The Gravitational N-body Problem

March 2021

Olof Jonnerby olofjonn@kth.se
Eric Söderberg ersode@kth.se

1 Introduction

The gravitational n-body problem is the problem of predicting motion and movement of independent celestial objects such as stars and planets. This report shows and evaluates the performance of four different algorithms that solves or approximates the n-body problem.

Processors used:

AMD Ryzen 5 3600 12 logical processors 6 physical cores

&

Intel Core i7-4790K, 8 logical processors, 4 physical cores

Operating System: Windows 10 Home x64

The atlantis.sys.ict.kth.se server was also used to assist in benchmarking, however it did not provide difference in relative scaling compared to our home computers.

2. Programs

We built and evaluated four different algorithms that solves or approximates the n-body problem.

Algorithm 1

The first algorithm uses a brute force approach with sequential execution. It uses Newton's law of universal gravitation over all sets, or tuples, of bodies in the simulation.

Algorithm 2

The second approach to the n-body problem optimizes the first algorithm by dividing the calculations to multiple threads. In this brute-force method, calculations are "first-heavy", meaning that bodies that are first in the list require a lot more calculations than the ones later in the list. To adjust for this, we used a "stripe" work distribution where one thread is assigned one body, another thread the next body etc. To do this, each thread reads and increments an atomic integer that represents the index to the list of bodies. Concurrently they will all perform calculations on the bodies that they have been assigned. After calculating and updating the current forces acting on each body, the threads wait for each other to finish. When done, they concurrently update the position of each body with the updates values derived from the previous calculated forces. All this is repeated for a number of steps, representing time, which is given by the user.

Algorithm 3

The third approach to the problem uses another type of optimization: the Barnes-Hut approximation. The way it works is to divide the bodies into a grid. The grid is represented by a quad tree - similar to a binary tree but with each node (if not a leaf) has four children instead of two. The grid has cubic cells which contain the average center of mass - calculated by average position weighted by mass, for all bodies within it. The distribution of the grid over the bodies is carried out recursively by dividing each cell/node in four equal parts until each subdivision contains one or zero bodies.

Figure 1: the grid containing the bodies

The algorithm iterates over all bodies in the system and compares them to adjacent neighbours (represented by nodes). When calculating the force affecting a body, other clusters of bodies that are sufficiently far away can be approximated as one single body with a single center of mass. This reduces the amount of calculations required, and so it performs faster than the earlier brute force method.

Algorithm 4

The fourth and final approach improves the Barnes-Hut approximation by parallelizing parts of it. In each iteration of the algorithm, a new tree is created to encapsulate the bodies after they have updated their positions. The creation of the tree, as well as the calculations for each body, can be parallelized. For the tree, recursive calls can be assigned as tasks to a work pool. Running threads can then extract tasks from the work pool and execute them in parallel. In this algorithm, blocks of bodies can be assigned directly to each thread. This is because calculations are not first-heavy as contrary to the brute force method where the first bodies in the list requires a lot more calculations compared to the last bodies.

