Concurrent Programming Project: N-Body Simulation

Erik Båvenstrand
KTH - Royal Institute of Technology

March 2019

1 Introduction

An N-Body simulation is the simulation of dynamical particles or bodies, under the influence of each other's gravity. It has applications in simulations from earth-moon-sun systems to understanding the large-scale structure of the universe.

2 Aim

The aim of this project is to create four versions of N-Body simulation programs and evaluate them against each-other. The first one is a sequential brute-force method with $O(n^2)$ time complexity where n is the amount of bodies to simulate. The second one is a parallel implementation of the same brute-force method. The third version is the Barnes-hut approximation method using quad-trees and the last one is a parallel implementation of Barnes-hut. The input values for the programs are mostly the same and consisting of, number of bodies and number of simulation iterations. Number of threads and the length of what is considered far away in Barnes-hut methods are also some values that should be send as arguments.

3 Software and computing platforms

All four programs are written in C and the parallel implementations are using Pthreads. The testing was done on a 2018 MacBook Pro 2,3 GHz intel core i5 quad-core processor and 8 GB RAM on macOS Mojave. Since the program only has Pthreads as its only dependency it should run completely fine on a UNIX machine.

4 Design and implementation

This section of the report will focus on explaining the four different implementations of the simulation in the order they were presented in section 2. The versions in sections 4.2 and 4.3 were written using the pseudocode provided in the project description while the two latter versions were only described in Section 11.2.4 (page 569) of the G.Andrews textbook "Foundations of Multithreaded, Parallel, and Distributed Programming".

4.1 Sequential Brute-Force

This is the most straight forward approach of N-Body simulation, but at the same time it is also completely useless at computing a large number of bodies. The main loop consists of a calculation of forces that, for each body, calculates the forces that it is affected by and the effect it has on all the other bodies. This has $O(n^2)$ in time complexity. When all the forces have been calculated the bodies has to be moved according to the forces. This is easily done by iterating all bodies and moving them with a $O(n)$ time complexity. In total this algorithm has $O(n^2)$ in time complexity since the calculation of forces is the bottleneck.

4.2 Parallel Brute-Force

Parallelizing the brute-force algorithm is straight forward since we can have a shared array of forces for each thread. The calculation of forces is done with threads for each body with id: $thread\_id + (num\_threads * i)$. This means that the workload is almost perfectly divided among the available threads. The forces on and by the body is calculated and placed in the array of forces for each thread. When all threads are finished calculating forces they move on to the movement of bodies. The division of bodies is the same as the calculation of forces. For each body, the array of forces for each thread is summarized to find the sum of forces that the body is affected by and then moved correspondingly.

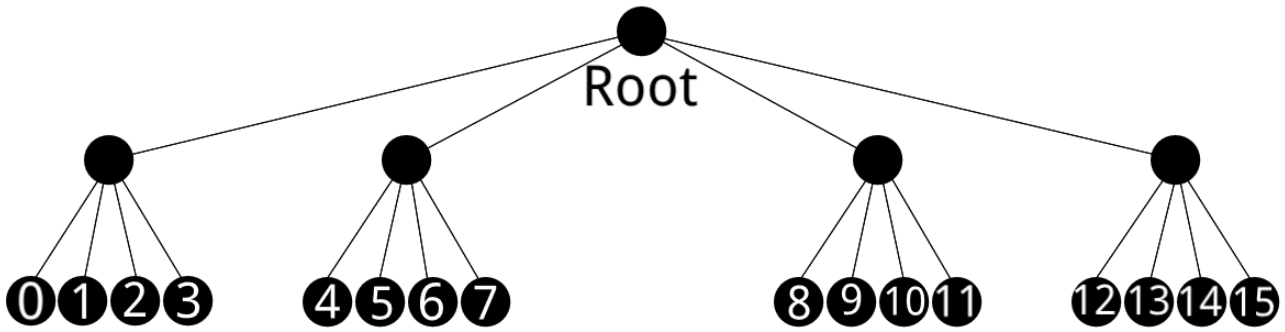

Figure 1: Quadtree division into sections 0-15

4.3 Sequential Barnes-Hut

The Barnes-hut algorithm is based on the fact that clusters of bodies far away can be seen as a large body with the mass of all bodies combined at the center of mass for the cluster. A quad-tree is used for dividing a node into four child quadrants. The data structure only allows one body to occupy the same node at a time so when another body is inside the same node, it is divided into four child quadrants and the bodies are placed in their corresponding quadrant.

