COMP90025 Parallel and Multicore Computing

Project 2 Report: N-Body Problem

Hanbin Li *hanbinl1*

Wenqing Xue *wenqingx*

October 2019

1 Introduction

The goal of this project is to simulate the two-dimensional N-body problem by implementing a MPI with OpenMP program. The N-body problem is aimed to simulate the individual motions of a group of particles under the influence of forces in physics and astronomy area. In a precise system, there are N particles moving around based on the impact of mutual gravitational attraction. According to Newton's second law and Newton's law of gravity, the gravitational force between two particles can be calculated as:

$$F_{ij} = \frac{Gm_i m_j (p_j - p_i)}{\left\|p_j - p_i\right\|^3}$$

where m is mass, p is position vector and G is gravitational constant occurred [1]. Inside the system, the computations of simulation for N particles are massive, since that position and velocity need to be re-calculated for each particle in each time step. For N particles, each particle is impacted by $N-1$ distinct forces that total to produce a net force (see Algorithm 1). In general, $N(N-1)$ times of calculations are worked out, which gives the complexity of $O(N^2)$.

Algorithm 1 N-Body Basic Algorithm

```
1: for each time step do
2:     for each particle $i$ do
3:         compute force
4:     end for
5:     for each particle $i$ do
6:         update position
7:         update velocity
8:     end for
9: end for
```

2 Methodology

There are two algorithms presented in detail; one is the basic algorithm with $O(N^2)$ complexity and the other one is the Barnes-Hut algorithm with $O(N \log N)$ complexity. The Barnes-Hut algorithm makes use of a tree data structure so that essentially nodes only interact with its own nearest neighbours at each level of the tree. It reduces the complexity with a trade-off between precision and cost [2].

2.1 Basic OpenMP+MPI Algorithm

Our main design idea for basic OpenMP+MPI algorithm for achieving $O(N^2)$ is to let communication only occur among the processes when the algorithm is computing the forces F_{ij}. In order to compute the forces, each process needs the mass and position of all other particles at every time step. Therefore, we used $MPI\_Allgather$ to deal with this type of communication as it could gather data from all process and distributes the data to all processes. Our design decisions are described as following.

2.1.1 Parallelization with OpenMP

Assume the work done per step is approximately equivalent, we expect a well-balanced load by using a block partitioning that assigns N/P particles per process. From the basic sequential algorithm explained above, we could see that the two inner loops are both iterating over particles. It is straightforward to put in a parallel for construct as there is no risk of race conditions due to loop-carried dependencies. For the first inner loop, the parallel for construct ensures for any particle i only one thread will access $F[i]$. Thread i will access $m[j]$ and $p[j]$ with read-only actions. For the second inner loop, thread i will only access quantities associated with particle i. Therefore, no race conditions are introduced in this case. Additionally, a schedule clause is added to ensure that the block partition is used.

2.1.2 Parallelization with MPI

Data Structure In the sequential version of our program, we collect most of the data (velocity, position, mass) of a particle into a single struct. If we keep using this data struct in MPI, it would require a derived data type. However, communication with derived datatypes tends to be slower than with native MPI data types. Therefore, we decided to use general arrays to store the velocities, positions and masses. The positions of all N particles are stored in the array p. We also defined an MPI datatype MPI_VECTOR to store two contiguous doubles using $MPI\_Type\_contiguous$ and MPI_Type_commit. For the computational convenience, we assume that N (the number of all particles) could be evenly divisible by P (the number of processes). Thus the chuck size taken by each process will be N divide by P.

Process

- Each process stores all the m_i in an entire global array of parcels masses since the masses are always needed but never changed.
- Each process uses a single array with size N particles for position.
- Each process uses a pointer *recv_p* to the start of its block of positions.

Thus, on process p, *recv_p* = *positions* + *chunk*.

With above implementation ideas, the pseudocode for the OpenMP+MPI is shown below.

