THE BARNES-HUT ALGORITHM FOR GRAVITATIONAL N-BODY PROBLEM

High Performance Programming

Rafael Rodriguez Velasco

VT 2024

Contents

1 Introduction **1**

2 Problem description **2**

2.1 Complexity analysis 2

2.2 Integration methods 2

2.3 Parameters 3

3 Solution method **4**

3.1 Tree assembly 5

3.2 Force calculation 6

3.3 Main program 9

3.4 Parallelization 10

4 Experiments **12**

4.1 Election of `theta_0` and Complexity comparison 12

4.2 Integration error 13

4.3 Parallel performance 14

4.4 Performance optimization 15

5 Conclusions **17**

References **18**

1 Introduction

This report aims to describe the implementation of a simulation for the gravitational N-body problem in two dimensions, addressing the computational challenges of the problem and the solution method used to overcome them. More precisely, the report will focus on the Barnes-Hut algorithm, and how to achieve the best possible performance in contrast to more naive approaches.

The gravitational N-body problem is a classical problem in physics and astrophysics, and it consists of predicting the motion of a group of celestial bodies under the influence of their mutual gravitational forces. In this report, I use Newton's law of universal gravitation with Plummer's model modification to handle instabilities at small distances ($r_{ij} << 1$), which states that the force acting on the i-th body is given by:

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

where r_{ij} is the distance between the i-th and j-th bodies, m_i and m_j are their masses, and ϵ_0 is the Plummer radius, a small constant to avoid singularities at small distances. The gravitational constant G is set to $\frac{100}{N}$, with N the number of bodies in the simulation [1].

The simulation involves N bodies in a 1×1 dimensionless domain, with initial positions and velocities read from a file. I will explore two different integration methods: Symplectic Euler and Velocity Verlet. The simulation will be run for a fixed number of time steps, and the final positions and velocities of the bodies will be written to a file.

The Barnes-Hut algorithm is a hierarchical algorithm that approximates the forces acting on each body by recursively grouping distant bodies into a single body by using their center of mass. It uses a structure called *quadtree* to divide the domain into four quadrants and assigns bodies to the appropriate quadrant. The algorithm is particularly useful for the gravitational N-body problem, as it allows for a significant reduction in the number of force calculations required to simulate the system. Figure 1 illustrates the quadtree structure used in the Barnes-Hut algorithm [2].

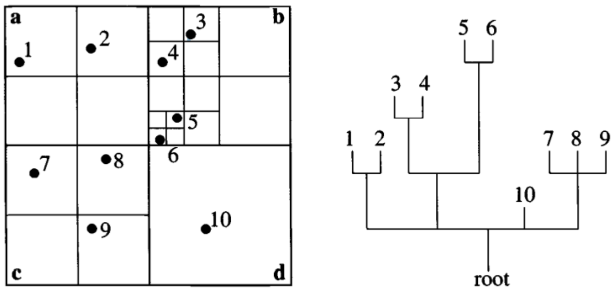


Figure 1: Quadtree structure used in the Barnes-Hut algorithm (Susanne Pfalzner and Paul Gibbon, 1997).

2 Problem description

2.1 Complexity analysis

If one were to calculate the force acting on each body by directly summing the forces from all other bodies (see equation (1)), the complexity of the problem would be $O(N^2)$. Even when taking into consideration Newton's third law, which states that the force exerted by body i on body j is equal in magnitude and opposite in direction to the force exerted by body j on body i, to implement it we would need an outer loop over all the bodies i and an inner loop over the bodies j from $i+1$ to N. This implies that the total number of force calculations is some multiple of

$$\underbrace{(N-1)+(N-2)+...+1+0}_{N \text{ terms}} = \sum_{k=1}^{N-1} k = \frac{(N-1)N}{2} = \frac{1}{2}(N^2 - N)\,, \tag{2}$$

which is $\mathcal{O}(N^2)$. This is clearly not feasible for large N, and it is the reason why the Barnes-Hut algorithm is used to reduce the number of force calculations required to simulate the system.

The Barnes-Hut algorithm relies on the subdivision of the domain into quadtrees, and thus its complexity can be estimated by knowing how many divisions are needed to reach the average cell. The average size of a cell can be assumed to be of the order of the interparticle spacing, so that the average area is A/N, with A the total area of the domain. In addition, the average length of a side of a cell must be the length of the side of the domain halved by the number of divisions, e.g.

$$\left(\frac{A}{N}\right)^{1/2} = \frac{A^{1/2}}{2^h}\,, \tag{3}$$

which implies that the height h of the tree is of order

$$h = \log_2 N^{1/2} = \frac{1}{2}\log_2 N\,. \tag{4}$$

This way, an average of $\mathcal{O}(\log N)$ force calculations is required to reach a leaf, and since there are N leaves, the complexity of the Barnes-Hut algorithm is $\mathcal{O}(N \log N)$ [2][3]. Thus, the Barnes-Hut algorithm is a significant improvement over the direct summation method, especially for large N. However, the algorithm is not without its own challenges, as the construction of the quadtree and the calculation of the forces require a significant amount of memory and computational resources, and it approximates the forces acting on each body, so it is not exact.