The first step in the main loop is to construct the quad tree by adding all the bodies to it. There is a simple loop that iterates all the bodies and adds them in $O(n)$ time complexity. After the tree is created, all the waste nodes are cleared form it. This is done by checking if a node contains bodies, and if not, all the child nodes are freed from memory by using free. Next, the centers of masses are calculated by recursively calculating the center of mass for the child nodes and them summing them to find the center of mass for the root node. The calculation of forces is basically the same except that if a body is sufficiently far away from a node in the tree the force is calculated from the center of mass and mass of the node instead of its containing bodies. This is done in $O(n * log(n))$ time complexity. The movement of bodies is done the same way as in previous versions except that if a body is far away of the workspace it is simply left alone since the forces it is involved in is so small. This is done with a simple loop that iterates the bodies in $O(n)$ time complexity. The final step of the main loop in the sequential Barnes hut is the recursive tree traversal that sets the nodes mass to 0 and the hasBody flag to false. This is to allow the tree to be constructed again with the same heap address without unnecessary malloc and free system calls. This is why the waste nodes are removed after the tree is created since some of the nodes are probably empty.

4.4 Parallel Barnes-Hut

Intuitively it seems like the parallelization of the Barnes-hut is as simple as the brute-force, but that is not the case. To really parallelize the Barnes-hut, several threads has to be allowed to access the tree at the same time. To achieve this the tree is initialized to the third level before the main loop starts. Since each node has four children, this means that there are 16 child nodes when the initialization is complete. The implementation therefore has a limit to 16 threads in the tree at once. This can easily be increased to the next level which will allow $16 * 4 = 64$ threads at once inside the tree. Each of these 16 child nodes are named sections $0 - 15$ and can bee seen in figure 1. The assignment of sections to threads is done in the same way bodies are assigned to threads in previous versions. This will hopefully balance the workload, but there are some issues that will be brought up in section 5. Thread number 0 is seen as leader and is therefore tasked with special instructions. Before the main loop is started, thread 0 initializes the tree and all other threads are waiting for its completion inside the main loop. Once all threads are ready, the construction of the tree is beginning. Each thread iterates all the bodies and checks if the body is located within any of its sections. If that is the case the thread adds the body to the section. When all threads are done, the next step is removing waste nodes. All threads loop the sections and if the thread is assigned to that section it removes the waste nodes of that section. Once all threads are complete with that, thread 0 summaries the weights from all 16 sections to the two layers above. Thread 0 also calculates the center of mass for all nodes directly after this. The forces are then calculated in parallel by only calculating the forces on bodies that are assigned to that thread by the formula $thread\_id + (num\_threads * i)$. The same thing is then done to move the bodies in parallel. The only thing left is the recursive traversal of the tree and setting the nodes mass to 0 and the hasBody flag to false. This

is done in parallel the same way that waste node removal is, by only processing the sections that are assigned to that specific thread. Thread 0 then takes care of the two upper levels since they also have to be reset.

5 Performance Evaluation

In the project description it said that the performance was to be evaluated using 120, 180 and 240 bodies in such a way that 120 bodies took 15 seconds to compute. This works well with the first two versions of the program since the algorithms are straight forward to parallelize. However, in the last two version the overhead of calculation was too much to see any reliable results. Therefore, the tests were modified to better test the performance of the four versions. The different tests can be seen in table 1 was run in the ranges found in the table. Each individual run of the programs was running five times and averaged.

5.1 Sequential Brute-Force

It is known that the algorithm has $O(n^2)$ in time complexity beforehand. We can see that in figure 2 the performance seems to trend towards just that. The y axis displays the execution time while the x axis displays the number of bodies.

5.2 Parallel Brute-Force

Each of the lines displayed in figure 3 correspond to a specific number of bodies. The y axis displays execution time while the x axis displays the number of threads used. It is evident in the figure that more threads equal speedup of the simulation. However, it seems like the speedup is decaying for each new thread that is added. This might be because the overhead of having threads created and sharing data structures is too costly and the calculation is becoming too fine grained.

5.3 Sequential Barnes-Hut

It is known that Barnes-hut has $n * log(n)$ in time complexity, and it is what can be observed in figure 5. The increase in execution time is almost linear. The y axis displays the execution time while the x axis displays the number of bodies.