Algorithm 2 N-Body OpenMP+MPI Algorithm

```
1: get and boardcast input data
2: for each time step do
3:     #pragma omp for schedule(static, $N/P$)
4:     for each local particle do
5:         compute force on $i$
6:     end for
7:     #pragma omp for schedule(static, $N/P$)
8:     for each local particle do
9:         compute position and velocity on $i$
10:    end for
11:    allgather position to global array pos
12: end for
```

Note that the process 0 as the root node will read the inputs and prints all the results. The initial conditions are read into three arrays with size of N. Then the masses and positions are broadcasted since they are always needed for computing the forces while the velocities array is scattered since the velocities are only ever used locally.

2.2 Barnes-Hut Algorithm

The main difference between from our previous implementation is that, in Barnes-Hut algorithm, we used a tree based approximation scheme instead of directly adding all the forces on a single particle, which reduces the computational complexity from $O(N^2)$ to $O(N \log N)$.

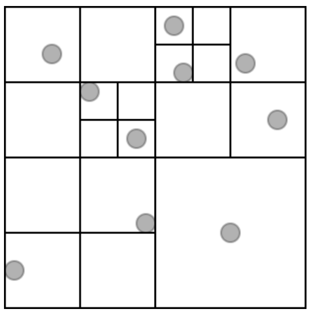

Figure 1: Subdivision in 2-Dimension (as quadtree)

The Barnes-Hut algorithm depends on the divide-and-conquer algorithm to find clusters of particles and a hierarchical tree-based representation to construct them. The main idea is to group and approximate nearby particles as a single particle. Initially, a quad-tree is created by recursively subdividing the cell into four smaller subcells of equal size, until there is at most one particle in each subcell. The individual particle will be inserted into the quad-tree through the main node. Each node in the quad-tree has 4 children and represents a region of the 2-dimensional space. The subcell will be discarded if there is no particle existed inside. Then, each cell in the quad-tree consists of the total mass and centroid of all particles downwards. Once the entire tree is constructed, the net force acting on each particle would be calculated, starting from the root of the quad-tree. If the group is quite far away, by using its centre of mass, its gravitational effects will be approximated. The centre of mass of a group is calculated by the average position weighted by mass of a particle in this group.

For instance, if two particles have masses m_1 and m_2, and positions (x_1, y_1) and (x_2, y_2), their total mass and centre of mass (x, y) are given by [3]:

$$m = m_1 + m_2;$$
$$x = (x_1 m_1 + x_2 m_2)/m;$$
$$y = (y_1 m_1 + y_2 m_2)/m.$$

At each step, if the size of the cell divides the distance of its centroid is smaller than a given attribute θ, the centroid will be used to compute the force on the particle. Otherwise, the subcell it contains will be looped in the same way.

2.2.1 Data Structure

We created a data struct *body* to represent a single particle in the system with records of mass, position, velocity and force. And another data struct *node* is used to represent a single node in the quad-tree with records of total mass, the centre of mass and positions.

2.2.2 Process

We used static distribution, therefore each process will get equal number of particles and do the same number of iterations.

Algorithm 3 Barnes-Hut Algorithm

```
1: get input data
2: for each time step do
3:     construct quad-tree
4:     compute force
5:     update positions
6:     update velocities
7: end for
```

3 Evaluation

3.1 Setup

All the measurements are taken place in Spartan platform by batching the *slurm* scripts. The amounts of particles we investigated are from 1200 to 6000, with a gap of 1200.

3.2 Performance

3.2.1 Basic OpenMP+MPI Algorithm

The execution times tend to be halved by doubling the number of processors and the speedup is approximately 2.

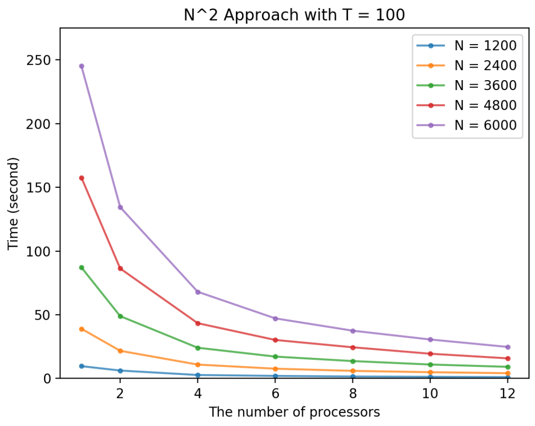

Figure 2: Basic OpenMP+MPI Algorithm Performances

3.2.2 Barnes-Hut Algorithm

The execution times do not follow the trend shown in N2 approach, which is not expected.

