Dark Matter Simulator

Aditi Raju[1], Dennis Du[1], Jared Ni[1], Rafay Azhar[1], and Sezim Yertanatov[1]

[1]Harvard University

May 7, 2024

Abstract

The N-body simulation has become a classic and important problem in modern high-performance computing. It involves simulating the dynamic movement of N discrete bodies in an isolated vacuum space, and in this paper, we will specifically focus on simulating the gravitational force between those bodies. The significance of this project goes beyond just experimenting with different parallelization techniques as N-body simulation is extremely helpful for theoretical research in many spheres such as astrophysics, chemistry, and molecular biology, for simulating space missions and predicting possible celestial events that might affect our planet or solar system. In this work, we will mainly focus on parallelizing the naive $O(n^2)$ algorithm using both shared and distributed memory parallelization along with SIMD vectorization, for which we request the computing resources, while also discussing a promising alternative approach for parallelization that involves dividing the spatial domain into octrees and thus aims to reduce the time complexity to logarithmic level.

1 Background and Significance

Considerable research in the natural sciences relies on simulations of N body systems. One such N body problem is modeling gravity (or, more accurately, the interactions of dark matter). This problem is naturally of great interest in physics, owing to both its complex nature and potential usefulness in different applications. Fundamentally, the problem deals with the challenge of calculating, to a great degree of accuracy and as quickly as possible, the positions, velocities, accelerations, and forces experienced by all objects in a universe.

This has many potential applications: being able to understand the complex dynamics of interactions between celestial bodies is quite useful for research in theoretical physics and for supporting astrophysical discoveries. Such knowledge is also crucial for predicting celestial events, which is pivotal for things like space mission planning.

What makes this problem particularly exciting is the fact that it has great potential for spillover effects into numerous other fields. What we treat as celestial objects in a universe could, without too much effort, be adapted to be atoms and molecules in a thermodynamics simulation. Likewise, treating such particles as individuals of a population, where some individuals are infected with a disease, could yield important healthcare insights by modelling the spread of the disease. In fact, this is exactly the technique that was employed by one of the earliest and most heavily cited papers on simulating the spread of Covid-19 [1].

Most existing work on such simulations tends to focus on developing theoretical frameworks for modelling the motion of these 'particles' (which may be anything from an asteroid to humans in a city). For instance, the Oxford study on Covid-19 [1] concentrates mostly on building robust methods for mathematically modelling the spread of Covid. Similarly, Vandenhaute et al. [2], in their acclaimed treatment

of molecular dynamics simulations, largely focus only on capturing the structural transformations of soft porous crystals upon exposure to external stimuli.

Whilst such extensive modelling work is certainly quite important, an equally crucial, and sometimes overlooked, factor for the success of such simulations is their efficiency of implementation. In order to be able to generate reproducible, generalizable results, such simulations needs to be on the order of thousands, if not hundreds of thousands of 'particles'. Trying to run them sequentially would be a disaster, in terms of both the time and financial overhead. Whilst there exists work on parallel implementations, much of it is very specific to the nature of the simulation, relying, for instance, on such things as the chirality of atoms in some treatment mixture. Our hope with this project is to develop a scalable, easy-to-understand pipeline for efficient computation that generalizes well to other N-body problems. Modeling gravity is an ideal task for our purpose. The mathematical computations involved (see section Algorithms and Code Parallelization below) are widely-understood, being based entirely on Newtonian mechanics, and can easily be adapted for a range of other tasks. Moreover, as discussed above, this simulation is also hugely important in its own right. As such, this problem allows us to dive deeper into reaping the fruits of parallelism in a way that it intuitive and flexible for many scientific applications. In keeping with this spirit, we provide a parallel implementation of the most general N-body algorithm, which has a runtime of $O(n^2)$. There has been some work on developing algorithms for simulating gravity with a lower runtime. As a further extension, we also build upon the work by Brandt [3] to consider the Barnes-Hut algorithm, supplemented by Morton's ordering for load imbalance, as a more elegant solution to our specific gravity problem.

