# ID1217 Project Report
# The Gravitational N-body Problem

JINYU YU

***Abstract*—This project is to implement several programs to solve the N-body problem, and evaluate different algorithms, parallel methods and optimizations to increase performance. We implement the normal algorithm which is $O(n^2)$, and the Barnes-Hut algorithm which is $O(nlogn)$, and developed paralleled version of both programs.**

## I. Introduction

The N-body problem studies the motion law of the interaction of multiple particles. It is one of the basic problems of mechanics and astrophysics. It has a wide range of applications, and can be used to study cosmic celestial bodies in astrophysics. In many other fields like mathematics and biological, the N-body problem is also important, such as molecular dynamics research. Because the N-body problem is a chaotic phenomenon, there is no law for the future movement of the object. Therefore, the N-body simulation can only use the Brute Force strategy.

The naive algorithm is easy to understand and implement, which is to calculate the force between every pair bodies, then the velocity and position. Because we need to process every pair of bodies, so the time complexity is $O(n^2)$.

There's 2 ways of improve the performance, which the first is to make a parallel program to run on several thread concurrently, and the second is to change the algorithm itself. In this project, we choose Barnes-Hut which has a time complexity of $O(nlogn)$. So the following 4 programs are been written:

- A sequential $O(n^2)$ program.
- A parallel $O(n^2)$ program.
- A sequential $O(nlogn)$ program using the Barnes-Hut method.
- A parallel $O(nlogn)$ program using the Barnes-Hut method.

In the evaluation part, we will concern the impact of both different number of bodies and different number of worker threads on the programs. We use C with Pthreads library using shared-memory programming model to do the concurrent stuff, and only run it on a single computer, so an estimate an amount of communication is provided in the evaluation part.

## II. Implementation

### *A. Sequential $O(n^2)$ program.*

In the N-body problem, every body has a mass and a initial position and velocity, Gravity causes the bodies to accelerate and move. The motion of N-body system is simulated by through discrete instant of time. At each time step, we calculate the force on every body, and then update their velocity and position.

```
for [time = start to finish by DT]{
  calculate forces;
  move bodies;
}
```

In calculating force step, the only thing that need to do is find all pairs of the bodies, and then use Newton formula

$$F = \frac{Gm_i m_j}{r^2}$$

To calculate the *magnitude* of the gravitational force between two bodies $i$ and $j$. Where $r$ is the Euclidean distance between the two bodies, given by

$$\sqrt{(p_i \cdot x - p_j \cdot x)^2 + (p_i \cdot y - p_j \cdot y)^2}$$

To find each pair of bodies, we use two quantifiers $i$ and $j$, where $i$ loops from 1 to $n-1$, and $j$ loops from $i+1$ to $n$. This will bring a $O(n^2)$ time consumption for the force calculate step.

The gravitational forces on a body cause it to move, and we use the following $O(n)$ formulas to calculate the velocity and position

$$F = ma$$

$$dv_i = a_i dt$$

$$dp_i = v_i dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2})dt$$

There are still some steps to do to prevent the program from errors.

a) Collision problem.

If two bodies comes too close to each other, then r will be very small, and will lead to numerical instability in calculation forces. In the extreme case, r may reach 0, and cause the division of 0 error when calculating the gravitational force. To solve this, we use a value that indicates the diameter of every bodies. If the distance between two bodies is smaller then the diameter itself, we use the diameter to indicate the actual distance.

b) Border problem

There is no border in the actual universe, but if there's no border in the simulation, a problems emerged: when a body accidentally get a position where is very far from other bodies, the force between the body and others will be extremely small. Again, numerical instability may generated, and the tiny force itself cause another problem: the body will not change it velocity, and will keep the velocity for the rest of its life. So we add an argument of the program, that indicates the size of the universe. every bodies that reach the border will keep its velocity but position unchanged. Bodies will not be bounced off.

c) Initialize problem

A good initial velocity, position and mass can lead a easier way to check whether the program is run properly. To simplify this, we use same mass ($m = 10000kg$) and same initial velocity ($v = 0$) for every body. To make the universe more compact, we let every body have a initial position on a squared grid, where the nearest distance between all the bodies are the same.