2.2 Integration methods

I will discuss and compare two different integration methods to solve the equations of motion for the bodies in the simulation: Symplectic Euler and Velocity Verlet. They are particularly useful for the gravitational N-body problem, as they are symplectic methods, which among other properties are time-reversible and conserve energy. Symplectic Euler is defined by the following update equations:

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + a(t)\Delta t \, , \tag{5}$$
$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t+\Delta t)\Delta t \, , \tag{6}$$

where $\mathbf{r}_i$ and $\mathbf{v}_i$ are the position and velocity of the i-th body, and $a(t)$ is the acceleration of the body. The Velocity Verlet method on the other hand is defined by:

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 \, , \tag{7}$$
$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \frac{a(t)+a(t+\Delta t)}{2}\Delta t \, . \tag{8}$$

I will not derive the order of accuracy of these methods, but one can convince oneself that the Velocity Verlet method is more accurate than the Symplectic Euler method, as the position of the former is Taylor expanded to second order, and the velocity is updated using a midpoint step. It turns out indeed that Velocity Verlet's error behaves as $\mathcal{O}(\Delta t^2)$, while Symplectic Euler's error behaves as $\mathcal{O}(\Delta t)$.

2.3 Parameters

The Barnes-Hut algorithm relies on a parameter θ to determine the accuracy of the force calculation. The parameter θ is a measure of the ratio of the size of the cell s to the distance between the cell's center of mass and the body d for which the force is being calculated. If the relation

$$\theta = \frac{s}{d} \leq \theta_0 \tag{9}$$

is satisfied, where θ_0 is a tolerance parameter, then the internal structure of the cell is ignored and the force is approximated by its center of mass. As a consequence, if θ is small, the algorithm will be more accurate, but it will require more force calculations. If θ is large, the algorithm will be less accurate, but it will require fewer force calculations. Figure 2 illustrates the meaning of the parameter θ [2].

Thus, for the task of achieving the best possible performance, the parameter θ is a key parameter to tune, since it determines the trade-off between accuracy and computational cost. In addition, the number of time steps and Δt are also important parameters to consider, as for a given time of simulation $T = \texttt{nsteps} \times \Delta t$, a bigger Δt will require fewer force calculations, but it will be less accurate.

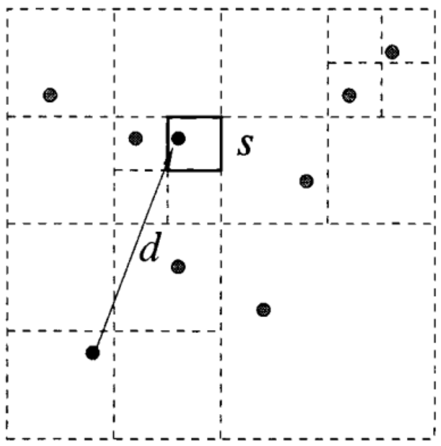


Figure 2: Illustration of the parameter θ in the Barnes-Hut algorithm (Susanne Pfalzner and Paul Gibbon, 1997).

3 Solution method

Here I will describe and explain each section of the code. It mainly follows the structure indicated by Tom Ventimiglia & Kevin Wayne [4], with some inspiration from the work of Kim Torberntsson [5].

The code starts by defining ϵ_0 since it will remain constant and declaring global variables that will be used by the different functions. Then, the structure for the quadtree is defined with the following members:

- `particle_index`: The index of the particle in the cell.
- `is_leaf`: A boolean that indicates whether the cell is a leaf of the tree.
- `x, y, width`: Coordinates of one corner of the cell and its width.
- `node_mass`: The total mass of the bodies in the cell.
- `cm_x, cm_y`: The center of mass of the bodies in the cell.
- `children`: A list of pointers to the child cells.

The use of a bool type for `is_leaf` instead of an integer reduces the memory usage of the program, as it only requires one byte instead of four. The use of a list of pointers for the children of the cell allows for a more flexible implementation, as the number of children is not fixed and can be changed easily.

The code is divided into three main sections: tree assembly, force calculation, and the main program where the simulation is run. This code uses many auxiliary functions to build the quadtree, which could potentially be inefficient due to successive calls. However, this is fixed by the compiler flag `-O3`, which allows the compiler to inline the functions.

3.1 Tree assembly

In this section, the functions to build the quadtree and fill it with particles are defined. This section starts by creating a function that creates and initializes the nodes.

