Parallel N-Body Simulation

Abdellah BELAID

January 2023

Abstract

This study discuss the problem of N-Body systems and propose a simulation following the **Barnes-Hut** algorithm, it then compares a dynamically balanced algorithm to the sequential one, while evaluating it and discussing a possible improvements.

1 Introduction

The N-body problem refers to the simulation of the motion of a large number of particles under the influence of gravity. It is a fundamental problem in physics and has many practical applications, such as simulating the motion of celestial bodies, the formation of galaxies, and the behavior of fluids. Simulating large numbers of particles is a computationally challenging task that requires the use of efficient algorithms and parallel computing. The Barnes-Hut algorithm is a widely used algorithm for simulating the N-body problem that achieves good performance by approximating the force on a particle using the center of mass of a group of particles. In this paper, we present an implementation of the Barnes-Hut algorithm in C++ that uses cells to represent the distribution of particles and is parallelized using the MPI library. The code uses a load balancing strategy based on a space-filling curve to ensure that the processors are evenly busy when the particles are moving.

2 Algorithm Description:

The Barnes-Hut algorithm is based on the idea of representing the distribution of particles using a tree structure. The algorithm uses a recursive tree structure to divide the space into smaller and smaller regions, and uses the center of mass and size of the cells to approximate the force on a particle. The implementation uses a cell class to represent the tree structure and a vector class to represent the particles. Each cell contains the center of mass and total mass of the particles contained within it. The algorithm uses the relative size of a cell to the distance of the particle being simulated to decide whether to compute the force due to all the particles in the cell or just the center of mass. The algorithm uses the opening criterion 'theta' to decide whether to compute the force due to all the particles in the cell or just the center of mass. A value of theta = 0.5 is used in the given code. This means that if the ratio of the size of the cell to the distance between the particle and the center of mass of the cell is less than 0.25, the force due to all the particles in the cell is computed. If the ratio is greater than 0.25, the force due to the center of mass of the cell is computed.

3 Implementation Details:

The code uses a cell class to represent the tree structure of the Barnes-Hut algorithm. The cell class contains the center of mass and total mass of the particles contained within it, as well as information about the size and location of the cell. The cell class also contains functions for adding particles to the cell, calculating the center of mass, and calculating the force on a particle.

The code uses a vector class to represent the particles under the name **Vec2**. it also contains some useful functions and operators to simplify the calculations the distance between two particles, and for updating the position and velocity of a particle based on the force acting on it.

The code uses the MPI library for parallelization, it uses the MPI_Bcast function to broadcast the mass and position arrays to all members in the group. This ensures that all the processes have the same initial state.

Then, we use the MPI_Scatter function to divide the initial velocity array and scatter it to all processes. Each process will receive a portion of the array, which will be stored in the velocity variable.

Next, the code runs a loop for a certain number of iterations, during each iteration the following steps are performed:

1. The tree_generator() function is called to generate the Quadtree.
2. The compute_cell_prop() function is called to compute the properties of each cell in the Quadtree.
3. The compute_force_space() function is called to compute the forces acting on each particle.
4. The delete_tree() function is called to delete the Quadree.
5. The compute_velocity() function is called to update the velocities of each particle.
6. The compute_positions() function is called to update the positions of each particle.
7. The MPI_Allgather function is used to gather the updated position array from all processes and store it in the position array.

Finally, if the rank is equal to 0, the code calls the write_positions() function to write the final positions to a file, and also prints the elapsed time.

The load balancing strategy used in the code is based on a space-filling curve (SFC). In this approach, the particles are mapped to a 1-dimensional space using an SFC, and the particles are partitioned among the processors based on this mapping. The specific SFC used in this code is the Morton (Z-order) curve. This approach allows the particles to be distributed evenly among the processors, ensuring that the processors are kept busy.

The code also uses an opening criterion 'theta' to decide whether to compute the force due to all the particles in the cell or just the center of mass. The value of theta = 0.5 is used in the given code. This means that if the ratio of the size of the cell to the distance between the particle and the center of mass of the cell is less than 0.25, the force due to all the particles in the cell is computed. If the ratio is greater than 0.25, the force due to the center of mass of the cell is computed.

