Uppsala University

High Performance Programming

1TD062

Assignments 4 & 5 Report:
The Barnes-Hut Method for Solving the 2D Gravitational N-Body Problem

Students:
Anders Köhler
Benjamin Bucknall
Julián Ramón Marrades Furquet
Omar Ghulam Ahmed Malik

Teachers:
Jarmo Rantakokko
Benjamin Weber

May 6, 2021

Contents

Introduction

The motion of planets, stars, and other bodies in a galaxy can be accurately described by Newton's 2nd law:

$$\mathbf{F}_{net} = m\mathbf{a} = m\frac{d^2\mathbf{r}}{dt^2}, \tag{1}$$

where $\mathbf{F}_{net}$ is the net gravitational force acting on the body, m is its mass, $\mathbf{a}$ is its acceleration, and $\mathbf{r}$ is its position as a function of time. Eq. (1) says that each body's trajectory in space can be determined once the net force acting on it is known. This trajectory can be found analytically for one- and two-body systems — where the latter can be solved for the *relative* trajectories of the bodies to each other. However, for systems containing three or more bodies, Eq. (1) must be solved numerically. Fortunately, numerous algorithms are available to simulate such systems, but difficulties arise in devising a program that scales feasibly with the number of bodies in the system.

The objective of this assignment was to solve Eq. (1) for a two-dimensional N-body galaxy, where the gravitational force $\mathbf{f}_{ij}$ — which is exerted on body i by body j — takes the following form:

$$\mathbf{f}_{ij} = -G\frac{m_i m_j}{r_{ij}^3}\mathbf{r}_{ij}, \tag{2}$$

where G is the gravitational constant, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the separation vector between body i and body j, and $r_{ij} = \sqrt{\mathbf{r}_{ij} \cdot \mathbf{r}_{ij}}$ is the distance between them. Since the computational domain is 2-dimensional, all vectors have only x- and y-components. Summing $\mathbf{f}_{ij}$ over all j bodies yields the net force on body i:

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

where the *net* subscript has been dropped from $\mathbf{F}$ for the sake of convenience in the rest of this report. The denominator in Eq. (2) was slightly modified in the program — to prevent dividing by near-zero values — by adding a small positive number ε to r_{ij}. Therefore, the total force acting on body i becomes:

$$\mathbf{F}_i = -Gm_i \sum_{j=1,\ j\neq i}^{N} \frac{m_j}{(r_{ij}+\varepsilon)^3}\mathbf{r}_{ij}. \tag{4}$$

After finding $\mathbf{F}_i$, the next step was to solve Eq. (1) numerically for body i's position at some point in time, $\mathbf{r}_i$, provided that its initial position and velocity are known. The *symplectic* Euler method was used to do this, which first calculates its velocity at the next timestep and then uses this new velocity to calculate its position at the next timestep:

$$\begin{aligned} \mathbf{a}_i^n &= \frac{\mathbf{F}_i^n}{m_i} = -G \sum_{j=1,\ j\neq i}^{N} \frac{m_j}{(r_{ij}^n+\varepsilon)^3}\mathbf{r}_{ij}^n \\ \mathbf{u}_i^{n+1} &= \mathbf{u}_i^n + \Delta t\mathbf{a}_i^n \\ \mathbf{r}_i^{n+1} &= \mathbf{r}_i^n + \Delta t\mathbf{u}_i^{n+1}, \end{aligned} \tag{5}$$

where $\mathbf{u}_i$ is the velocity of body i, Δt is the size of each timestep from one point in time to the next, and the n and $n+1$ superscripts denote the given and next timesteps, respectively (they are not exponents). Eq. (5) can be repeated for as many timesteps as needed, thereby simulating the galaxy of interest.

Methodology

In this report, two approaches to simulating an N-body system were investigated. Firstly, an exact (up to computer accuracy) implementation is studied, whereby the the sum of forces acting on each body is computed by individually considering every other body in the system. Secondly, the Barnes-Hut algorithm is investigated, which approximates the forces by a group of bodies on a single body — provided that the distance is large enough — as a single, compound force. This assumption has the potential to greatly reduce the number of necessary computations and consequently, the run-time of the simulation. For both algorithms, the effects of relevant optimization techniques, including parallelization, will be evaluated in terms of the resulting performance and accuracy of the simulations.

