Distributed Barnes-Hut Algorithm: Planet Simulation

Christopher Mertin

May 10, 2016

Introduction

The N-body problem is important for many aspects of physics and scientific simulations. This is due to the fact that each particle interacts with every other type of particle, which results in an $\mathcal{O}(n^2)$ algorithm if you take the naïve approach.

There are alternative ways to approach these types of problems, one of which is known as the *Barnes-Hut Algorithm* [1]. While the original algorithm was sequential, modern day methods have parallelized this approach to be used with MPI for use in clusters.

The problem with parallelizing this sort of algorithm is that it is difficult to build the tree efficiently in parallel. This was accomplished with the use of DENDRO. The actual force calculations themselves are embarassingly parallel as the position update has no dependencies.

Numerical Calculations

The gravitational force on one planet by another is given by

$$\vec{F} = G\frac{m_1 m_2}{r^2}\hat{r} \tag{1}$$

we also know from Newton's laws that

$$\vec{F} = m\vec{a} \tag{2}$$

from these above two equations, we can figure out the force on a given planet by first calculating the force on it, and then the direction of the acceleration. This can be accomplished with something trivial such as the *forward-euler method* or some other basic time-stepping scheme. However, methods such as the forward-euler method perform horribly at conserving energy of the system.

However, we can use another numerical method to achieve $\mathcal{O}(\Delta t^3)$ accuracy, and is much better in conserving energy, with a method known as the *Hermite Scheme*. Something such as Runge-Kutta-4 can give $\mathcal{O}(\Delta t^4)$ accuracy, but does take longer. As this is a simple test case, and the function itself was made modular in the code, the Hermite algorithm was chosen. This hermite algorithm was adapted from [2].

As we know from Equation (2), the acceleration is the second derivative of the position. The Hermite algorithm takes this further and calculates the *jerk* of the system, which is the time derivative of the acceleration. With this, we can get the position in the following order

$$\frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \vec{a}_i = G\sum_{\substack{j=1\\ j\neq i}}^{N} \frac{m_j}{\left|\vec{r}_j - \vec{r}_i\right|^3}\left(\vec{r}_j - \vec{r}_i\right) \tag{3}$$

$$\frac{\mathrm{d}^3\vec{r}_i}{\mathrm{d}t^3} = \vec{j}_i = G\sum_{\substack{j=1\\ j\neq i}}^{N} m_j\left[\frac{\vec{v}_j - \vec{v}_i}{\left|\vec{r}_j - \vec{r}_i\right|^3} - 3\frac{(\vec{r}_j - \vec{r}_i)\cdot(\vec{v}_j - \vec{v}_i)}{\left|\vec{r}_j - \vec{r}_i\right|^5}(\vec{r}_j - \vec{r}_i)\right] \tag{4}$$

$$\frac{\mathrm{d}\vec{r}_i}{\mathrm{d}t} = \vec{v}_i^{(n+1)} = \vec{v}_i^{(n)} + \frac{1}{2}\left(\vec{a}_i^{(n)} + \vec{a}_i^{(n+1)}\right)\Delta t + \frac{1}{12}\left(\vec{j}_i^{(n)} - \vec{j}_i^{(n+1)}\right)\Delta t^2 \tag{5}$$

$$\vec{r}_i^{(n+1)} = \vec{r}_i^{(n)} + \frac{1}{2}\left(\vec{v}_i^{(n)} + \vec{v}_i^{(n+1)}\right)\Delta t + \frac{1}{12}\left(\vec{a}_i^{(n)} - \vec{a}_i^{(n+1)}\right)\Delta t^2 \tag{6}$$

So first the jerk and accleration need to be calculated, and then the velocity, followed by the position being updated. This needs to be done for each of the particles to have the desired result. The exponents in Equation (5) and Equation (6) denote the time step. $(n+1)$ is for the *new* time step, while (n) is for the previous time step, and the subscript i denoting the i^{th} planet.

Verifying Numerical Results

It was quite easy to verify to make sure the code was performing correctly. For the code to perform correctly, the *total energy* of the system had to be conserved as there were no outside forces. This relied on the *potential energy* and the *kinetic energy* of the system.

The potential energy (U) can be calculated as

$$U = -G\frac{m_1 m_2}{\left|\vec{r}_2 - \vec{r}_1\right|}$$

with G being the gravitational constant, m_i being the mass for the two planets, and $\vec{r}$ being their position. The kinetic energy (T) can be represented as

$$T = \frac{1}{2}m\left|\vec{v}\right|^2$$

where v is the velocity of a planet. In 3D, this is transformed into

$$T = \frac{1}{2}M\left(v_x^2 + v_y^2 + v_z^2\right)$$

The total energy of the system can be computed as $U + T$ at the start of the simulation, and again after the final time step. The total energy of the system should be conserved within some percent error.

