Uppsala University

High Performance Computing and Programming

The N-Body Problem

Authors:

Anton Sjöberg

Alessandro Piccolo

Carl Christian Kirchmann

May 22, 2016

UPPSALA UNIVERSITET

1 The problem

The N-body problem is a problem encountered when trying to calculate the gravitational interaction between celestial objects. The solution of this problem can be used to predict the behavior of planets, stars and moons. The force, velocity and position is calculated for each object in each time step. The complexity of solving this problem with a double for-loop is $O(N^2)$. For massive systems the complexity has to be reduced by using an alternative way of computing the forces and update the velocities and positions. The algorithm that will be used in this assignment is the Barnes-Hut algorithm which has a complexity of $O(Nlog(N))$.

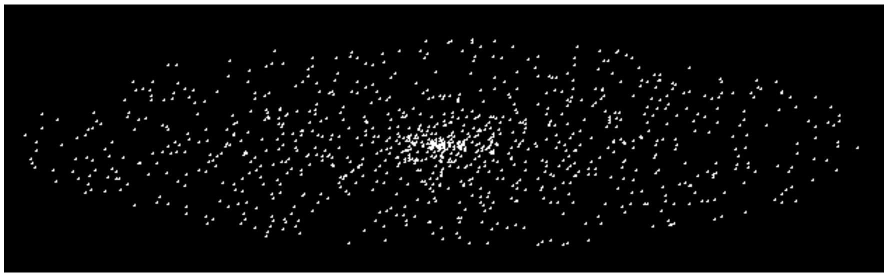

Figure 1: Shows a visualization of a galaxy with 1000 particles.

2 The solution

The general idea of the Barnes-Hut algorithm is to regard multiple objects clustered together at a large distance from another object as one object with a single center of mass that is the sum of the cluster of objects. One has to set the limits for ratio of the size of the cluster and the distance between the center of mass in the cluster and the object it is gravitational interacting with. The limit has to be set so that the necessary precision is obtained. By doing this the number of objects interacting will be reduced since there is a chance that clusters of objects will be considered a single object.

The Barns-Hut algorithm can be divided into the three following steps as seen in algorithm 1. These steps are described more in depth in the following subsections.

Algorithm 1 Barnes Hut Algorithm

```
1: Structuring the particles into a quadtree.
2: Calculating the center of mass and mass for each node in the quadtree
3: Calculating the force on each particle by traversing the quadtree.
```

2.1 Building the quadtree

The root node of the quad tree is the entire square in the plane. The rootNode branches out into four children, which corresponds to one of four squares that the big square can be broken into. Each of these squares can in turn be broken down with four corresponding children and so on until the tree is completed. A node that has no children is defined as a leaf. Only leaves can have a particle and there can only be one particle in each leaf.

When creating the tree, the particles are being inserted one by one. The procedure is described in algorithm 2 and 3.

Algorithm 2 QuadtreeBuild

```
1: Empty Quadtree
2: for i = 1 : n do
3:     Insert Particle into root using QuadInsert.
4: end for
5: Traverse the tree removing leaves that have no children.
```

Algorithm 3 QuadInsert

```
1: if The subtree contains more than 1 particle then
2:     Calculate what Child the particle lies in
3:     QuadInsert the particle into that Child
4: else if The subtree contains 1 particle then
5:     Set node to leaf
6:     Add four children to the node
7:     Move the particle already in the node to the children which corresponds
   to its coordinates.
8:     Insert the new particle into the child that corresponds to its coordinates.
9: else if The subtree contains no particles then
10:    Store the particle in the node
11: end if
```

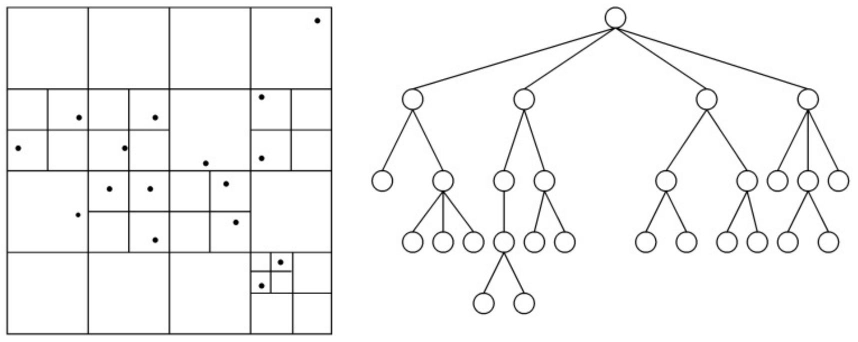

Figure 2: Shows a visualization of a quadtree with 16 particles.