Specifications

The simulation was done on the AMD Opteron$^{\text{TM}}$ 8-core, 64-bit processor running at 2.6 GHz, with capability to handle 2 threads per core. The machine uses the Scientific Linux release 6.10 (Carbon) operating system. Regarding the C programming language, the C99 standard was used with the GCC compiler release 4.4.7. All of the C codes use the following additional flags to allow the compiler to make further optimizations:

- `-O3`: Tells the compiler to perform aggressive optimizations when building the assembly code from the C code.
- `-march=native`: The compiler should produce a code that matches the current CPU's processor.

The usage of the following flags was avoided:

- `-funroll-loops`: Automatically unrolling loops can be dangerous, especially if the workload within them is heavy. Since this was the case, loop unrolling was manually added at a later stage, when the code was vectorized.
- `-ffast-math`: Since the positions of the planets are spread close to the origin, it is not safe to risk a substantial impact on accuracy if values get flushed to 0.

Summary of Exact Method

Algorithm 1 presents the exact method of computing the evolution of an N-body system. The key observation to make here is that, in order to compute the forces acting on a particular body (indexed by i), the algorithm iterates through every other body in the system (indexed by j), considering the contribution from body j explicitly, before summing all contributions to calculate the resulting net force on body i. The method thus operates with a $\mathcal{O}(N^2)$ complexity.

In the implementation of Algorithm 1, several manual optimizations were made in order to minimize execution time:

- By employing the keywords `restrict` and `const`, the compiler is notified that two or more pointers do not point to the same memory location, and that a variable won't change its value during its life-span, respectively. This way, it is likely that the compiler will be able to perform optimizations when transforming into assembly code.

Algorithm 1 Exact galaxy simulation algorithm

```
Input: ${\mathbf{r}_i^0, m_i, \mathbf{u}_i^0 : i = 1, 2...N}$ - positions, masses, and velocities of all $N$ bodies at initial time.
    $T$ - number of timesteps in the simulation
    $\Delta t$ - size of each timestep
Output: ${\mathbf{r}_i^T, \mathbf{u}_i^T : i = 1, 2...N}$ - positions and velocities of all $N$ bodies at the final, $T$-th point in time.

1:  Initialization: read ${\mathbf{r}_i^0, m_i, \mathbf{u}_i^0}$, the initial positions; masses; and velocities, from the input file in a 2D array.
2:  for $t = 0, 1, ..., T-1$ do
3:
4:      Calculate the accelerations acting on each body at a given time. The accelerations are stored in a 2D array.
5:      for $i = 1, 2, ..., N$ do
6:          $\mathbf{a}_i^t := \mathbf{0}$
7:          for $j = 1, 2, ..., N$ do
8:              if $j \neq i$ then
9:                  $\mathbf{r}_{ij}^t := \mathbf{r}_i^t - \mathbf{r}_j^t$
10:                 $r_{ij}^t := \sqrt{\mathbf{r}_{ij}^t \cdot \mathbf{r}_{ij}^t}$
11:                 $\mathbf{a}_i^t := \mathbf{a}_i^t + \frac{m_j}{(r_{ij}^t + \epsilon_0)^3}\mathbf{r}_{ij}^t$
12:             end if
13:         end for
14:         $\mathbf{a}_i^t := \mathbf{a}_i^t * (-G)$
15:     end for
16:
17:     Update the positions of each body using the calculated accelerations.
18:     for $i = 1, 2, ..., N$ do
19:         $\mathbf{u}_i^{t+1} := \mathbf{u}_i^t + \Delta t \mathbf{a}_i^t$
20:         $\mathbf{r}_i^{t+1} := \mathbf{r}_i^t + \Delta t \mathbf{u}_i^{t+1}$
21:     end for
22:
23: end for
```

