0
7:          for each child m of n do
8:              total_force ← total_force + CalculateForce(m, p)
9:          end for
10:     end if
11: end procedure
```

Alternative options on data structures and algorithms:

- **Storing all the data on separate arrays**. It is possible if we use a straightforward $\mathcal{O}(N^2)$ algorithm. However, in this assignment, since the structure of quadtree is used, it is much more convenient for the operations on quadtree to store some properties of the particles in another struct.
- **Full struct of particle: we could use a full struct of the particle**. However, for the purpose of caching, we only want to call the most frequently-accessed properties, which are coordinates and mass. Properties like brightness are even not used in the computation; and we only have to update the velocity once at the last iteration.
- **Recursive or Iterative function?** If implemented correctly, the time complexity is the same for both implementations. However, if the size of the problem is really large, then it is necessary to avoid the recursive functions since it will use a huge amount of stack memory and it could lead to a stack overflow. However, in this assignment, because of the size of the problem, it is possible to use the recursive version since it is much easier to understand. We did both implementations and give some comparisons in Section 4.

4 Experiments, Optimizations and Complexity

The content in this section involves some performance results in real time measurements. The time (wall) was measured using the same function *get_wall_seconds()* as in the lab. The total time includes both I/O time and calculation time.

4.1 Choice of $\theta_{\max}$ for the given target accuracy

Since the suggestion of $\theta_{\max}$, we calculated the pos_maxdiff with different values of $\theta_{\max}$ between $[0.02, 0.5]$. Intuitively, the larger $\theta_{\max}$ is, the bigger error (or pos_maxdiff) will be. Thus, the line graph in Figure 1 should be monotonously increasing. Therefore, we draw a red line of error threshold to illustrate the maximum tolerance for pos_maxdiff. The maximum value of $\theta_{\max}$ satisfying the target accuracy we found is **0.256**. We could have found more precise value but we decided to stop there.

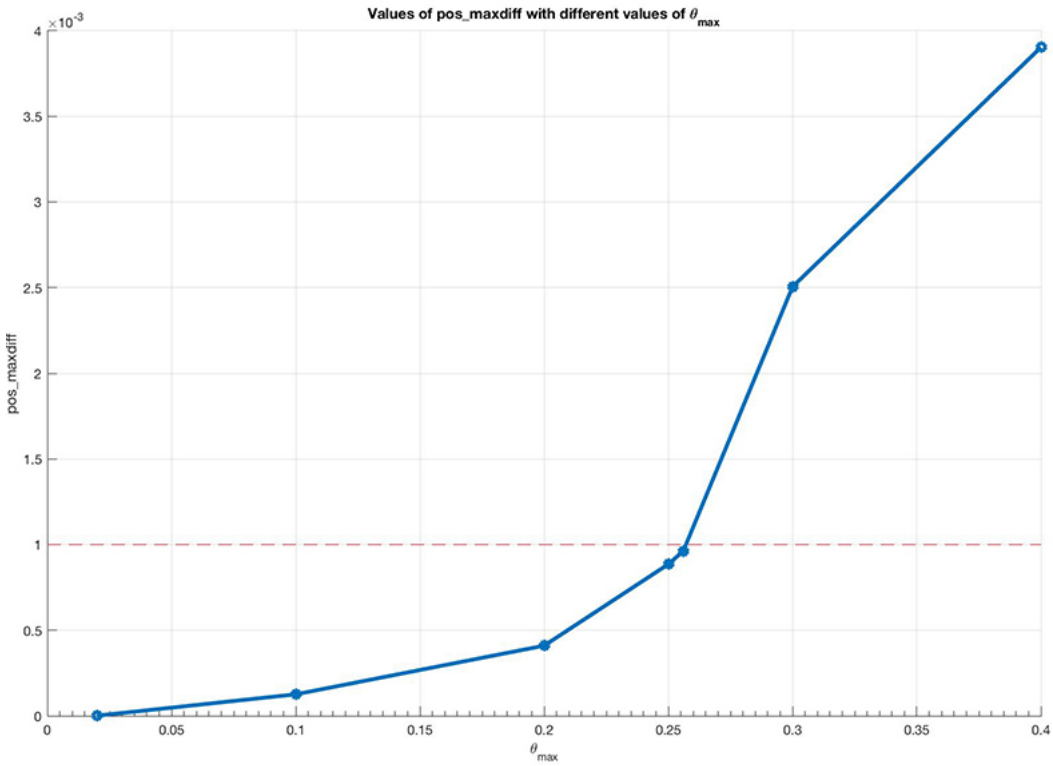

Figure 1: Values of pos_maxdiff resulted by different values of $\theta_{\max}$

4.2 Final results

The wall time was measured using the **time** command. Each result is the best time among five trials. We did these performance testings on Macbook 2016 and Berzelius of which information was stated in Section 1.

Table 5: Final performance results with different optimization flags (in seconds)

Optimization flag	**Macbook 2016**	Speedup	**Berzelius**	Speedup
No flag	8.34s	1	14.89s	1
-O2	3.94s	2.12	6.97s	2.14
-O3	3.91s	2.13	6.97s	2.14
-O3 *-march=native -ffast-math*	3.93s	2.12	6.67s	2.23

We can observe that adding flags has significant improvement in the performance of the program. However, the results for -O2 and -O3 flags are quite similar in both machines. For **Macbook 2016**, the *-march=native -ffast-math* flag does not make any improvement in comparison with -O3 but it does in

the other one **Berzelius**. We also include speedup results along with the time results to illustrate the efficiency.

