N-Body Simulation Project Report

Duc Nguyen, Tu Nguyen

June 11, 2024

Abstract

This report documents the implementation and analysis of an N-body simulation project. The project simulates the dynamics of several bodies under gravitational forces in a 2D space. We implemented both a straightforward simulation algorithm and the Barnes-Hut tree algorithm as a way of improving the running time. The report encompasses the implementation, parallelization, and visualization of the simulation, along with a study of the performance of the running time of the algorithms implemented.

1 Overview and Introduction

The Github repository of the project can be found here: `https://github.com/Combi2k2/CSE305-3BodyProblem`. In the repository there are the main code of the project which is stored in the src and include folder and benchmark plots which are stored in the benchmark plot folder.

The N-body problem involves predicting the individual motions of a group of celestial objects interacting with each other gravitationally. This project focuses on simulating these interactions in a 2D space, using Newton's laws of motion and universal gravitation. The challenge lies in efficiently computing the forces and updating the positions of the bodies, especially as the number of bodies increases.

2 Straightforward Simulation Algorithm

The straightforward algorithm calculates the gravitational force exerted on each body by every other body and updates their positions accordingly.

2.1 Algorithm Description

For each body b_i with mass m_i, position (x_i, y_i), and velocity (u_i, v_i), we compute the force $f_{i,j}$ between b_i and every other body b_j using:

$$f_{i,j} = G\frac{m_i m_j}{(x_i - x_j)^2 + (y_i - y_j)^2}$$

where G is the gravitational constant. The net force on b_i is the vector sum of all pairwise forces. We then update the velocities and positions based on these forces using a user-specified timestep Δt:

$$u_i = u_i + \frac{f_i}{m_i}\Delta t, \quad v_i = v_i + \frac{f_i}{m_i}\Delta t$$

$$x_i = x_i + u_i\Delta t, \quad y_i = y_i + v_i\Delta t$$

2.2 Parallelization of the Straightforward Algorithm

To optimize the straightforward algorithm, we implemented parallelization using multi-threading. The idea is to divide the bodies into group, and each thread will be in charge of computing the acceleration acting on each particle of its group. After all the forces are calculated, the positions of the bodies are updated sequentially. In short, we will only parallelize the process of computing the force, which is the most time consuming step in the simulation. We calculate the acceleration instead of the force in order to avoid multiplying the mass multiple times, which helps reduce running time.

We created a function `computeThread` to compute gravitational forces for a subset of bodies. This function calculates the acceleration exerted on each body in the subset by all other bodies. The subset is defined by start and end indices, making it easy to parallelize the workload in the main update function. This idea is taken from lab 1 and 2 of the course.

The `update` function is used for the parallel computation. It divides the bodies into blocks, each handled by a separate thread, which use the `computeThread` above and launches these threads to compute accelerations concurrently. After the accelerations are computed, the function updates the positions and velocities of the bodies based on the accelerations.

3 Initialization and Simulation Output

After implementing the parallelization of the force computation and position update, we move on to the simulation of the bodies. We first initialize the bodies, then use the update function to extract the positions of the bodies after each timestep. We then save these positions in a txt file and use that to get the simulation and analyse the running time.

3.1 Initialization

In our project, we have 2 types of initialization functions. The first (initialize random) is the generalized version, which randomly generates the bodies where the amount of bodies is put into the function. Secondly, we have 2 specific twist to the N-body simulation problem, which are the 4 galaxies version and the solar system simulation version, with the initialize functions initialize 4galaxies and initialize solarsystem accordingly. Using these initialization, we can simulate these 2 real natural phenomena.

In each of these 3 initialization functions above, we initializes each body's mass, radius, position, and velocity according to the desired output. The variables are randomly generated in the function initialize random to ensure a diverse distribution of bodies within the simulation space and generated using real natural data in the other 2 to ensure the movement of the bodies reflect the real life movement of the galaxies and the planets.