- To avoid branching in line 8, the loop in line 5 was split into two consecutive ones. The first iterates from $j = 1$ to $i - 1$ and the second one from $j = i + 1$ to N.
- In line 11, there is an array accessed by index i in the inner j loop (line 7). Hence, in order to make it easier for the compiler to perform optimizations within said loop, a loop invariant was created by accumulating all acceleration terms for a given i in a variable `sum`, leaving line 11 as $sum := sum + \frac{m_j}{(r_{ij}^t + \epsilon_0)^3} \mathbf{r}_{ij}^t$, and line 14 as $\mathbf{a}_i^t := sum * (-G)$.
- Thinking ahead, when the code is vectorized, the state matrix will be accessed column-wise; the velocities, masses and positions of two planets will be accessed at once. Thus, it is beneficial to store the matrix column-wise, so that cache-lines are fully exploited. Furthermore, the columns themselves were concatenated into a 1D array to ensure that they are placed contiguously in memory.
- The inner j loop can be vectorized by accessing several j's at a time. SS3 intrinsics, with `__m128d` 128-bit registers to store pairs of 64-bit doubles, were employed. In order to take full advantage of this capability, the required data structures `state` and `acc` storing the states of the bodies are declared and initialized as arrays aligned to 16 bytes. This allows memory access to be performed more efficiently than if assumed to be unaligned. Finally, it is worth noting that special treatment is required for instances in which the number of bodies to consider is odd. In this case, vector operations are performed for all possible pairs of bodies, with the influence of the remaining individual being computed through regular scalar arithmetic.

Parallelization

When it comes to parallelizing Algorithm 1, the first concern is which loop to break down: i or j? If the outer loop is split, a thread will compute the (whole) accelerations for some given planets, and then update the state with them (see Fig. 1). On the other hand, splitting the inner loop will provide a thread with the duty to compute some given acceleration terms for all planets (see Fig. 2).

In both scenarios, if the work among threads is balanced, thread k will handle the exact same number of (i, j) pairs. However, threading j would require either some protection because several threads will update the accelerations for a given i or a serial code which merges the local sums from the treads for each i. Moreover, the state update (lines 18-20 in Algorithm 1) would be executed by serial code.

On the other hand, threading i creates much less overhead. One can let each thread compute the accelerations for its planets, wait for all threads to finish that part, then let each thread update its planets for the next time step and synchronize the threads again. With this simple procedure, there is no worry about any threads interfering with variables required by others.

Hence, the outer loop parallelization is implemented distributing the work evenly among threads. I.e. if there are `n_threads` threads and `N` planets, the number of planets per thread is `N/n_threads`. However, if the division leaves a remainder `rem`, there will be `rem` threads handling an extra planet. The thread synchronization is implemented as a barrier with `pthread_cond_wait` and `pthread_cond_broadcast` so that waiting for other threads does not consume CPU time.

The Barnes-Hut Method

The Barnes-Hut algorithm is an alternative method for simulating an N-body problem. It is based on the exact method given above with the addition of one extra assumption: that the force exerted

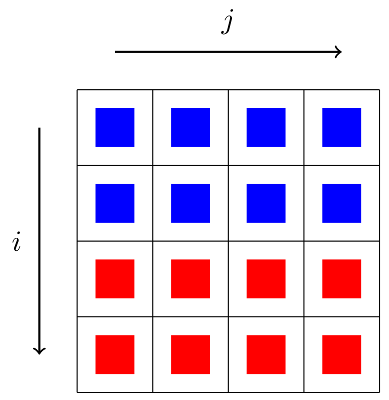


Figure 1: Threading the outer loop. Each color represents the tasks of a particular thread.

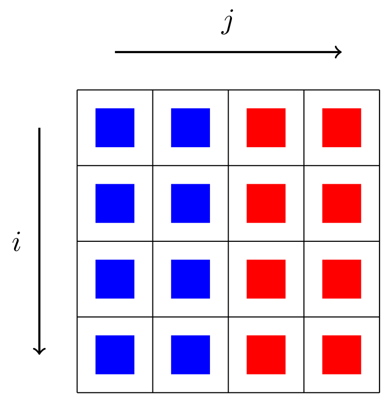


Figure 2: Threading the inner loop. Each color represents the tasks of a particular thread.

on an object by a distant group of objects can be approximated by the force that would be exerted by an object located at the center of mass of the group, with mass equal to the total sum of masses of the group. This approximation allows for a significant reduction in the computational load when calculating forces between bodies at the expense of accuracy of the simulation. In practice, this compromise is balanced through observation of the value of θ, defined as the following ratio:

$$\theta := \frac{\text{spatial extension of a group of particles}}{\text{distance between the the group and the particle}}$$

When computing the forces acting on a given body, one iterates through groups of bodies calculating the value of θ for each group. If θ is large, this indicates that the body under consideration is close to the group, and so the resulting approximation will not be sufficiently accurate. Likewise, if θ is found to be small, this shows that the group is far enough away from the particle in question for the approximation to be of sufficient accuracy. It is up to the user to determine the exact meanings of θ being 'large' or 'small' through defining a threshold parameter $\theta_{\text{max}} \in [0, 1]$ that defines the boundary between 'large' and 'small' values of θ. Note that for large values of θ_{max}, many particles will be grouped together, increasing performance of the simulation at the expense of accuracy. Similarly, a small value of θ_{max} will mean that many particles will be treated individually, increasing both the simulation's run-time and accuracy. At the limiting case of $\theta_{\text{max}} = 0$, the Barnes-Hut performs identically to the exact method outlined above.

