# Parallel N-Body Simulations: Comparing Sequential and Parallel Brute-Force, Barnes-Hut, Fast Multipole, and Hilbert Curve Sorted Bounding Volume Hierarchy Algorithms

**Rohan Phanse**
Computer Science '27
rohan.phanse@yale.edu

**Areeb Gani**
CS & Applied Math '27
areeb.gani@yale.edu

**Vishak Srikanth**
CS & Econ, Math '27
vishak.srikanth@yale.edu

## 1 Introduction

N-body simulations play a central role in cosmology by modeling gravitational interactions between celestial bodies. Beyond cosmology, these simulations have gained traction in fields such as physics and chemistry for modeling plasma, molecular dynamics, and fluid dynamics to name a few, as well as in machine learning for visualizing high-dimensional data with the t-SNE approach [7]).

In general, $N$-body simulations model the behavior of $N$ bodies under a mutual force such as gravitational attraction. Celestial systems with $N = 2$ bodies can be fully described using solutions derived from Newton's laws and Kepler's equations. However, systems with $N \geq 3$ bodies cannot be described with closed-form equations due to their generally chaotic behavior. Therefore, every intermediate time step would need to be simulated in order to obtain future states of the system, making larger $N$-body simulations expensive to compute.

For example, the time evolution of a chaotic system with $N = 3$ bodies is depicted in Figure §1. Given the state of an $N$-body simulation at time $t$, we can obtain the next state at time $t' = t + \Delta t$ using the following physics equations:

$$\vec{F}_i = \sum_{j \neq i} \frac{G m_i m_j}{\|\vec{r}_j - \vec{r}_i\|^3} \cdot (\vec{r}_j - \vec{r}_i), \quad \vec{a}_i = \frac{\vec{F}_i}{m_i}, \quad \vec{v}_i(t') = \vec{v}_i(t) + \vec{a}_i \Delta t, \quad \vec{x}_i(t') = \vec{x}_i(t) + \vec{v}_i(t') \Delta t$$

$N$-body simulations were first studied by Erik Holmberg in 1941, who used an analog optical computer to simulate gravitational forces between a few bodies (represented as lightbulbs) using the luminosity of the lightbulbs [5]. Since then, computers have rapidly advanced, and the main bottlenecks for these simulations now encompasses both the hardware and the software. Using a brute force all-pairs algorithm, each frame of an $N$-body simulation can be calculated with full accuracy in $O(N^2)$ time complexity. However, this approach quickly becomes intractable when considering systems with a large number of bodies such as galaxies. Because the force $\vec{F}_i$ on the $i$th body can be computed independently of the forces on other bodies, this problem is embarrassingly parallel and thus can be optimized with parallel programming algorithms and GPU acceleration.

Additionally, most practical implementations rely on approximations like the Barnes-Hut algorithm [1] and Fast Multipole Method (FMM) [3], which can reduce the time complexity to $O(N \log N)$ by leveraging hierarchical tree structures and even to $O(N)$ under certain conditions. These approximation methods can be parallelized to achieve further gains in performance.

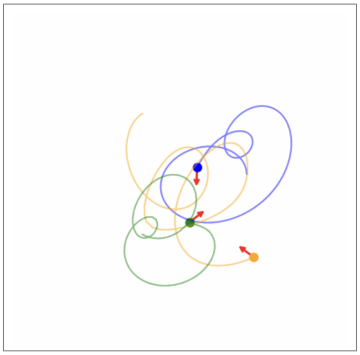

Figure 1: Example of an $N$-body simulation for $N = 3$.

This project focuses on developing sequential and parallel C++ implementations of three hierarchical tree-based algorithms—Barnes-Hut (BH), Fast Multipole Method (FMM), and Hilbert-sorted Bounding Volume Hierarchy (BVH)—and evaluating the relative performance gain compared to sequential and parallel implementations of the brute-force all-pairs algorithm for $N$-body simulations.[1]

# 2 Theory

## 2.1 Barnes-Hut Algorithm

The Barnes-Hut algorithm, proposed in 1986, approximates the forces in an $N$-body simulation in $O(N \log N)$ time [1]. The algorithm uses a hierarchical spatial decomposition to group particles into specific regions of space. The algorithm then approximates long-range interactions by aggregating multiple bodies into a single equivalent body located at their center of mass. As particles move to their new positions, the spatial decomposition is updated.