Algorithm 1: `insert`: Creation of nodes

```
Data: The dimensions of the cell
Result: The node is created and initialized
1  Allocate memory for the node
2  Set the node's coordinates and width
3  for index 0 to 3 do
4      Set children[index] to NULL
5  end
```

For inserting the particles in the tree, the strategy is the following:

Algorithm 2: `insert`: Insertion of particles in the quadtree

```
Data: The node, the particle to insert and the size of the cell
Result: The particle is inserted in the quadtree
1  if there is no node then
2      Create it using create_node
3      Set particle_index to the one of the particle
4      Set the mass and center of mass of the node to the mass and position of the particle
5      Set is_leaf to 1
6  end
7  else
8      if the node is a leaf then
9          Set is_leaf to false
10         Use the function get_size to get the size of the cell and the index of the correct
             child
11         Use insert recursively to insert the old particle in the corresponding child
12     end
13     Use the function get_size to get the size of the cell and the index of the correct
         child
14     Use insert recursively to insert the new particle in the corresponding child
15     Use update_mass_and_cm to update the mass and center of mass of the node
16 end
```

This function is called for each particle in the simulation, and it is used to build the quadtree. It relies on the auxiliary functions `update_mass_and_cm` and `get_size` to update the mass and center of mass of the node and to obtain the correct size and child index where the particle should be inserted, respectively.

An unoptimized previous version of the code subdivided the node into four children every time a particle was inserted, which was inefficient as not all nodes were occupied by particles. The current version of the code only allocates memory for the children where the particle is inserted, which reduces the memory usage of the program.

The function `get_size` uses the following logic

Algorithm 3: `get_size`: Calculates the right dimensions and index of the child where the particle should be inserted

```
Data: The node to subdivide, the particle to insert and pointers to an index variable
      and an array of dimensions
Result: The values of the dimensions and the index of the child are updated
1 Calculate the midpoint of the node
2 Calculate the midpoint of the node
3 Decide the index of the child based on the position of the particle
4 Decide the attributes x and y of the child based on the index of the child
```

Instead of making multiple comparisons, the above function gets the index by compactly using the following expression:

```
int index = (pos_x[particle] >= mid_x) *2 | (pos_y[particle] >= mid_y)
```

which compares the x and y positions of the particle with the midpoints of the cell and returns the index of the child by cleverly using the bitwise OR operator.

The function `update_mass_and_cm` is used to update the mass and center of mass of the node.

Algorithm 4: `update_mass_and_cm`: Update of the mass and center of mass of the quadtree

```
Data: The node to update and the inserted particle
Result: The mass and center of mass of the node are updated
1 Calculate total mass by summing the node's mass and particle's mass
2 Calculate the center of mass by multiplying the previous node mass by the previous
    center of mass, adding the particle's mass times the particle's position, and then
    dividing by the total mass
3 Update the mass and center of mass of the node
```

Finally, a small function `free_tree` is defined to free the memory of the quadtree. It is a simple recursive function that frees the memory of the children of the node.

3.2 Force calculation

In this section, the functions to calculate the forces acting on each body are defined. Before delving into the `update_forces` function, the auxiliary function `get_r` is defined to calculate

Algorithm 5: `free_tree`: Free the memory of the node and its children

```
Data: Double pointer to the node to free
Result: The memory of the children is freed
1 if the node is NULL then
2     End
3 end
4 if the node is not a leaf then
5     for each child do
6         Use free_tree recursively to free the memory of the children
7     end
8 end
9 Free the memory of the node
```

the distances between a particle and the center of mass of a node. This will help to improve readability and reduce the number of repeated calculations.

Algorithm 6: `get_r`: Calculates the distances between the particle and the center of mass of the node

```
Data: Pointer to the node and an array where distances are stored, and the particle
      for which the force is being calculated
Result: The distances are stored in the array
1 Calculate the x component of the distance
2 Calculate the y component of the distance
3 Calculate the norm of the distance
4 Store the distances in the array
```

The function `update_forces` is defined as follows:

Algorithm 7: `update_forces`: Updates the forces acting on the particles

```
Data: The node and the particle for which the force is being calculated
Result: The force acting on the particle is updated
1  Create an array to store the distances between the particle and the center of mass of
     the node
2  Use get_r to calculate the distances
3  if the node is a leaf then
4      Use calculate_force to calculate the force
5  end
6  else
7      Calculate $\theta$;
8      if $\theta$ is less than the threshold $\theta_0$ then
9          Use calculate_force to calculate the force
10     end
11     else
12         for each not NULL child do
13             Use update_forces recursively to calculate the force acting on the particle
                 using the node's children
14         end
15     end
16 end
```

This function is called for each particle in the simulation, and it is used to calculate the forces acting on each body. It relies on the auxiliary function `calculate_force` to calculate the force acting on the particle. The function `calculate_force` simply calculates the force acting on the particle using equation (1) and updates the force acting on the particle.

```c
void calculate_force(int particle_index, node_t* node) {
    double r_x = pos_x[particle_index] - node->cm_x;
    double r_y = pos_y[particle_index] - node->cm_y;
    double r_squared = r_x * r_x + r_y * r_y;
    double r_plummer = sqrt(r_squared) + EPSILON_0;
    double r_cubed = r_plummer * r_plummer * r_plummer;
    double force_factor = -G * mass[particle_index] * ...
                                    * node->node_mass / r_cubed;

    fx[particle_index] += force_factor * r_x;
    fy[particle_index] += force_factor * r_y;
}
```

Note the use of repeated multiplication instead of division to calculate the force factor, which is a common optimization technique.

