ASTROPHYSICAL MODELING WITH PARALLEL BARNES-HUT

A PREPRINT

Bryson Banks
The University of Texas at Austin
Austin, Texas
brysonbanks@utexas.edu

David Campbell
The University of Texas at Austin
Austin, Texas
campbeda@utexas.edu

Jeffrey D Nelson
The University of Texas at Austin
Austin, Texas
jdn2283@utexas.edu

August 14, 2020

ABSTRACT

Efficiently performing an *n*-body simulation is an important problem in the world of scientific computing, especially the domain of astrophysics, and the Barnes-Hut algorithm is the most common solution. Here we present a parallel implementation of the Barnes-Hut algorithm and explore its performance as parameters of the simulation are modified. We discuss the relevant trade-offs, show how our parallel implementation scales, and present areas for future work. Our implementation is designed to model the movement of large bodies in a simulated solar system.

1 INTRODUCTION

The Barnes-Hut Algorithm is a tree-based approximation algorithm for simulating the *n*-body problem, commonly present in scientific based modeling. The time complexity for calculating the force applied to every body in a set by every other body scales with $O(n^2)$. In many settings, high precision is unnecessary, and the scalability and speed of the algorithm is more critical. For example, astrophysicists simulating the gravitational force applied on a planet can approximate the force exerted by objects in a distant galaxy. The entire galaxy, from the perspective of the planet, can be modeled as one object with some center of mass. Using a modified parallel implementation of the Barnes-Hut algorithm as a base, we construct a model for simulating the *n*-body problem and explore its functionality and scalability.

2 OUR MODEL

The model that we construct assumes a shared memory system and is implemented in C++ with OpenMP support for parallel computations. Our input is a given number of time steps and a set of particles, where each particle is composed of a mass, initial position, initial velocity, and initial acceleration. Our output is the position, velocity, and acceleration of each particle at each time step of the simulation. As a first step, our implementation constructs an **Oct-Tree**, covered in the following section. Each time step is then simulated, where the algorithm for each time-step can be broken into the following phases:

- the total force applied to each particle is calculated in parallel
- the position, velocity, and acceleration of each particle is updated in parallel
- the Oct-Tree is updated in parallel

2.1 OCT-TREE

An Oct-Tree is the fundamental data structure employed by the Barnes-Hut algorithm to trade off precision for scalability. Consider the left picture in **Figure 1**, which depicts a set of particles within some simulation bounds. Think of each particle as a large body (star, planet, etc.) and the simulation bounds as the galaxy these particles are contained by. Physics tells us that the gravitational force decreases exponentially as the distance between two particles increases. Therefore, we can approximate the force applied on a particle by grouping particles that are far enough away. The

representation of this grouping, the Oct-Tree, also depicted in **Figure 1**, is a tree in which each node has at most eight children. Note that our figure actually shows a Quad-Tree, since a two-dimensional space is easier to visualize. Each child is responsible for representing the center of mass in an octet of the three-dimensional space owned by its parent node. Nodes with no children are called leaf nodes, and they represent the actual particles in the simulation. Calculating the force applied on any given particle involves traversing the Oct-Tree and determining whether each octet is far enough away to approximate forces for particles within that octet, or whether the next level must be explored.

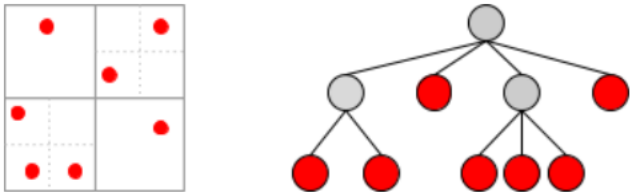


Figure 1: Oct-Tree

2.2 OCT-TREE CONSTRUCTION

The parallelism in constructing an Oct-Tree comes from each octet being independent of one another. A particle inserted into octet 0 of the root can occur in parallel with a particle inserted into octet 1 of the root. Parallelism can also be applied per level, but our implementation only exploits parallelism at the root of the tree. For brevity, we only show pseudocode for the insertion function, which is implemented recursively in **Algorithm 1**.