Some of the data structures that can be used for the hierarchical spatial decomposition include quadtrees for dividing a bounding 2D space into quadrants, octrees for dividing a bounding 3D space into octants, K-D trees for handling $d$-dimensional spaces, and BVH trees used in the Hilbert-sorted Bounding Volume Hierarchy (BVH) extension to the Barnes-Hut algorithm. A potential issue with these spatial decompositions is that they may result in highly irregular tree traversals when computing the forces and thus require careful parallelization.

The Barnes-Hut Algorithm is outlined below as a step-by-step procedure executed at each time step $1, 2, ..., T$:

1. **Calculate Bounding Box**: Determine the bounding box of the root node of the tree, which is the smallest box containing all bodies in the coordinate space, using a parallel reduction over all body positions.

2. **Build Tree**: Construct a quadtree, octree, or BVH tree where each leaf node contains at most one body.

3. **Calculate Moments**: Create a multipole expansion akin to a Taylor series expansion to calculate moments at each tree node. For calculating forces, only the the center of mass of each node is needed, which can be computed recursively from each child's center of mass.

4. **Calculate Forces**: Traverse the tree from the root in DFS order and use long-range interactions along with moments to calculate forces. If the node's center of mass is larger than some threshold distance, the interactions with bodies covered by that node are approximated by that node's center of mass.

5. **Update Positions**: Update body positions by integrating the approximated forces according to Newton's laws.

An example of a quadtree is provided in Figure §2, which was adapted from Cassell et. al. (2024) [4]. Each node stores an offset to its first child (or "E" denoting empty). Each node in the tree directly maps to a bounded region in 3D space, with child nodes contained within the parent's bounding box.

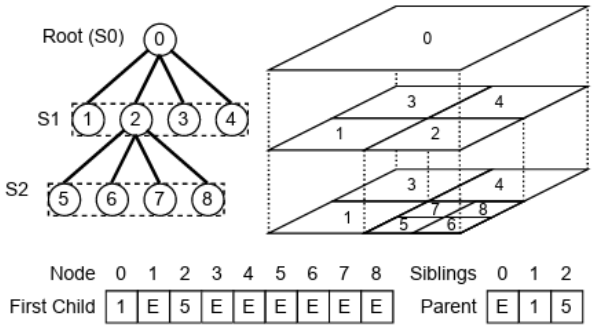


Figure 2: Quadtree structure mapped to 3D space and memory layouts [Figure adapted from Cassell et. al. (2024)].

[1] Full implementations are available at https://github.com/areebg9/cpsc424-final.

## 2.2 Fast Multipole Method

The Fast Multipole Method (FMM) improves upon the $O(N^2)$ brute-force approach to obtain a time complexity of $O(N \log N)$ and even a quasi-linear complexity of $O(N)$ under certain ideal conditions [3]. Similar to Barnes-Hut, FMM groups distant points together and approximates their combined effect by constructing a quadtree or an octree and running multipole and local expansions on it. The FMM algorithm is based on six operators named in the format of $X2Y$, where $X$ is the source and $Y$ is the destination of the operator.

The operators are listed below, where $P$ stands for particle or body, $M$ for multipole, and $L$ for local.

1. **P2M (Particle-to-Multipole):** Computes a multipole expansion from particles within a node.
2. **P2P (Particle-to-Particle):** Computes interactions between nearby particles when approximation is not sufficient.
3. **M2L (Multipole-to-Local):** Converts a multipole expansion of a "distant" node into a local expansion.
4. **M2M (Multipole-to-Multipole):** Combines and transfers many multipole expansions of children into parent node.
5. **L2P (Local-to-Particle**): Evaluates a local expansion at the locations of particles in a node.
6. **L2L (Local-to-Local):** Passes a local expansion from a parent node to its children.

The algorithm is illustrated in Figure §3, which was adapted from Bramas (2020) [2].