For all the different tests, the error remained within 2% error for any number of iterations. This is well within the bounds of the simulation and can be treated as correct. This error comes about from the approximation resulting in the Barnes-Hut algorithm.

Barnes-Hut Algorithm

The Barnes-Hut algorithm works by using an adaptive quadtree, which is shown in Figure 1. In this figure, each color represents one "branch" of the quadtree. The blue branch is refined more than the others due to the point located in it. It is known as an *adaptive quadtree* because it only refines the grid up to the point where particles can be approximated reasonably well as a single particle.

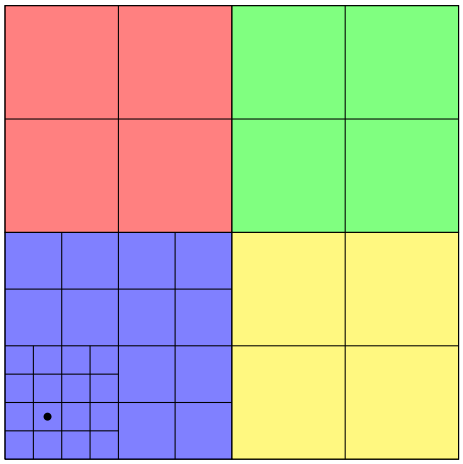

Figure 1: Adaptive Quadtree

To approximate the single particle, the center of mass is calculated for all the particles in that octant. The total mass in that octant would be the sum of the masses. In other words:

$$\vec{r}_i = \frac{\sum_{i=1}^{\zeta} m_i \vec{x}_i}{\sum_{i=1}^{\zeta} m_i} \tag{7}$$

where ζ is the total number of particles in the given octant. This new position and mass are used to approximate the particles that are reasonably far enough away to be approximated as a single point.

The colors in Figure 2 represent those particles that are "far enough away." Those in red are approximated as larger particles, and they decrease in size as you go down to finer grains. The larger the particles in the grid, the more that were approximated/averaged in that octant. The ones in the yellow grid are down to the finest grain size where each dot represents a single particle. The finest octant size is given for those in yellow, which are the octant's neighbors.

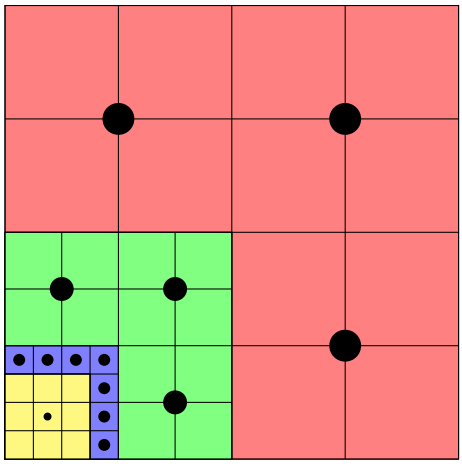

Figure 2: Adaptive Quadtree Coarsening

The code can be found [3] under the github branch `CS6320_Semester_Project`. The complexity of building the tree in DENDRO is $\mathcal{O}\left(\frac{N}{p}\log\left(N/p\right)\right)$. Since the positions of the particles are updated after each time step, the octants change as well. This can be prevented by altering/adjusting the tree after each time step. However, the implementation that was used was rebuilding the tree after each iteration. Therefore, the *time complexity* is $\mathcal{O}\left(t\frac{N}{p}\log\left(N/p\right)\right)$, where t is the number of time steps.

DENDRO is "probably, yea" work-optimal [4]. The calculation part is work optimal as it can be parallelized easily and not have to do any extra work.

Results

The results of the implemented Barnes-Hut algorithm can be seen in Figures 3 and 4.

In Figure 3, the large jump and irregularities in the plot are most likely due to the inefficient communication that was implemented. In the current implementation, at the start of each iteration an `MPI_Allgather` was performed so that each node/process had all the paricles so they could see which it needed for what octants. After implementing the code, it was realized that there could be an *interprocessor sorting* done between the nodes, removing the `MPI_Allgather` requirement. There were a few other `MPI_Allgather` and `MPI_Allreduce` commands that were performed which were realized after that they could be removed.

With more time, these could be fixed so that the code was more efficient. The code and timings were performed on MIR since TANGENT did not have PETSC installed and was unable to be installed locally.