## *B. Parallel $O(n^2)$ program*

There is two step in the sequential program, where calculateForce have two quantifiers $i$ and $j$, where $i$ loops from 1 to $n-1$, and $j$ loops from $i+1$ to $n$, and moveBodies have only one quantifires $i$ that loops from 1 to $n$.

The moveBodies step can be easily paralleld by slice 1-$n$ by $numWoekers$, and each worker thread process only one of the slices.

However, there are two main problem in the calculateForce program.

a) Shared memory problem

calculateForce program use shared data *position* and *mass* to generate and add up the *force* data. *position* and *mass* will not change in this step, so they can be directly used, but the *force* data is frequently updated in this step, to prevent mutual access problems, we can use a critical area for *force*, but it will be very slow and increase the sequential fraction. To solve this, we use local arrays *force[worker]* to save temporary data, and add all of worker datas in the moveBodies Step.

b) Imbalanced workload

We can assign each worker a contiguous block of tasks, worker 1 handle first $n/numWorkers$ bodies, and worker 2 handles next $n/numWorkers$ bodies, and so on. However, this will lead to a very imbalanced workload. Worker 1 need to calculate forces between body 1 and every other body, then between body 2 and all other bodies except for 1, and so on. On the other hand, the last worker has relatively little to do because it only need to calculate the later bodies.

To solve this, we use a cyclic allocation strategies, which means workers process bodies in a loop. For example, when there is 4 workers, worker1 handles bodies 1,5,9,.. and worker2 handles bodies 2,6,10,.. and so on. This will bring a more balanced workload for all workers.

## *C. Sequential $O(nlogn)$ program using the Barnes-Hut method.*

In short, Barnes-Hut method is to approximate far bodies as a single body, and it has 4 steps instead of 2 in previous problem.

- Construct a quadtree representing the current distribution of the bodies.
- Compute the total mass and the center of mass for each cell.
- Calculate the forces using the information in the tree. If the center of mass of the cell is far enough away, approximate the entire subtree by that cell; otherwise, visit subcells. A user defined parameter $\theta$ is used to determine this.
- Move the bodies as before.

The number of nodes in the quadtree is $O(nlogn)$, so phases (1) and (2) have this time complexity. Calculate forces is also $O(nlogn)$, and move bodies is $O(n)$ as before.

a) Constructing quadtree

To construct the Barnes-Hut tree, insert the bodies in sequence. To insert a body b into the tree rooted at node x, use the following recursive procedure:

If node x is a leaf node and is empty, put the new body b here.

If node x is a parent node, Recursively insert the body b in the appropriate quadrant.

If node x is a leaf node and have value, say containing a body named c, then there are two bodies b and c in the same region. Subdivide the region further by creating four children. Then, recursively insert both b and c into the appropriate quadrant(s). Since b and c may still end up in the same quadrant, there may be several subdivisions during a single insertion.

b) compute total mass and the center of mass

This can be done by making an upward pass from the leaf node to the root.

If the node is empty, then its mass is 0, and have no position.

If the node is a leaf node and is not empty, then its mass is the same to the body it contains, and the same with the mass center position.

If the node is a parent node, its mass equals all of its children, and its mass center equals

$$x_p = (x_1 \cdot m_1 + x_2 \cdot m_2 + x_3 \cdot m_3 + x_4 \cdot m_4)/x_m$$

$$y_p = (y_1 \cdot m_1 + y_2 \cdot m_2 + y_3 \cdot m_3 + y_4 \cdot m_4)/y_m$$

c) calculating the force on a body

To calculate the net force acting on body b, use the following recursive procedure, starting with the root of the quad-tree:

If the current node is a leaf node (and it is not body b), calculate the force exerted by the current node on b, and add this amount to b's net force.

Otherwise, calculate the ratio $s/d$ where $s$ is the quadtree size and $d$ is distance. If $s/d < \theta$, treat this internal node as a single body, and calculate the force it exerts on body b, and add this amount to b's net force. Otherwise, run the procedure recursively on each of the current node's children.

d) empty the quadtree

because we need to regenerate the tree every time, we need to empty tree and free memory every time to prevent memory leak. To do this, simply:

If the node is a leaf node, delete itself and free memory.