3.2 Simulation Output

The `main` function initializes the bodies, runs the simulation for a specified number of steps, and outputs the positions and velocities of the bodies at each step. This allows us to track the evolution of the system over time and analyze the results.

4 Barnes-Hut Algorithm

The Barnes-Hut algorithm optimizes force computation by grouping nearby bodies and approximating them as a single body. In our project, we will implement the idea of the Barnes-Hut algorithm described in the paper `http://arborjs.org/docs/barnes-hut`.

4.1 Algorithm Description

The algorithm uses a quad-tree structure to recursively divide the space into quadrants. In our project, we represent each region in the space by a box (TreeNode) and stores the number of particles (nParticles), the center of mass (COM), the position of the center of the region (box center), the size of the box (box size) and the list of 4 child quadrants of that region (children).

These above parameters allow us to define 2 important methods of the TreeNode structure, which are insertion and compute the force exerted on a particle, which is the main usage of the tree (query).

4.1.1 Constructing the Barnes-Hut Tree

Bodies are inserted into the tree one by one and we traverse from the root node of the tree. If the node is empty, or nParticles = 0, we simply put the particle in the node and update the number center of mass. If the node already contains 1 body (external node), the region is subdivided, and both bodies, the previous body and the new body, are recursively inserted into the appropriate quadrants. If the node is already divided into 4 quadrants which are not all empty (internal node), the new body is simply put in the appropriate quadrant and the node itself is updated to reflect the new center of mass and total mass.

4.1.2 Calculating Forces

First, we build a simple function used in the straightforward algorithm, which calculate the acceleration exerted on a body by another body (`computeAcceleration`).

To calculate the force on a body in the Barnes-Hut algorithm, we traverse from the root node of the tree and there are 2 cases:

- If the current node is an external node, we simply use the function `computeAcceleration` to calculate the acceleration as the effect of the node on the subject body.
- If the node is an internal node (there are at least 2 bodies in the tree rooted at the current node), then we check if the node is "far enough" from the subject body, using the threshold parameter θ. We calculate the ratio $\frac{s}{d}$ where s is the node width and d is the distance from the center of mass of the node to the body.
  - If $\frac{s}{d} < \theta$, the node is far enough and we consider the node as a body represented by its center of mass `COM` and use the function `computeAcceleration` to compute the acceleration exerted on the subject body.
  - If $\frac{s}{d} \geq \theta$, the node is not far enough and we have to traverse through each children of the current node and update the acceleration recursively. After all children nodes are traversed, we sum up the acceleration by the x-axis and the y-axis to get the total acceleration of the node.

In our project, to avoid calling the tree recursively, we decide to use a stack to store the traversed node and do a depth first search inspired run on all the nodes of the tree. The acceleration is fully updated when the stack is empty.

4.1.3 Parallelization of the Barnes-Hut algorithm

In the Barnes-Hut algorithm, we concurrently insert bodies into the tree and concurrently compute the acceleration in case of using multiple thread.

For insertion, we use a lock to lock the node that we are considering to put a body in. By doing this way, we can avoid data racing among threads and allow bodies which belongs to different children of a node to be inserted concurrently.

For computing acceleration, we use the same approach as in the parallelization of the straightforward algorithm. We divide the bodies into groups, and assign each thread to compute the acceleration exerted on the bodies within its group.

5 Rendering the simulation

To generate a window of simulation, we use `Magick++` and `SDL2` libraries:

- `Magick++` gives the image data of one logical state (positions and radius of bodies).
- `SDL2` receives the image data and renders it on the window at real time.

5.1 Implementation details

`renderSimulation` function: This takes in 3 parameters:

- `std::vector<Body>` constant reference,
- `Magick::Image`'s reference

This function will fills the background with white color, and sets the stroke color to black. It then enters a loop to draw a circle for each body in the simulation. The circle's center coordinates are calculated based on each body's position and scaled using the scaling factor. Finally, it draws a circle with a radius equal to each body's radius at its calculated position.

