Optimizing Cache Locality with Z-Order Curves to Parallelize Barnes Hut Tree Construction

Nick Twum

1 Introduction

N-body simulations were something that captivated me ever since we had that lesson about them. These simulations appeared simple at first; iterating through each and every single pair of particles seemed easy. However, diving into the rabbit hole of the myriad of extremely sophisticated optimization techniques that have developed in the past decades overwhelmed me. From spatial partitioning to fast multipole methods, I became obsessed in trying to implement my own version of an N-body simulation. After long hours of research, I came across the paper, "Scalable parallel formulations of the Barnes–Hut method for n-body simulationss" by Anantha Garma, and "Optimizing the gravitational tree algorithm for many-core processors" by Tomoyuki Tokuue. These two papers, had something in common: Z Curves. Z Curves also known as Morton Curves are a form of spatial partioning, which map multidimensional space in a single dimension and are useful for subdividing a domain of particles to be fed into multiple threads for parallelizing Barnes Hut Trees. Z curves also manage to arrange the particles in way such that they are close together in memory, improving memory locality[1]. Enchanted by the possibilities of such a simple pattern in filling grids, I started my journey in trying to implement it in an N-body simulation, one which is built using the Barnes-Hut tree algorithm. Thus my research question was born, and I ventured out to figure out the potential of Z-level curves in speeding up N-body simulations

2 Proposed Research Question

To what extent can N-body simulations be made efficient through the synergy of parallelized Barnes-Hut tree construction and Z-level curves.

3 Methodology

This section will discuss initial attempts at developing this N-Body Simulation. Analysis and comparisons will be made on different algorithms and optimizations that were implemented initially.

3.1 Brute Force N-Body Method

The simulation intially employed the Brute Force N-body Method, which is a direct approach whereby the gravitiational attraction between each pair of particle is calculated. At a given timestep, the net force acting on a particle is computed, and its positions and accelerations are updated using Euler's Method for numerical integration as follows:
For body i at time step n, the net acceleration due to all other bodies j is [2]:

$$\mathbf{a}_i^n = -G \sum_{\substack{j=1 \\ j \neq i}}^{N} m_j \frac{\mathbf{r}_j^n - \mathbf{r}_i^n}{\left(\|\mathbf{r}_j^n - \mathbf{r}_i^n\|^2 + \epsilon^2 \right)^{3/2}}$$

$$\textbf{Velocity update :} \quad \mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t$$
$$\textbf{Position update :} \quad \mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \mathbf{v}_i^n \Delta t$$

1. Grama, Ananth, et al. "Scalable Parallel Formulations of the Barnes–Hut Method for N-Body Simulations." Parallel Computing, vol. 24, no. 5–6, 1998, pp. 797–822, doi:10.1016/S0167-8191(98)00011-8.

2. Gregor Hartl Watters (https://physics.stackexchange.com/users/336755/gregor-hartl-watters), Initializing positions of n-body simulations, URL (version: 2023-02-09): https://physics.stackexchange.com/q/749288

- G: Gravitational constant
- $\mathbf{r}_i^n$: Position of body i at step n
- $\mathbf{v}_i^n$: Velocity of body i at step n
- ϵ: Softening length (avoids singularities when $r_{ij} \to 0$)
- Δt: Time step size

3.1.1 Object Class

It was necessary to create an object class, which had the attributes of position, velocity, mass and radius; factors that will be taken into consideration in the calculating the gravitional force.

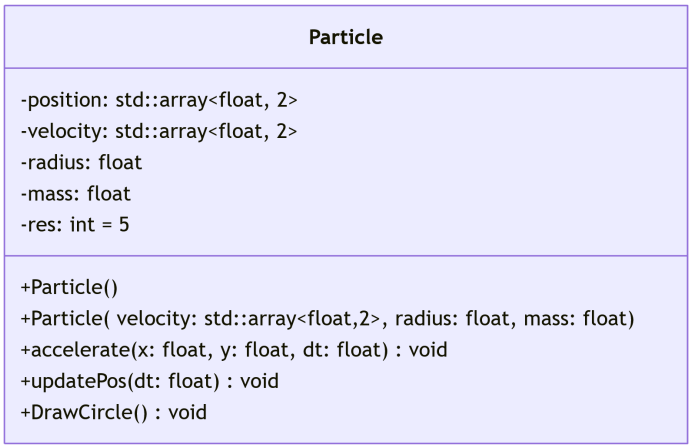

Figure 1: Particle UML diagram

The position and velcoity are rerpesented as C++ standard library arrays in order to optimize memory allocation. DrawCircle() renders the particle using OpenGL,which offers fast real time visualizations as the positions of the particles are updated. The acceleration function works in accordance to Eulers Step Method discussed above and computes the new velocity, which is then used to compute the position. We calculate the acceleration of each pair of masses through a double for loop leading to $O(\mathrm{N}^2)$ time complexity

3.1.2 Performance of Brute Force Method

The resulting N-body simulation had decent performance and averaged around 25ms per time step for 3000 bodies. However, due to the $O(\mathrm{N}^2)$ time complexity of the algorithm (from the double for loop), there is an exponential decrease in performance as the number of bodies (N bodies) increases.