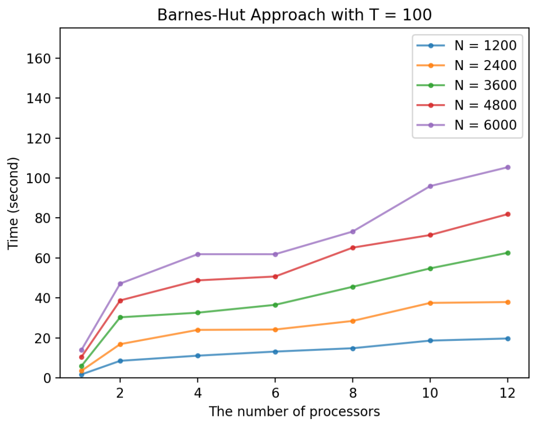

Figure 3: Barnes-Hut OpenMP+MPI Algorithm Performances

4 Discussion

For the basic OpenMP+MPI approach, the results of execution time shown in the Figure 2 are expected. The execution time reduces when the number of processors increases and the speedup tends to be 2.

However, for the Barnes-Hut OpenMP+MPI approach, the execution time did not follow the trend described in the previous approach. The running time increases as the number of processors increases. According to algorithm complexity, the implementation of Barnes-Hut approach achieves $O(N \log N)$ while the first approach has complexity of $O(N^2)$. Theoretically, the Barnes-Hut implementation should be much efficient than the basic approach. The main reasons that cause low performance is the load balance. This is a major performance issue. Computational work can not be evenly divided over all processors. In our implementation, we used static distribution to assign same number of arbitrarily distributed particles to each process, but the work in the simulation is determined by the number of interactions each process is computing. When there is high particle density variation, the distribution of work will be uneven and the communication is quite expensive at scale since many processors might have to wait on a single overloaded processor. To improve the performance of the system, we would further consider to implement using dynamic distribution by adding load balance algorithm at the end of each time step. Once partitioned, the particles will be redistributed and interactions will be assigned. By doing so, it will reduce the communication overhead.

5 Conclusion

In general, the Barnes-Hut algorithm improves the complexity from $O(N^2)$ to $O(N \log N)$. It performs much better in sequential time, compared to the basic OpenMP+MPI approach. Nevertheless, it is harder to parallelize the Barnes-Hut algorithm to achieve good run-time performance due to its non-uniform properties and dynamic changing behaviour. Thus, the further enhancement to obtain good speedup is to consider data locality and load balance.

6 Extension

6.1 Orthogonal recursive bisection

Besides the original Barnes-Hut algorithm above, there are a few variants implemented for better optimization in specific ways. As discussed in lecture, the orthogonal recursive bisection is introduced to provide a more balanced division of space by finding vertical and horizontal lines that divide the particles into two each group recursively, instead of merely splitting into four square-shape subcells in Figure 1.

6.2 Fast Multiple Method

Greengard and Rokhlin introduced the Fast Multiple Method (FMM) in 1987 [4], which was ranked as one of the top 10 algorithms of the 20th century. The algorithm has a better complexity of the order of N to evaluate all simulations within round-off error. The idea of this implementation is computing interactions between cell and cell, instead of interacting between particle and cell, like in the Barnes-Hut algorithm [2]. Alternatively, it uses the balanced octree concept for the computation. In general, the FMM brings the complexity of $O_\epsilon(N)$ for a fixed acceptable accuracy ϵ.

References

[1] Wikipedia contributors. n-body problem, 2019. [Online; accessed 13-October-2019].

[2] Guy Blelloch and Girija Narlikar. A practical comparison of n-body algorithms. *Parallel Algorithms: Third DIMACS Implementation Challenge, October 17-19, 1994*, 30:81, 1997.

[3] Wikipedia contributors. Barnes–hut simulation — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-October-2019].

[4] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.