5.2 Parallelization of rendering process

We notice a chance for optimizing the rendering process:

- One thread (1) for logical computation: Updating `std::vector<Body>` after each step,
- One thread (2) for drawing the image data given the correspond `std::vector<Body>`,
- One thread (3) for rendering the image data on the window.

This goal can be obtained using a thread-safe queue following the producer-consumer model. This model separates the problem into two parts: the producer, which generates data, and the consumer, which uses the data. This separation allows data to be produced and consumed asynchronously, which in our case means asynchronous simulation and rendering.

We developed a `ThreadSafeQueue` class template that employs a mutex for synchronization.

As thread (1) and (2) simulate one producer-consumer problem, and thread (2) and (3) simulate one, we used the `ThreadSafeQueue` as follow:

- `ThreadSafeQueue` (1) stores logical data of the simulation that require rendering, provided by thread (1), and consumed by thread (2),
- `ThreadSafeQueue` (2) stores image data produced by thread (2), and rendered on the window by thread (3).

Thread (1) and (2) will push new data into their correspond queue whenever they finish computing the data. And thread (3) will consume the image data, render on window at a fixed intervals.

6 Performance and Efficiency

6.1 Technical Detail

Core(s): 4
Thread(s): 8
Socket(s): 1
NUMA node(s): 1
Model name: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz

6.2 Constants used in simulations

Constant	Value	Description
DEFAULT_N	10000	Number of bodies
DEFAULT_TIME	50	Number of iterations
G	1.0	Gravitational constant, m^3 kg^{-1} s^{-2}
XBOUND	1.0e6	Width of space
YBOUND	1.0e6	Height of space
RBOUND	10	Upper bound on radius
DELTAT	0.01	Time increment
THETA	0.5	Threshold in BH Tree
NDIM	2	Dimension of space

Table 1: List of constants used in the project

In order to have diverse comparision in the efficiency of the algorithms, we simulate with a range of number of bodies: 2, 50, 500, 1000, 5000, 10000 and for multi-thread simulation, we use the number of thread from 1 to 10.

6.3 Straightforward algorithm and parallelization benchmark

Threads	2	50	500	1000	5000	10000
1	2e-06	0.000426	0.040321	0.156043	4.30371	17.8043
2	0.002915	0.004633	0.028392	0.111486	2.23824	10.7874
3	0.002597	0.004887	0.024244	0.090948	1.73967	8.99601
4	0.004527	0.006857	0.026632	0.080721	1.63472	8.02614
5	0.004177	0.008717	0.022395	0.067154	1.40085	7.50173
6	0.002894	0.010888	0.023592	0.060742	1.56123	8.11619
7	0.00278	0.011977	0.024751	0.059837	1.59785	7.4944
8	0.002932	0.015722	0.028084	0.09626	1.617	6.73321
9	0.00303	0.027007	0.031023	0.111609	1.80472	7.37573
10	0.003951	0.026306	0.033627	0.099434	1.80747	6.81549