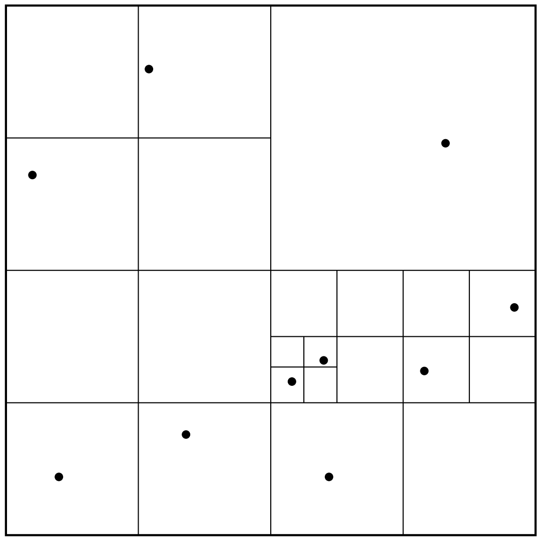


Figure 3: A graphical representation of a quadtree containing the locations of 10 particles in a square computational domain.

A further consideration to take into account when implementing the Barnes-Hut method is that of how to define ‘groups’ of particles. This is done through utilization of a *quadtree*. A quadtree is a data structure consisting of nodes representing square regions of the computational domain. The root of the tree is taken to be the entire computational domain which is recursively subdivided into four equal quadrants until each leaf node contains either one or zero particles. If a node is not a leaf node, then it stores the sum of masses and center of mass of all particles contained in it. The groups of particles in question are then represented by the nodes of the quadtree, where the extension of a group is given by the width of the region represented by the node. Fig. 3 shows the quadtree that would be created for an example configuration of 10 bodies in a square computational domain.

In the implementation developed for this investigation, the quadtree is generated at each timestep by first initializing the root as the entire computational domain devoid of any bodies, before adding each body $\mathbf{x}_i$ in turn as follows, starting from the root node.

1. If the node is an empty leaf node, add $\mathbf{x}_i$ and initialize the node’s children.
2. Else, if the node is a leaf node already containing a body $\mathbf{x}_j$, add $\mathbf{x}_i$ and $\mathbf{x}_j$ to their relevant child nodes by repeating from step 1.
3. Else, if the node is not a leaf, repeat from step 1, considering the child node that represents the quadrant containing $\mathbf{x}_i$.

With a quadtree in place, the Barnes-Hut method of computing the forces on a particular particle $\mathbf{x}$ proceeds as follows. Beginning from the root of the quadtree, compute the value of θ for the current node. If $\theta > \theta_{\max}$, consider the node’s four children, computing their θ-values, and performing the same comparison to $\theta_{\max}$. Continue recursively until reaching either a node for which $\theta < \theta_{\max}$, or a leaf node. If this node is a leaf containing a unique particle, compute the force on $\mathbf{x}$ from this particle. Else, compute the force on $\mathbf{x}$ from the hypothetical particle located at the centre of mass of the node, with mass equal to the sum of the masses of all particles located within the node.

Due to the method's recursive partitioning of data into four subnodes, the method would be expected to operate with a $\mathcal{O}(N \log N)$ complexity. This hypothesis was confirmed through correlation testing, accounted for in subsection "Evaluation of Barnes-Hut Approximation".

The full Barnes-Hut algorithm for running an N-body simulation is given below in Algorithm 2.