If the node is a parent node, recursively delete all its children, then delete itself and free memory.

e) move bodies

This is the same to the previous program.

### D. *Parallel $O(nlogn)$ program using the Barnes-Hut method.*

The Barnes-Hut algorithm is harder to parallelize efficiently than the basic algorithm, because (1) the spatial distribution of bodies is in general nonuniform and hence the tree is imbalanced, (2) the tree has to be reconstructed at each time step because the bodies move, and (3) the tree is a connected data structure that is harder to distribute than simple arrays.

a) Construct the quad tree

This step is recursive, and every time it insert a node into the tree, the full tree need to be updated, thus it cannot be efficiently parallelized.

b) calculate mass and mass center

This step can be parallelized by simply assign 4 children nodes of the root node to 4 different workers. But as is said before, the tree can be extremely unbalanced, and this task allocation cannot be balanced.

c) calculate the force

This step can easily be parallelized by assign different bodies to workers.

d) delete the tree

This step can be parallelized by delete 4 children nodes in 4 workers.

e) move the bodies

This is the same to previous program

## III. Evaluation

### A. *$O(n^2)$ program.*

A result of $O(n^2)$ program showed in Fig. 1-3. This is the output of the sequential $O(n^2)$ program, with parameters *gnumBodies*=120 and DT=1.

From the result we can see that in the start, every body were posited on the grid, and after many timesteps, they begin to contract because of the gravity. Finally after 10000 steps, they form a galaxy-like pattern.

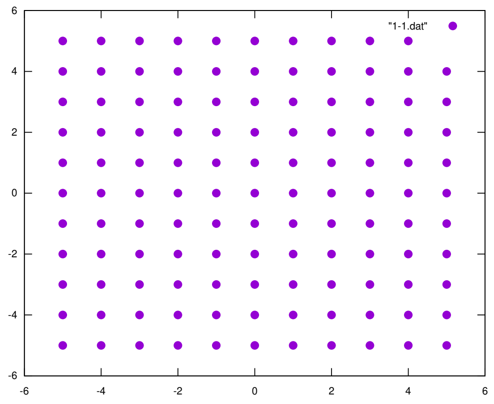

Fig. 1. Initial universe

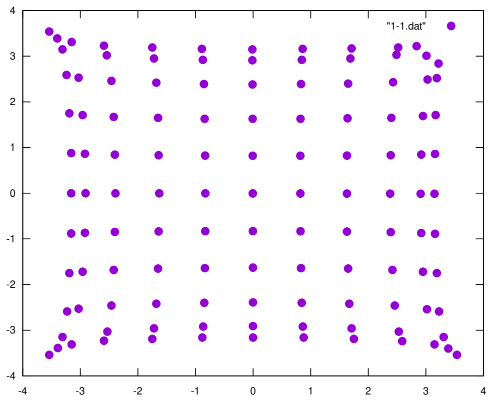

Fig. 2. $O(n^2)$ program after 1000 steps

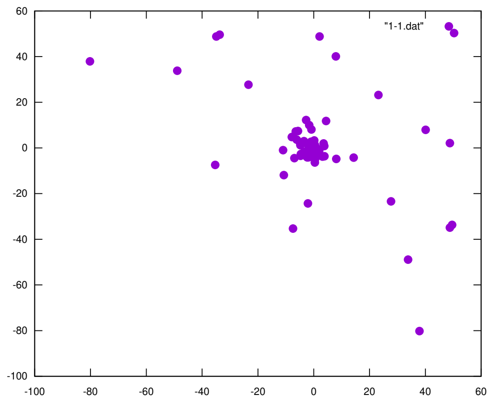

Fig. 3. $O(n^2)$ program after 10000 steps

Raw time data of different number of workers showed in Table 1. This is run with the arguments: numSteps = 50000, DT = 1. ”1w 120b” indicates 1 worker and 120 bodies.