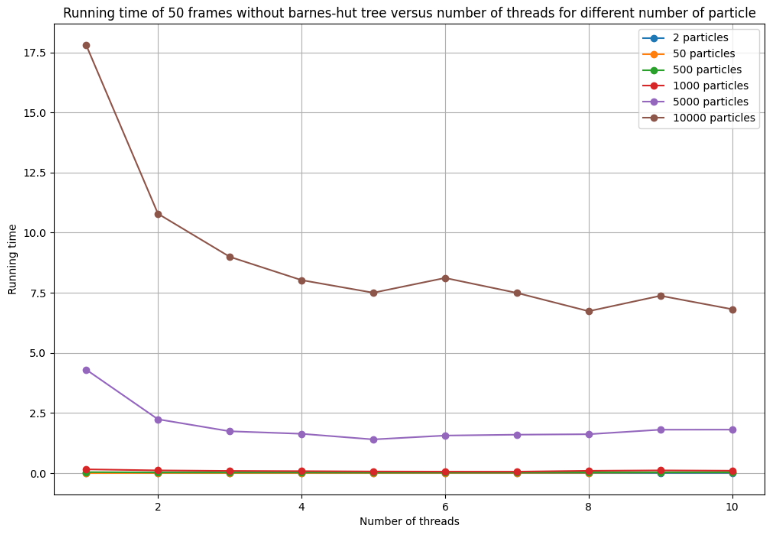

Figure 1: Straightforward algorithm running time of 50 frames

We can check that for the number of bodies to be small such as 2, 50, and 100, there is hardly any effect of applying multi-thread in the running time of the simulation, which is the same phenomenon as in lab 3 for the calculation of the sum where the size of the array is small. The effect is much clearer for high number of bodies, as in nParticles = 5000 or 10000 where the running time almost halved by using 2 threads and gradually get faster when using more threads. The best number of thread is around 5 in this case.

6.4 Barnes-Hut algorithm benchmark

When simulating the N-bodies with Barnes-Hut tree, we test the running time for 3 different values of θ, the threshold to decide if the node is far enough from the subject body, a default θ with value 0.5 and 2 others θ with values 1 and 2.

6.4.1 Default theta ($\theta = 0.5$)

Threads	**2**	**50**	**500**	**1000**	**5000**	**10000**
1	1.7e-05	0.001488	0.022246	0.060249	0.376483	0.928514
2	0.008477	0.009044	0.030843	0.05758	0.283518	0.625817
3	0.007955	0.011516	0.022497	0.045374	0.253917	0.980059
4	0.009071	0.013021	0.023577	0.042801	0.292032	0.640643
5	0.006663	0.017935	0.023815	0.04797	0.287478	0.685229
6	0.010239	0.022708	0.031208	0.044886	0.320232	0.884894
7	0.008135	0.032137	0.030606	0.04734	0.461905	1.04765
8	0.010318	0.042662	0.035858	0.049427	0.300958	0.929662
9	0.00779	0.050056	0.056621	0.060912	0.427927	0.866439
10	0.009038	0.042841	0.066263	0.083136	0.465345	1.06699

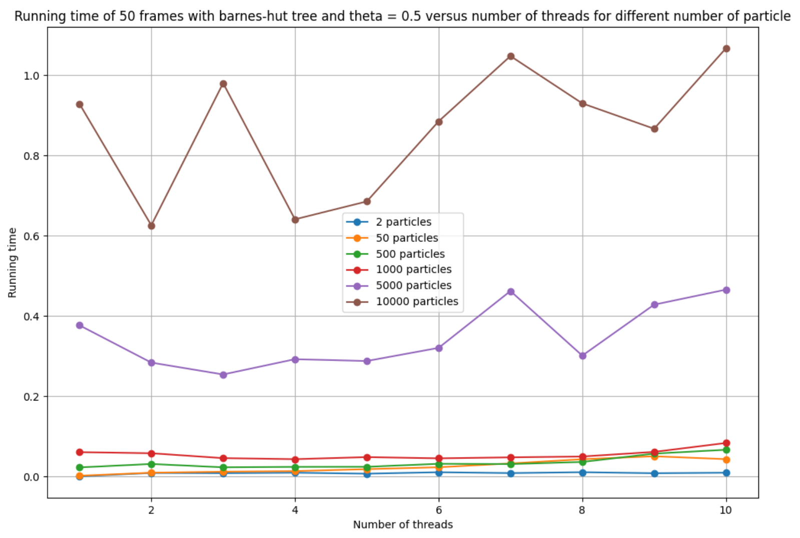

Figure 2: Barnes-Hut algorithm running time of 50 frames with $\theta = 0.5$

6.4.2 Theta 1 ($\theta = 1$)

Threads	**2**	**50**	**500**	**1000**	**5000**	**10000**
1	1.7e-05	0.001488	0.022246	0.060249	0.376483	0.928514