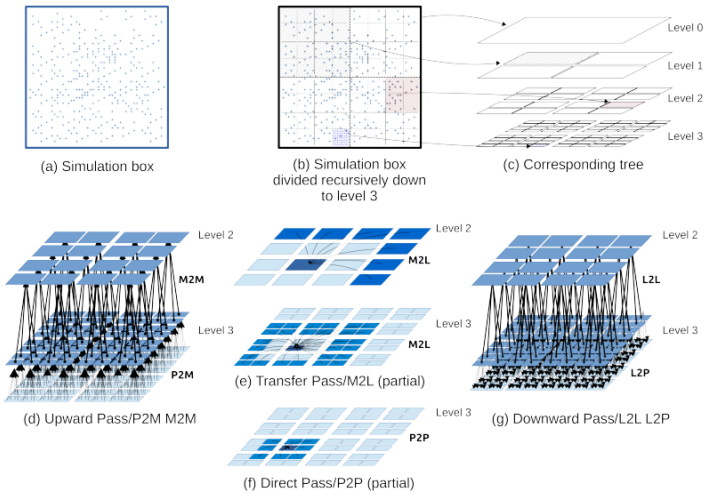


Figure 3: Schematic of FMM Algorithm. Steps (a, b, c) show how the octree is built while steps (d, e, f, g) show the various operators and their sequence in the algorithm [Figure adapted from Bramas (2020)].

## 2.3 Hilbert-sorted Bounding Volume Hierarchy (BVH)

The BVH is a balanced binary tree where the number of leaf nodes is a power of two. This means that the number of levels, nodes per level, and total number of nodes are fixed. The tree structure functions like a skip list and enables us to efficiently traverse without storing connectivity information in memory explicitly (i.e. stackless traversal algorithm). This algorithm sorts all bodies along a 1-dimensional Hilbert space-filling curve, building a BVH and reducing the moment calculations within a single leaf-to-root traversal.

The BVH algorithm (built upon Barnes-Hut) is shown below as a step-by-step procedure executed at each time step.

1. **Calculate Bounding Box:** Locates a bounding box within a Cartesian grid capable of containing all $N$ bodies.
2. **Hilbert Sort:** Computes the Hilbert index which is a 1D mapping of each body's grid cell and sorts the bodies based on their index. In practice, the Hilbert index is precomputed for all bodies. The Hilbert index of each body's grid cell is computed with the Skilling's Grey code algorithm [6].
3. **Build Tree and Calculate Moments:** Constructs the BVH leaf nodes from the bodies and then constructs each higher level using reductions of the bounding boxes and moments of their children. Since the reductions at each node are independent and per-level, this process can be efficiently parallelized.

4. **Calculate Forces:** Similar to Barnes-Hut except in two aspects: the algorithm can jump from a leaf node to the next node in the DFS traversal across multiple levels without traversing nodes in-between because it are traversing in Hilbert sort order. Secondly, since bounding boxes may overlap, moments may need higher order terms in the expansion, so the accuracy may vary from Barnes-Hut for the same distance threshold.

5. **Update Positions:** Same as Barnes-Hut by integrating the approximated forces according to Newton's laws.

# 3 Methods

This section details the specific implementation strategies for the sequential and parallel versions of the $N$-body algorithms we developed. All experiments were conducted on the zoo cluster except for the CUDA runs. The zoo nodes were typically 20-32 cores, and we used standard C++17 libraries (GNU v11.4) along with OpenMP libraries with shared-memory parallelism and Parlay library with task-based parallelism for the implementations. We used the (header only) version of Parlay library that was used for the class assignments and the default OpenMP version available on the zoo.

## 3.1 Parallel Brute-Force Approaches

$N$-Body simulations can be calculated with full accuracy using a brute-force all-pairs approach that involves going through all pairs of the $N$ bodies. For each pair, the gravitational force of one body on the other body can be calculated in constant time. Therefore, sequential brute-force approaches for $N$-body simultations run in $O(N^2)$ time. Parallelizing these approaches by evenly distributing the total workload among $P$ processors achieves a work of $O(N^2)$ and a span of $O(N^2/P)$.

We developed three parallel brute-force approaches: a memory-efficient approach using $O(N)$ space, a faster but more memory-intensive approach requiring $O(N \cdot P)$ space, and our fastest brute-force approach using CUDA. For the first two approaches, we created two separate implementations each using OpenMP and ParlayLib respectively.

### 3.1.1 Parallel Brute-Force Memory-Efficient Approach

We first parallelized the outer loop of $i$ for computing each $\vec{F_i} = \sum_{j \neq i} G \cdot m_i m_j / r_{ji}^2 \cdot \hat{r}_{ji}$. Each workload is independent since it completely calculates the force on one body, enabling parallelization with efficient $O(N)$ space complexity.