Parallelization

In addition to utilizing many of the serial optimization techniques mentioned above, a parallel implementation of the Barnes-Hut method was constructed. In order to identify the sub-procedure of the algorithm with the highest potential for speed-up through parallelization, it is useful to split Barnes-Hut into a sequence of steps, each of which is carried out at every time iteration:

1. Construct the quadtree corresponding to the system's current state using an `add` function in a loop over all particles.
2. Traverse down the quadtree to calculate the centers of mass and the total mass in each node; this was done using the `compute_tree` function.
3. Update each body's position via the symplectic Euler method to get the next state of the system using the `getNextState` function. This function also calls the `computeForce` function, which computes the net forces on each body by the nodes in the quadtree.
4. Delete the quadtree from memory using the `delete_tree` function.

Of these four steps, Step 3 was found to be the slowest by a significant margin — on the order of 100 milliseconds — as shown in Table 1, and was therefore chosen as the bottleneck to be parallelized. Furthermore, according to Algorithm 2, there are two likely targets for parallelization in `getNextState`: the outer loop over i on line 9 and traversing down the q nodes in the quadtree on line for each i — via the `computeForce` function. However, this traversal is done recursively and if each thread executes the `computeForce` function, then this would generate a large number of threads while traversing down the quadtree. Creating a single thread has a large overhead, which would accumulate as more threads are recursively created; this is highly inefficient. On the other hand, for the outer loop, a fixed number of threads can be created by the master thread and divided amongst an arbitrary number of iterations; in this way, the thread overhead can be controlled. For this reason, the outer loop was parallelized and so each thread was responsible for independently traversing down the quadtree to calculate the net force on the bodies for which they are responsible.

Function	Average time (s)	Standard dev. (s)
`add` (loop)	0.005100	0.000079
`compute_tree`	0.000606	0.000003
`getNextState`	0.256170	0.000806
`delete_tree`	0.000899	0.000007

Table 1: Execution times of the four main steps in the serial Barnes-Hut implementation, averaged over 10 runs for a single timestep. Here $\theta_{\max} = 0.25$ (see next section), $N = 10000$, and $\Delta t = 10^{-5}$.

Unlike the exact method, the Barnes-Hut method was not implemented with vectorization. To do so would be difficult due to the recursive nature of traversing the quadtree, as opposed to the fixed inner loop of the exact method. Particularly, the data are not stored contiguously within the quadtree, which is required to use the SIMD intrinsics such as SS3.

Algorithm 2 Barnes-Hut galaxy simulation algorithm

```
Input: $\{\mathbf{r}_i^0, m_i, \mathbf{u}_i^0 : i = 1, 2...N\}$ - positions, masses, and velocities of all $N$ bodies at initial time.
    $T$ - number of timesteps in the simulation
    $\Delta t$ - size of each timestep
    $\theta_{\max}$ - Barnes-Hut approximation parameter
Output: $\{\mathbf{r}_i^T, \mathbf{u}_i^T : i = 1, 2...N\}$ - positions and velocities of all $N$ bodies at the final, $T$-th point in time.

1:  Initialization: read $\{\mathbf{r}_i^0, m_i, \mathbf{u}_i^0\}$, the initial positions; masses; and velocities, from the input file in a 2D array.
2:  for $t = 0, 1, ..., T-1$ do
3:
4:      Generate the quadtree $Q$ for the configuration of bodies at the current time.
5:
6:      Compute the total mass $m_q$ and center of mass $\mathbf{R}_q$ of each node $q$ of the quadtree.
7:
8:      Calculate the accelerations acting on each body at a given time using the Barnes-Hut method. The accelerations are stored in a 2D array.
9:      for $i = 1, 2, ..., N$ do
10:         $\mathbf{a}_i^t := \mathbf{0}$
11:         for node $q$ in $Q$ do
12:             $\mathbf{r}_{iq}^t := \mathbf{r}_i^t - \mathbf{R}_q$
13:             $r_{iq}^t := \sqrt{\mathbf{r}_{iq}^t \cdot \mathbf{r}_{iq}^t}$
14:             $\tilde{\mathbf{r}}_{iq}^t := \mathbf{r}_i^t -$ center of $q$
15:             $\tilde{r}_{iq}^t := \sqrt{\tilde{\mathbf{r}}_{iq}^t \cdot \tilde{\mathbf{r}}_{iq}^t}$
16:             $\theta := \frac{\text{width of } q}{\tilde{r}_{iq}^t}$
17:             if $\theta < \theta_{\max}$ then
                    $\mathbf{a}_i^t := \mathbf{a}_i^t + \frac{m_q}{(r_{iq}^t + \epsilon_0)^3} \mathbf{r}_{iq}^t$
18:             else
19:                 Return to line 11 for each of $q$'s children.
20:             end if
21:         end for
22:         $\mathbf{a}_i^t := \mathbf{a}_i^t * (-G)$
23:     end for
24:
25:     Update the positions of each body using the calculated accelerations.
26:     for $i = 1, 2, ..., N$ do
27:         $\mathbf{u}_i^{t+1} := \mathbf{u}_i^t + \Delta t \mathbf{a}_i^t$
28:         $\mathbf{r}_i^{t+1} := \mathbf{r}_i^t + \Delta t \mathbf{u}_i^{t+1}$
29:     end for
30:
31: end for
```