3.3 Main program

Before delving into the time step calculation, the main function first reads six command line inputs:

- `N`: The number of bodies in the simulation.
- filename: The name of the file containing the initial positions, velocities, masses, and brightness of the bodies.
- `nsteps`: The number of time steps to run the simulation.
- `delta_t`: The time step of the simulation.
- `theta_0`: The parameter θ of the Barnes-Hut algorithm.
- `processes`: The number of threads to use in the parallelization.

The command line inputs allow easy testing of the program with different parameters and initial conditions. The program then allocates memory for the arrays that will store the positions, velocities, masses, and forces of the particles, since the functions previously defined require these arrays to be globally defined, but their size depends on the input N.

Moreover, four extra arrays are defined using stack memory to store the inverse of masses, accelerations and brightness. This is done since stack memory is faster than heap memory, and the arrays are small enough to fit in the stack. The inverse of masses is pre-calculated to avoid extra divisions in the force calculation since the mass value will remain constant throughout the simulation. The brightness array is used to store the brightness of the particles, which is not used in the simulation, but it is written in the output file.

The program then reads the initial conditions from the file and initializes the quadtree. The quadtree is built using the `insert` function, and the forces acting on the particles are calculated using the `update_forces` function. The forces are then used to update the velocities and positions of the particles using the chosen integration method. The program then writes the final positions, velocities, and brightness of the particles to a file.

Finally, the program iterates over time steps to simulate the galaxy. Depending on the chosen integration method, the structure of the code changes slightly since Velocity Verlet method is a bit more complex than Symplectic Euler, requiring information of both $a(t)$ and $a(t+\Delta t)$. The structure of both methods is shown in algorithms 8 and 9 respectively. For the case of Velocity Verlet, $\frac{1}{2}\Delta t^2$ is precomputed to avoid redundant multiplications in the position update.

As commented before, the Velocity Verlet method requires also information on the acceleration at the next time step. A naive code implementation would require calculating the forces twice, thus making the code twice as slow. However, it is enough to calculate the force just one extra time before the loop starts, as shown below.

Algorithm 8: Symplectic Euler method

```
1  for each time step do
2      Create the root pointer to NULL
3      for each particle do
4          Set the force acting on the particle to zero
5          Use insert to insert the particle in the quadtree
6      end
7      for each particle do
8          Use update_forces to update the forces acting on the particle
9      end
10     for each particle do
11         Update the velocity of the particle using the force acting on the particle using
            equation (5)
12         Update the position of the particle using the velocity previously calculated
            (equation (6))
13     end
14     Free the memory of the quadtree
15 end
```

After the simulation is complete, the code stores all the resulting positions and velocities in a result file.

3.4 Parallelization

I chose to parallelize the code using OpenMP, as it is a simple and effective way to parallelize loops.

Due to the way the particles are inserted in the tree, parallelization is challenging since multiple threads could try to insert a particle in the same node at the same time. One could try to solve this by using locks, but this would introduce a significant overhead. In addition, if one inspects the performance of the code (with *Operformance analyzer*, for example) it is clear that the time spent in the `insert` function is negligible compared to the time spent in the `update_forces` function. Thus, I chose to parallelize the `update_forces` function, which is the most time-consuming part of the code.

It turns out that the `update_forces` function is a perfect candidate for parallelization, as the forces acting on each particle are independent of each other once the tree is already built. A line with the number of threads to use is added at the beginning of the main function, which may be changed conveniently depending on the number of cores of the machine. This way, the code is parallelized by adding

```
#pragma omp parallel for
```

before the loops that iterates over the particles in the `update_forces` function. This line tells the compiler to parallelize the loop, and the compiler will automatically distribute the iterations

Algorithm 9: Velocity Verlet method

```
Initialize the root to NULL
for each particle do
    Set the force acting on the particle to zero
    Use insert to insert the particle in the quadtree
end
for each particle do
    Use update_forces to update the forces acting on the particle
end
Free the memory of the quadtree
for each time step do
    for each particle do
        Calculate and store the acceleration values with the previously calculated force
        Update positions using equation (7)
    end
    Allocate memory for the root of the quadtree
    Initialize the root
    for each particle do
        Set the force acting on the particle to zero
        Use insert to insert the particle in the quadtree
    end
    for each particle do
        Use update_forces to update the forces acting on the particle
    end
    for each particle do
        Use update_forces to update the forces acting on the particle
    end
    for each particle do
        Update the velocity of the particle using the force acting on the particle using
          equation (8)
    end
    Free the memory of the quadtree
end
```

of the loop among the available threads.