5.4 Parallel Barnes-Hut

Each of the lines displayed in figure **??** correspond to a specific number of bodies. The y axis displays the execution time while the x axis displays the number of threads used. The speedup of this implementation is not as drastic as the parallel brute-force. This can be because of the need of syscalls which cannot be done in parallel. It is because of that reason that the structure is preserved during the main loop. Another reason for the speedup is that some things that currently are in parallel could be faster to do in sequential fashion. One problem with this implementation of parallelization is that the workload is not evenly distributed. If all bodies are in the upper right quadrant in the top left corner, the thread associated with that section will do all the work whilst the rest of the threads just idle. This could be prevented by doing division of workspace based on number of bodies in clusters instead of quadrants.

6 Conclusions

The implementation and parallelization of the brute-force algorithm for N-Body simulation is really easy and straight forward while the Barnes-hut is way more complex. The Barnes-hut reached a significant speedup thanks to the parallelization of the construction of the tree as it is the costliest operation in the main loop. One interesting thing was that without the syscalls the parallel speedup of the Barnes-Hut was significantly more.

Version	Bodies	Bodies Increment	Iterations	Threads
1	250-500	50	110000	1
2	250-500	50	110000	1-4
3	80000-230000	30000	100	1
4	80000-230000	30000	100	1-4