2 Scientific Goals and Objectives

The scientific goal is to be able to simulate the motion of n-body celestial objects in a vacuum overtime using real-world physics, and include as many celestial objects as possible in our simulation. Our end goal is to generate a video recording of the states of the universe across different timestamps, where we are able to scale to 100,000+ celestial objects and still record the video in a reasonable time frame, and benchmark the performances of the parallelized and sequential algorithm when simulating different numbers of celestial bodies.

Due to the need to calculate the location and motion of objects in space at each timestamp of the simulation based on each object's gravitational force, acceleration, and velocity, intense floating point calculations on large quantity of data are needed to make the simulation realistic. However, this problem is parallelizable using high performance computing. By dividing the floating point calculations of each of the n celestial objects into c regions, we can achieve a much higher performance. Based on Amdahl's Law, we can expect speedup for p processors as S_p, where f is the fraction of sequential code, and $1 - f$ is the fraction of parallelized code.

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

We expect our fully sequential version of the algorithm to be $O(n^2)$. Our goal is to reduce this runtime to but a fraction using a combination of MPI, OpenMP, and SIMD, to divide the parallelizable portion of the code among different processes to significantly reduce the runtime.

Overall, there are two reasons why we feel this project deserves compute hours on a HPC architecture. Firstly, the nature of the problem being explored (n-body simulation) means that analysis resulting from this project has potential for great impact in many different fields in the sciences. Secondly, the sheer scale and size of the problem means that it could benefit massively from the parallelization afforded by use of such HPC architectures.

3 Algorithms and Code Parallelization

In order to further dive into the parallelized code, we should first understand the mechanics behind the sequential code. Our goal is to simulate the gravitational interaction between N-bodies in an isolated system, where each body is supposed to be a point particle with its position, velocity, and acceleration in three directions (x, y, z). Then, the total force that each body i experiences from other bodies in the system is described by Newton's law of universal gravitation:

$$\vec{F_i} = \sum_{j \neq i} \vec{F_{ij}} = \sum_{j \neq i} \frac{G m_i m_j}{\|\vec{r_j} - \vec{r_i}\|^2} \frac{\vec{r_j} - \vec{r_i}}{\|\vec{r_j} - \vec{r_i}\|}$$

where $\vec{F_{ij}}$ is the force applied by the $j-th$ particle, G is the gravitational constant, m is the mass, and $\vec{r}$ is the position vector. $\frac{G m_i m_j}{\|\vec{r_j} - \vec{r_i}\|^2}$ represents the magnitude of the force, while $\frac{\vec{r_j} - \vec{r_i}}{\|\vec{r_j} - \vec{r_i}\|}$ represents the direction in which that force is applied, as a unit vector pointing from body i to body j. After that, we use Newton's second law of motion to obtain the acceleration that results from gravity:

$$\vec{a} = \frac{\vec{F_i}}{m_i}$$

Since this simply cancels out the factor of m_i in the force calculation, in practice we compute the acceleration directly by taking out the m_i in the force calculation. Finally, we find the change in the velocity and position vectors via numerical integrations:

$$\Delta v = \vec{a} \Delta t$$

$$\Delta pos = \vec{v} \Delta t$$

Δt represents the length of a single time step in the simulation and is set to a small number (in our experiments, we used 0.001). The updated positions and velocities of all the objects are taken as the new state of the universe, and this procedure of calculating accelerations and integrating to get velocities and positions is repeated at each time step over the course of the simulation.

The naive way of implementing this simulation is to calculate the interaction for each particle by iterating through the list of other bodies, and it gives us the $O(n^2)$ time complexity. Our team utilizes the distributed memory parallelization that would reduce it to $O(\frac{n^2}{p})$, where p is the number of processes involved in the calculations.

We divide the parallelization process, which was implemented using MPI, into these steps:

1. We have a root process that reads the initial state of all n bodies from a special text file and creates the universe by assigning each body its initial state, consisting of its mass, radius, position, and velocity. Note that we randomly generate the initial states by a Python script, which allows us to test the robustness of our simulation given all possible positionings of the objects.

