COMP90025
Parallel and Multicore Computing
Project2 Report

Mengchen Wang
mengchenw1@student.unimelb.edu.au

Chuan Yang
chuany1@student.unimelb.edu.au

Introduction

For our project topic, we choose to improve the performance of N-body problem. The problem is to predict the motion of numbers of physical object due to gravitational interactions with each other. This problem has application in predicting motion of celestial object such as moons and stars.

In order for the prediction to work, we need a quasi-steady state for the N-body system, which means each body has an initial speed. Otherwise the whole system will collapse to a single point due to the gravitational force. This property will be generated randomly for each body at the beginning of the program. We will also consider each body to be a point mass that does not occupy any space. Lastly, we will place a heavy body in the center of the system to acting as the axis of the system.

During calculation, we will assume that during the time frame, the force acting on each body is constant. This assumption is only valid if the interval is small. In our case, we use an interval of 1E5 second, which is larger than 1 day. But since the universe we defined is on the magnitude of 1E18m, this time frame is valid.

Method

The straight forward approach is to calculate the force acting on each body from every other body at a given time, thus calculating the net force acting it. For a system of n bodies, this method has the complexity of $O(n^2)$.

A better solution is given by the Barnes-Hut algorithm (Barnes and Hut, 1986). In this approach, we will construct a Barnes-Hut tree. The root node will be the entire universe, each child node divides the space of its parent node into octant (or quadrant for 2D cases). This process is repeated when all nodes contains one or zero body. For each node, we could calculate its mass by summing up the mass of its child node, and a center of mass calculated from weighted average over masses of child node. In figure 1 there is a simulation of constructing a Barnes-Hut tree.

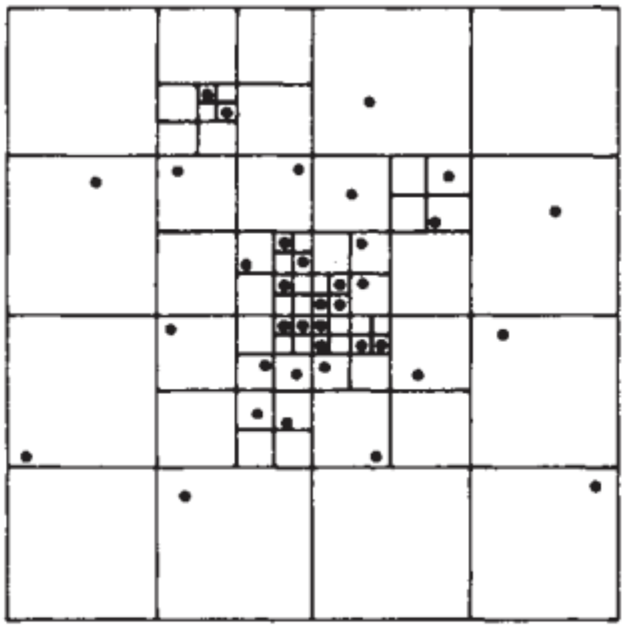
Figure 1. Constructing a 2D Barnes-Hut Tree

When calculate the force acting on a body p, we will check if the dimension of the cube l divided by the distance from p to that node, d is smaller than a threshold value q. This means that the node which encompasses the cluster of body is sufficiently far from p. If so, we can approximate all bodies inside that node to a single body. Otherwise, we will perform it recursively to its child nodes. For an average case, the Barnes-Hut algorithm provides a complexity of O(nlogn). The process is illustrated in figure 2.

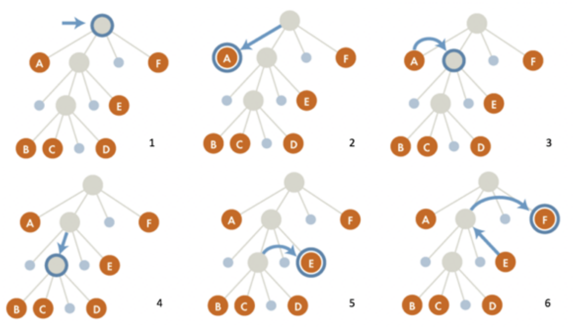

Figure 2. Process of approximating force force acting on point A using Barnes-Hut Algorithm

Here are the detailed steps listed in figure 2. We would like to compute the net force acting on point A.