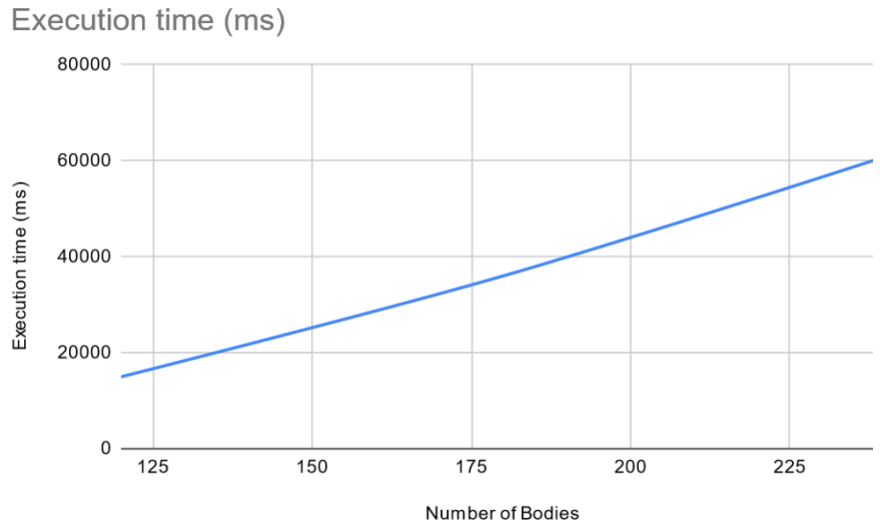
A custom executor, GatedExecutor, was created to deal with the issue of recursive parallelism when constructing the quadtree containing the bodies. A custom executor was needed for two reasons: Firstly, we want to be able to determine when all tasks are finished, similar to the process of joining on a single thread, without actually terminating the threads. This is because the overhead of creating the threads in each timestep is undesirable. Secondly, the GatedExecutor allows for threads to temporarily lend their computational power to the queued tasks, if any such are available. This was implemented because the center of mass and total mass of each cell can only be known once the full tree is calculated. As such, a worker would need to wait for all child nodes to complete their calculations, before being able to calculate this. Not parallelising this part would result in a larger sequential portion of the program. However, this call cannot block, since that would result in executor threads waiting for each other to take the subtasks, effectively causing a deadlock.

This solution allows each executing thread to suspend their current execution in order to assist in performing tasks while waiting for some condition, such as the subtasks being completed. However, it has significant overhead, as our executor must synchronize a large portion of its methods, to ensure that these temporary assisting threads never end up blockingly waiting for a task; the assisting threads *must* eventually return to their original work.

Evaluation

Table 1: run times for the first brute force algorithm (steps: 230k)

BF sequential	120 (bodies)	180	240
Execution time (ms)	14983.0471	36007.8084	60864.498

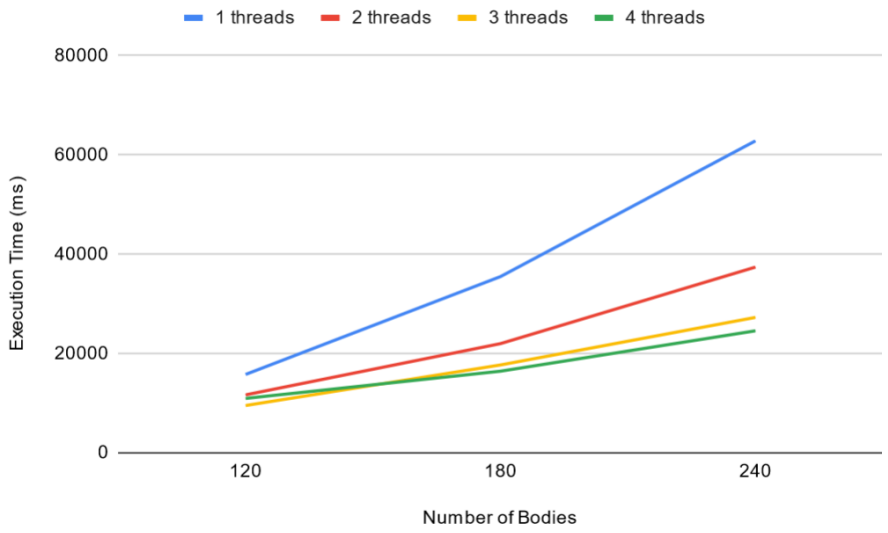


The first algorithm scales $O(n^2)$ with the number of bodies in the system, which is why the execution time increases quite dramatically (>100%) for a smaller increase of 50% in problem size.