Results and Discussion

Evaluation of Barnes-Hut Approximation

Accuracy

As noted above, one key feature of the Barnes-Hut algorithm is the trade-off between accuracy and performance, as determined by the parameter value $\theta_{\max}$. This leaves the user in a situation of having to decide on a value of $\theta_{\max}$ that minimizes run-time, yet does not lead to unacceptable error values. In order to select such a value for further investigations, a maximum position difference ϵ_0 for a single body (when compared to the exact method) of 10^{-3} was selected as an acceptable upper error bound, as per given instructions. As is shown in Fig. 4, the maximum value of $\theta_{\max}$ that satisfies this error bound was found to be ≈ 0.25. Therefore, $\theta_{\max} = 0.25$ represents a fair compromise between performance and accuracy, and thus was chosen for all further analysis.

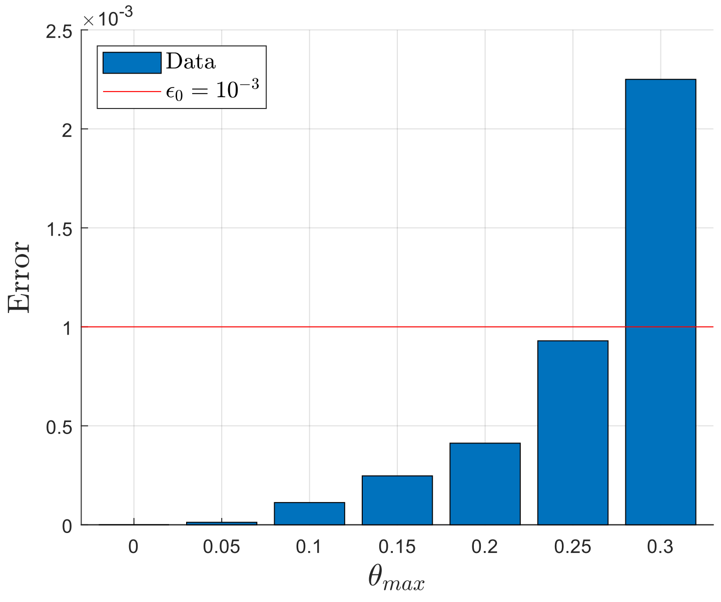


Figure 4: Error rates for different values of $\theta_{\max}$, tested in steps of 0.05 on a system of size $N = 2000$. Parameters are set to: $nSteps = 200$, $\Delta t = 10^{-5}$. Note that a value of 0 generates no error at all as this parameter setting will cause Barnes-Hut to generate the same results as the exact method, though at a much slower pace.

Performance

As Fig. 5 shows, the Barnes-Hut implementation managed to outperform the exact method for systems of $N = 3000$ and upward. The overhead introduced by the management of the quadtree renders the algorithm unsuitable for systems below this threshold – and even slightly above it due to the loss of precision, with time gains being minuscule – but for system sizes reaching into five digit numbers, the efficiency benefits offered by Barnes-Hut become all the more clear.

As hypothesized in subsection "The Barnes-Hut Method", the algorithm ought to operate with a time complexity of $\mathcal{O}(N \log N)$, which is indeed hinted at by the trend described by Fig. 5. In order to conduct proper testing, further execution times were collected for all systems that were available for simulation, whereupon the data was subjected to a Pearson correlation coefficient test against the N values as well as their linearithmic ($N \log N$) and quadratic (N^2) counterparts. As demonstrated by the linear fits in Fig. 6 and their correlation values, it was confirmed that Barnes-Hut indeed is a $\mathcal{O}(N \log N)$ algorithm.