### 3.1.2 Parallel Brute-Force Memory-Intensive Approach

While the memory-efficient approach above requires $N(N-1)$ force calculations in total, we can instead compute $\frac{N(N-1)}{2}$ forces $\vec{F_{ij}}$ (i.e. the force of body $j$ on body $i$) and trivially compute the remaining forces with $\vec{F_{ji}} = -\vec{F_{ij}}$. We use the same parallelization of the outer loop of $i$ as described in the memory-efficient approach. While this halves the total workload, it also means that the workloads are no longer independent as forces are now collectively computed by multiple workers. In order to prevent race conditions, we create thread-local storages for partial values of $F_i$ and merge them at the end to obtain the values of $F_i$. While this approach is roughly twice as fast, it is much more memory-intensive as it requires $O(N \cdot P)$ space.

### 3.1.3 Parallel Brute-Force CUDA Approach

To leverage GPU acceleration, we implemented a brute-force approach using CUDA. Each GPU thread is assigned a single body $i$ and computes the force $\vec{F_i}$ on that body due to all other bodies. Because reading from global memory is much slower than reading from shared memory in CUDA, we employed a tiled-approach, where a shared memory tile is filled with a block of bodies at a time. Each thread loads its unique body $B$ into shared memory, and computes the partial forces from all the tile bodies on $B$. The `__syncthreads()` function is called before calculating forces for the current tile to ensure that all threads have finished loading their part of the tile before proceeding. Furthermore, synchronization is also used after calculating forces to ensure that all threads finish using the current tile before it is overwritten by the next tile.

## 3.2 Parallel Barnes-Hut Implementation

The Barnes-Hut (BH) algorithm, offering an average-case time complexity of $O(N \log N)$, approximates forces by hierarchically grouping particles. Its main computational stages involve constructing a spatial tree (an octree in 3D or quadtree in 2D), calculating the center of mass and total mass for each node in the tree, and then traversing the tree for each particle to compute forces. The `BARNES_HUT_THETA` parameter, set to 0.25 in our simulations, dictates the accuracy and extent of this approximation by controlling the multipole acceptance criterion.

**OpenMP Implementation** In the OpenMP version of the Barnes-Hut algorithm, parallelization is primarily concentrated on the force calculation phase. After the spatial tree is constructed (often sequentially or with limited parallelism in simpler OpenMP models), it is treated as a shared, read-only data structure by all threads. The loop iterating over each of the $N$ bodies to calculate its net force is parallelized using the `#pragma omp parallel for` directive. Each thread independently traverses the tree for its assigned subset of bodies to compute their respective forces.

**ParlayLib Implementation** The ParlayLib implementation of Barnes-Hut aims for more comprehensive parallelism. Both the tree construction phase and the force calculation phase are parallelized. ParlayLib's functions for parallel tree building (e.g., leveraging recursive parallelism and work-stealing schedulers) are employed for efficient octree/quadtree generation. For force calculation, `parlay::parallel_for` is used to distribute the work of computing forces for each particle. The inherent work-stealing capabilities of ParlayLib are particularly advantageous for balancing the computational load, given that tree traversals for different particles can vary significantly in depth and complexity.

## 3.3 Parallel Bounding Volume Hierarchy (BVH) Implementation

The Bounding Volume Hierarchy (BVH) method, also achieving $O(N \log N)$ complexity, involves sorting particles along a 1D Hilbert space-filling curve to enhance data locality, followed by constructing a binary tree (the BVH) over these sorted particles. Forces are then computed by traversing this BVH.

**OpenMP Implementation** In the BVH algorithm, the tree building function creates the BVH structure recursively by splitting bodies along the longest axis and building left and right subtrees. This can be parallelized using OpenMP tasks with each recursive call becoming a task with taskwait to ensure both subtrees are built before returning the node. In the force calculation phase, forces can be calculated independently for all bodies, making it an embarrassingly parallel operation with #pragma omp parallel where each thread processes a subset of bodies, computing their forces independently. This provides good scalability also. The function to split bodies involves finding bounds and sorting bodies, which can be parallelized using reductions and parallel sort. In the OpenMP version, the primary parallelization occurs during the force calculation phase. Once the tree is built, the loop iterating over each particle to compute its net force via BVH traversal is parallelized using the `#pragma omp parallel for` directive. During this phase, the BVH structure is treated as a shared, read-only data structure by the threads. While the initial sorting of particles (e.g., by Hilbert curve index, as stated in the project's overall design) could be parallelized using OpenMP, the provided wrapper focuses parallelism on the force computation step post-construction.

