Project

N-Body Simulation

Olof Eliasson

Spring Term 2024

Introduction

The N-body problem is a problem in the realm of physics that describes how celestial bodies interact with each other through gravitational forces. A solution to the problem exists for two bodies interacting, where we can easily predict where either of them will be at any point in time. A solution for three bodies also exists, however, with some constraints. In the case that this report will focus on, a world with N bodies in it, it is yet not possible to predict how the bodies will interact. We instead need to rely on actually simulating the interactions that occur from the start and stepping time forward in small increments in order to capture some of the details of what would happen in the future. The shorter we step time forward, the more accurate an image we capture.

Platforms

The simulation program described in this report was implemented using purely Java. The performance evaluations shown later in the report were taken on both a linux laptop and a desktop running Windows using the timing info logged by the simulation software.

Implementations

There are three important parts that are shared among all of the variations of algorithms for simulating the N-body problem. These are the `Simulation`, `Window`, and `Body` classes which are responsible for creating most of the structure that exists in the program.

The `Simulation` class is the biggest part of the program since it is responsible for encapsulating the code that performs and handles every step of the simulation. It is implemented as an abstract class and is done so to easily provide common functionality to subclasses that inherits it but also to expose a more common interface to interact with the simulation from other parts of the program. The biggest part of this parent class is the `Run` method it provides which starts the simulation and handles calling the methods that perform the simulation and also calculating and displaying runtime info to the user. The four types of algorithm we want to implement are all child classes to this one and overrides its `CalculateForces` and `UpdatePositions` methods to provide for example the Barnes-Hut or the brute force algorithm to the programs main loop.

The other important class is the `Window` class which handles the initialization and updating of the window that displays a visualization of how the simulation is going at each time step. It is implemented as a `Singelton` to provide easy access to the window instance. The `Window` class uses the Java Swing framework to instantiate the window as a `JFrame` and creates a `JComponent` that is used as a canvas that the program can draw to using the data linked to from the simulation.

The `Body` class is meant as a wrapper for important data that the program needs to keep track of regarding each body and giving us an easy way of accessing and passing that data around.

Sequential Standard Method

The easiest method to simulate the N-body problem is using a normal brute-force algorithm, which is the slowest, most inefficient way of doing it. It works by calculating all gravitational forces acting on a body by all other bodies and performing this operation for all bodies. These forces are calculated using Newton's gravitational law, which gives us the force between two bodies. Using the total force exerted on a body we can calculate how its velocity will be changed and from that we can find out where in the simulation region it should appear.

Parallel Standard Method

The parallelized version of the brute force algorithm is simply that same algorithm just split up into multiple chunks that are run in parallel. This is done by creating a `Runnable` class containing the code that performs the calculations that brute force algorithm does. The difference with this code is that the indices that the for loops that loop over all bodies in the simulation

haven been changed so that each thread that is created is assigned different portions of the bodies data structure. We then create a set amount of threads and supply them with the `Runnable` class and start their execution. There are no critical sections for the threads to execute but they do however to be synchronized between sections of code since certain parts of the code require the result computed in the previous section. This is done using two sets of barriers, one internal barrier to the threads which is solely responsible for synchronizing the threads with each other. This is needed between the section that computes the forces of each body and the section that updates the position in each thread, since we need to know all of the forces in order to compute an accurate position. The second barrier is used to synchronize all threads with the main loop of the simulation program and is located both in the main loop and at the end of each threads execution loop. This barrier is meant to halt the execution of the main loop and preventing the program from proceeding to next simulation step until the one we are computing right now is done, which is when all executing threads have reached the end of their computation. At the end of the simulation a flag is set which signals all threads that they should stop executing.

Sequential Barnes-Hut Method

The Barnes-Hut algorithm is a much more efficient way of solving the N-body problem than the brute-force approach and it does this by sacrificing some of its accuracy for performance. This inaccuracy comes from the approximations that the Barnes-Hut algorithm sometimes makes to the forces at work in the simulation realm. However, these inaccuracies are quite negligible since these approximation are only applied when they are deemed to be almost the same as the real thing. The core of the algorithm is the Barnes-Hut tree which is simply a normal quad tree data structure with some extra data stored in each cell other than the points within the cell.

The quad tree is built up by inserting the bodies in the simulation one by one. As a body is inserted it is first passed to the root cell of the tree which covers the entire simulation area. When a body is inserted into a cell it first checks if it has room to store this body inside of it, since the Barnes-Hut tree has a condition that each cell in tree may only contain one body. If this cell is empty then we simply insert this body into this cell and proceed to the next body. However, if this cell already contains one body then we need to subdivide the space that this cell occupies into four quadrants and then insert both the body already stored in this cell and the new body we want to insert in whichever child cell that each of the bodies lies in, by calling the insert function again on that cell. The last case to handle is the case where multiple bodies has already been inserted into this cell in which case

its child cells have already been created. In this case we simply check in which child's space the body lies in and call insert again on that child cell.

The next step is computing the data used for the approximations which is essentially calculating a sort of pseudo body for each cell in the tree. This body only exists to be used in calculations and is not actually part of the simulation and is in reality not one body but a combination of all bodies inside of cell. The theory is that over a long distance the force applied to a body by multiple bodies inside a common cell would be pretty close to the same force applied to a body that resembled the combination of all those bodies. This is due to the fact that the distances between the bodies we want to combine could be viewed as negligible considering the distance between them and the body we want to calculate the force for. Therefore we calculate a center of mass and total mass for each cell in the tree, which will act as our sort of pseudo body that we can use to approximate a group of bodies inside of a cell. This can dramatically reduce the number of force calculations that needs to be performed since for example four or even hundreds of bodies can be represented as one single body which would mean only one force calculation instead of hundreds. Computing these pseudo bodies is done by recursively traveling the tree until we reach the bottom of the tree where each cell only contains one body at which point we know that the approximations will simply be equal to the bodies stored in those cells. All cells above can calculate their center of mass and total mass values simply by creating a weighted average of its children's center of mass values and the total mass of a cell is simply the sum of its children's total masses.