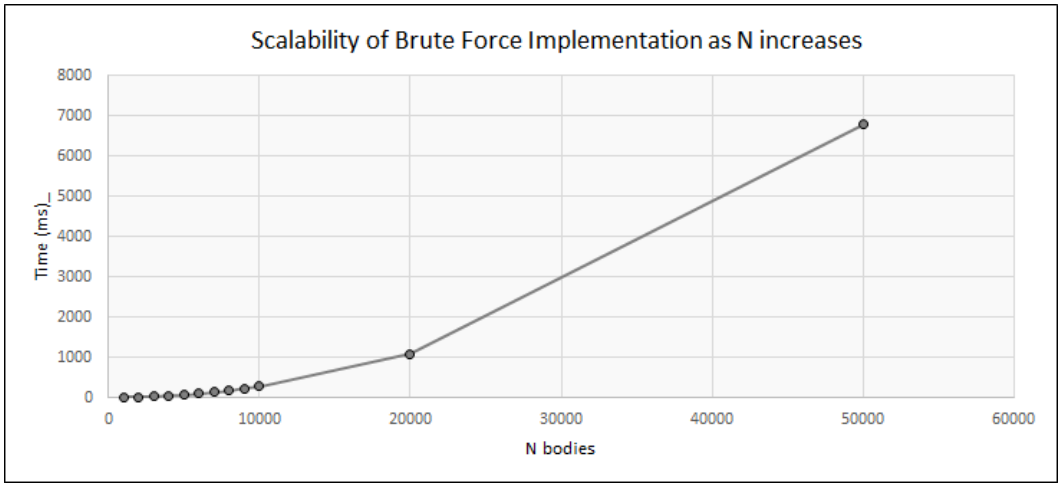

Figure 2: Exponential increase of computational time as number of bodies increases

While the brute force approach is rendered unusable at higher number of bodies, it provides a decent foundation to implement the Barnes-Hut method which uses spatial partitioning to substantially increase performance.

3.2 Barnes-Hut Implementation

The Barnes-Hut Method, implements a Quad/Oct Tree, which is a spatial database that recursively divides a given space into quadrants. Quad trees are extremely useful in N-body simulations as they provide a way to estimate long range gravitational forces. In a given cluster of particles, if a particle is far away enough from the cluster, The cluster of particles can be estimated to be one particle with its mass being the sum total of the particles which make up the cluster. The Barnes Hut Tree, thus reduces the time complexity of N-body simulation from $O(N^2)$ to O(NlogN) complexity which is substantially better than the Brute Force implementation[3].

3.2.1 Implementing the Barnes-Hut Tree Structure

The implementation of the Barnes-Hut Tree boils down to creating a Quad-Tree/Oct-Tree with the appropriate force calculation algorithm as a method. The tree starts with a root node extends downwards to 4 children nodes representing the quadrant divisions of the area our simulation takes place.

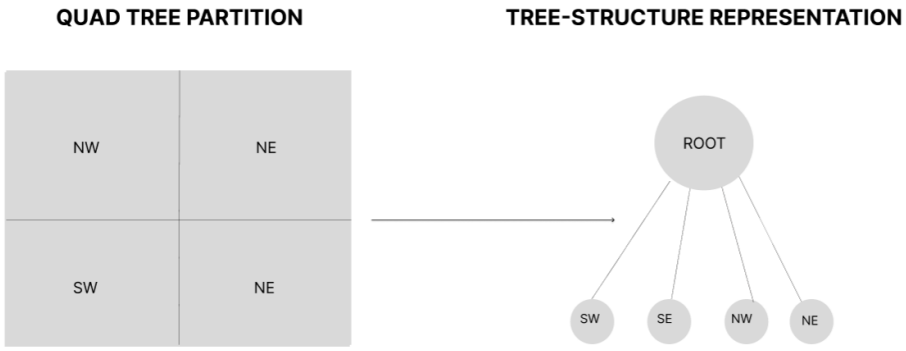

Figure 3: Paritioned Space Being Represented as a Quad-tTree

In creating the Barnes-Hut Tree, we first recursively partition the simulation space. The partitioning must follow the rule of atomicity, i.e., there must always be at most one particle per leaf node. Thus, we continously divide the space till each particle is in its own leaf cell.

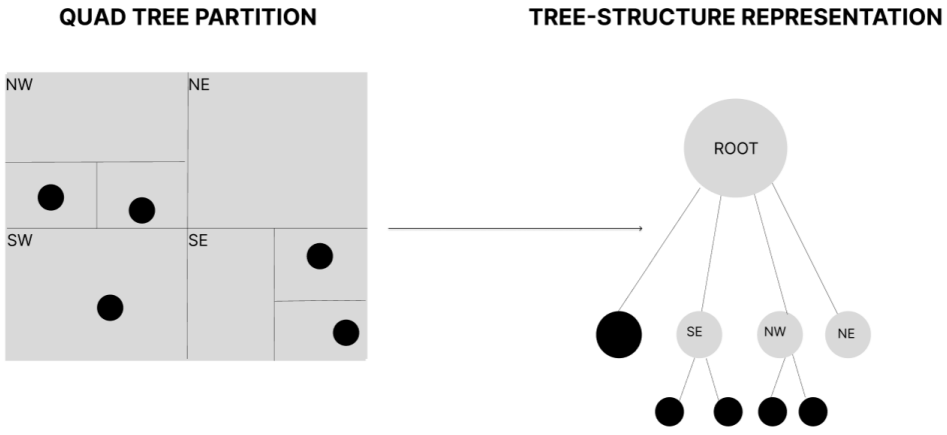

Figure 4: Below shows the partition while maintaining atomicity of leaf cells

3. Grama, Ananth, et al. "Scalable Parallel Formulations of the Barnes–Hut Method for N-Body Simulations." Parallel Computing, vol. 24, no. 5–6, 1998, pp. 797–822, doi:10.1016/S0167-8191(98)00011-8.