Algorithm 1: *insertParticle(Node n, Particle p, Int octet)*

Input: n : Node, p : Particle, $octet$: int

```
1  child := n.children[octet];
2  if child = null then
3  |   n.children[octet] := p;
4  else
5  |   if child.isLeaf then
6  |   |   newNode := newNode(getBounds(n, octet));
7  |   |   n.children[octet] := newNode;
8  |   |   insertParticle(newNode, child, findOctet(newNode, child));
9  |   |   insertParticle(newNode, p, findOctet(newNode, child));
10 |   else
11 |   |   insertParticle(child, p, findOctet(child, p));
12 |   end
13 end
```

2.3 OCT-TREE CENTER OF MASS

Once the Oct-Tree is constructed, the center of mass for each non-leaf node needs to be set. The algorithm to do so recursively calculates the center of mass for each child node before updating the node itself. Parallelism is exploited at every level of the tree, so the time complexity for this algorithm scales with $O(log\ n)$. **Algorithm 2** shows the pseudocode for this.

2.4 FORCE CALCULATION

At the core of the algorithm is the computation of the gravitational forces produced on each individual particle by all other particles in the system. When the number of particles in the system is small, these calculations can be done with full accuracy in a brute force pairwise manner with time complexity $O(n^2)$. This method does not scale well to systems with a large numbers of bodies or time steps. Instead, Barnes-Hut calculates the force exerted on each individual particle by traversing the constructed Oct-Tree and approximating forces from sets of distant particles.

Algorithm 2: *centerOfMass(Node n)*

Input: n : Node
for $child$: $n.children$ **do**
  if $!child.isLeaf$ **then**
    $omp\ task$;
    $centerOfMass(child)$;
  end
end
$omp\ taskwait$;
$n.mass := 0$;
$pos := (0, 0, 0)$;
for $child$: $n.children$ **do**
  pos += ($child.mass$ * $child.pos$);
  $n.mass$ += $child.mass$;
end
$n.centerOfMass$:= (pos / $n.mass$);

For each particle, the algorithm starts at the root of the tree and moves downward. At each node, the ratio D/r is calculated, where D is the size of the octet represented by the node and r is the distance between the particle and the node's center of mass. If this ratio is less than $theta$, traversal down this path of the tree stops, and the force exerted on the particle is approximated by this node, which represents all particles within its spatial bounds. Otherwise, the traversal recursively continues to each child of this node and the D/r calculation is repeated. Total exerted force on the particle in question is recursively accumulated for each explored downward path of the tree.

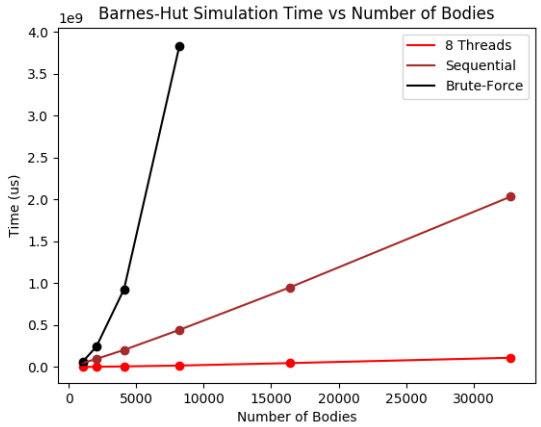
In this manner, the time complexity of each tree traversal is bounded by the height of the tree. Since the tree traversal needs to be performed for each particle, the time complexity of the sequential Barnes-Hut force calculation algorithm is bounded by $O(n * log\ n)$. By parallelizing the independent tree traversals, the time complexity can be reduced further to $O(log\ n)$. Even further incremental speedup can be gained by parallelizing the independent traversals of node children. **Algorithm 3** shows pseudocode for a single independent parallel tree traversal.