It uses by default static scheduling, which is the most efficient for this problem, as the iterations of the loop are independent of each other and have a similar cost. It allows for some overhead reduction and better cache utilization than dynamic scheduling. The chunk size could be a parameter to set and experiment with, but since the program may be run with different numbers of particles, it is better to let the compiler decide the chunk size.

4 Experiments

In this section, I will describe the experiments I performed to show the correctness and performance of the code. I will compare the performance of the Barnes-Hut algorithm with the direct summation method, and I will also compare the asymptotic behavior of the error when using Symplectic Euler and Velocity Verlet methods.

All the time measurements were based on the time taken to run the entire executable, compiled with `gcc` version 13.1.0, with the `-O3` and `-ffast-math` flags to optimize the code. The code was run on a machine with an Intel Core i5-8250U CPU @ 1.60GHz with 4 cores (8 logical processors) and 8 GB of RAM.

To debug the code and check its correctness, I used an auxiliary code that compares the maximum difference in position of particles `posmaxdiff` between two result files (the ones written at the end of the simulation).

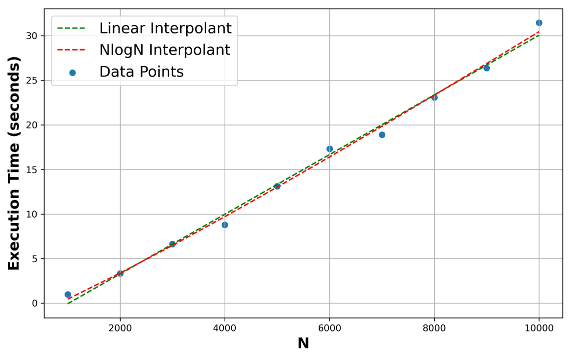
4.1 Election of `theta_0` and Complexity comparison

The first validation test is to check if the Barnes-Hut algorithm is correctly implemented. For this, the code was run with Euler integration and $\theta_0 = 0$ and the posmaxdiff was checked to be identical to the direct summation method.

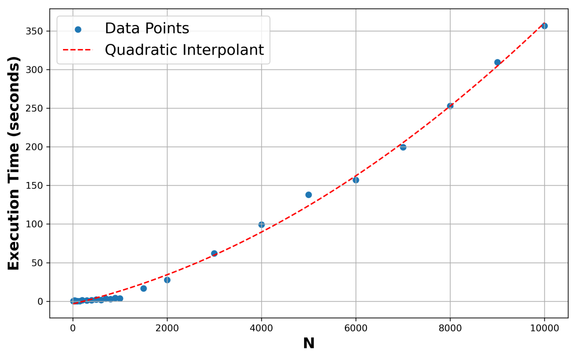
Another way to check if the code is correctly implemented is by checking the behavior of the execution time and the position errors `posmaxdiff` as a function of the parameter θ_0. The execution time is expected to decrease as θ_0 increases, as the number of force calculations required to simulate the system decreases. However, the position errors are expected to increase as θ_0 increases, as we are approximating the forces acting on each body more and more. By running with a symplectic Euler simulation with 200 time-steps and $\Delta t = 10^{-5}$, a python program was used to run the program with different values of θ_0 and store the execution time and `posmaxdiff` between the Barnes-Hut algorithm result and an "exact" Newton algorithm in a file. The maximum value of θ_0 that guarantees a `posmaxdiff` below 10^{-3} was found to be $\theta_0 = 0.2565$, which is a reasonable value for the parameter.

To further reassure the correctness of the code, one should expect the complexity of the Barnes-Hut algorithm to be $\mathcal{O}(N \log N)$ in contrast to the direct summation method, which is $\mathcal{O}(N^2)$

(see Complexity analysis). In Fig. 3, the execution time of the Barnes-Hut algorithm and the direct summation method are plotted as a function of the number of particles N, along with the expected asymptotic behavior of the execution time.



(a) Barnes-Hut



(b) Newton

Figure 3: Execution time against the number of particles used for a) Barnes-Hut algorithm with θ_0=0.2565. b) Newton algorithm.

It is clear that the execution time of the Barnes-Hut algorithm not only grows slower than the direct summation method, but it also runs faster. By fitting the execution time of the Barnes-Hut algorithm to a function of the form $aN \log N$, one can observe the expected behavior which, although similar to $\mathcal{O}(N)$, fits much better to $\mathcal{O}(N \log N)$. The execution time of the direct summation method, on the other hand, grows clearly as $\mathcal{O}(N^2)$, as expected.

4.2 Integration error

Now, let us check if the Velocity Verlet method is properly implemented. As discussed in Integration methods, the Velocity Verlet method is expected to have an error that behaves asymptotically as $\mathcal{O}(\Delta t^2)$, while the Symplectic Euler method should behave as $\mathcal{O}(\Delta t)$. To check this, I created another auxiliary Python script to run the program with different values of Δt while maintaining the total time of simulation $T = \texttt{nsteps} \times \Delta t$ constant using $\Delta t = 10^{-8}$ and $\texttt{nsteps} = 2 \cdot 10^6$ as a reference for zero error. This was done for both Velocity Verlet and Symplectic Euler methods. The results are shown in Fig. 4, along with two lines of slopes 2 and 1 for Verlet and Euler respectively, to show the expected behavior. In addition, a straight line was included as a reference error of 10^{-5} to compare the Δt values that guarantee the same error for both methods.