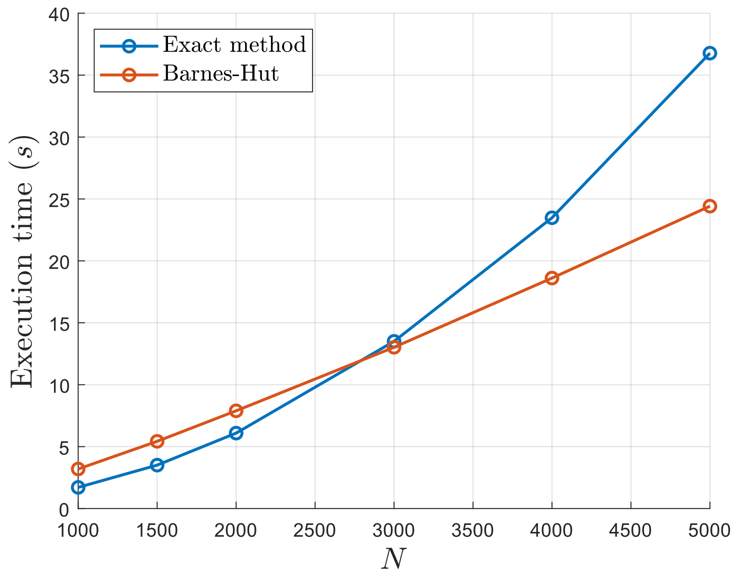
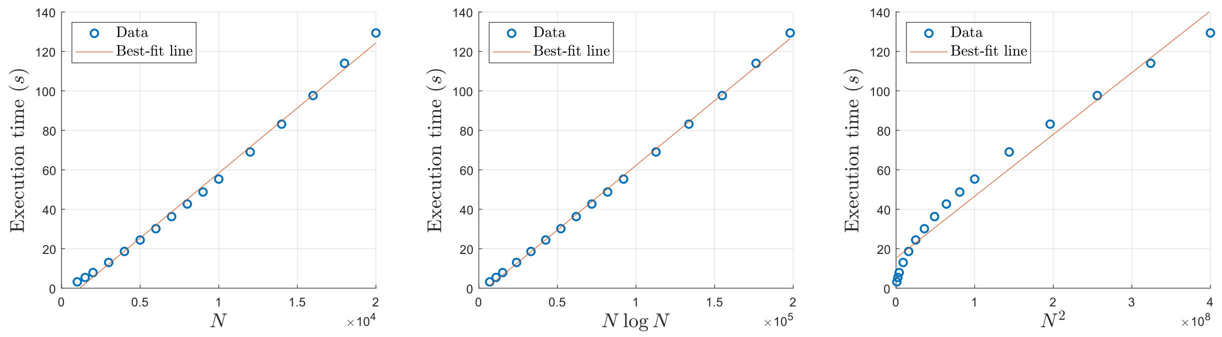


Figure 5: Performance metrics for the serialized versions of the exact method and Barnes-Hut. For each N the experiment is performed 10 times and the results are averaged. Parameters are set to: $nSteps = 200$, $\Delta t = 10^{-5}$, $\theta_{\max} = 0.25$.



(a) N, $r = 0.9977$ (b) $N \log N$, $r = 0.9993$ (c) N^2, $r = 0.9816$

Figure 6: Pearson correlation coefficient (r) test for different complexities in terms of the number of bodies N. Parameters are set to: $nSteps = 200$, $\Delta t = 10^{-5}$, $\theta_{\max} = 0.25$.

Evaluation of Parallelization

Fig. 7 shows plots comparing the execution times and speed-up factor for parallelized implementations of both the exact method, and the Barnes-Hut algorithm as a function of the number of threads used.

Looking at the trend for 1 to 16 threads, it can be observed that the exact method initially scales almost perfectly with the number of threads. Then, from 9 threads onwards, the scaling worsens.

Both behaviors are expected. Since computing the acceleration for each planet should take the same amount of time, n threads should reduce the running time by a factor of n. However, when the number of threads becomes large, it takes more time to create them, synchronize them, etc. Thus, the scaling factor becomes smaller as n keeps growing.