The mass, and center of mass of each quadrant is adjusted as nodes are continously added. This is helpful in the final force calculation as quadrants can be estimated to be single particles, reducing the number of needed computations. Performance of a Barnes-Hut Tree is determined by the θ variable, which is the threshold that determines if a group of distant particles can be approximated as one. θ can be computed through the following equation

$$\theta = \frac{s}{d} \tag{1}$$

Where s is the width of the node the cluster is found, and d is the distance between the particle and the center of mass of the node.

3.2.2 Barnes Hut Data Structure

The UML diagram below shows the structure of our implemented Barnes-Hut Tree data structure

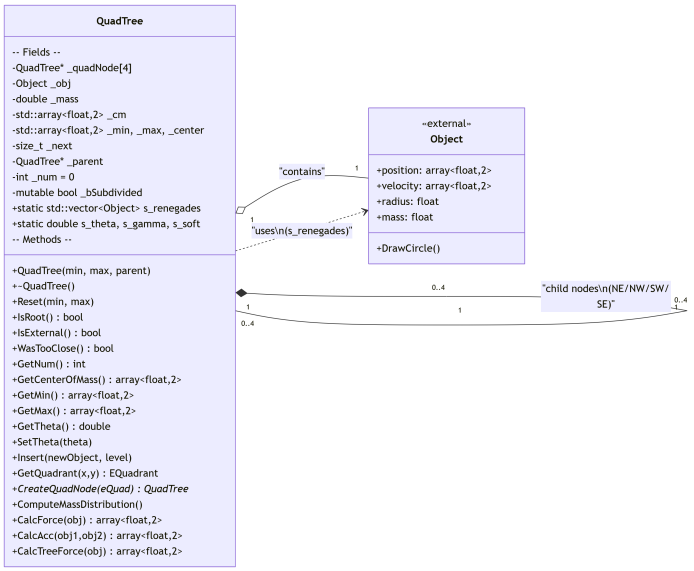

Figure 5: Barnes Hut Tree UML Diagram

The tree contains an array of pointers to other QuadTree nodes. These represent the children nodes or divided quadrants of the parent Node. The mass distribution of every node is calculated at the end of construction. Each quad node has a parent pointer, except the root node. This is useful in traversing the tree while inserting bodies and calculating the force.

3.2.3 Explanation of Insertion and Force Calculation Algorithms

Algorithm 1 QuadTree Particle Insertion

```
1:  function Insert(node, particle, level)
2:      if node.isEmpty() then
3:          node.particle ← particle
4:          node.mass ← particle.mass
5:          node.center_of_mass ← particle.position
6:          return
7:      if node.isInternal() then
8:          if node.isLeaf then                    ▷ Subdivide if containing another particle
9:              Subdivide(node)
10:             quadrant ← GetQuadrant(node, node.particle.position)
11:             Insert(node.children[quadrant], node.particle, level + 1)
12:             node.particle ← None
13:         quadrant ← GetQuadrant(node, particle.position)
14:         Insert(node.children[quadrant], particle, level + 1)
15:     UpdateCenterOfMass(node, particle)         ▷ Propagate COM upward
```

For insertion, as seen above, a vector of particle objects is iterated through and each body is passed as a parameter to the insert method. The algorithm checks if a leaf node is empty, and assigns the particle to that node. If node already contains a particle, the node is subdivided into quadrants, and a recursive call is made, passing the specified child node the particle belongs to as the parameter.

Algorithm 2 Barnes-Hut Force Calculation

function CalculateForce($node, target, \theta$)
  if $node.isEmpty()$ **then**
    return $(0, 0)$
  if $node.isLeaf$ **and** $node.particle \neq target$ **then**
    return GravitationalForce($target, node.particle$)
  $d \leftarrow \|node.center\_of\_mass - target.position\|$
  $s \leftarrow node.bounds.width$
  if $s/d < \theta$ **then** ▷ Approximate as pseudo-particle
    $pseudo \leftarrow$ Particle($node.center\_of\_mass, node.mass$)
    return GravitationalForce($target, pseudo$)
  else
    $force \leftarrow (0, 0)$
    for $child$ **in** $node.children$ **do**
      if $child \neq$ None **then**
        $force \leftarrow force +$ CalculateForce($child, target, \theta$)
    return $force$
function GravitationalForce(p_1, p_2)
  $r \leftarrow p_2.position - p_1.position$
  $distance \leftarrow \sqrt{r_x^2 + r_y^2 + \epsilon^2}$ ▷ ϵ: softening
  return $G \cdot \frac{p_1.mass \cdot p_2.mass}{distance^3} \cdot r$

The Barnes-Hut algorithm approximates gravitational forces in O(N log N) time by recursively traversing a Quad/Oct Tree spatial hierarchy. For each particle, it starts at the root node and checks whether to treat a node as a single pseudo-particle or traverse its children, based on the θ criterion ($\theta = \frac{\text{Node Width}}{\text{Distance To Particle}}$). If $\theta <$ threashold (typically 0.5), the node's center of mass and total mass are used to compute an approximate force, bypassing individual particle calculations. Otherwise, the algorithm recurses into the node's children. Gravitational force is calculate with a softening parameter (ϵ) to avoid singuarities. Empty nodes are skipped, while leaf nodes (containing particles) compute direct forces[4].

