Lab 5: Barnes-Hut Report

Julian Canales
CS 378H Concurrency Honors
Spring 2025
April 9, 2025

1 Introduction

The goal of this lab is to implement the Barnes-Hut algorithm for solving the N-body problem using MPI for parallelization. We first provide context of the N-body problem and then explain how the Barnes-Hut algorithm works to solve the problem.

According to the lab guide, the N-body problem represents an astrophysical simulation that models the continuous interaction of each body with every other body in a system. The interactions are caused by the gravitational force that each body exerts on the other bodies, where each body could represent a star, planet, galaxy, or other celestial bodies. The N-body problem is not analytical solved in closed form beyond 2 objects and it is standard to simulate interactions in order to determine how objects will interact with one another.

The naive approach to solving the N-body problem would be to compute all $n \times n$ interactions between every pair of bodies, resulting in $O(n^2)$ computational complexity. However, this lab will make use of the Barnes-Hut algorithm, which reduces complexity to $O(n \log n)$ by computing interactions only for pairs of bodies that are sufficiently close, while approximating the effects of distant groups of bodies by treating them as single bodies with their centers of mass. We elaborate on the algorithm and its implementation later in the report.

This report is organized into the following sections:

1. Algorithm Overview
2. Optimizations
3. Analysis of Performance

Note: The program outputs the end to end time. Also, a flag, `-q`, was added that when used, activates the sequential implementation of the algorithm.

1.1 Time Spent

The total time I spent on this lab is approximately 20 hours, broken down as follows:

- 5 hours researching the Barnes-Hut algorithm and understanding the problem, algorithm, and mathematical foundations
- 5 hours implementing the sequential version of the algorithm
- 5 hours implementing the MPI parallel version and debugging issues
- 5 hours testing, benchmarking, and writing this report

1.2 Hardware Specifications

All performance testing was conducted on a machine with the following specifications:

- CPU: Intel(R) Core(TM) i5-14600
- Architecture: x86_64
- Cores: 14 physical cores (20 logical threads)

- CPU MHz: 2700.000 (max 5200.0000)
- L1d cache: 336 KiB
- L1i cache: 224 KiB
- L2 cache: 14 MiB

These values were retrieved by running the command `lscpu` on the machines terminal.

2 Algorithm Overview

The following breakdown of the algorithm makes use of three main sources:

1. The lab specs for high level understanding.
2. Warren and Salmon's paper titled "Astrophysical N-body simulations using hierarchical tree data structures" for understanding the unique tree data structure that needed to be implemented.
3. Singh et al. paper titled "Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity" for a deeper understanding of the Barnes-Hut algorithm.

The first phase constructs a tree (quadtree in 2D) that represents a spatial partitioning of the bodies in the system. Each interior node in this tree represents the center of mass of all bodies contained within the subtree rooted at that node.

The second phase computes force interactions for each body using the Multipole Acceptance Criteria (MAC), which determines whether bodies in a subtree are sufficiently far away that their net force can be approximated by the center of mass of the entire subtree. The MAC is expressed mathematically as a ratio between the size of a cell (l) and the distance from the body to the cell's center of mass (d). Note that a cell in this context is what a node in the quadtree is meant to represent geometrically, and the size of the cell equates to the length of a side of the cell. If $l/d < \theta$, where θ is a user-defined threshold parameter, then the approximation is considered valid. Smaller values of θ yield more accurate results but require more computation, while larger values provide faster computation at the cost of accuracy.

Once the MAC determines which interactions to compute directly and which to approximate, the gravitational forces are calculated. For each interaction, the force components are projected onto the x and y axes ($F_x = G \cdot M_0 \cdot M_1 \cdot dx/d^3$ and $F_y = G \cdot M_0 \cdot M_1 \cdot dy/d^3$), where a minimum distance threshold (r_{limit}) prevents numerical instabilities when particles are very close. The total force on each particle is the sum of all these individual force contributions.

Finally, the velocity and position of each body are updated using the Leapfrog-Verlet integration scheme. This method first calculates acceleration from force ($a_x = F_x/M_0$, $a_y = F_y/M_0$), then updates position using the current velocity and half-step acceleration ($P'_x = P_x + V_x \cdot dt + 0.5 \cdot a_x \cdot dt^2$), and finally updates velocity with the full acceleration ($V'_x = V_x + a_x \cdot dt$).

2.1 Data Structures

The central data structure for the Barnes-Hut algorithm is the quadtree, represented by the `Node` struct:

```
struct Node {
    bool emptySpace;
    double xCenter;
    double yCenter;
    double length;
    Particle* p;
    double xCenterMass;
    double yCenterMass;
    double quadrantTotalMass;
    Node* children[4];
};
```

Each `Node` represents a spatial region in the simulation domain. Internal nodes store the center of mass of all particles in their subtree, while leaf nodes contain individual particles. The `Particle` struct stores the physical properties of each body:

```
struct Particle {
    int index;
    double x_pos;
    double y_pos;
    double mass;
    double x_vel;
    double y_vel;
};
```

2.2 Core Functions

The implementation consists of several key functions listed below.