Algorithm 3: *treeForce(Node n, Particle p)*

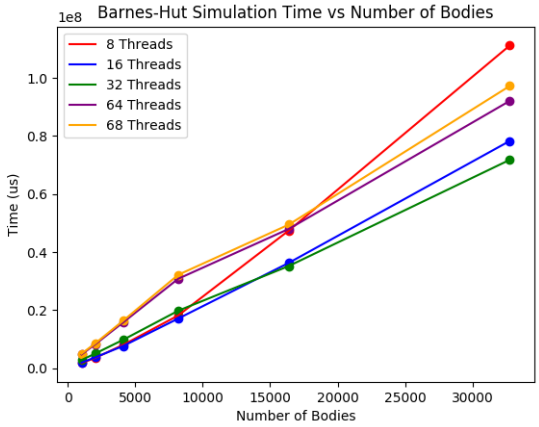
Input: n : Node, p : Particle, θ : user defined float (usually < 1)
$totalForce := 0$;
$cm := n.centerOfMass$;
if $n.numChildren$ == *1* **then**
  $totalForce := pairwiseForce(p, cm)$;
end
else
  $r := distance(p, cm)$;
  $D := n.size$;
  if $D/r < \theta$ **then**
    $totalForce := pairwiseForce(p, cm)$;
  end
  else
    for $child$: $n.children$ *in parallel* **do**
      $totalForce := totalForce + treeForce(p, child)$;
    end
  end
end
return $totalForce$;

2.5 RUNTIME

As mentioned above, a brute-force solution to the *n*-body problem will scale with $O(n^2)$ time complexity for each time step, while a sequential implementation of the Barnes-Hut algorithm will scale with $O(n * log\ n)$. Parallelizing the



(a) Brute Force, Sequential vs Parallel Simulations of N-body Simulation Problem



(b) Scalability of Parallel Barnes-Hut Algorithm

Figure 2: Runtime Characterization of Parallel Barnes Hut

algorithm causes the time complexity to scale with $O(log\ n)$ per time step, allowing us to simulate much larger systems for longer periods of time. The $O(log\ n)$ factor within each time step comes from the force calculation walking the tree for each particle, as well as the center of mass updates on the tree.

3 RESULTS

All results were collected on a single node of Stampede2, a supercomputer managed by the Texas Advanced Computing Center. Each node is an Intel Xeon Phi 7250 with 68 cores and 96GB of DDR4 RAM. Runtime statistics were collected as the number of threads, particles, and time steps were scaled. The input for each experiment was generated by an input generator that we constructed. The following sections go into more detail on the input generator and evaluate our results.

3.1 INPUT GENERATION

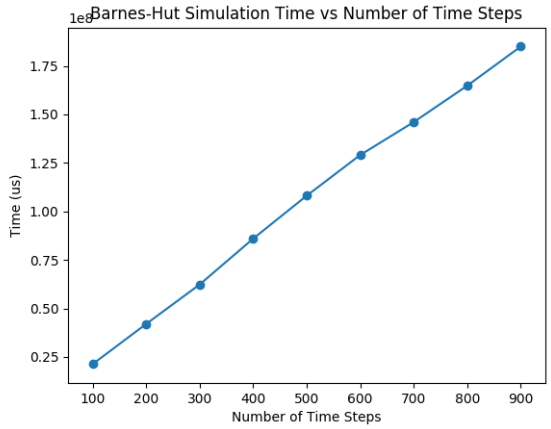
Our goal in generating input was to construct data that closely represented a real solar system. The simulation bounds were set to the approximate size of our solar system, and the mass of each particle was randomly generated to be on the order of magnitude of the mass of Jupiter. The initial position of each particle was randomly selected within the simulation bounds, while the initial velocity and acceleration of each particle was set to 0. Since we do not model particle collisions in our implementation, we attempted to reduce collisions by setting a minimum starting distance between each particle. The size of the data sets ranged from 2^{10} to 2^{20} particles.

3.2 GRAPHS AND EXPLANATIONS

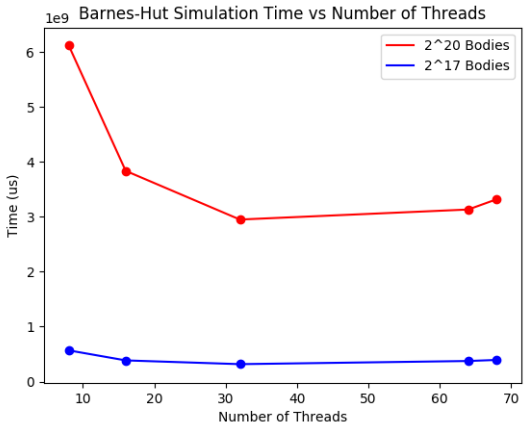
From **Figure 2A**, we were able to observe the expected runtime scaling of the brute force algorithm vs the sequential and parallel implementation of Barnes-Hut. As the number of bodies increases, the brute force algorithm scales with $O(n^2)$, while sequential BH scales with $O(n * log\ n)$ and parallel BH scales with $O(log\ n)$.