3.2.4 Performance Analysis of Barnes-Hut Tree

As mentioned the Barnes-Hut method offers significant performance improvement compared to the brute force method.

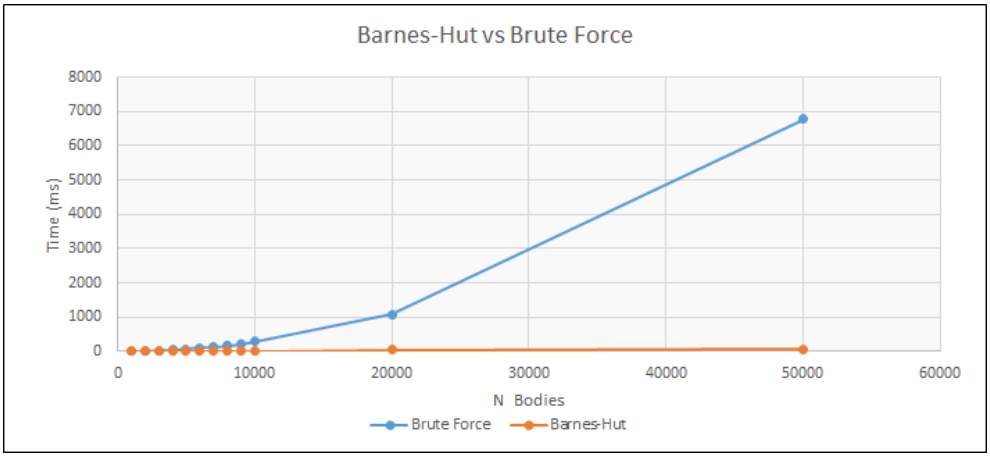

Figure 6: Comparison of Barnes-Hut and Brute Force N-body methods

4. Tokuue, T., Ishiyama, T. (2023). Optimizing the gravitational tree algorithm for many-core processors. Monthly Notices of the Royal Astronomical Societ, 528(1), 821-832. https://doi.org/10.1093/mnras/stad4001

The above graph demonstrates the efficiency the Barnes-Hut tree algorithm over the Brute-Force method. The exponential time complexity of the Brute-Force method is made evident in the above graph. The Barnes-Hut method starts off equivalent to the Brute-Force method, likely due to the relatively cheap cost of computing bodies, under 10,000 bodies. However, as the N-bodies move past 10,000 bodies the compute time increases expoenentially. The Barnes-Hut Tree method, is substantially more efficient and thus, will be used for the remainder of this project. However, what differs the Barnes-Hut method, even more from the brute force, is that a significant chunk of compute is not used for calculating the force, but rather used to construct the Quad-Tree and calculate the mass distribution.

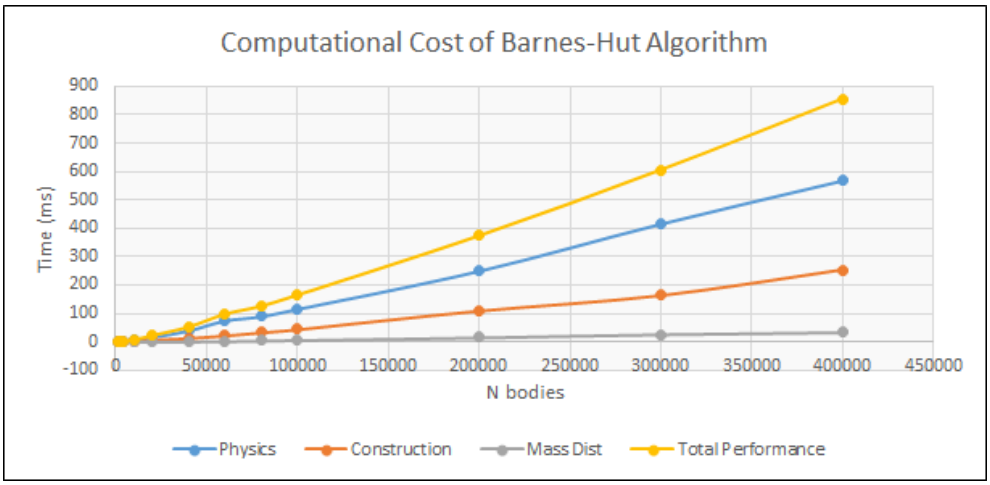

Figure 7: Computational Cost of Barnes-Hut Algorithm

The graph above shows the computational cost of each stage of the Barnes-Hut algorithm. As expected the, physics calculations take up most of the compute. However, the construction accounts for approximately 29% of the compute time, and thus is a vital factor to consider in making further optimizations to the model.