2	0.008477	0.009044	0.030843	0.05758	0.283518	0.625817
3	0.007955	0.011516	0.022497	0.045374	0.253917	0.980059
4	0.009071	0.013021	0.023577	0.042801	0.292032	0.640643
5	0.006663	0.017935	0.023815	0.04797	0.287478	0.685229
6	0.010239	0.022708	0.031208	0.044886	0.320232	0.884894
7	0.008135	0.032137	0.030606	0.04734	0.461905	1.04765
8	0.010318	0.042662	0.035858	0.049427	0.300958	0.929662
9	0.00779	0.050056	0.056621	0.060912	0.427927	0.866439
10	0.009038	0.042841	0.066263	0.083136	0.465345	1.06699

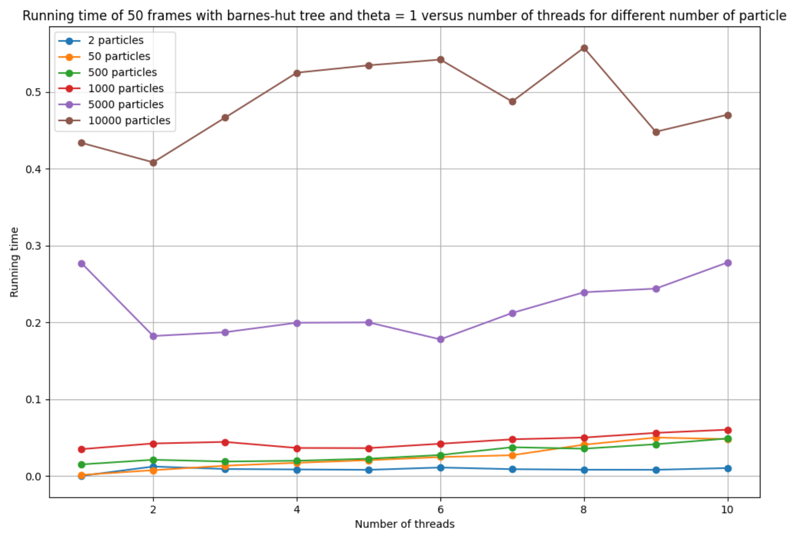

Figure 3: Barnes-Hut algorithm running time of 50 frames with $\theta = 1$

6.4.3 Theta 2 ($\theta = 2$)

Threads	**2**	**50**	**500**	**1000**	**5000**	**10000**
1	9.8e-05	0.001635	0.010702	0.022679	0.135903	0.352444
2	0.022362	0.008177	0.021372	0.039194	0.146922	0.342867
3	0.014498	0.014751	0.016237	0.029803	0.141832	0.345441

4	0.013039	0.016894	0.017054	0.031446	0.133186	0.377271
5	0.011866	0.021311	0.019037	0.03313	0.138627	0.453775
6	0.006278	0.039319	0.023977	0.040664	0.148711	0.415626
7	0.010357	0.033628	0.028092	0.03923	0.161773	0.403772
8	0.009179	0.033071	0.032116	0.04748	0.18578	0.412995
9	0.009026	0.041143	0.037937	0.043931	0.183906	0.606356
10	0.007922	0.053506	0.048682	0.059494	0.219278	0.686188

We can check that with the Barnes-Hut algorithm, the effect of multi-threading almost disappear as there is no improvement in the running time as the number of thread increase.

6.4.4 Theta comparision

Theoretically, the higher the value of θ, the lower the running time regardless of the number of bodies and the number of threads, as there is higher chance a node is far enough from the subject value and the acceleration can be computed straightaway instead of having to traverse to the 4 child quadrants. We will test if the hypothesis is correct or not.

For each value of theta (including the straightforward algorithm where theta can be considered 0), we plot the running time using a single thread and 10 threads for objective judgement

We an check that for small number of bodies, there is little effect. However, as the number of bodies increases, the running time gets lower when we increase the value of theta.

Moreover, the effect of Barnes-Hut algorithm is very clear, as the gap between the black line (representing the straightforward algorithm) and the other 3 colors (representing the running time using Barnes-Hut) is large.

7 Conclusion

After benchmarking the simulation with both algorithm and multiple-threading, we can notice a few things about the running time.