1. Calculate D from A to the root, and checks for l/D<q. It fails so we will perform the recursive call to its child nodes
2. The first child is A. Since A cannot exert force on itself, we can skip this node.
3. Next is the internal nodes at the same level of A. Again, we check for l/D<q. It fails, and we move on to its child node.
4. The first non-empty child is the node that encompasses B, C, and D. This time l/D is indeed less than q. We can approximate these 3 points using this node.
5. Moving on to the next node at the same level. It is the external node E, no approximation is needed.
6. We have finished the function all at this level. Now we need to go back to the recursive call at root level, which leads to another external point F.

Performance Improvement

We start the project with a sequential implementation of the brute force algorithm and comparing the result from each parallelization implementation with it to check for performance improvement.

The first thing we tried to do is to distribute the workload into several threads for from the nested for loop update force method. We can parallelize the outer loop using Openmp to distribute the workload of calculating changes in the points.

Next, we defined a Barnes-Hut Tree structure and try to implement the Barnes-Hut algorithm. We did the naive Openmp optimization here by spreading the workload of calculating points among threads. The Barnes-Hut Algorithm also takes advantage of MPI in the following way. The master nodes get the n-body input (number of bodies, mass, initial velocity, initial position) and broadcast it to all the slave nodes. Upon receiving this information, all nodes start building the same Barnes-Hut tree independently on their own nodes and update a portion of the points according to their own process id. Then these nodes will broadcast the change it makes to other fellow nodes for the next iteration to begin. After the last simulation is done all results are return to the master node where user can get the result on bodies' velocity and positions.

Result and Discussion

Implementation	N=1000	N=5000	N=10 000
Brute Force (BF)	2.8	70.14	283.3
Barnes Hut (BH)	0.75	4.458	9.706
BF + OpenMP (4 threads)	0.794	16.6	66.18
BH + OpenMP (4 threads)	0.69	3.978	8.511

Table 1. Performance of Different Algorithm with Different Input Size

As we can see, BH contributes a big improvement in performance, and the improvement is more obvious when data size gets larger, demonstrating the reduced complexity of O(NlogN).
When we run Brute force with OpenMP, we can also see an obvious speed up about 4 times. Because in brute force, the main complexity is updating the bodies, which is O(N*N), by using OMP to parallel

the updating part with 4 threads, we can get up to 4 times speed up.
When we are using Barnes Hut tree, the complexity of building the tree is about O(NlogN), and that of updating the bodies is also about O(NlogN). The process of building the tree is not able to be paralleled because the tree is dynamically increasing, and if we use ‘omp parallel atomic’ to parallel it, the grammar is not allowed. Then we tried to parallel the phase of updating bodies with OMP by using 4 threads, so the updating part can get a speed up of 4. However, the percentage of time consumed by building tree is larger than updating, so the total speed up is not quite obvious.
So, without multiple processes, we can get a quite good speed up against original sequential with Barnes Hut tree and OpenMP.

Implementations	Building Tree(s)	Update Force(s)	Total (s)
BH(Sequential)	0.4695	0.268	0.736
BH(OpenMP with 4 threads)	0.6208	0.079	0.703
BH(MPI with 2 nodes)	0.4958	0.137	0.644
BH(MPI with 4 nodes)	0.648	0.091	0.871
BH(OpenMP+MPI with 2 nodes)	0.7488	0.070	0.839
BH(OpenMP+MPI with 4 nodes)	0.8538	0.054	0.943

Table 2. OMP vs MPI at N=1000 Average Results

Table 2 gives us the detailed performance on various Barnes-Hut implementations, which are combinations of MPI and OMP. For this test, all inputs are at size n = 1000 and the result are averaged over 5 runs. We did see a weird trend with different implementation, which is that the time of building tree increases when we try to use OMP or MPI. We list some analysis to find the possible reasons here. However, we did see update force method taking less time as more computing resource is involved.
First, looking at the building tree column alone, the time varies with different implementation. However, In our code, building tree is implemented independent of threads or nodes. A possible reason for this is that when we use OMP on multiple processes, the multiple threads take lots of memory, so the host process encounters insufficient memory when building the tree, which means the process has to wait for available memory, causing this process to be slower with sequential implementation. Also, we notice the performance degrade with increasing number of nodes. Taking a closer look at these results, we can see the calculation part (Update Force) has an increase in in the speed, but because building the tree takes the majority of the time, the final outcome is slower than using less nodes.

Reference

Barnes, J., & Hut, P. (1986). *A hierarchical O (N log N) force-calculation algorithm*. nature, 324(6096), 446.