THE NBODY PROBLEM

COURSE PROJECT REPORT, ID1217, SPRING 2018

DAVID JACOBSSON
DJACOB@KTH.SE
ID1217

1.INTRODUCTION

The n-body problem, in physics, is the problem of predicting the individual motion of n celestial bodies and the gravitational effect they have on each other.

In this report I will document an attempt at implementing a software solution to simulate n bodies in a 2 dimensional space. I will present two different algorithms for solving the problem, both in sequential form, and parallel. The two methods are “The bruteforce method”, which is a $O(n^2)$ solution, and the “Barnes Hutt method” which is a O(n*log(n)) solution. The programs are implemented in Java.

I will present the background for solving the issue in general and then describe the two programs in detail. I will also present execution data derived from performance test, studying speedup between the two methods and between serial and parallel.

Lastly I will attempt to draw conclusions from my findings and discuss what could potentially be changed for further improvement.

2. PROGRAMS

The basic concept of solving the issue is to first create a initial state for all the bodies in the simulation. For the solution to be solvable, all gravitational impact must be known for all bodies. To acquire this, current position and momentum must be known for each body. From that information, all acting forces can be derived, which themselves over time affect a change in momentum.

2.1 PHYSICAL PRINCIPLES

The following physical principles are used in the programs.

The gravitational force F: $F=\frac{G\times m1\times m2}{d^2}$

where **F** is the force two bodies are exerting on each other. **G** is the gravitational constant(6.67e-11). **m1** & **m2** are the individual masses of the two bodies and **d** is the distance between them.

The distance between two objects d: $\sqrt{(x1-x2)^2+(y1-y2)^2}$

Given two positions in two dimensions $P(xi,yi), i=1,2$.

Derived via the Pythagorean theorem $a^2+b^2=c^2$, the distance between them is acquired by the calculating the the component vectors of the vector between the two points which can be found by subtracting the axis values of body1 and body2 with each other.

Change in velocity ?v(deltav): $dv = \frac{f}{m} \times dt, \{m \times a = f\} \rightarrow dv = a \times dt$

Derived from Newtonian formula for momentary acceleration. **dt** is the momentary change in time (deltat).

Change in position ?p(deltap): $d = \left(v + \frac{deltav}{2}\right) \times dt$

Where **v** is the current velocity of the body.

2.2 BRUTE FORCE ALGORITHM

As the name suggests, this method solves the problem by simply iterating over all existing bodies, performing all necessary calculation one by one.

The program is built up by a main loop and three supporting function calls. The value for change in time(deltat) is hardcoded into the program. The programs main loop is called with user inputing N bodies and numStep calculation sets. It then sets up and initializes the bodies with a starting position and velocity. The data is stored as an object containing all bodies contained within an array.

It then runs a for loop for <numStep> iterations. For each iteration it runs the function calls; "public void calculateForces()" and "public void moveBodies()" in that order. They calculate the forces acting on all bodies within the object and then move them to their new position. Once the loop is complete the program prints execution time and exits.

For the **parallel** implementation, the main method creates numWorkers, supplied by user, threads and calls the thread classes. The threads to essentially the same operations as the serial implementation, both calculateForces(int id) and moveBodies(int id) are called in that order, but the methods now take an id value which is used to separate the bodies in between the threads. Effectively the threads will alternate in a cyclic fashion, see table below.

Iteration	Body1	Body2	Body3	Body4	Body5	body6
1	w1	w2	w3	w4	w1	w2
2	w3	w4	w1	w2	w3	w4
3	w1	w2	w3	w4	w1	w2

The table above refers to a simulation with 6 bodies and 4 threads.

2.3 BARNES-HUTT ALGORITHM

This method makes use of the fact that if considering a set of bodies, they can be replaced with a placeholder body with the combined mass and a new center of gravity, effectively decreasing the amount of calculations needed when clusters of bodies are a sufficient distance from each other.