Using the same amount of steps for all other algorithms and amount of bodies:

Table 2: run times for the parallel version of the brute force algorithm (BF Parallel)

Threads	120 (bodies)	180	240
1	15791.9567	35505.8806	62844.2237
2	11674.7114	21999.4833	37416.1752
3	9532.3046	17700.7557	27279.9041
4	10962.5331	16444.4491	24579.0268



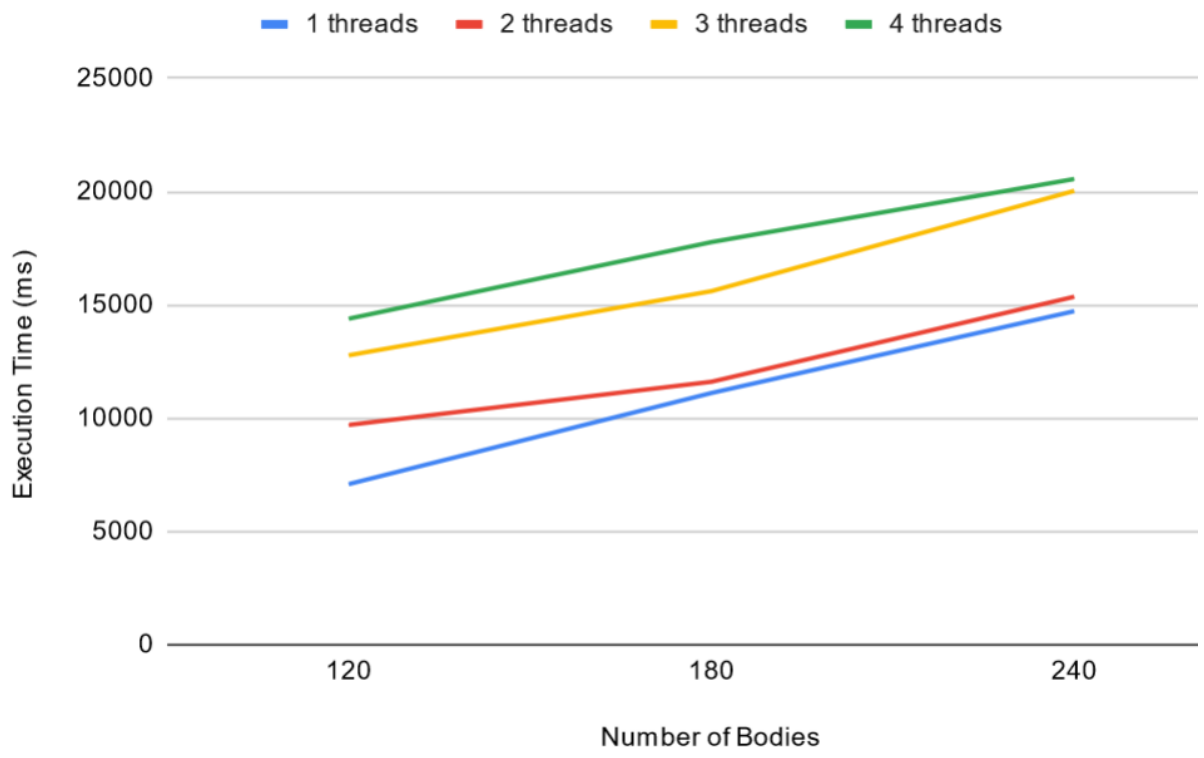
Somewhat surprisingly, executing the algorithm with 4 cores runs slower than with 3 cores on 120 bodies. This might be due to the extra overhead from having an additional thread. After each calculation phase, threads wait for each other in a barrier. The more threads, the more synchronization needed thus increasing the waiting time in the barrier. When the number of bodies increases however, the parallelization gains greater effect and we get a speedup when running extra threads.