2.2 Calculating the center of mass

Basically the idea is to sum up particles into groups and calculate the total mass and coordinates of that group. Each node will therefore contain the total mass and weighted sum of its sub-tree. Algorithm 4 is done by a simple post order traversal. The cost is no more than O(n) with n representing the number of nodes.

Algorithm 4 Calculating the center of mass (CoM) and total mass

```
1: Returns structure returnV
2: procedure CENTRE_OF_MASS($treeNode*\ node$)
3:     if (node contains one particle) then
4:         Save particle coordinates and mass in node
5:         Save particle coordinates and mass into returnV
6:         return returnV
7:     else
8:         for i = 1 to 4 do
9:             $CoM_{sum} += child_i.coord \cdot child_i.mass$
10:            $mass_{sum} += child_i.mass$
11:        end for
12:        Save particle coordinates and mass in node
13:        Save particle coordinates and mass into returnV
14:    end if
15:    return returnV
16: end procedure
```

2.3 Calculating the forces

The forces were calculated by using the algorithm described in algorithm 5. The cost of the algorithm is mainly set by the value of θ_{max}. If precision is not of as much concern as the speed one can set the value of θ_{max} to the following values:

$\theta_{max} \in (0, 1)$. For this assignment the error of of the position for the particles was not to exceed 10^{-3}. The value for θ_{max} was therefore iteratively changed until the error was close to 10^{-3}. The maximum value for θ_{max} was found to be 0.19.

Algorithm 5 TreeForce

```
for i = 1 to particle do
    f(i) =TreeForce(particle,node)
end for
procedure TreeForce(particle,node)
    f = 0
    if subtree contains one particle then
        r = distance from particle to node
        $\bar{r}$ = vector between particle and node
        $f = G * m * m_{cm} * \bar{r}/r^3$
    else
        r = distance from particle to center of mass in subtree
        $\bar{r}$ = vector between particle and center of mass in subtree
        D = size of subtree box
        if $D/r < \theta_{max}$ then
            $f = G * m * m_{cm} * \bar{r}/r^3$
        else
            for i = 1 to 4 do
                f = f + TreeForce(particle,node(i))
            end for
        end if
    end if
end procedure
```

2.4 Data Structures

The particle struct was used to hold all information for each particle. The returnV struct was used by the center of mass algorithm to return the center of mass and its coordinates. To build the quadtree the treenode struct was used, it had a pointer to the particle it contained. This pointer was set to NULL if it had no particle. It also had center of mass and its coordinates, this was the calculated center of mass for that node and its subtree. The attribute leaf was used to remove leaves from the tree which had no particles. To compute the position of each box *llc* (lower left corner) and *lvl* were used. The treeNode struct also contained pointers to its children.

3 Other possible solutions

One option that was looked into but was not implemented was the use of switch cases. In the function QuadInsert, as sen in algorithm 3, a lot of if-statements are used to find where to move the particle already in the node and where to insert the new particle. If switches are implemented instead of these if-statements a performance increase is to be expected. This is partly due to the fact that the if the branch predictor misses a performance decrease follows.

Another method instead of using a quadtree built with treenode structs would have been to rearrange particles in a vector containing all particles. This would mean that no child struct pointers would be needed and the vector we used for particles would have been sufficient. We would have had to add some more attributes to each particle and also sort them in such a way so that they would represent a tree. Unfortunately there was not time enough to implement this solution as well.

Earlier versions had bad scaling of the tree building algorithm. In every step of inserting a particle it counted how many other particles where in the subnode that the particle were added to, which was time consuming. In the final version of the code this issue is solved by introducing a particle counter in the struct of the node. Time went down dramatically and made the final code scale much better.

4 Replicating results

Replicate the results by typing the following commands into the terminal.

```
$ make brutegalsim
```

```
$ make galsim
```

`$ brutegalsim N input_/filename nsteps` Δt θ_{max}.

`$ galsim N input_/filename nsteps` Δt θ_{max}.

Where N stands for the number of particles to be simulated and nsteps stands for the number of time steps used. The brutegalsim will produce a result file called Bruteoutput.gal and the galsim barnes hut algorithm will produce a BHoutput.gal file. For all the measured time cost the parameter ϵ was set to 10^{-3}, $nsteps = 200$, $\Delta t = 10^{-5}$ and $\theta_{max} = 0.19$. When running the compare code with parameter values as above and a particle size of 2000, "ellipse_N_02000.gal" the maximum difference in position becomes $0.85 \cdot 10^{-3}$ which is less than the required limit of 10^{-3}.