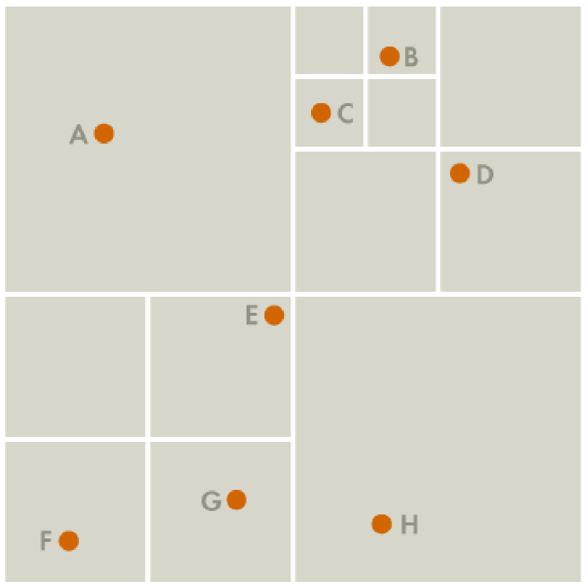


Illustration 1: Universe of quadrants, with bodies A-H

The universe of bodies is represented by a tree structure where the top node contains the replacement mass and center of gravity is stored as as single Body object within the node. Each node also has four children, all representing a quadrant of the universe. And the bottom of the tree, in the external nodes(leafs) the individual bodies are stored. See illustration 1 and 2.

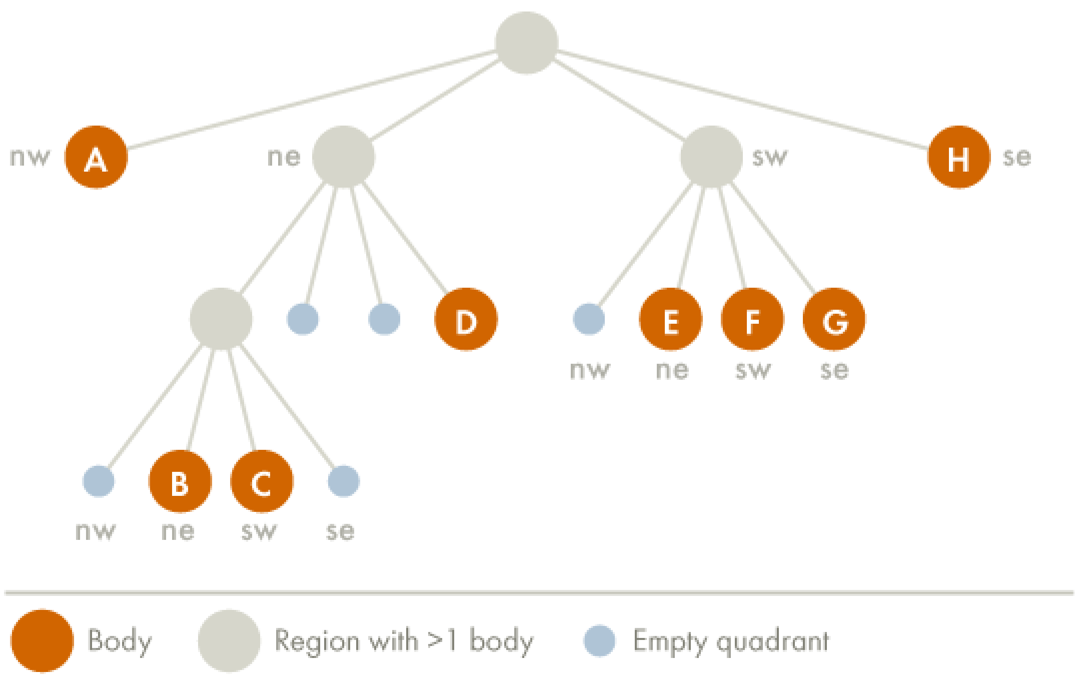


Illustration 2: A tree structure

The **serial** implementation of the Barnes-Hutt method uses a Quadrant, Tree and Body class.

- The Quadrant class contains coordinates on its own borders and methods to determine if a point lies within itself.
- The Tree class contains one Body object and 4 Quadrant objects. It contains a recursive function that calculates all forces on all bodies, one at a time, by parsing the tree. If $s \div d < Theta$ then the forces are calculated from the replacement node. **s** is the length of the quadrant for some internal node, **d** is the distance between a body and the internal nodes center of gravity and **Theta** is some given fraction that can vary from one simulation to the next. If Theta is small then more calculations will be performed, giving higher accuracy but slower performance. The class also contains methods for inserting bodies into the tree.
- The Body class contains values representing postion, velocity and forces acting upon itself. It contains methods for determining the distance between itself and other bodies, update forces acting on itself and creating a replacement body with combined mass and new center of gravity.

The main method creates the bodies and starts a loop that creates a new tree, inserts the bodies in their new positions, calculates all forces, moves all bodies and repeats for given number of steps.

For the **parallel** version, the threads are spawned by the main loop and iterate over essentially the same loop, using the same classes and method calls, but is assigned bodies to calculate by a bag of tasked contained within a monitor. The threads are synchronized with a barrier at three locations because each step must be complete before any thread can move one to the next.