**ParlayLib Implementation** The ParlayLib wrapper mirrors the OpenMP strategy in terms of where parallelism is applied. The recursive BVH tree construction naturally maps to Parlay's fork-join parallelism where the building of left and right subtrees can be independently parallelized using parlay::par_do:. The force calculation phase can be parallelized with Parlay's parallel_for: distributing the work across available cores, with automatic work-stealing for load balancing. While in our algorithm particles are used to construct the BVH object sequentially. Subsequently, `parlay::parallel_for` is employed to distribute the force calculation tasks, where each task involves a particle traversing the BVH. Although the broader ParlayLib ecosystem offers primitives for parallel tree construction (e.g., through parallel sorting for initial ordering and recursive parallel tasking for node creation), the current wrapper utilizes these for the force calculation loop over a sequentially built tree.

## 3.4 Parallel Fast Multipole Method (FMM) Implementation

The Fast Multipole Method (FMM), with its potential for $O(N)$ complexity, achieves efficiency through a hierarchical decomposition of space and a sequence of distinct computational passes: tree construction, an upward pass (P2M, M2M), a transfer pass (M2L), a downward pass (L2L, L2P), and a direct interaction pass (P2P). Our parallel FMM implementations build upon a common sequential FMM tree and interaction list construction logic, which are then used by specialized OpenMP and ParlayLib classes that parallelize the subsequent computational phases.

**OpenMP Implementation** The OpenMP FMM parallelizes each phase primarily using loop-level parallelism via `#pragma omp parallel for`. After initial setup and creation of node-per-level lists, the P2M (Particle-to-Multipole) calculations are parallelized over leaf nodes. The M2M (Multipole-to-Multipole) and L2L (Local-to-Local) translations are processed level-by-level (bottom-up for M2M, top-down for L2L), with parallel loops over the relevant nodes at each level. For the M2L (Multipole-to-Local) pass, parallelism is applied over all tree nodes that possess interaction lists, with each thread then sequentially processing its assigned node's list. The L2P (Local-to-Particle) evaluations and P2P (Particle-to-Particle) direct computations are parallelized by distributing work over the leaf nodes. Synchronization is implicitly managed by the sequential execution of these phase-specific parallelized methods.

**ParlayLib Implementation** The ParlayLib FMM version, after the same initial sequential setup, also parallelizes each computational phase, often employing more adaptive or fine-grained tasking. P2M operations on leaf nodes and M2M translations (processed level-by-level) are parallelized using `parlay::parallel_for`. The M2L pass iterates over tree levels, filters nodes with interaction lists, and then applies `parlay::parallel_for`; the processing of a node's interaction list itself uses an adaptive strategy. L2L translations are implemented with a recursive `parlay::par_do` structure for hierarchical parallelism. The L2P evaluations are parallelized over all bodies. For P2P interactions, a flat list of all direct interaction work items is generated and then processed in parallel as well. ParlayLib's work-stealing scheduler assists in balancing load across these varied parallel patterns.

# 4 Results

## 4.1 Simulation Setup

All simulations were run in triplicate with N ranging from 1000 to 5 million bodies and run for 2D and 3D scenarios with distances uniformly distributed in the range $[1, 10^7]$ and masses uniformly distributed in the range $[1, 10^8]$. These represent reasonable ranges for simulations of galactic forces. The distances are expressed in Astronomical Units (AUs) (distance between earth and sun $\sim 1.5 \times 10^{11}$ m) and masses are expressed in Earth masses ($M_E = 6 \times 10^{24}$ kg). The corresponding value of $G = 6.674 \times 10^{-11} \frac{m^3}{kg^2 s^2} = 4.471 \times 10^{-21} \frac{\text{AU}^3}{M_E \cdot s^2}$

## 4.2 Algorithm Runtime Performance Comparison

Figures 4 and 5 below show how each N-body simulation algorithms performs as the number of bodies increases for 2D and 3D scenarios.