5 Hardware and compiler

The following specs were used for the tests.

gcc (GCC) 4.9.3
Scientific Linux 6 x86_64: gullviva.it.uu.se
AMD Opteron (TM)

6 Performance and discussion

In Figure 3 the percentage of time spent in each part of the Barnes hut algorithm 1 is visualized. Several parts are hard to identify since the time it takes to run that part of the code is minimal compared to other parts. The most time consuming parts were building the tree, calculating the force and calculating the center of mass. Time commands were put before and after the force calculation, the center of mass calculation, the tree building, the update of positions as well as before and after the update of velocity. The major computation time, as can be seen in figure 3, is for calculating the force.

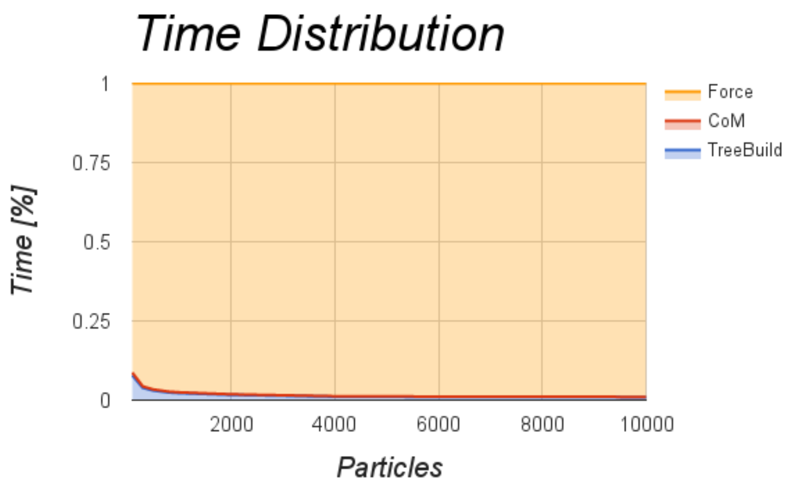

Figure 3: Shows the time distribution for various parts of the Barnes Hut algorithm vs number of particles.

The reason for this performance of the algorithm is probably that the force calculations does not increase as much in cost when the tree gets bigger since it uses the approximation with θ_{max} mentioned in algorithm 5. The tree build time cost is minimal even when the problem size is increased.

The code was run for brute force and barnes hut five times for each number of particles. That means that N was set to: $100, 200, 300, 400, 500, 600, 700, 800, 900$ $, 1000, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 1000$ and the time it took for each execution was saved. In order to not spend too much time on this the timesteps *nsteps* was adapted to the number of particles so that the runtime did not exceeded one minute.

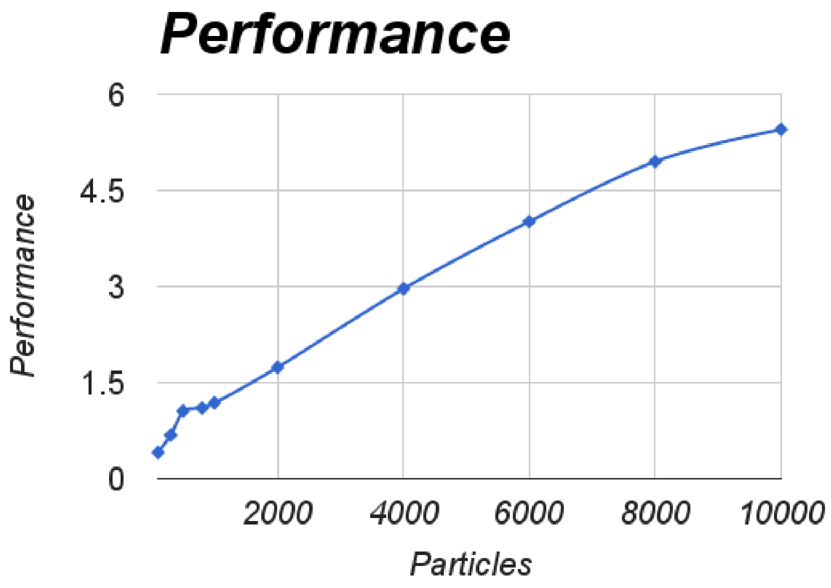

Figure 4: Computation time ratio for Barnes Hut and Brute Force vs number of particles.

When the performance in figure 4 is above 1 the barnes hut is better than brute force. As can be seen in figure 4 that happens when the number of particles is more than 500. This is due to the fact that barnes hut complexity is $Nlog(N)$ and brute force is N^2. The performance of barnes hut vs brute force is increased as the number of particles increases.