2. The root process then sends the created list (universe) of n objects to all other ranks.

3. At this point, all ranks identify the range of objects that they are responsible for and update their positions and velocities by calculating the corresponding acceleration vectors that arise from the gravitational influence of all other bodies. In a simulation with p processes, each rank is assigned approximately n/p objects on which to compute updates.

4. All ranks send the updates performed on their assigned set of $\approx n/p$ objects to the root process.

5. Root process collects the updated bodies from all other processes.

6. Repeat the steps 2-5 for the given length of the simulation.

Calculating the force requires a huge number of floating-point operations. Moreover, our goal is to perform the simulation for at least 100000 objects, where each object has a mass, radius, and acceleration, velocity, and position vectors in three directions - all in the double data type. It means that we need 88 bytes per object in our memory, thus, the total memory capacity we would need is 8800000 bytes at minimum, which is more than 8 MB.

One might notice from the parallelization process described earlier that at each time step there is an abundance of communication between the root process and all other processes to get the most updated state of all objects before and after calculations. However, this excessive communication causes overhead and negatively affects the scalability of the program. Hence, to mitigate this issue and reduce the latency, our team decided to use non-blocking MPI communication methods. When the root process sends objects' states to other ranks, it does so through non-blocking sends, so that it can start computing the updates for its assigned set of objects before all the sends have completed. Similarly, the root process posts non-blocking receives that request updates from each of the other ranks, in order to initiate communication from whichever rank finishes first.

We also used OpenMP to divide the work within a process among multiple threads. In the same way that the full set of objects can be distributed across multiple MPI ranks, the subset of objects for which a single rank is responsible can also be assigned to multiple threads within that rank.

Finally, we took advantage of SIMD to parallelize vector operations, such as vector additions and subtractions, scalar multiplications, and dot products, since working in (x, y, z) coordinates requires many such operations. Additions are needed to total all accelerations, subtractions are needed to calculate the difference between two positions, scalar multiplications are needed to take masses into account, and dot products are needed to calculate vector magnitude.

SIMD vector operations are performed by storing the double-precision (x, y, z) coordinates in 256-bit AVX registers. Since each such register holds 4 double-precision values, the fourth lane was left unused and filled with 0 as a placeholder. In the case of scalar multiplications, the scalar value by which to multiply each coordinate of the vector is repeated in all four lanes of an AVX register. For dot products, the coordinate-wise multiplications are first performed in parallel, and then the accumulation of the products is performed using horizontal additions.

Libraries and Tools

We used CImg and ffmpeg to visualize the results of the simulation. CImg is a library that was used in the code to draw the positions of the objects on an image and output the result as pixel data. ffmpeg is a program that can be used to compile the image pixels generated by CImg into a video.

Alternatives

Apart from the naive $O(n^2)$ algorithm, we have explored many other potential ways to optimize the n-body simulation. Now, we would like to focus on one of the prominent methods - the Barnes-Hut simulation, which would reduce the time complexity to $O(\frac{n}{p}log(\frac{n}{p})$, where n is the total number of bodies and p is the number of processes.

The key idea here is to divide the simulation space into octrees, such that each node is dedicated to a specific region (North, North-West, North-East and so on). We would keep building octrees recursively until all bodies are assigned to leaf nodes. The tree is structured so that each internal node carries the

total center of mass and other coefficients that describe the collection of its children nodes, while every leaf node contains the mass and position of a single body. The process of building an octree is well depicted on Figure 1.

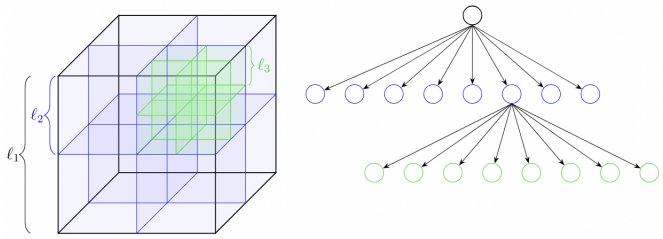


Figure 1: Example of dividing a space into 8 regions recursively[3].