3.2.5 Finalized Implementation

After implementing the Barnes-Hut Tree, significant performance gains where observed, allowing for more robust physics simulations. The diagram below shows a fully working implementation of the Barnes-Hut tree.

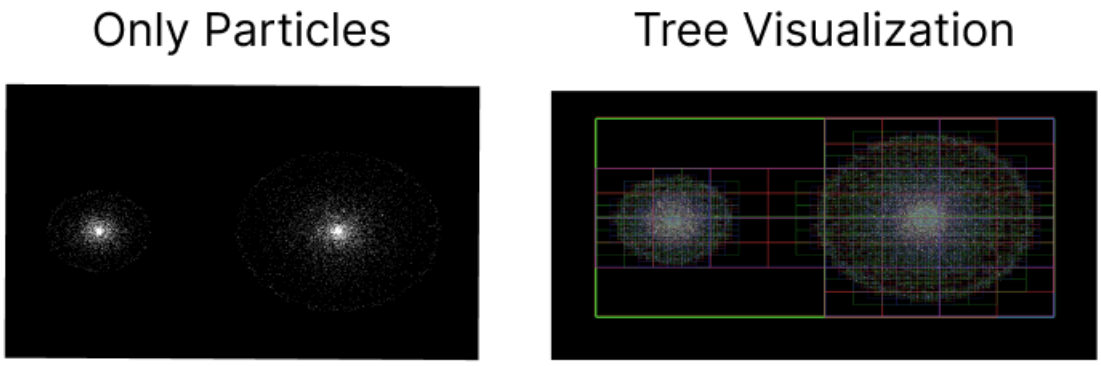

Figure 8: Quad Tree Visualization in Real Time Simulation

The spatial partitioning of the simulation space based on particle positions is easily observable. The visualization confirms that the tree algorithm is working as intended since cells maintain their atomicity.

4 Parallelizing the Barnes Hut Tree

One way we can significantly increase the performance of the Barnes-Hut Tree is through parallelization. OpenMP has proved valuable for parallelizing both the force and tree construction phases on the Barnes-Hut algorithm. By

distributing work using the OpenMP **pragma** directives, within 8 threads, the speed of force calculation can be brought down significantly.

4.1 Parallelizing the Acceleration

The parallelizing of the acceleration was straightforward as it is just a simple for statement iterating through the list of bodies. For this we use the #**pragma omp parallel for** directive, which divides the loop iterations among available threads. This approach is particularly effective for the force calculation phase since each body's acceleration is computed independently.

4.1.1 Performance Evaluation

The graph below compares the single threaded and multithreaded force compute time.

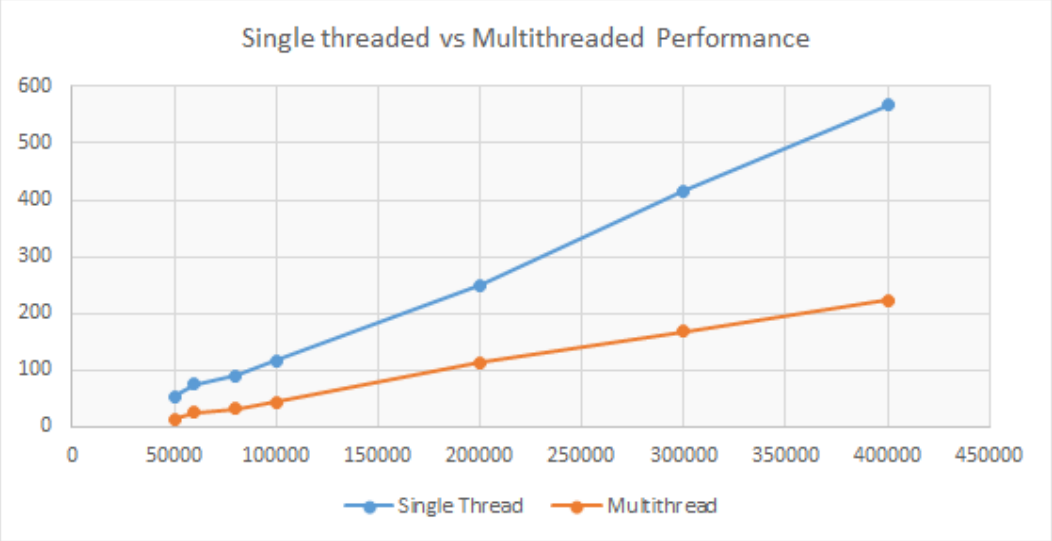

Figure 9: Comparing Single Threaded and Multi-threaded Performance

Parallelized performance seems to perform better. However, in calculating the average performance gain, multi-threaded seems to be approximately 2.80 times faster. This, given the number of cores we used, is low, as we should have expected an 8x increase in performance. This could be possibly due to memory bottle necks.

4.2 Parallelzing the Tree Construction

Unlike force calculation, parallelizing tree construction is challenging. While force calculations operate on independent particles, tree construction involves shared data structures that can lead to race conditions where multiple threads attempt to modify the same tree nodes. The unpredictable and non-uniform spatial distribution of particles, results in unbalanced workloads across processing threads[5].
To address this challenge, Z-order curves (morton ordering), can be used to sort the particles before insertion. Z-order curves provide an effective solution by mapping multidimensional data to one dimension while preserving spatial locality. Encoding morton particles, provides a predictable insertion pattern, which can be taken advantage off to parallelize the tree. Arranging the randomly distributed particles into a spatially coherent sequence allows us to split the sorted array into segments that can be processed independently. Each thread can then construct a subtree for its assigned segment, with these subtrees later merged into a complete structure minimizing the imbalance.