On the other hand, the Barnes-Hut parallelization does not scale as well. This is because the Barnes-Hut algorithm itself attempts to reduce the total number of computations by averaging over clusters of closely-spaced bodies — which are defined by the chosen θ_{max} value — and this would inevitably result in a load imbalance amongst the threads. Consider the following example with two threads: suppose that the first thread computes $\mathbf{F}_{ij}$ (the force on body i exerted by body j), while the second thread computes $\mathbf{F}_{ji}$ (the force on body j by body i). Now suppose further that body j is actually a cluster of closely-spaced bodies that lie in a node whose θ value (for body i) is less than θ_{max}. The second thread will therefore have more work to do than the first as it will have to iterate over the cluster of bodies that constitute body j, calculating $\mathbf{F}_{ji}$ for each of them. On the other hand, the first thread only needs to consider the center of mass and total mass of the bodies in j to calculate $\mathbf{F}_{ij}$. So if body j is an aggregate of M bodies, then the first thread would perform 1 computation while the second would perform M computations instead.

If one looks at 17 threads, the speed-up drops down. This fallback may be due to a combination of two factors:

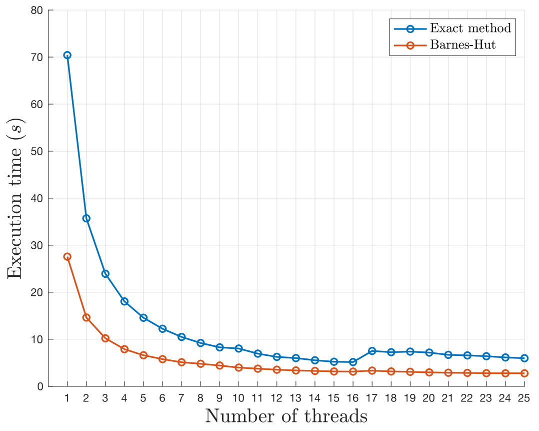
1. There are more threads than physical cores. Thus, not all threads can have simultaneous access to the CPU resources.
2. A point has been reached where the overall data used by the threads doesn't fit in the caches. This leads to a thread replacing existing data in a cache by its own working data, when that previous data was going to be needed in the near future by another thread.

This pitfall is much more noticeable for the exact method since the caches in Barnes-Hut were already being cleaned up due to the recursive nature of the algorithm used to traverse the quadtree; in a given ith iteration, the cacheline that contained the data of i and its neighbors would more likely be replaced once the `computeForce` function is recursively called several times.

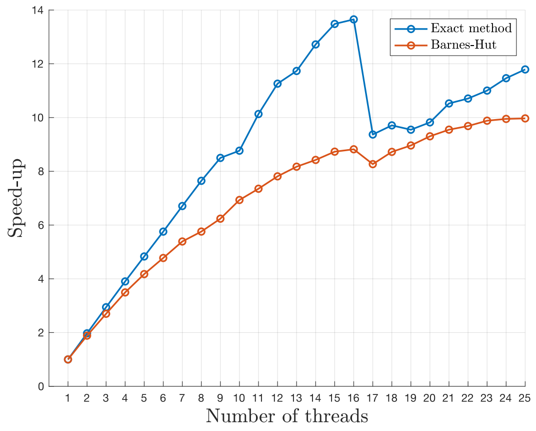
For 18+ threads, the performance of the exact method improves without reaching the previous peak at 16 threads. Regarding Barnes-Hut, its speed-up increases similarly as it did in 1-16 threads but seems to reach a plateau for 23-25 threads. This may be the point where the saved `user` time via parallelization is balanced by the required `system` time to manage the large number of threads.

Conclusion

In this report, two related yet distinct approaches to numerically solve for the evolution of an N-body system have been implemented and evaluated. The first of these approaches is an exact method that considers the effect that each body has on every other body in the system. Although this method is very accurate, for systems containing a very large number of bodies it can be wasteful to consider each and every body individually. The second method is the Barnes-Hut algorithm that allows the effect of distant groups of bodies to be approximated by that of a single, hypothetical body. This approximation vastly reduces the total number of required computations, and therefore simulation run-time, by forfeiting some of the accuracy of the exact approach.



(a) Execution time



(b) Speedup

Figure 7: Performance metrics for the parallelized versions of the exact method and Barnes-Hut for a $N = 10000$ body system. For each number of threads, the experiment is performed 10 times and the results are averaged. Parameters are set to: $nSteps = 100$, $\Delta t = 10^{-5}$, $\theta_{\max} = 0.25$.

Furthermore, various ways of implementing these two algorithms have been explored. In both cases, serial and parallel programs have been written to carry out the simulation, with the performance benefits of parallelization being evaluated in depth.