Then, to find a force executed on object i, we could just iterate the tree in a top-to-bottom manner. As we know, each internal node represents a region (cell), so for each node we visit, we would estimate the the value $\frac{l}{d}$, where l is the side length of the cell and d is the distance between the center of mass of the cell and object i. If the resulting value is less than a parameter θ (Multiple Acceptance Criterion), then we can consider the object i and the cell as far enough, hence we can just approximate the gravitational interaction between them instead of calculating it for each body-body pair in the region covered by the node. Otherwise, we need to open the node, and further iterate downwards until we can do the approximation or we are forced to do the body-body calculations (in case we reach the leaf nodes) [3]. So, this method allows us to avoid n^2 calculations, and approximate the gravitational force of whole regions on a single body, which reduces the number of required calculations to $O(\log(N))$.

One way to parallelize the Barnes-Hut method is to use MPI libraries and assign $\frac{n}{p}$ bodies to p processes such that each process builds its own octree. After that, all processes merge their local octrees in a root process, which then distributes the global octree to everyone. Finally, every process calculates the executed force and updates velocity and position for only its own assigned objects by traversing the global octree in parallel. At the first glance, the algorithm sounds straightforward, nevertheless, there are several crucial caveats. First of them is the way we assign bodies to each process. If we just equally divide regions and give them to each process, due to the dynamic character of our simulation space, it is possible that some processes will get regions with few or no bodies at all, while others might get very densely populated areas. This leads to a huge work-imbalance since several processes now have to perform almost all of the required force calculations. In order to prevent this scenario, our team has considered Morton's ordering algorithm that converts coordinates of each process into an integer and sorts them. This linear ordering creates a space-filling curve and ensures that processes get bodies that are located nearby to each other. Only thing left to do is partition the sorted list such that each process is responsible for exactly $\frac{n}{p}$ bodies. Figure 2 illustrates Morton's ordering in a 2-dimensional space.

Last important point to discuss about parallelizing Barnes-Hut method is merging local octrees to get one global octree. The naive approach would be that each process adds its own local octree into the global tree one by one; however, this could possibly cause large communication overhead. To prevent it, our team focused on performing a pairwise reduction of trees, and in the end, root process will have the global octree, which it will broadcast to everyone.

The main source of the Barnes-Hut idea and its optimizations is the paper on N-body simulations by Brandt [3], and our team utilized its core parallelizing techniques for this approach. As we have worked on this method for the good half of the semester, our team has made decent progress: we have developed the octree structure, Morton's ordering, and parallelized the force calculation process. However, the merging process of local octrees took longer time than we initially expected, which made us switch to the $O(n^2)$

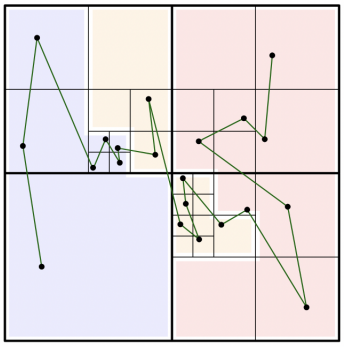


Figure 2: Bodies in 2D are connected with a space-filling curve[3].

algorithm due to the time restrictions. However, as a team we are willing to further continue developing the first method as we believe that it encompasses significant potential optimizing the N-body simulations.