4.2.1 Morton Ordering

In order to sort the particles in morton order, the x and y coordinates of the particles must be converted into morton keys. This involves taking the integer x and y coordinate of a particle and interweaving the bits as shown below [6].

5. Tokuue, T., & Ishiyama, T. (2023). Optimizing the gravitational tree algorithm for many-core processors. Monthly Notices of the Royal Astronomical Society, 528(1), 821-832. https://doi.org/10.1093/mnras/stad4001

6. Tokuue, T., & Ishiyama, T. (2023). Optimizing the gravitational tree algorithm for many-core processors. Monthly Notices of the Royal Astronomical Society, 528(1), 821-832. https://doi.org/10.1093/mnras/stad4001

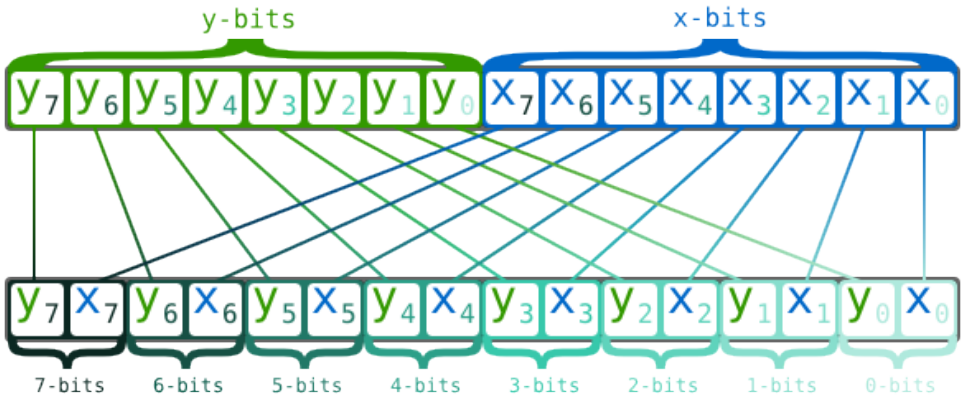

Figure 10: Interweaving of morton code

Unfortunately, in the current simulation implementation, the coordinates of particles are represented via floating point digits, and hence cannot be directly encoded. To account for this problem, a function is created which maps the floating point numbers onto a discrete grid.

```
int toGrid(int grid_size, float coord, float min, float max){
    unsigned int position = (coord - min) / (max - min) * (grid_size - 1);
    return static_cast<unsigned int>(position);
}

inline uint64_t encodeFloat(float x, float y, float min, float max){
    uint32_t int_x = toGrid(1048576, x, min, max);
    uint32_t int_y = toGrid(1048576, y, min, max);
    return encodeMortonKey(int_x, int_y);
}
```

The toGrid function transforms each floating-point coordinate into an integer grid cell index by normalizing the coordinate within its range [min, max] and scaling it to fit within our grid resolution. We use a grid size of 1048576 (2^{20}), providing 20 bits of precision per dimension, which offers sufficient spatial resolution for our simulation domain while avoiding floating-point comparison issues.Once the floating-point coordinates are mapped to their respective grid cells, the encodeMortonKey function interleaves the bits of these integer indices to produce the final Morton key. This transformation preserves spatial locality while enabling efficient sorting and partitioning of particles, forming the foundation for our parallel tree construction approach.

4.2.2 Filling the tree with morton order

The morton order is also known as a z level curve because it fills out a grid in a ”Z-shape.” The order typically moves from the SW to SE to NE to NE.

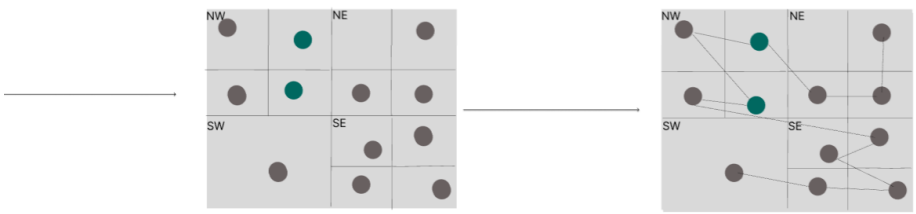

Figure 11: The Z shape Order of Particle Insertion

The line from the southwestern-most cell to the northeastern-most cell, is the order with which cells are inserted. In morton order, the southwestern-most cell is always filled first and the northeastern-most cell last. The diagram below shows the order of particle insertion in the simulation after implementing morton order.