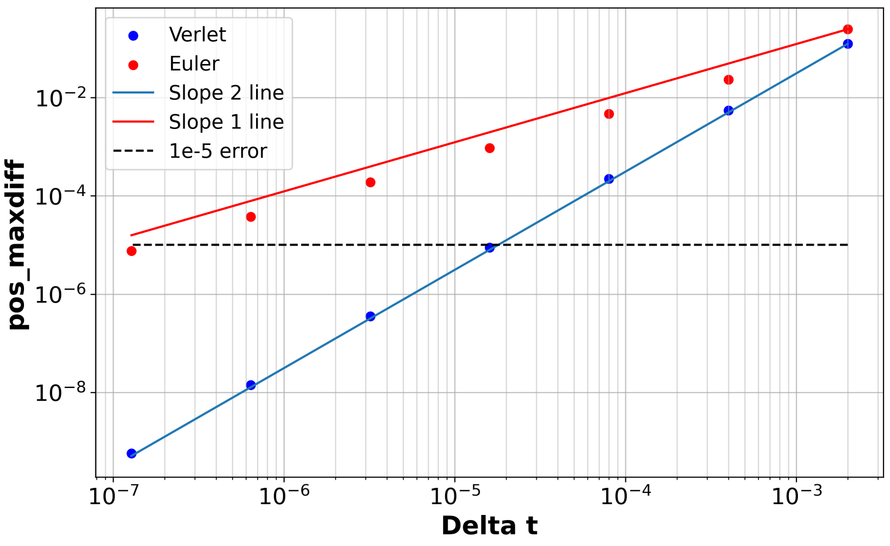
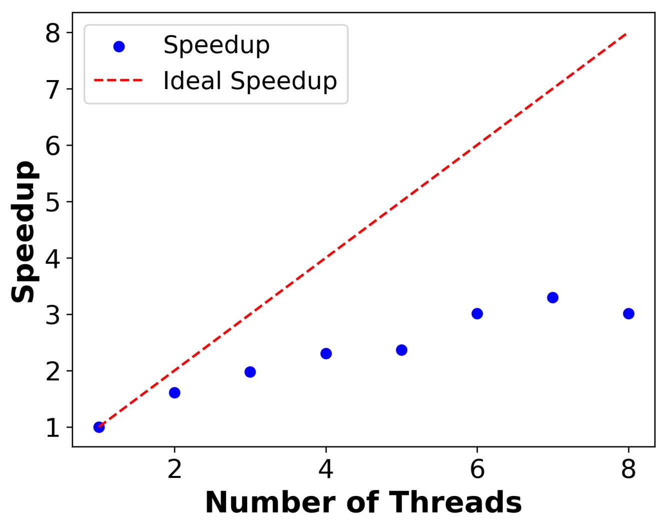


Figure 4: Position error `posmaxdiff` against Δt for Velocity Verlet method and Symplectic Euler method.

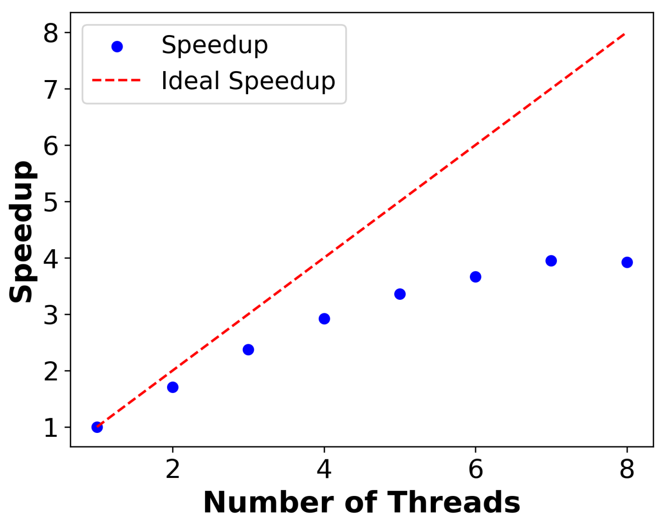
Once again, the results are consistent with the expected behavior. Note that to achieve an error of 10^{-5}, the Velocity Verlet method requires a Δt of roughly 10^{-5}, while the Symplectic Euler method requires a Δt of approximately 10^{-7}. For smaller values of the error, the difference in Δt required for both methods to achieve the same error would be even more significant.