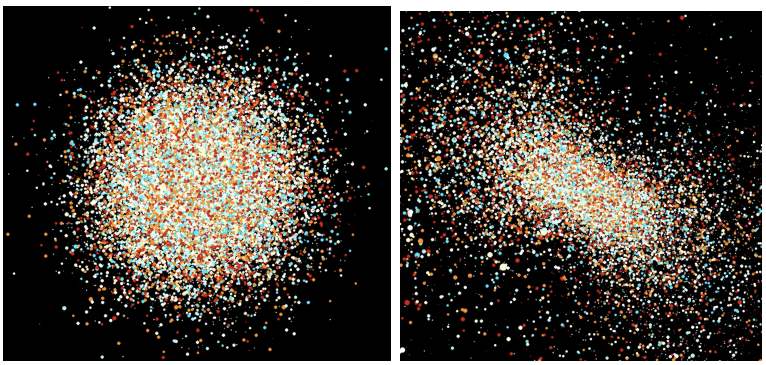
Validation, Verification

Since our simulation purports to represent a phenomenon (gravitation) that abides by the laws of physics, we validate our simulation by testing that it does in fact follow other physical laws that we know to be true of the world but that were not explicitly modeled. Specifically, we test that the law of conservation of momentum holds in our simulated universe. Momentum is a vector quantity that is equal to mass times velocity. Though the velocity (and thus momentum) of an object will change at every time step, the total momentum across all objects must remain constant at all times.

We tested that momentum is conserved by randomly generating an initial state of the universe and then running the simulation for 30,000 time steps. In our random initial state, the total momentum was calculated to be $(-392839, 553118, -732516)$. During the simulation, the total momentum was continually recalculated, and each time it was found to be equal to the initial value. Thus, the simulation conserves momentum, providing evidence that the simulation correctly models the physical process of gravitation.

4 Performance Benchmarks and Scaling Analysis

Below are all of our recorded data for bench-marking and analysis. We ran our simulation with each of the following object count: n = 256, 1024, 4096, 16384, 65536, and 100000. For each object count group, we ran simulations on different combinations of nodes and tasks per node on the cluster.



Simulation video snapshot at timestamp 0 (left) and timestamp 30000 (right), for n = 65536.

# nodes	# tasks	# cpus	runtime (secs) for n = 256	1024	4096	16384	65536	100000
1	1	1	2.84809	44.3785	591.24	9633.03	110793.89	N/A
1	4	1	1.43926	22.5374	386.179	6340.81	N/A	N/A
1	8	1	0.797374	12.2905	183.969	3013.06	N/A	N/A
2	8	1	1.34829	7.9695	102.065	1502.19	N/A	N/A
3	8	1	1.81154	7.34421	74.0812	1043.59	N/A	N/A
4	8	1	1.96253	8.13751	66.5432	818.66	N/A	N/A
5	8	1	2.43752	9.84152	77.4443	680.816	N/A	N/A
6	8	1	2.60329	17.1898	77.4291	616.913	N/A	N/A
7	8	1	3.20337	17.5231	98.4151	565.497	N/A	N/A
8	8	1	3.62409	15.9157	108.191	537.856	7660.59	17191.1

Table 1: Scaling data for n ranging from n = 256 objects to n = 100000 objects.

	n = 256	1024	4096	16384	65536	100000
Typical wall clock time (hours)	0.00022	0.002	0.018	0.149	2.128	4.775
Typical job size (nodes)	1	3	4	8	8	8
Memory per node (KB)	22.5	30.0	90.1	180,2	720.9	1100
Maximum number of input files in a job	1	1	1	1	1	1
Maximum number of output files in a job	1	1	1	1	1	1
Library used for I/O	ffmpeg	ffmpeg	ffmpeg	ffmpeg	ffmpeg	ffmpeg

Table 2: Workflow parameters of the six test cases used during project development.

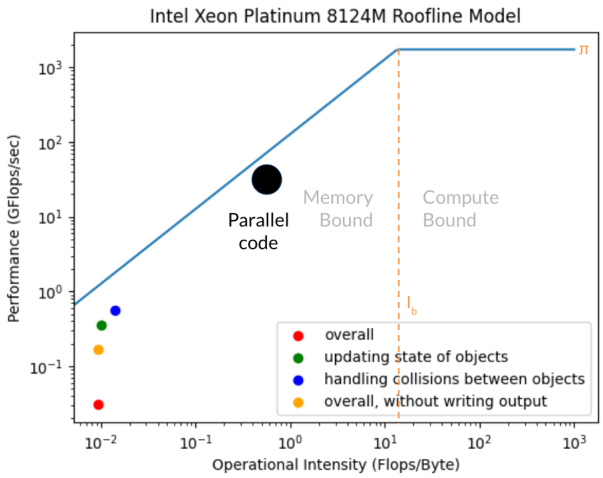


Figure 3: Roofline model.