TABLE I
RAW TIME DATA IN MS

| | Test1 | Test2 | Test3 | Test4 | Test5 | median |
|---|---|---|---|---|---|---|
| 1w 120b | 16009.185 | 15932.189 | 15964.554 | 15937.842 | 15956.122 | 15956.122 |
| 1w 180b | 35918.044 | 35870.673 | 35837.728 | 35849.626 | 35839.485 | 35849.626 |
| 1w 240b | 63796.918 | 63715.948 | 63730.888 | 63737.819 | 63683.147 | 63730.888 |
| 2w 120b | 9208.315 | 9097.996 | 9147.564 | 9098.945 | 9124.117 | 9124.117 |
| 2w 180b | 19250.523 | 19312.208 | 19384.834 | 19279.02 | 19332.115 | 19312.208 |
| 2w 240b | 33662.927 | 33758.506 | 33622.617 | 34724.001 | 34341.705 | 33758.506 |
| 4w 120b | 6521.459 | 6316.89 | 6409.129 | 6556.56 | 6603.324 | 6521.459 |
| 4w 180b | 13076.813 | 12962.148 | 13043.606 | 13038.055 | 13086.832 | 13043.606 |
| 4w 240b | 22457.44 | 22231.272 | 22189.452 | 22024.197 | 22326.63 | 22231.272 |

TABLE II
TIME CONSUMING DATA

| | 120b | 180b | 240b |
|---|---|---|---|
| 1worker | 15956.122 | 35849.626 | 63730.888 |
| times | 1 | 2.247 | 3.994 |

Time consuming compare showed in Table 2, and Speedup data in Table 3.

From the table we can find several interesting points.

a) the time complexity

TABLE III
SPEEDUP DATA

| | 1w | 2w | 4w |
|---|---|---|---|
| 120b | 15956.122 | 9124.117 | 6521.459 |
| speedup | 1 | 1.75 | 2.45 |
| 180b | 35849.626 | 19312.208 | 13043.606 |
| speedup | 1 | 1.86 | 2.75 |
| 240b | 63730.888 | 33758.506 | 22231.272 |
| speedup | 1 | 1.89 | 2.87 |

from table 2 (i.e. time consuming when only 1 worker), we can find that the algorithm perfectly obeys its $O(n^2)$ complexity, where $(180/120)^2 = 2.25$ and $(240/120)^2 = 4$, where the actual value is 2.247 and 3.994.

b) the speedup

the ideally speedup for 2 workers should be 2 and 4 workers is 4. Actually, these are less. There are several reasons. First, when use multiple workers, we should add temporary forces from all the worker outputs. The more worker is, the more calculation it has to be perform. So it lowers the speed. Second, the program iterated 50000 times, fork-join and barrier process time take up a lot amount of time. These all add to the sequential part of the program. That's also explains why speedup of 240 bodies is higher than 180 and 120 bodies, because it need more time to take the actual calculation, and the percentage of barrier lowers.

c) communication estimate

because we design the program with pthread and shared memory model, it cannot be distributed, so we need to estimate the amount of communication.

the calculateForces method needs to access several data: body position data $p$, body mass data $m$, body force data $f$. However, not all of this should be communicated. Since the mass of the stars stays unchanged, workers can have a local copy of mass data, so that they do not need to get this every time. Then, the force data is local datasource according to the algorithm. So the only data that need to be communicated is the position data.

the moveBodies needs to access several data: body position data $p$, body mass data $m$, body force data $f$ and velocity data $v$. However, not all of this should be communicated. The same to calculateForces, mass data $m$ can be omitted. For the velocity data, since every worker always handle the same bodies, this is also a local data. The moveBodies method need communicate position data $p$ and temporary force data from other workers.

## *B. $O(nlogn)$ program using the Barnes-Hut method.*

A result of $O(nlogn)$ program using the Barnes-Hut method showed in Fig. 4-6. This is with parameters *gnumBodies*=120 and DT=1 and $\theta$=0.5, the only difference is Fig. 5 use $\theta$=0.9

From the result we can see that if $\theta$=0.5, after 1000 steps, the approximation is nearly the same with the basic algorithm. But if $\theta$=0.9, it will be a bit different. After 10000 steps, due to cumulative error, the result of Barnes-Hut is different from the basic algorithm.