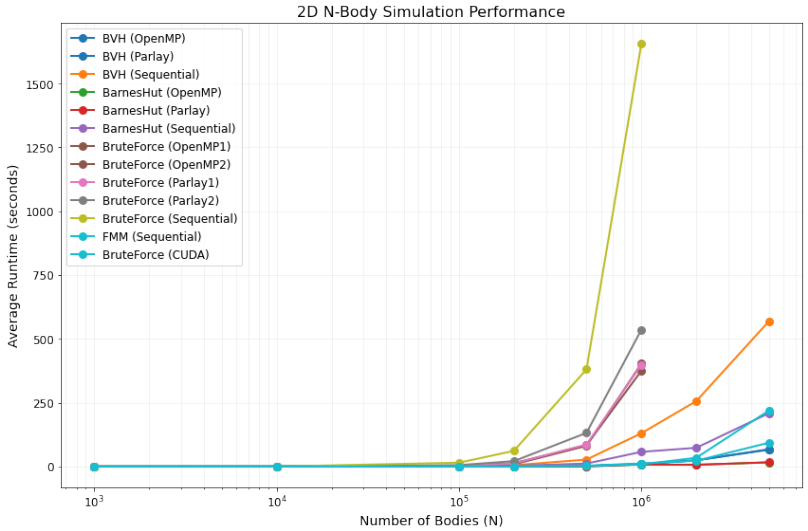


Figure 4: Average Runtimes for each algorithm for 2D as the number of bodies increases

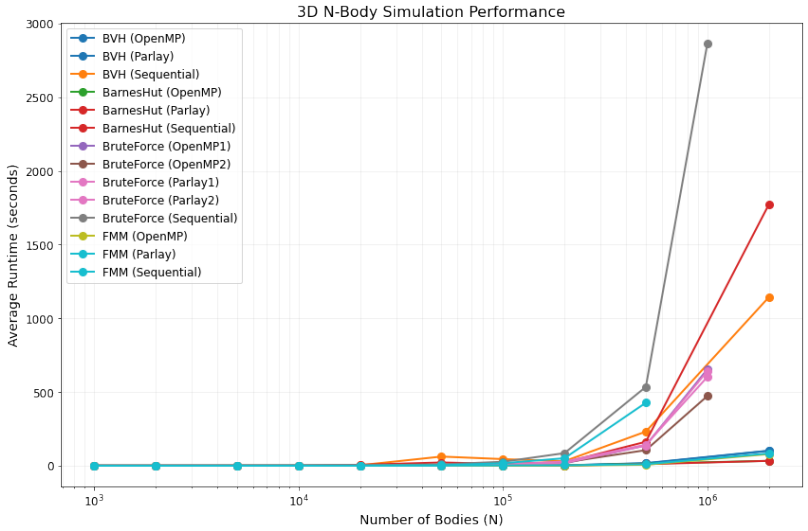


Figure 5: Average Runtimes for each algorithm for 3D as the number of bodies increases

From these results, we observed the following:

1. Parallelization Benefits: OpenMP and Parlay implementations clearly show better performance compared to their sequential counterparts especially for N larger than $10^5$
2. Barnes-Hut generally had superior performance compared to BVH and FMM approaches across most problem sizes across 2D and 3D simulations. In particular, there were numerous stability issues with FMM at smaller N values for 2D and in most cases the algorithm ran into convergence issues with multipole expansions or exceptions. We will need to further optimize this algorithm so that the algorithm produces more reliable and accurate results.
3. Runtimes for 3D simulations were 4 times those needed for 2D over all methods and all sizes of N. Performance was similar for each algorithm across all problem sizes between 3D and 2D and scaled close to the expected asymptotic limits.
4. Brute force CUDA implementation shows performance gains comparable to optimized parallel versions of Barnes Hut or BVH running on CPUs for most problem sizes.

## 4.3 Speedup Analysis

Figure 6 and 7 below are heatmaps to visualize the speedup factors of each algorithm relative to the brute force sequential implementation.