Table 1: Performance tests

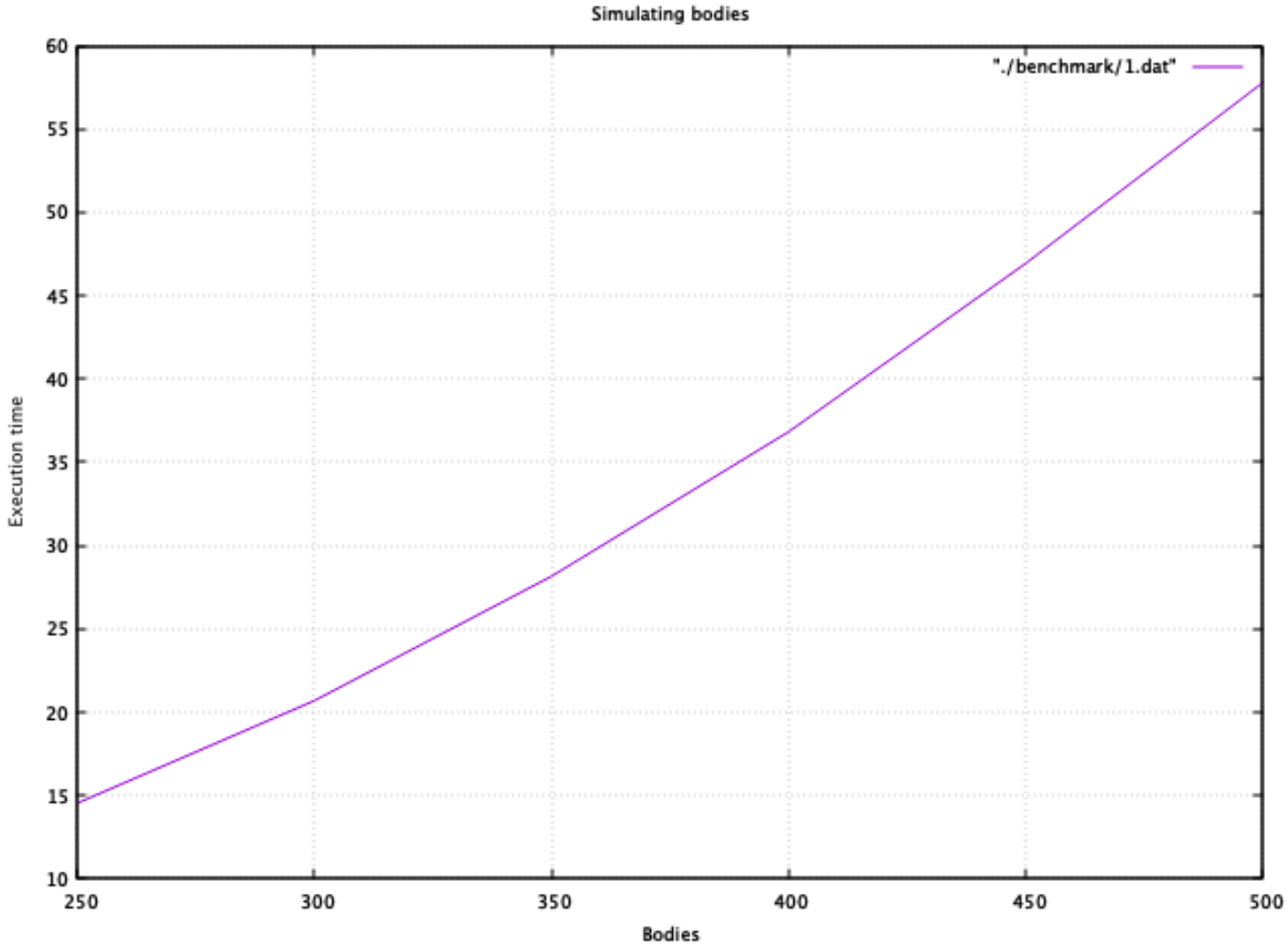

Figure 2: Performance of sequential brute-force