The code also has a main function that sets up the simulation, initializes the particles, and runs the simulation for a specified number of iterations.

4 Results and Evaluation

To Evaluate our parallelism approach we conducted some Tests over a wide range of particles between 100 and 3000 randomly distributed, for 1000 iterations with time pace of $0.01s$, using a Quad Core processor which means an optimal speedup would be less than 400% over the sequential algorithm , As shown in the Figures below , the performance obtained was quite satisfying with achieving up to 300% in time complexity for big number of particles.

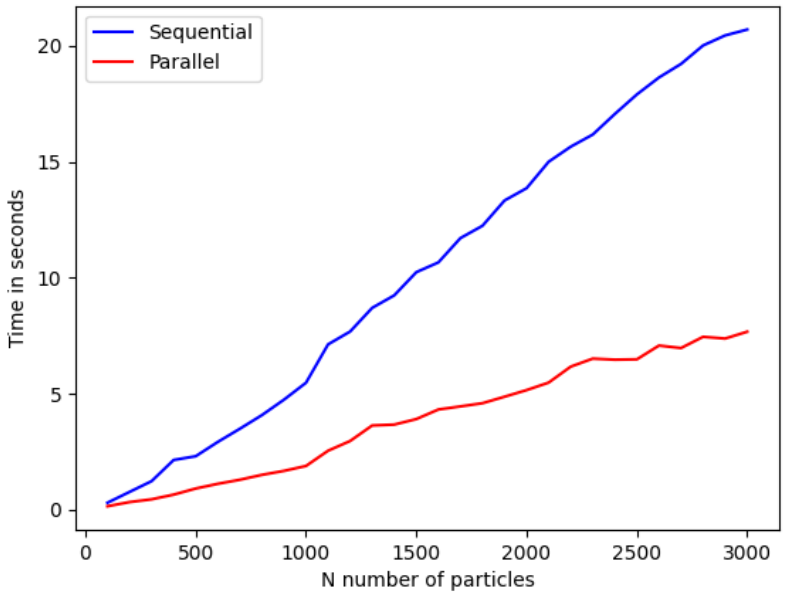

Figure 1: Graph of time elapsed for N-Body simulation

Input	Sequential	MPI ($size = 4$)	speed-up
N = 100	0.3093s	0.1576s	196%
N = 1000	5.4775s	1.8929s	289%
N = 5000	38.8818s	13.2679s	293%
N = 10000	88.7389s	34.5772s	260%

Table 1: Comparaison of performance.

5 Evaluation

Even with MPI, load balancing can still be a challenge for the Barnes-Hut algorithm because the Quadtree structure of the algorithm can lead to uneven workloads among the different processes. Some processes may have more leaf cells to handle than others, which can result in uneven workloads

that means depending on the particle distribution this can have a significant impact on the overall running time.

For that, a more sophisticated approach described by J. Salmon uses the method of orthogonal recursive bisection (ORB) to recursively decompose the system into sub-volumes similar to the approach described above. However the strategy for decomposition is such that an equal number of particles lie on either side of the division. The decomposition into sub-volumes continues for as many levels as required, producing a well balanced tree. Each of the processors are then assigned one or more sub-volumes, balancing the computational load on the cluster.

Or in general we can use Space-filling curve partitioning:
One way to balance the load is to use a space-filling curve, such as the Hilbert curve, to partition the particles into subregions. Each processing unit can then be assigned a subregion and work on the particles within that subregion independently.

6 Conclusion

In this paper, we have presented the Barnes-Hut algorithm and discussed its implementation, parallelization, and optimization techniques. The simulation results indicate that the proposed approach can improve the performance of the algorithm and make it more efficient for simulating large-scale gravitational systems. It is worth noting that the Barnes-Hut algorithm is not the only N-body algorithm and some problems may be more suited to other algorithms such as direct N-body or tree code.