Using the tree to calculate the force applied to a body can be done by traversing the tree recursively until we've accounted for all forces that might affect this body. We stop travelling down a branch if we either hit the bottom, i.e, a cell with just one body in it, in which case we use the normal gravitation law to get the force. We also stop going down a branch and use the approximation stored at current cell if we satisfy the theta condition. The theta condition is a ratio between the side length of the current cell and the distance between the body we're calculating for and the center of mass of the current cell. If this ratio is less than the theta threshold we then utilize the approximation to approximate all the bodies that are stored with in this cell instead of going down the tree further and performing more calculations. If the condition is not satisfied we are to close to the pseudo body approximation and would risk too great of an error in the computations, so we go down another level in tree and repeat.

The simulation code now follows the pattern of first calculating the pseudo

bodies after which we can calculate the forces being applied to each body. The next step is updating the positions of each body which is the same code from the previous implementations and after the positions have been updated we need to reset the quad tree and insert all bodies again since it is possible that the structure has changed.

Parallel Barnes-Hut Method

The parallelized version of the Barnes-Hut algorithm uses the same methods used to parallelize the brute force algorithm mentioned before. We simply create a `Runnable` that has the task of calculating the forces and updating the positions of a subset of bodies assigned to it using the Barnes-Hut tree. We again use two barriers to synchronize all the threads with each other and also the main execution loop. There is however one extra bit of synchronization necessary in this case, which is at the beginning of the threads execution. We use the barrier used for synchronization with the main loop to ensure that all threads wait to begin their computations until the main thread has finished computing the pseudo bodies of the tree, since they are necessary for the force calculations to function correctly.

Performance Results

380 000 time steps The test results shown below in table 1 by letting the algorithms run for 380 000 time steps before finishing and recording the amount of time that was needed to finish a simulation of that size. Each algorithm was tested with a varying amount of bodies either being 120, 180 or 240 bodies.

Algorithm	1 Workers	2 Workers	3 Workers	4 Workers
Seq. Brute (120)	14.74			
Seq. Brute (180)	33.36			
Seq. Brute (240)	58.67			
Par. Brute (120)	19.67	19.46	25.11	30.97
Par. Brute (180)	38.38	30.68	36.18	42.01
Par. Brute (240)	64.4	45.77	51.51	60.71
Seq. Barnes (120)	34.98			
Seq. Barnes (180)	60.16			
Seq. Barnes (240)	86.27			
Par. Barnes (120)	44.75	35.73	36.69	39.38
Par. Barnes (180)	75.09	53.51	52.54	52.62
Par. Barnes (240)	104.44	73.61	66.66	65.23

Table 1: Runtime in seconds for a simulation with specified number of bodies

As we can see from the results shown in table 1 we can see that the brute force method is superior during all testing scenarios and that in most cases the parallelized versions gets worse as we increase the worker count. This does make sense if we compare them to the results in table 2.

In table 2 we can see the same tests being performed, however with more bodies present in the simulation which meant that the simulation was only run for 4000 time steps in order to keep the runtimes sort of in the same range.

Algorithm	1 Workers	2 Workers	3 Workers	4 Workers
Seq. Brute (1200)	15.33			
Seq. Brute (1800)	34.05			
Seq. Brute (2400)	60.04			
Par. Brute (1200)	15.09	8.65	6.69	6.69
Par. Brute (1800)	33.91	18.9	14.37	13.5
Par. Brute (2400)	59.95	32.97	23.46	21.89
Seq. Barnes (1200)	4.75			
Seq. Barnes (1800)	8.34			
Seq. Barnes (2400)	12.33			
Par. Barnes (1200)	5.65	3.45	2.86	2.6
Par. Barnes (1800)	9.6	5.78	4.63	4.02
Par. Barnes (2400)	12.31	7.38	6.4	5.32

Table 2: Runtime in seconds for a simulation with specified number of bodies

If we look at both tables at once we can see that the results from the first table shows that in the case of the Barnes-Hut approximations there probably aren't enough bodies to the point where each approximation significantly reduces the number of calculations to perform, which means that the overhead added by the quad tree slows us down. In the case of the parallelization either not giving us better results or getting faster and then slower its probably again due to the lack of bodies in the simulation. This means that each thread gets assigned a pretty small chunk of data to process which they finish quickly however all the overhead of synchronizing makes them overall perform worse than if we skip synchronizing and simply let one thread handle it.

However if increase the scale of the simulation as is the case with the results from table 2 then we see more of what we expect to see. Here we can the see the expected time complexities of each algorithm more clearly and

also the speedup that we can gain from utilizing more cores. We can see the exponential $O(n^2)$ complexity of the brute force algorithm and how the $O(n \cdot lg(n))$ complexity of the Barnes-Hut algorithm has a slower time increase. We can also almost see a linear relationship of the performance increase from adding another thread and that it mostly plains out at around 4 workers.

Conclusion

In this report we have looked at what the N-body problem is and at two ways that one might try to solve it. We have also compared these ways to each other and discussed some of the performance issues that they have and arrived at the conclusion that the Barnes-Hut algorithm is mostly only worth it if the scale of the simulation is large enough.

During this project i got to learn more about the parallelization workflow that exists in Java which is something that i have not worked with before. In addition to this i also got to learn what the Barnes-Hut algorithm is and how it works.