- `createNode`: Initializes a new node with given spatial coordinates and parameters
- `getQuadrant`: Determines which quadrant a particle belongs to within a node
- `addParticle`: Iteratively adds a particle to the quadtree
- `constructBarnesHutTreeSeq`: Builds the complete quadtree from an array of particles
- `calculateForces`: Computes the gravitational forces on a particle using the quadtree
- `deleteTree`: Cleans up the dynamically allocated tree

2.3 Sequential Implementation

In this section, I will list the general steps I took for the sequential version of the algorithm.

1. Read input particle data from a file

2. For each simulation time step:
   (a) Construct the Barnes-Hut quadtree
   (b) For each particle:
      i. Check if the particle is within domain boundaries; mark lost particles
      ii. Calculate forces from other particles using the tree
      iii. Update velocity and position using Leapfrog-Verlet integration
   (c) Delete the tree
3. Write final particle data to output file

2.4 MPI Implementation

The MPI parallel implementation was more involved and follows the following general steps:

1. Rank 0 reads input particle data and creates vector of particles
2. Rank 0 broadcasts total particle count to all processes
3. Each process calculates particle distributions/ workload for themselves
4. Define custom MPI datatype for `Particle`
5. Rank 0 broadcasts initial particles to all processes
6. Each process identifies the subset of particles they will update
7. For each simulation step:
   (a) Check if the particle is within domain boundaries; mark lost particles
   (b) Each process constructs the complete Barnes-Hut tree using global particle data
   (c) Each process updates positions and velocities for its local particles
   (d) All processes share updated particle data via `MPI_Allgatherv`
8. Gather final results back to rank 0
9. Rank 0 writes final particle data to output file

The key to this implementation is properly divying up the work among the processes and using `MPI_Allgatherv`, which ensures that all processes have access to the complete and updated particle data at the end of each time step:

```
// share updated particles to all processes
MPI_Allgatherv(localData.data(), localCount, MPI_PARTICLE,
               globalData.data(), counts.data(), displacements.data(),
               MPI_PARTICLE, MPI_COMM_WORLD);
```

3 Optimizations

Only two notable optimization techniques were added during/after development to the base algorithm improve performance.

3.1 Non-Recursive Tree Traversal

Initially, I implemented recursive functions for tree traversal in force calculation and tree deletion. However, I was receiving a bus error when dealing with large numbers of particles (100,000) and had a hypothesis that this could be due to stack overflow. To address this, I replaced recursion with stack-based iterative approaches, which can be found from lines 163 - 237 in `barneshuttree.cpp`. This change resolved the bus error issues.

3.2 Inline Center of Mass Calculation

Instead of building the tree first and then computing the center of mass in a separate pass, as the algorithm in Singh et al.'s paper suggested, I calculate and update the center of mass during tree construction. This is done by maintaining running totals of mass and mass-weighted positions in each node:

```
// Update center of mass and total mass
double totalMass = node->quadrantTotalMass + p.mass;
node->xCenterMass = (node->xCenterMass * node->quadrantTotalMass + p.
   x_pos * p.mass) / totalMass;
node->yCenterMass = (node->yCenterMass * node->quadrantTotalMass + p.
   y_pos * p.mass) / totalMass;
node->quadrantTotalMass = totalMass;
```

This optimization eliminates the need for an additional tree traversal and reduces the overall computational cost.

4 Analysis of Performance

Performance was measured across different particle counts (10, 100, 100,000) and MPI process counts (1, 2, 4, 8, 16, 20). For these tests, the following simulation parameters were held constant: `-s 1000` (number of iterations), `-t 0.35` (MAC threshold), and `-d 0.005` (timestep). I initially attempted to run with more than 14 MPI processes but resulted in an error because the MPI implementation defaulted to using the 14 physical cores as available "slots." Therefore, the `-use-hwthread-cpus` flag was required to allow MPI to utilize the available logical processors for tests involving 16 and 20 processes. The collected data was under one singular run and I used the command `make tests`. A better approach would be to average the results over several iterations, but collecting the data once suffices for our purposes.

4.1 Execution Times

Below is a table that holds all the collected data.

Process Count	10 Particles	100 Particles	100,000 Particles
1 (Sequential)	0.0029102	0.0456502	305.485
2	0.0042925	0.0301819	182.886
4	0.0057551	0.0256832	133.365
8	0.00676168	0.0334261	145.383
16	0.00918005	0.050714	138.731
20	0.00955446	0.0313843	138.053

Table 1: Execution times in seconds for different particle and process counts

4.2 Speedup Data & Figures

The speedup relative to the sequential implementation was calculated and is shown below.