- **Overall:** The running time increases quadratically with the number of bodies in almost any cases, due to the fact that the complexity of the straightforward update function is O(n^2)

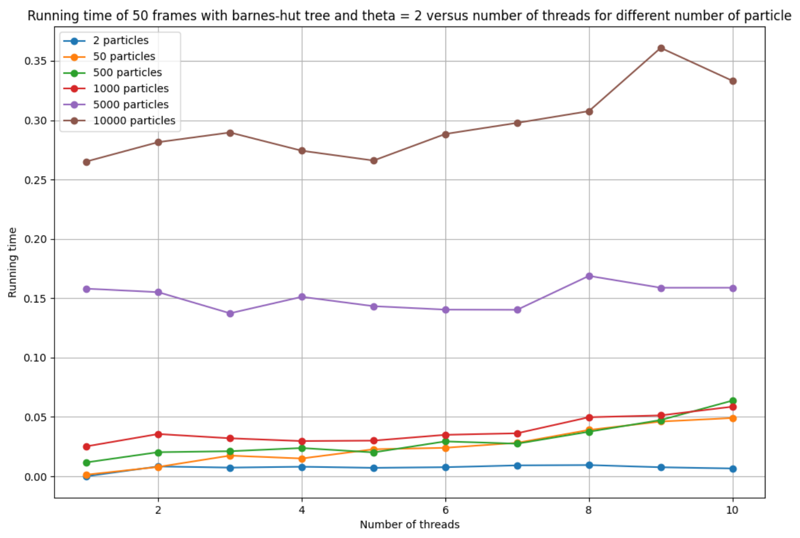

Figure 4: Barnes-Hut algorithm running time of 50 frames with $\theta = 2$

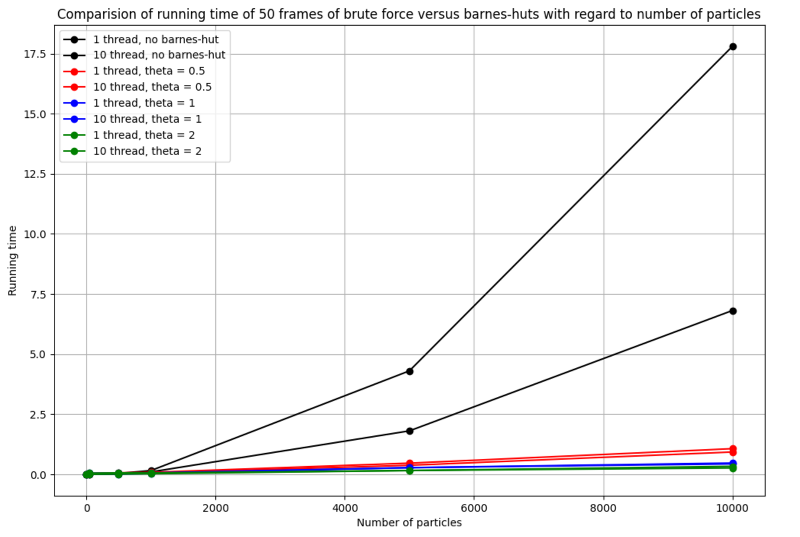

Figure 5: Barnes-Hut algorithm running time of 50 frames with regard to θ

- **Straightforward versus Multithreaded:** .The running time decreases as the number of threads increases, which is what we expected and prove that parallel computing works. As can be seen in lab 3 when we have to compute the sum of an array, the decrease of the running time is not linear due to the extra time required to manage multiple threads. As can be seen from the graphs, there are cases where the running time decrease a lot whereas in other cases, the effect of multi-threading is minimally.
- **Straightforward versus Barnes-Hutt Algorithm:** The running time of the Barnes-Hutt algorithm, in general, decreases as the value of theta increases. Using the Barnes-Hut has a huge impact on the running time as it can enormously reduce the running time of the straightforward algorithm, depending on the number of bodies simulated, the more number of bodies simulated, the more the power of the Barnes-Hut algorithm is showcased.