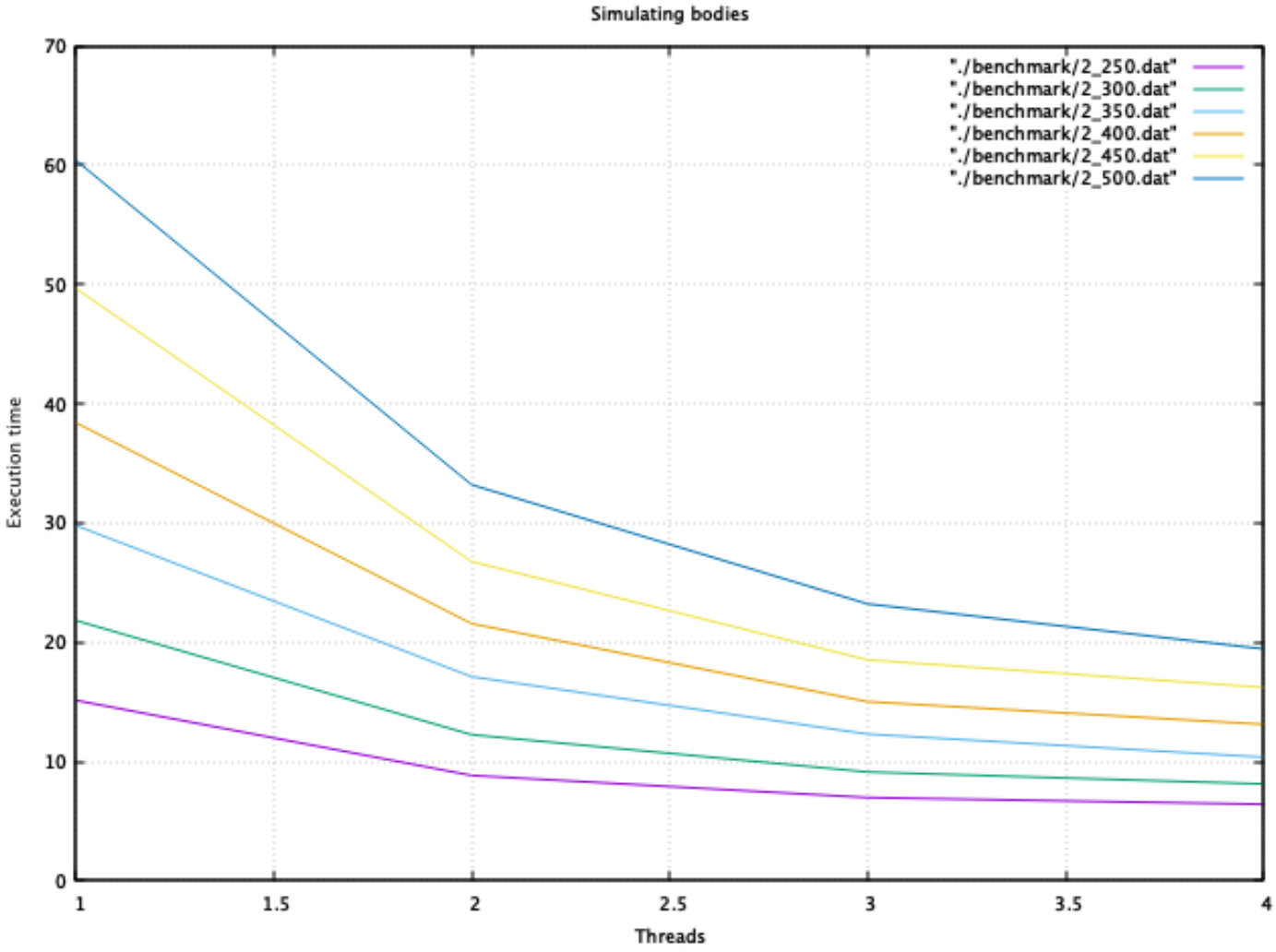

Figure 3: Performance of parallel brute-force

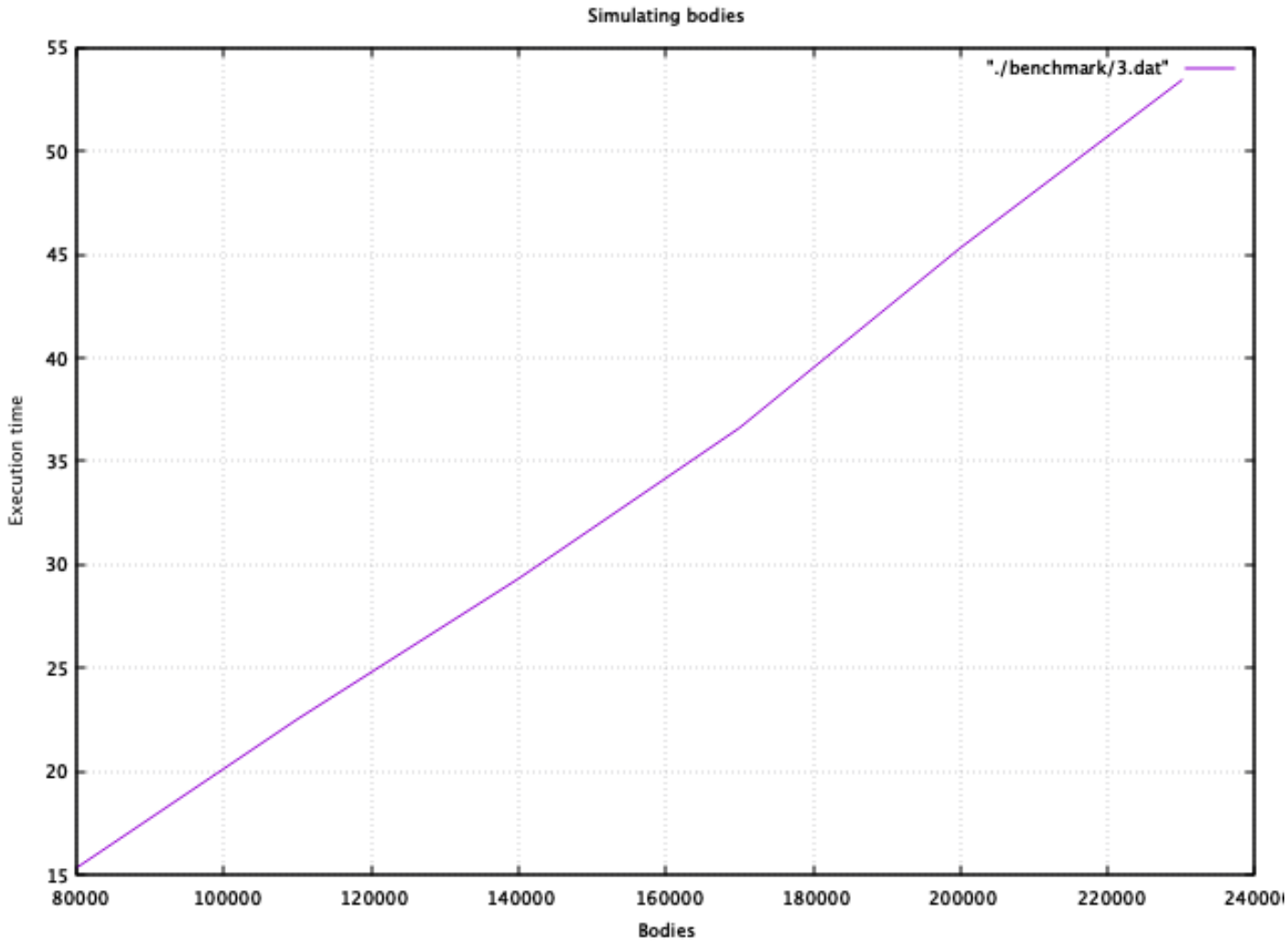