4.3 Experiments

4.3.1 Replacing recursive functions by iterative functions

-Same time complexity

-Better space complexity (use memcheck valgrind)

-Avoid stack overflow

4.3.2 Complexity of each operator on the quad tree

4.4 Program complexity

The performance measurements in this section were done on **Berzelius** with -O3 optimization flag. Table 6 indicates the computation time for sets with different number of particles. We got the time in cases $\theta_{\max} = 0$ for $\mathcal{O}(N^2)$ complexity, and $\theta_{\max} = 0.256$. The time was measured by placing the **get_wall_seconds()** which is similar to the one in the lab, at the beginning and the end of the program, except the I/O procedures. This is because on our server **Berzelius**, the I/O time is really random and it does not make sense if we include those to evaluate the complexity. The slots marked with "–" are bigger than 100s. We also plotted the computation time versus the number of particles for checking the real complexity in Figure **??**.

As can be seen from Figure 2, when setting $\theta_{\max} = 0.256$ (Barnes-Hut implementation), we obtain the expected $\mathcal{O}(N\log(N))$ complexity while in case $\theta_{\max} = 0$, we have the shape of $\mathcal{O}(N^2)$ complexity.

5 Conclusion

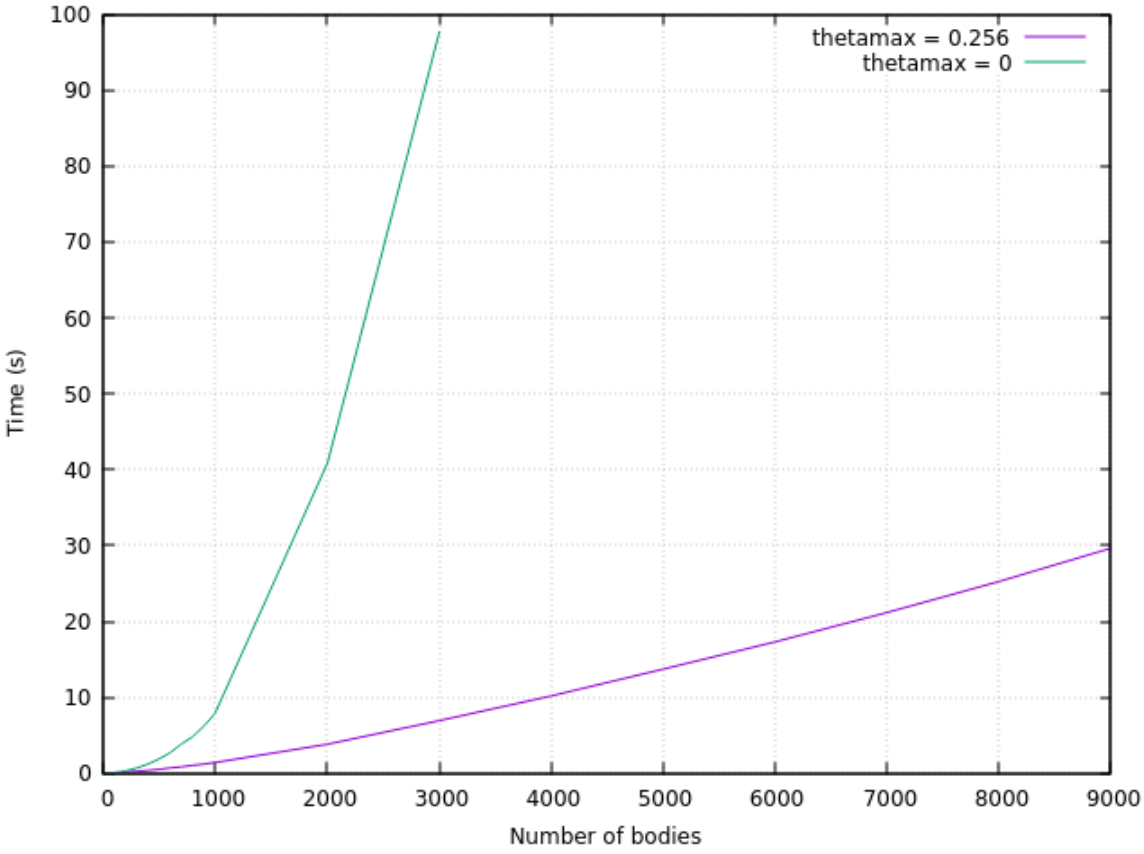

Figure 2: complexity

Table 6: Computation time on **Berzelius** with different values of N

N	Wall time with $\theta_{\max} = 0.256$	Wall time with $\theta_{\max} = 0$
10	0.0009370	0.0010390
20	0.0031040	0.0035601
30	0.0089421	0.0069938
40	0.0110779	0.0115230
50	0.0173981	0.0178390
60	0.0223758	0.0280061
70	0.0287378	0.0335340
80	0.0335600	0.0423331
90	0.0392060	0.0534279
100	0.0442431	0.0651462
150	0.0805900	0.1430421
200	0.1214850	0.2762249
300	0.2295508	0.6667540
400	0.3666220	1.1795640
500	0.5067809	1.8718028
600	0.6812260	2.6995649
700	0.8349972	3.8379309
800	1.0206718	4.8004501
900	1.1958830	6.1728532
1000	1.4003611	7.8991399
2000	3.7918429	40.7159750
3000	6.8999462	97.6599860
4000	10.1695912	–
5000	13.6909249	–
6000	17.2950299	–
7000	21.1669080	–
8000	25.2523329	–
9000	29.6465890	–