4.3 Parallel performance

To check the performance of the parallelized code, a setup with high enough particles is chosen, with the aim of getting an acceptable efficiency (if one splits the total time as $T = T_{ideal} + T_{overhead}$, the last term grows much slowly compared to the first when n increases). The code is run with $\Delta t = 10^{-5}$, and the execution time is measured for threads in a range from 1 to 8. Two experiments were carried out, one where $\theta_0 = 0.2565$, 10000 particles and 200 steps and another where $\theta_0 = 0$, 5000 particles and 100 steps. Each execution time is measured 5 times and the minimum one is chosen to avoid taking computer noise into account. The results are shown in Fig. 5.



(a) $N = 10000$, `nsteps`$=200$, θ_0=0.2565



(b) $N = 5000$, `nsteps`$=100$, θ_0=0.1

Figure 5: Speedup plots for the parallelized code with 5000 particles. a) $N = 10000$, `nsteps`$=200$, θ_0=0.2565 b) $N = 5000$, `nsteps`$=100$, θ_0=0.1.

The results show that the speedup increases with the number of threads up to 7, and they do not follow the ideal behavior. However, parallelization seems to make the code up to roughly 4 times as fast, which is understandable since the machine running the code only has four cores. However, possible future optimizations could be implemented, such as partitioning and scheduling to further improve the parallel performance of the code [6] and adjusting workload imbalance: by assigning clusters of close particles to the threads, an improvement in cache locality could be possibly achieved.

4.4 Performance optimization

To conclude the experiments, I will try to show the best outperformance possible of the Barnes-Hut algorithm over the direct summation method. I will run the program with $N = 10^4$ particles to take advantage of the better complexity, and I will parallelize the code with 8 threads to further improve the performance. The idea is to compare the benefits of both Barnes-Hut algorithm and Velocity Verlet method over the direct summation method and Symplectic Euler method.

Since the Barnes-Hut algorithm is not exact, a maximum error tolerance must be chosen in order to perform comparisons. Before its election, it is wise to study the behavior of the error as a function of Δt while setting $\theta_0 = 0$, similarly to the previous experiment. This is done since the simulation may become unstable for large enough time steps, or for initial conditions that are too close to each other.

In my case, I chose a reference zero error simulation for 4000 particles with $\Delta t = 2 \cdot 10^{-8}$ and `nsteps`$= 5000$. The results are shown in Fig. 6.

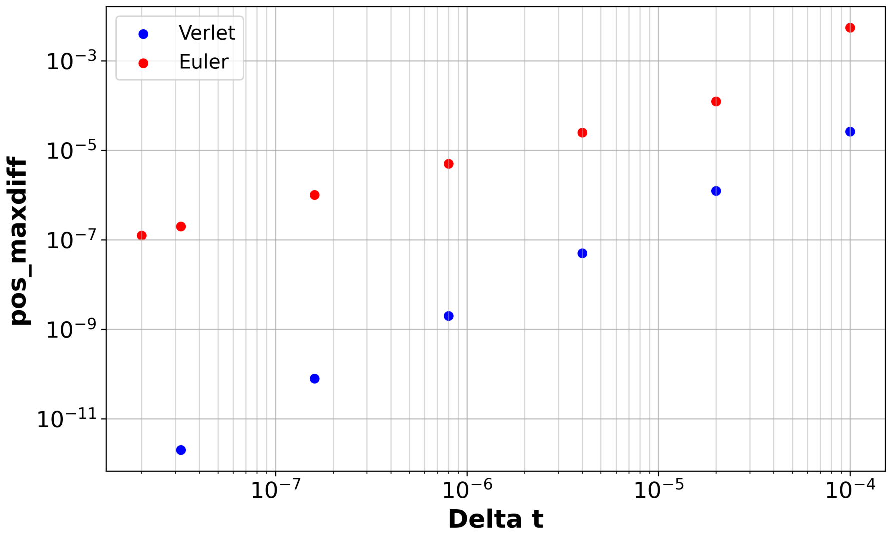


Figure 6: Position error `posmaxdiff` against Δt for Barnes-Hut with Verlet and direct summation with Euler.

Once again, we find similar results to those in Fig. 4. Note however that for Δt values higher than $2 \cdot 10^{-5}$, the error of the Euler method breaks its previous behavior, as the simulation becomes unstable.

Now, let's study how the error is affected by the parameter θ_0. By fixing $\Delta t = 2 \cdot 10^{-8}$ and `nsteps`$=5000$, the code is run with different values of θ_0 and the `posmaxdiff` is measured, as shown in Fig. 7.