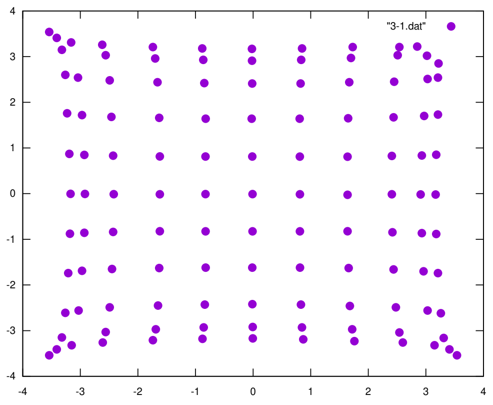

Fig. 4. Barnes-Hut program after 1000 steps $\theta$=0.5

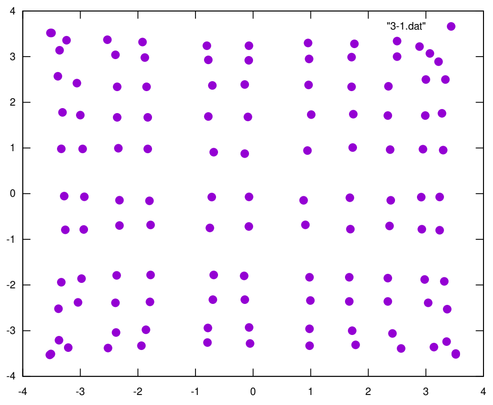

Fig. 5. Barnes-Hut program after 1000 steps $\theta$=0.9

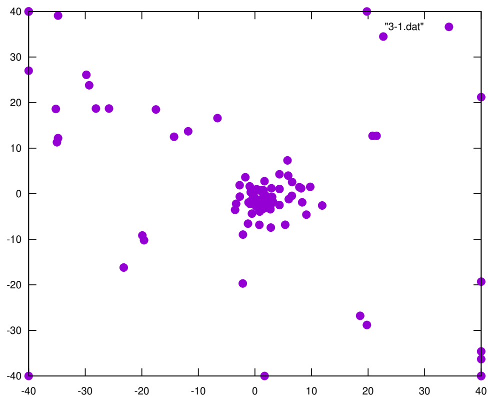

Fig. 6. Barnes-Hut program after 10000 steps $\theta$=0.5

From the table we can find several interesting points.

a) the time complexity

From table 4 "time consuming compare between sequential basic $O(n^2)$ and Barnes-Hut algorithm", we can find basic algorithm is same to the previous, with time consumption 2.250 and 3.992.

The Barnes-Hut is $O(nlogn)$, so the time consuming between 180 and 120 bodies should be $\frac{180log(180)}{120log(120)} = 1.63$, between 240 and 120 should be $\frac{240log(240)}{120log(120)} = 2.29$ , but the actually data is larger than this, means that the time complexity is higher. One possible reason is that the lower $\theta$ is, the more accurate, but the time complexity is higher. When $\theta$=0, it fallback to the basic $O(n^2)$ algorithm. So increase the value of $\theta$ can improve the speedup.

b) the speedup

Barnes-Hut is hard to parallelize. so the speed up is very low.

TABLE IV
TIME COMSUMING COMPARE BETWEEN SEQUENTIAL N2 AND BH $\theta$=0.8

| | 120 | 180 | 240 |
|---|---|---|---|
| basic | 15027.602 | 33813.247 | 59985.937 |
| time | 1 | 2.250 | 3.992 |
| BH | 12342.519 | 23380.573 | 33061.288 |
| speedup | 1 | 1.894 | 2.679 |

TABLE V
SPEEDUP OF BARNES-HUT

| | 1w | 2w | 4w |
|---|---|---|---|
| 120b | 13477.13 | 8931.077 | 8664.416 |
| speedup | 1 | 1.509 | 1.555 |
| 180b | 24785.01 | 16063.286 | 13012.87 |
| speedup | 1 | 1.543 | 1.905 |
| 240b | 34799.795 | 22961.342 | 18165.933 |
| speedup | 1 | 1.516 | 1.916 |

## IV. CONCLUSION

In this project, we implement 4 programs to solve the N-body problem, tried to use several ways to speedup the execution and parrallelize them. Make program run concurrently is not a simple thing, we need to consider many aspects, including race condition and memory sharing and message distribute.