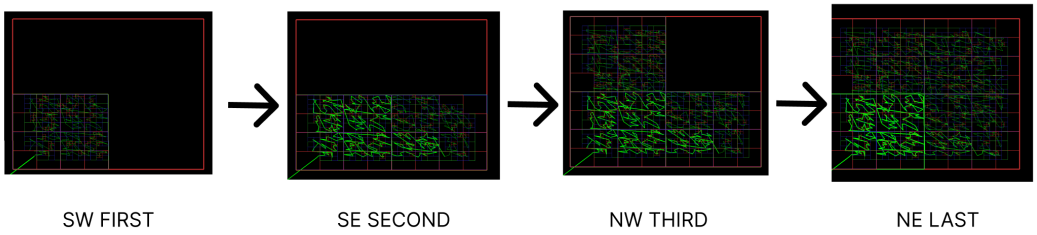

Figure 12: Order of Particle Insertion after Morton Sort

Knowing that particles are going to be inserted in that particular order, we can safely divide our list of bodies into four sections based on the location where particles switch to the next quadrant. Thus, we search for the first particle in a new quadrant, and use these transition points as the boundaries for parallel processing.

4.3 Implementing Parallel Tree Construction

For now, parallelization will be done using four threads. After obtaining the boundaries of the quadrants, we split the bodies into 4 distinct lists. Now, we create 4 new quad nodes, each corresponding to the 4 distinct lists, and we create a for loop, which iterates through the lists and their corresponding nodes. Using #**pragma omp parallel for**, we can parallelize this construction process, allowing each thread to independently build its respective quadrant of the tree structure. Once all four threads complete their work, the main thread attaches the four independently constructed subtrees to the root node, completing the full quadtree structure[7].

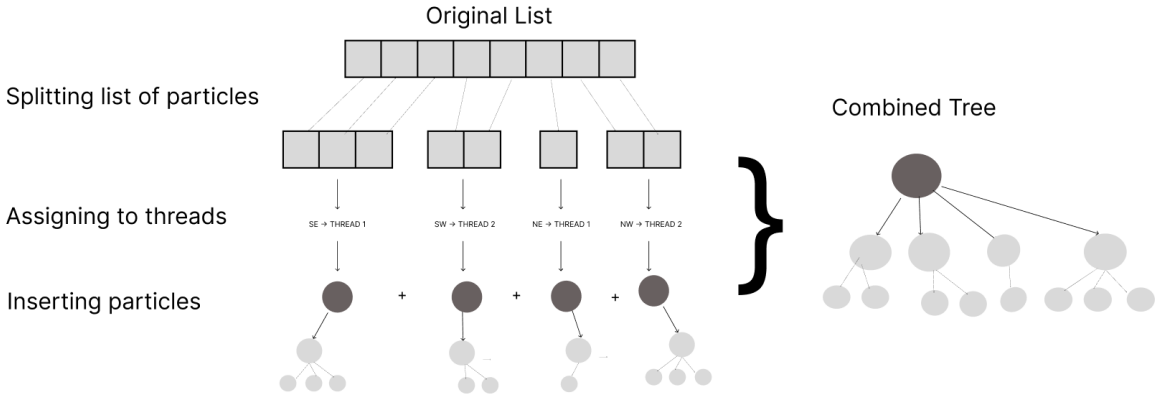

Figure 13: Process of Parallelizing Tree Construction

4.4 Performance Impact of Parallel Tree Construction

4.4.1 Memory Locality with Morton Order

Morton ordering significantly enhances memory locality during both tree construction and force calculation. The Z-order curve maps multidimensional spatial coordinates to one-dimensional values while preserving spatial proximity, meaning particles that are physically close in the simulation space remain close in memory after sorting. Thus, with morton ordering, these adjacent nodes, will also be adjacent in memory leading to reduced memory latency.

7. Tokuue, T., & Ishiyama, T. (2023). Optimizing the gravitational tree algorithm for many-core processors. Monthly Notices of the Royal Astronomical Society, 528(1), 821-832. https://doi.org/10.1093/mnras/stad4001

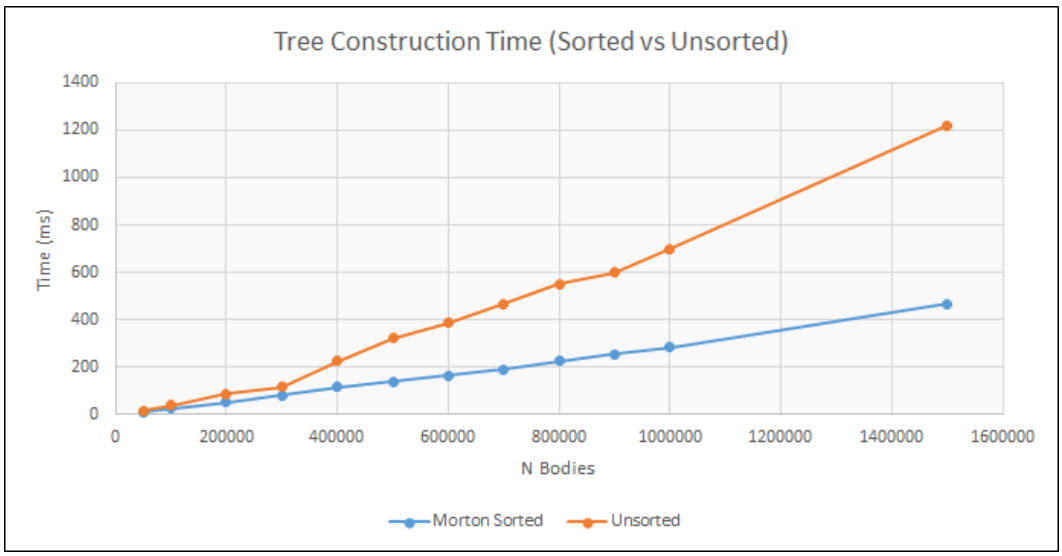

Figure 14: Impact of Optimizing Cache Locality with Morton Ordered Particles

From the graph, we see a significant impact on tree construction time. While the difference is nearly negligible at smaller numbers of bodies, as the number of bodies increases, so does the discrepancy. This scaling behavior aligns with theoretical expectations of cache efficiency benefits and offers a speed up of approximately 202%

4.4.2 Analyzing Impact Of Parallelization

The graph below shows the impact of performance on our parallel Barnes-Hut tree construction implementation.