Table 3: run times for Barnes-Hut approximation algorithm with sequential execution

	120 bodies	180	240
BH sequential (ms)	8159.5531	11771.0006	16935.2466

Table 4: run times for the parallel version of the Barnes-Hut approximation algorithm

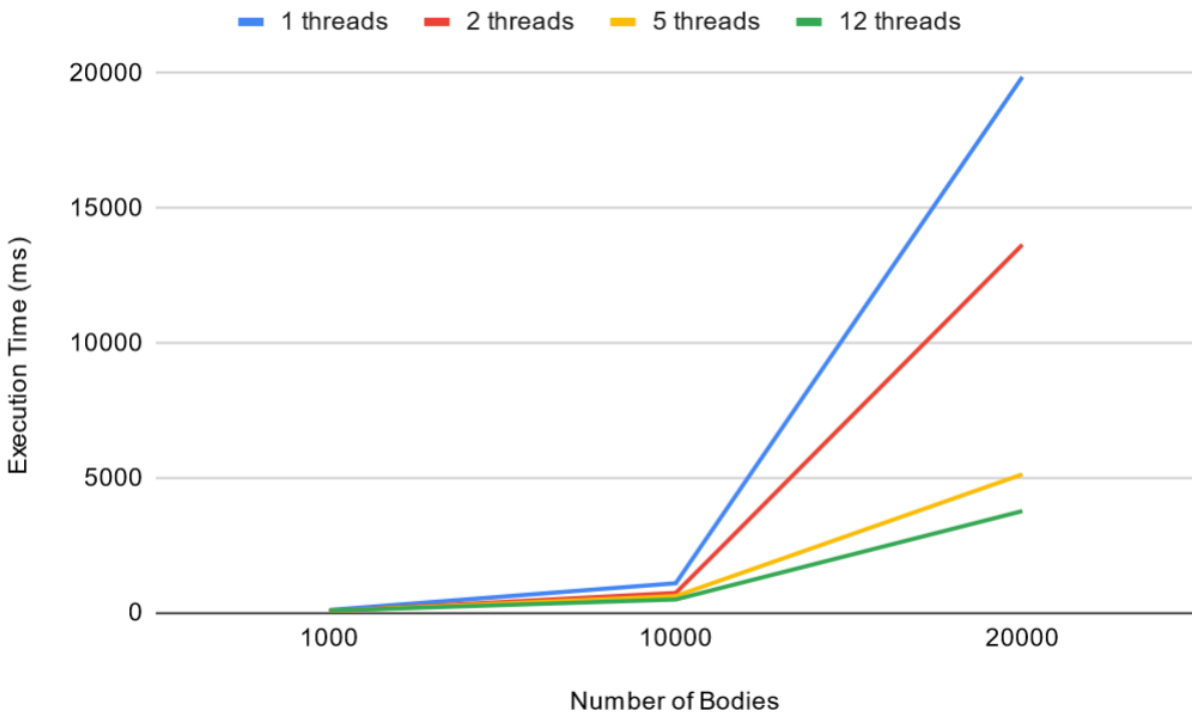
Threads	120 bodies (ms)	180	240
1	7089.5263	11111.3572	14723.1994
2	9701.8409	11606.9794	15359.5972
3	12775.2364	15607.2406	20033.3707
4	14389.6185	17771.8316	20555.6618



In this somewhat naive parallel implementation, blocks of bodies are assigned to each thread. Only one thread creates the tree, which means a large part of the program is still sequential. The time gained from parallelizing some of the work is therefore dwarfed by the extra overhead from synchronizing and running multiple threads.

Table 5: *Run times for the parallel version of the Barnes-Hut approximation algorithm with low steps but larger number of bodies, using parallelized quadtree creation*

Threads	1000 bodies (ms)	10000	20000
1	117	1108	19870
2	80	748	13650
5	82	632	5143
12	94	507	3782



Conclusion

The parallel version of the brute force algorithm achieved a notable speedup compared to the sequential version, but was still overall slower than the sequential Barnes-Hut approximation algorithm. The first parallel version of the Barnes-Hut algorithm performed worse overall, but in the second version (parallelized quad-tree) we found much better performance scaling with respect to number of bodies.

In conclusion, we found that the additional overhead of creating tasks appeared to greatly outweigh the gained performance, at least with our chosen methods. The individual tasks are small enough that the synchronization overhead becomes a significant part of the overall execution time.

Our main takeaway from this project is the importance of avoiding forced barriers if possible, as they can create bottlenecks that greatly throttles a program's performance due to improper scheduling.