Figure 6: Heatmap of speedup relative to Brute force (all-pairs) sequential for each algorithm for 2D simulations

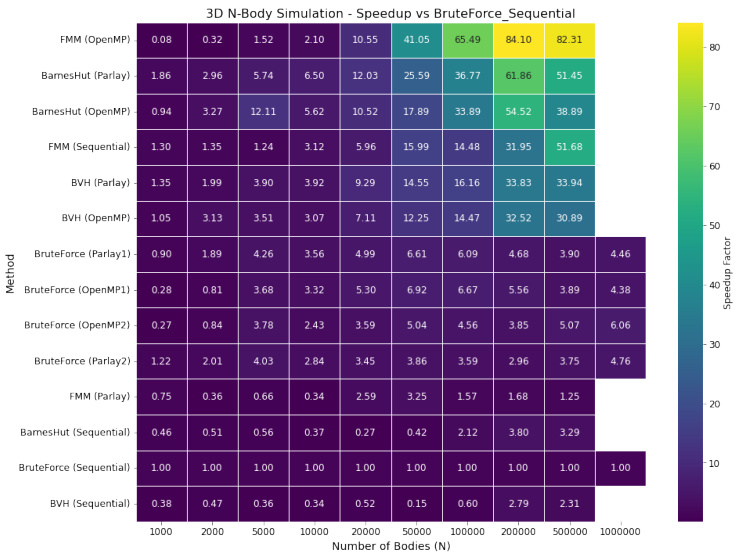


Figure 7: Heatmap of speedup relative to Brute force (all-pairs) sequential for each algorithm for 3D simulations

From the above results, we observed the following:

1. Both Open MP and Parlay parallel versions of Barnes-Hut produced consistently the highest speedup for 2D simulations ranging from 1.5-$\sim$390X across the range of N tested.
2. For 3D simulations, FMM with OpenMP was the clear winner for larger N (50,000 and above) while Barnes and Hut was better at smaller N and coming in second for larger N. Both parallel versions of BVH also consistently performed well coming in third with speedups of 1.5-30X. FMM Sequential was also surprisingly effective providing upto 50X speedup at 500K bodies.

# Conclusions

Based on the performance analysis and speedup measurements, Barnes-Hut and BVH algorithms consistently outperform brute force approaches for large body counts, with FMM showing competitive performance at scale. The parallel implementations (OpenMP and Parlay) provide substantial speedups over sequential versions, particularly at larger body counts. 3D simulations impose a significant performance overhead compared to 2D, with an average ratio of approximately 4x across all methods. Tree-based methods (Barnes-Hut, BVH) show higher 3D/2D ratios (5-11x) than brute force methods (1.4-1.7x). FMM with OpenMP achieves excellent speedups at medium to large scales (10K-500K bodies), reaching up to 80x faster than sequential brute force in 3D. FMM Parlay implementation shows lower performance than OpenMP, suggesting some implementation inefficiencies. Barnes-Hut parallel implementations show excellent speedups, with both OpenMP and Parlay versions achieving 30-60x speedup over sequential brute force for large body counts with Parlay slightly outperforming OpenMP at the largest scales (500K bodies).

# Future Optimization Opportunities

Our FMM implementation is the one that presents numerous optimization opportunities. For example, the Parlay implementation could benefit from better workload distribution and adjusting the multipole expansion order based on the problem size could optimize accuracy vs. performance. Applying vectorization optimizations to the multipole and local expansion evaluations and reorganizing tree traversal to improve cache utilization during the upward and downward passes. An interesting extension of this work could be hybrid algorithms that combine the strengths of different algorithms (e.g., FMM for far-field, direct Barnes-Hut calculation for near-field) that can effectively utilize both CPU and GPU resources. Another interesting extension could be runtime selection of optimal algorithm based on problem characteristics.

# References

[1] Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096):446–449, 1986.

[2] Bérenger Bramas. Tbfmm: A c++ generic and parallel fast multipole method library. *Journal of Open Source Software*, 5(56):2444, 2020.

[3] J Carrier, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM journal on scientific and statistical computing*, 9(4):669–686, 1988.

[4] Thomas Lane Cassell, Tom Deakin, Aksel Alpay, Vincent Heuveline, and Gonzalo Brito Gadeschi. Efficient tree-based parallel algorithms for n-body simulations using c++ standard parallelism. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 708–717, 2024.

[5] Erik Holmberg. On the Clustering Tendencies among the Nebulae. II. a Study of Encounters Between Laboratory Models of Stellar Systems by a New Integration Procedure. , 94:385, November 1941.

[6] J Skilling. Programming the hilbert curve. *American Institute of Physics*, 707:381–87, 2004.

[7] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.