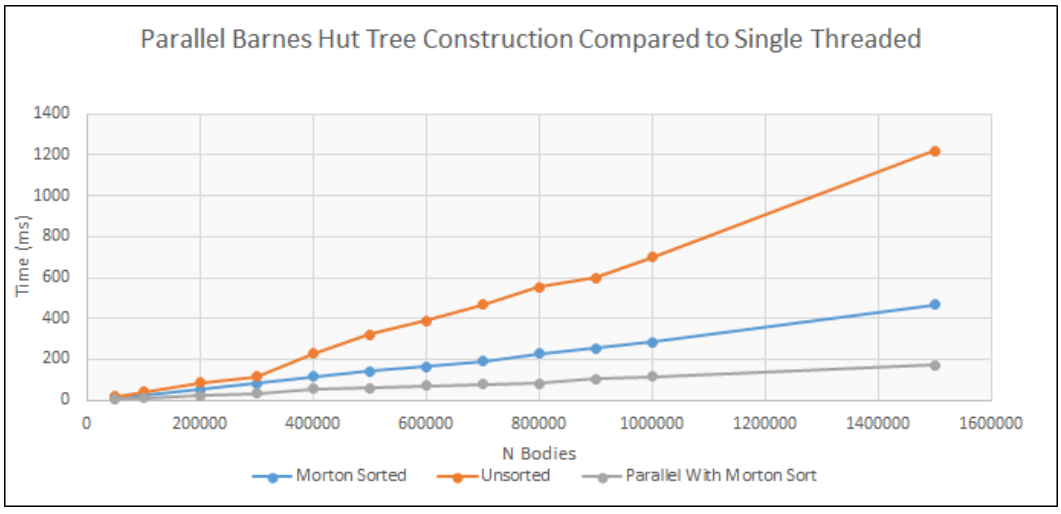

Figure 15: Performance Increase From Parallelized True Construction

The performance data demonstrates that parallelizing tree construction yields substantial benefits that complement our morton ordering optimization. The graph illustrates a clear correlation between increased particle count and greater performance gains from parallelization, with the speedup becoming more pronounced as simulation size grows. The average speedup compared to Unsorted seems to be approximately 5.14 times faster and 2.43 times faster when looking at just parallelization alone. The speedup from parallelization (combined with Morton ordering) transforms tree construction from being a major bottleneck into a much less time-consuming component of the overall simulation. Thus, with these optimizations in place, a real-time N-body simulation, of around 150, 000 bodies can run without issue, efectively, these optimization to the N-Body, sim as brought about a 15 times increase in the number of bodies we can run in real-time.

5 Limitations

There are some heavy limitations to this project, which would need to be addressed in a future revision.

1. The use of only 4 threads for the parallel tree construction. While substantial gains in performance were observed through parallelizing and using morton order, there is still the low handing fruit of using all the available cores of the computer. However, adjusting and figuring out how to partition the children nodes in half and handling their reassembly in the main thread, will prove a challenge in the future, hence why I started with 4 threads.

2. Algorithm for force calculation could be made efficient. This is especially important due to the fact that the force calcuation can specifically be optimized with Z-order curves as the order with which particles appear in, mimics the order of tree traversal to calculate the force.

3. Encoding the morton keys must introduce additional computational cost, which were not considered. These computational costs are much more pronounced in simulations with larger number of particles. More effective encoding algorithms, to quickly encode these floating point particles, should be considered in the future.

6 Conclusion

My journey with this project has been the most rewarding programming experience I have had. Diving into research papers, scouring countless Stack Overflow pages, and finally implementing the Morton order algorithm brought immense satisfaction. I have gained valuable insights into the world of simulations and discovered how seemingly abstract concepts can have unexpected practical applications. Learning about spatial databases and their interpretations while challenging myself has been truly fulfilling. I am proud of what I have accomplished and am committed to further refining this project.

References

[1] Grama, Ananth, et al. ”Scalable Parallel Formulations of the Barnes–Hut Method for N-Body Simulations.” *Parallel Computing*, vol. 24, no. 5–6, 1998, pp. 797–822, https://doi.org/10.1016/S0167-8191(98)00011-8.

[2] Gregor Hartl Watters. ”Initializing positions of n-body simulations.” Physics Stack Exchange, 2023-02-09, https://physics.stackexchange.com/q/749288.

[3] Tokuue, T., & Ishiyama, T. ”Optimizing the gravitational tree algorithm for many-core processors.” *Monthly Notices of the Royal Astronomical Society*, vol. 528, no. 1, 2023, pp. 821-832, https://doi.org/10.1093/mnras/stad4001.