The strong scaling results in Figure 4 seem to agree with how the scalability should look. The timings follow the general pattern for the time taken to solve the problem.

In order to compare the results, the sequential case for the $\mathcal{O}\left(N^2\right)$ algorithm was implemented and timed as well. Those timings can be seen in Table 1. The implemented code performed *worse* than the $\mathcal{O}\left(N^2\right)$ algorithm for $p < 12$ and $N < 8000$. This is likely due to

the extra requirement to build the tree with every iteration and savings not coming about except for large N.

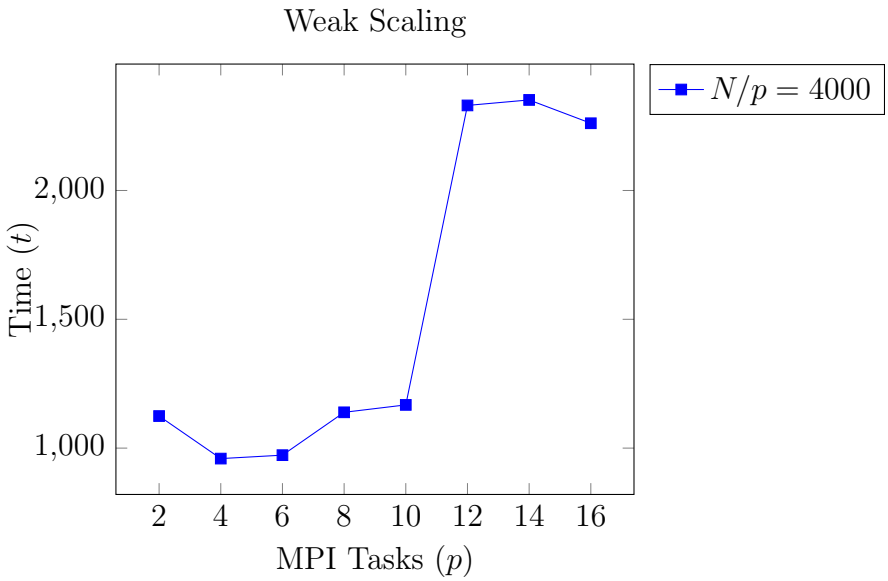

Figure 3: N/p remains constant

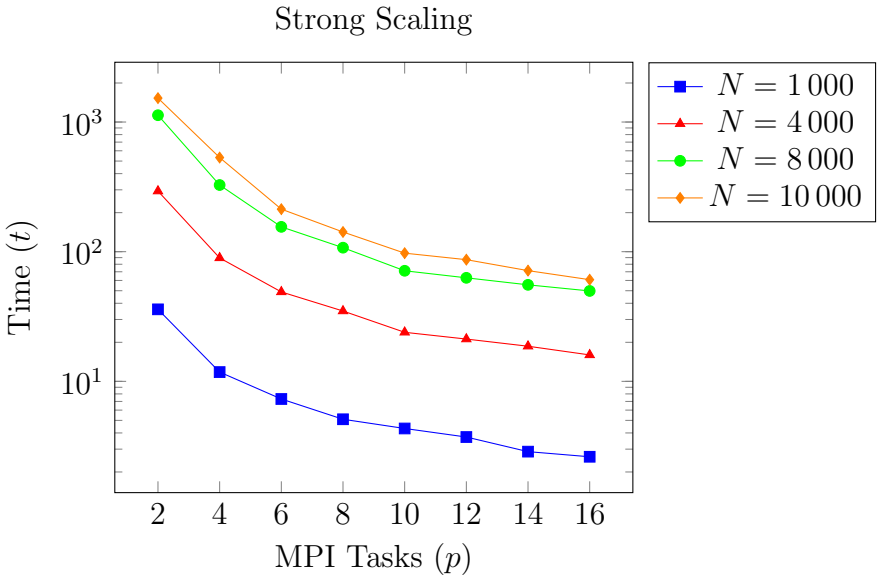

Figure 4: N remains constant

Table 1: Timings for Sequential

N	t
1000	1.04
4000	16.61
8000	66.07
10000	103.11

References

[1] Josh Barnes and Piet Hut, *A Hierarchial* $\mathcal{O}(N \log(N))$ *Force-Calculation Algorithm*, Nature **324** (4), (December 1986) 446-449

[2] Piet Hut and Jun Makino, *Moving Stars Around*, `http://www.artcompsci.org/`

[3] *Barnes-Hut:* DENDRO, `https://github.com/cmertin/dendro`

[4] Weerahannadige Milinda Shayamal Fernando, Personal Conversation, May 9, 2016