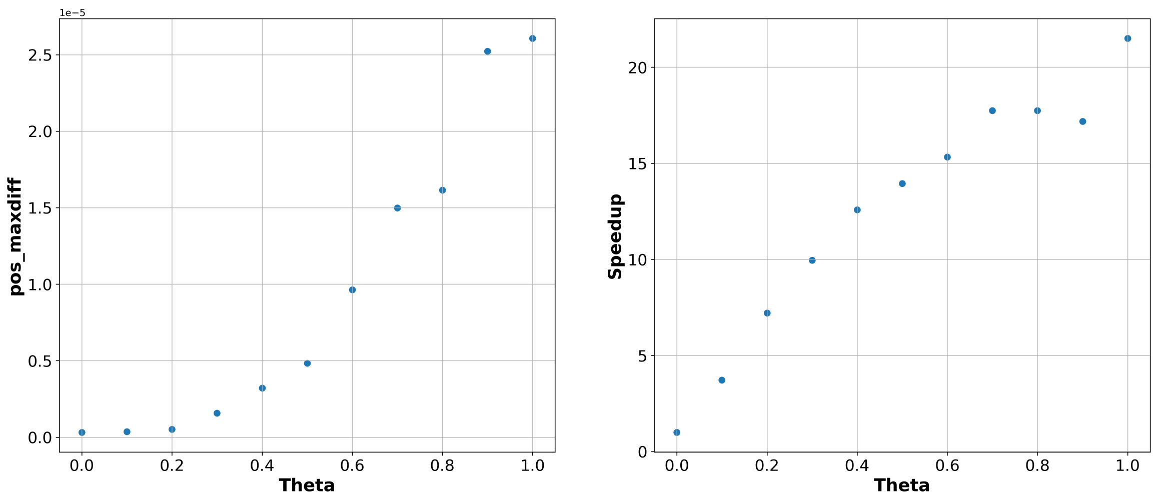


Figure 7: Position error `posmaxdiff` against θ_0 for fixed Δt and `nsteps`

This indicates that the error induced by θ_0 behaves somewhat similar to the error induced by Δt, as the error increases as θ_0 increases in a similar fashion while providing an approximately linear speedup. Note however that the error is of the order of 10^{-6} to 10^{-5}, so that if one chooses to compare the performance of the two algorithms with an error tolerance below 10^{-6}, once could achieve massive speedups adjusting the time-steps according to Fig. 6, but θ_0 would induce errors higher than the desired tolerance for almost any value. Conversely, if the tolerance is chosen too high, the simulation may become unstable due to the integration methods.

For those reasons, I chose to set the error tolerance to both 10^{-5} and 10^{-6}. The results from the optimal parameters and timings are shown in table 1 and 2.

Tolerance	θ_0	Δt	`nsteps`
10^{-5}	0.55	$5 \cdot 10^{-5}$	2
10^{-6}	0.25	$1 \cdot 10^{-5}$	10

Barnes-Hut with Verlet

Tolerance	Δt	`nsteps`
10^{-5}	$1.\bar{6} \cdot 10^{-6}$	60
10^{-6}	$1.\bar{6} \cdot 10^{-7}$	600

Direct Newton with Euler

Table 1: Optimal parameters for the chosen tolerances

Tolerance	B-H with Verlet (s)	Newton with Euler (s)	Speedup
10^{-5}	0.0225	1.3564	60
10^{-6}	0.12	13.44	112

Table 2: Timings and speedup of Barnes-Hut with Verlet against direct Newton with Euler.

5 Conclusions

In this report, I have described the implementation of a simulation for the gravitational N-body problem using the Barnes-Hut algorithm. I have shown that the Barnes-Hut algorithm is a significant improvement over the direct summation method up to a factor of 112 times faster for the same error tolerance. However, the Barnes-Hut algorithm can perform even better for larger N, as the complexity of the algorithm is $\mathcal{O}(N \log N)$ in contrast to the direct summation method, which is $\mathcal{O}(N^2)$. The Velocity Verlet method was also implemented and shown to have an error that behaves asymptotically as $\mathcal{O}(\Delta t^2)$, while the Symplectic Euler method behaves as $\mathcal{O}(\Delta t)$.

This algorithm however is not without its own challenges, as the construction of the quadtree and the calculation of the forces require a significant amount of memory and computational resources, and it approximates the forces acting on each body, so it is not exact. In addition, the parameter θ is a key parameter to tune, which may be tedious if one desires to implement a simple simulator for didactic purposes. One should also note that Velocity Verlet, while more accurate than Symplectic Euler, can still become unstable for large time steps.

Thus, the Barnes-Hut algorithm has been proven to be a powerful tool for simulating the gravitational N-body problem, especially when compared to the direct Newton method and when a large number of bodies are involved. However, there is still room for improvement as some techniques such as partitioning and scheduling could be used to further optimize the parallelization of the code [6].

References

[1] Henry Crozier Plummer. On the problem of distribution in globular star clusters. *Monthly Notices of the Royal Astronomical Society, Vol. 71, p. 460-470*, 71:460–470, 1911.

[2] Susanne Pfalzner and Paul Gibbon. *Many-body tree methods in physics*. 1997.

[3] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[4] Tom Ventimiglia and Kevin Wayne. The Barnes-Hut Algorithm. http://arborjs.org/docs/barnes-hut.

[5] Kim Torberntsson. The barnes hut algorithm for n-body simulation implemented in c. https://github.com/KimTorberntsson/Barnes-Hut.git, 2015.

[6] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 1995.