Process Count	10 Particles	100 Particles	100,000 Particles
1 (Sequential)	1.00	1.00	1.00
2	0.68	1.51	1.67
4	0.51	1.78	2.29
8	0.43	1.37	2.10
16	0.32	0.90	2.20
20	0.30	1.45	2.21

Table 2: Speedup relative to sequential implementation

Below is also a graph that assists in visualizing the speedup for all cases.

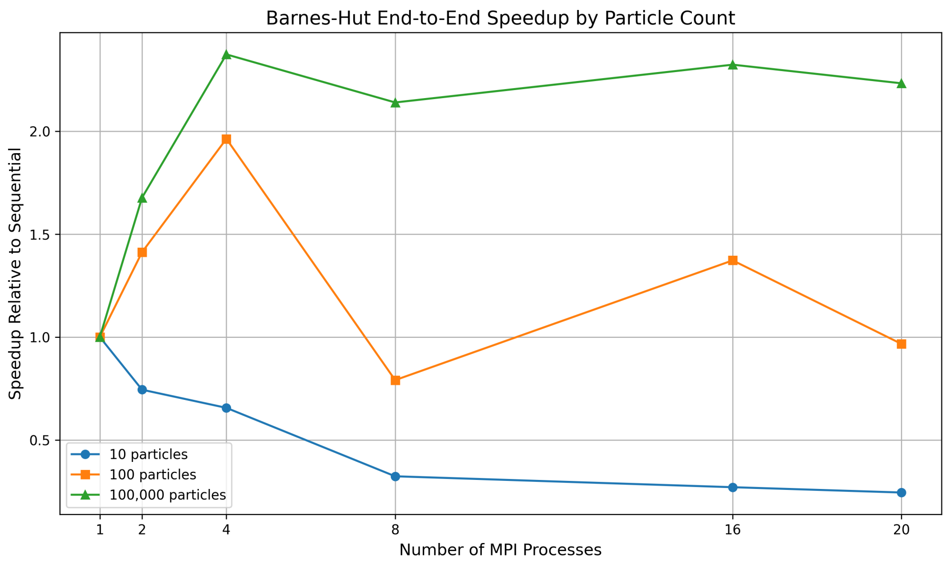

Figure 1: Graph showcasing the speedups.

4.3 Analysis & Discussion of Results

Several important observations can be made from the performance results. Below is a list of bullet points that has observations for each input size and one for overall observations regarding optimal process count.

1. **Small Particle Counts**: For 10 particles, parallel execution is always slower than sequential. This is expected due to the communication overhead of MPI, which outweighs the computational benefits for small problem sizes.
2. **Medium Particle Counts**: For 100 particles, we see moderate speedup with 2 and 4 processes (up to 1.78x), but performance degrades with 8 and more processes due to increased communication overhead.
3. **Large Particle Counts**: For 100,000 particles, speedup is achieved across all process counts, with the best performance at 4 processes (2.29x speedup). The speedup plateaus around 2.2x regardless of further increases in process count.
4. **Optimal Process Count**: The best performance was consistently achieved with 4 processes, regardless of particle count. This suggests a bottleneck, possibly due to memory bandwidth limitations or communication overhead.

The main thing I noticed for overall performance was the limited scalability beyond 4 processes. Some reasons that I could come up with for this are:

- **Global Tree Construction**: Each process constructs the complete tree in each step, which becomes a sequential bottleneck. Particularly, this could be a consequence of Amdahl's law.
- **All-to-All Communication**: The `MPI_Allgatherv` operation becomes increasingly expensive as the number of processes grows. More message complexity/latency is needed for each process to receive the up-to-date tree structure.
- **Memory Bandwidth**: With more processes, memory access becomes a limiting factor, especially as all processes need to access the complete particle dataset.

4.4 Further Optimization Opportunities

Based on the performance analysis, several optimization opportunities remain that could be explored in future iterations of this lab.

1. **Distributed Tree Construction**: Instead of each process building the complete tree, a natural progression would be a distributed approach where each process builds a partial tree and essential information is exchanged. This would increase the complexity of the code significantly, which is why I ultimately decided against it in my implementation described above.
2. **Domain Decomposition**: The provided implementation utilizes particle decomposition. A progression would be to implement spatial domain decomposition (i.e., each process gets assigned a certain area of the 4×4 square domain to handle). This could reduce communication overhead by possible allowing processes to focus primarily on particles within their assigned regions and bypass the necessary step by step update that is needed for the particles states.
3. **Hybrid Parallelism**: I am aware that combining MPI with multi-threading frameworks, such as OpenMP and pthreads, is a feasible option. Although we have not explored that thoroughly in the course thus far.