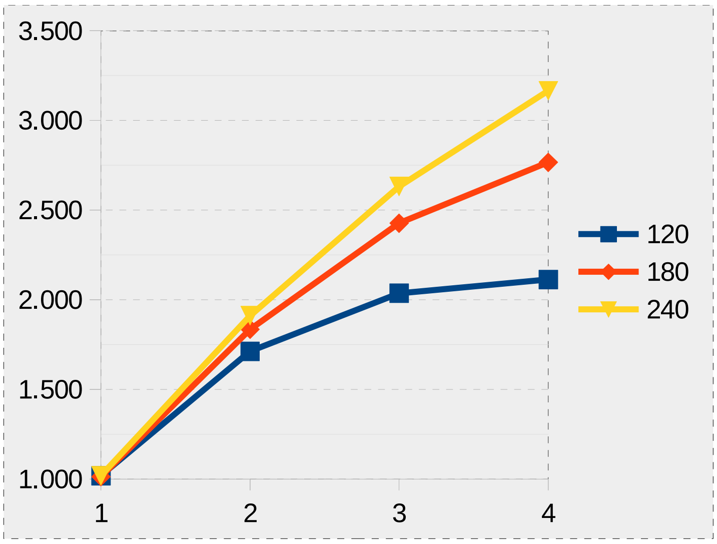
3. EVALUATION

Raw data from simulated runs using the following parameters:

numSteps = 54000

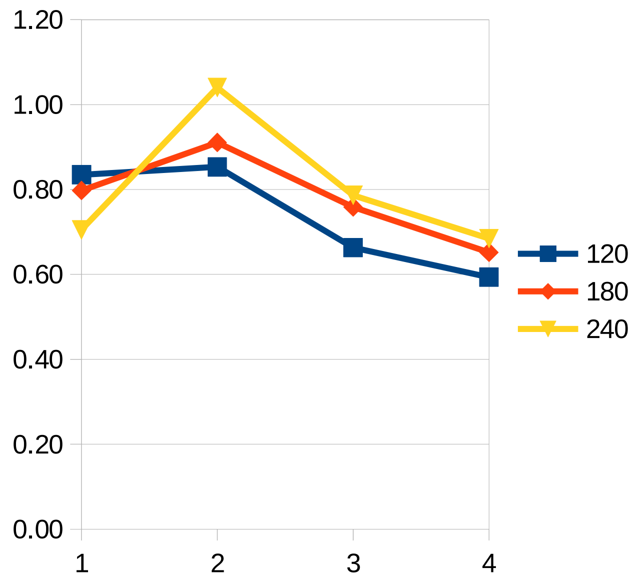
Mass 0.1, Theta 0.5, ?t 0.0005, length of base quadrant 4.

We can see from the data that for the brute force version the speedup is becomes double at around 4 threads, and even triples when the workload increases(240 bodies). The growth slows down considerably from 2 to 4 threads however.



NSQ PARALLEL SPEEDUP

LOAD	WORKERS	SPEEDUP
120	1	1.018
	2	1.712
	3	2.036
	4	2.112
180	1	1.013
	2	1.835
	3	2.427
	4	2.766
240	1	1.018
	2	1.911
	3	2.631
	4	3.165



BH PARALLEL SPEEDUP

LOAD	WORKERS	SPEEDUP
120	1	0.83
	2	0.85
	3	0.66
	4	0.59
180	1	0.80
	2	0.91
	3	0.76
	4	0.65
240	1	0.71
	2	1.04
	3	0.79
	4	0.68

For the Barnes-Hutt version we see negative speedup.

4. CONCLUSION

Although I was unsuccessful in developing a functioning parallel Barnes-Hutt version it was plain to see that the serial version outperformed the brute force version by a large margin. The error is likely situated in the thread class where a issue creating more and more performance overhead amasses. A large portion of the total runtime is spent inserting moved bodies into a new tree which is a process that can only be done by a single thread. There may also be some error in the distribution of bodies. If they are not evenly distributed the tree will become very deep which slows down force calculation.

Regarding the Brute force version, I would have achieved a better speedup if I had used a method called reverse strip when distributing the bodies over the workers. By changing the order of distribution for every cycle of delivery the overall workload gets more evenly shared between the threads.

5. REFERENCES

Illustration 1: The Barnes-Hut Algorithm, Tom Ventimiglia & Kevin Wayne

Illustration 1: The Barnes-Hut Algorithm, Tom Ventimiglia & Kevin Wayne