In **Figure 2B**, we see how the parallel implementation scales with the number of threads. Interestingly, we observed that the ratio of bodies to threads had a large impact in our experiments. As the number of threads increases, we know that the synchronization overhead increases, but this overhead should be amortized as the size of the data set grows. **Figure 3B** shows results for data sets up to size 2^{20}, and 32 threads is still the optimal value. Perhaps for even larger data sets, we would start to see better scaling with 64 or 68 threads.

In **Figure 3A**, we were able to confirm that our parallel algorithm scaled linearly with the number of time steps, as expected. This experiment was done for 10k bodies.



(a) Runtime by Number of Timesteps Simulated



(b) Number of Threads vs Runtime for Large N-Body Simulation

Figure 3: Runtime Characterization of Parallel Barnes Hut (Con't)

4 RELATED WORK

The seminal paper for the Barnes-Hut algorithm was written in 1986 by Josh Barnes and Piet Hut[1]. In 1993, Warren and Salmon improved upon the algorithm's performance by optimizing the data structure used to represent the Oct-Tree[2], and Warren further extended this in 2014[3]. Our research showed that the Barnes-Hut algorithm has been widely deployed on CPU-based and GPU-based supercomputers in various research contexts. The last 10 years has seen more GPU-based implementations, such as the Burtscher and Pingali paper from 2011[5]. One commonly cited CPU-based implementation, by Winkel et al.[4], is also included in our references.

5 CONCLUSION AND FUTURE WORK

While the sequential Barnes-Hut algorithm far outperforms the brute-force solution to the *n*-body problem, we have shown that parallelizing the Barnes-Hut algorithm can improve performance by a factor of *n* without any loss of precision or additional work. Our model, which targeted a shared memory machine and was implemented in C++ with OpenMP support, produced results that lined up with our initial expectations. The Oct-Tree data structure and pseudocode that we presented illustrates how the the algorithm is structured and explains where the performance improvements arise.

Lastly, we discuss areas for improvement and future work. First, we would like to support collision detection and improved bounds handling. Our implementation assumes each particle to be a singular, width-less density at its center of mass. With inertia and the lack of collision handling, it is possible that a particle moves outside the simulation bounds. For the inputs that we constructed and the number of time steps that we simulated, the chances of this happening is very small, but if it were to happen, we log an error and future positions of that particle are undefined. Second, note the user defined θ value in the $treeForce$ function of **Algorithm 3**. As θ decreases, we expect the precision and the runtime to increase. Thus, we would like to further explore this trade-off and allow for a configurable θ. Third, we would like to compare our CPU-based parallel implementation to a GPU-based implementation. We expect that a GPU-based implementation would see even more performance improvements. Finally, we would like to get access to real astronomical data sets and compare our accuracy to measured results.

References

[1] Josh Barnes and Piet Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature, 324*, pages 446—449. December 1986.

[2] M. S. Warren and J. K. Salmon. A parallel hashed Oct-Tree N-body algorithm. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*, pages 12–21. 1993.

[3] Michael S. Warren. 2HOT: An Improved Parallel Hashed Oct-Tree N-Body Algorithm for Cosmological Simulation Networks with Existing Applications. *Scientific Programming, 22(2)*, pages 109–124. 2014.

[4] Mathias Winkel, Robert Speck, Helge Hubner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A Massively Parallel, Multi-Disciplinary Barnes–Hut Tree Code for Extreme-Scale N-Body Simulations. *Computer Physics Communications, 183(4)*, pages 880–889. April 2012.

[5] Martin Burtscher and Keshav Pingali An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm *GPU Computing Gems Emerald Edition*, Chapter 6. December 2011.