Figure 4: Performance of sequential Barnes-hut

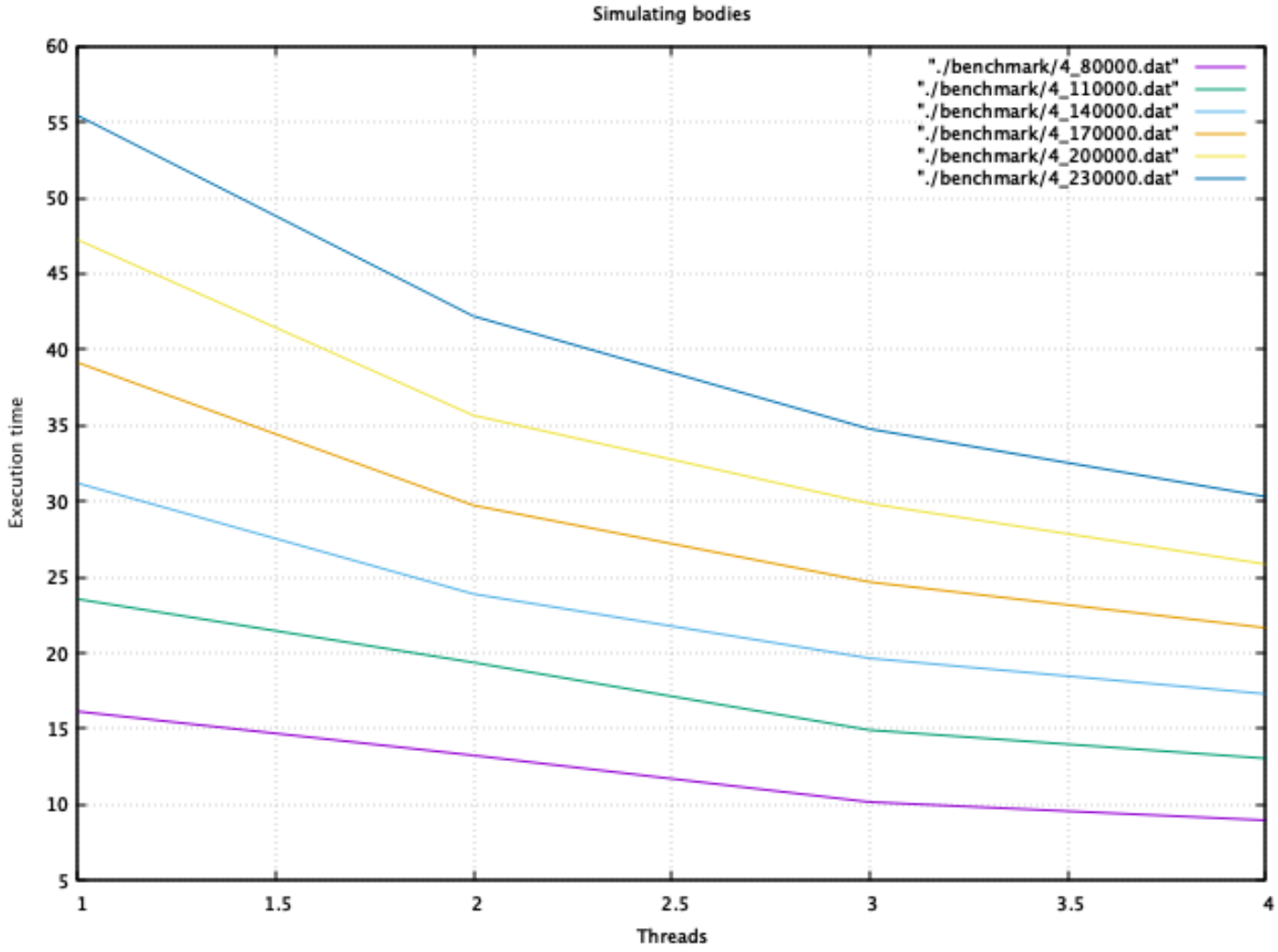

Figure 5: Performance of parallel Barnes-hut