Above we have graphed the roofline model for the cluster machine. The roofline model (figure 3) indicates a peak floating point performance of 1728 Gflops/s, with a peak memory bandwidth of 128 GB/s, and a ridge point = 13.5 Flops/byte. The parallel code, with 18 ranks and SIMD, takes 115762920000 flops,

1193056266984 bytes, and 4.23157 seconds. This results in an operational intensity of 0.776 flops/byte and 27.356 Gflops/second.

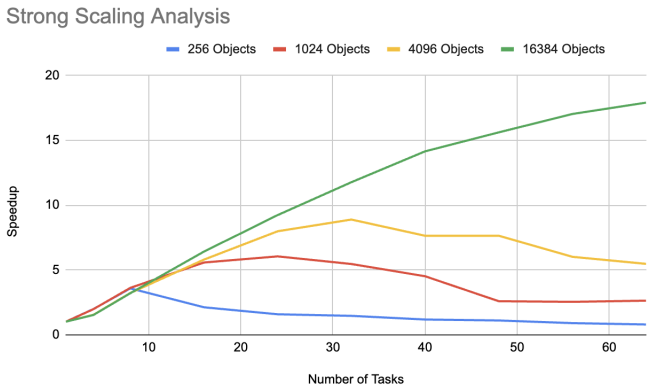


Figure 4: Strong scaling analysis of performance

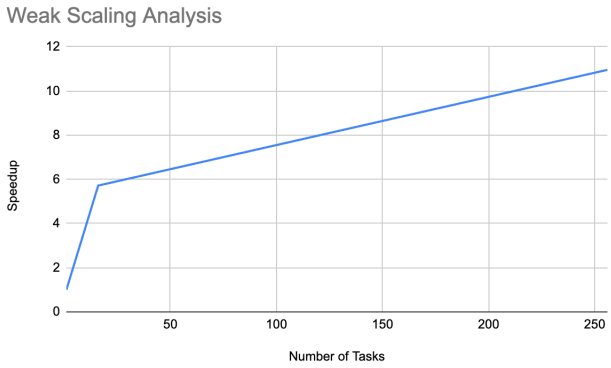


Figure 5: Weak scaling analysis of performance

A strong scaling analysis reveals that parallelization (8 nodes with 8 tasks per node) offers significant speedup gains. The speedup is most noticeable for larger simulations, with 16384 objects having a speedup of more than 18 times. For smaller simulations (such as 256 objects), increasing the number of tasks deteriorates speedup gains, as the amount of objects does not outweigh the overhead created by the parallelization. For weak scaling, we similarly see that although we do get performance gains from increasing the number of tasks, the communication overhead limits the gains as the problem size increases.

Our parallelized code runs orders of magnitude faster than our sequential code. For 16384 objects, there is a 18 times speedup, and a 14.5 times speedup for 65536 objects. For 100,000 objects, our sequential code timed out after 6 hours, whereas our parallelized code ran in 4.78 hours.

5 Resource Justification

Our benchmark resource use results are as follows:

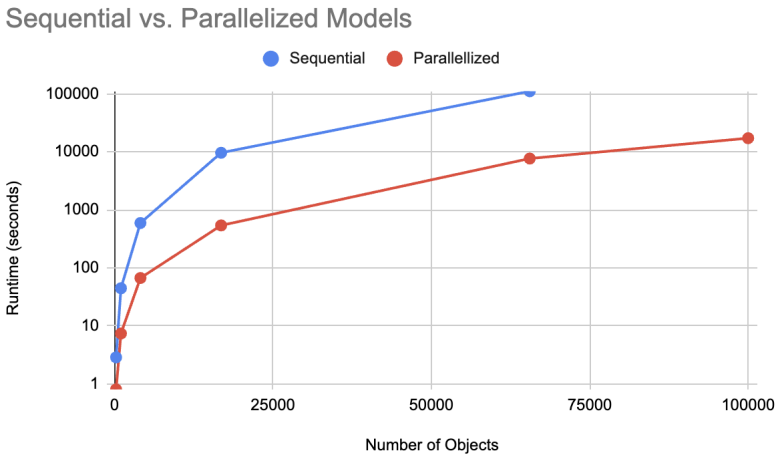


Figure 6: Runtime vs. number of objects in log-linear scale

objects count	256	1024	4096	16384	65536	100000
Trials per simulation	2	2	2	2	2	2
Simulations per task	10	10	10	10	2	2
Iterations per simulation	30000	30000	30000	30000	30000	30000
avg. runtime (s) per simulation	2.21	16.30	176.56	2475.24	7660.59	17191.1
avg. runtime (s) per iteration	0.000074	0.00054	0.0059	0.083	0.26	0.57
node hours per iteration	0.00000016	0.0000012	0.000013	0.00018	0.00058	0.0013
Total node hours	0.10	0.72	7.80	108	69.6	156

Table 3: Justification of the resource request

For 256 objects, the optimal job size of our benchmark is 8 nodes. Given 0.000074s average optimal runtime per iteration, the node hours is calculated as the following:

$$0.00000016 \text{ node hours per iteration} = 8 \text{ nodes} \times \frac{0.000074s}{3600\frac{s}{\text{hour}}}$$

$$0.096 \text{ total node hours} = 2 * 10 * 30000 * 0.00000016$$

For 1024 objects:

$$0.0000012 \text{ node hours per iteration} = 8 \text{ nodes} \times \frac{0.00054s}{3600\frac{s}{\text{hour}}}$$

$$0.72 \text{ total node hours} = 2 * 10 * 30000 * 0.0000012$$

For 4096 objects:

$$0.000013 \text{ node hours per iteration} = 8 \text{ nodes} \times \frac{0.0059s}{3600\frac{s}{\text{hour}}}$$

$$7.80 \text{ total node hours} = 2 * 10 * 30000 * 0.000013$$

For 16384 objects:

$$0.00018 \text{ node hours per iteration} = 8 \text{ nodes} \times \frac{0.083s}{3600\frac{s}{\text{hour}}}$$

$$108 \text{ total node hours} = 2 * 10 * 30000 * 0.00018$$

For 65536 objects:

$$0.00058 \text{ node hours per iteration} = 8 \text{ nodes} \times \frac{0.26s}{3600\frac{s}{\text{hour}}}$$

$$69.6 \text{ total node hours} = 2 * 2 * 30000 * 0.00058$$

For 100000 objects:

$$0.0013 \text{ node hours per iteration} = 8 \text{ nodes} \times \frac{0.57s}{3600\frac{s}{\text{hour}}}$$

$$156 \text{ total node hours} = 2 * 2 * 30000 * 0.0013$$

Our benchmark is the wall time it takes, in seconds, to generate each simulation result. For each group in objects count = 256, 1024, 4096, and 16384, we ran 10 simulations (with different nodes count and tasks per node count), and 2 trials per simulation per objects group. Each simulation includes 30000 timestamps, and the wall time starts from the beginning of the first timestamp and ends after recording the result of the 30000th timestamp. For group in objects count = 65536 and 100000, only two simulations are recorded, as we realized non-optimal simulations took too long to simulate 30000 steps and ultimately timed out on the cluster. Together all of our benchmarks require 342.2 total node hours on the cluster to cover all test cases.

Group Member Contributions

Aditi: Added SIMD to vector operations; designed performance benchmarks and visualizations.
Dennis: Implemented the sequential algorithm for gravity simulation and added MPI parallelism.
Jared: Implemented tree for Barnes-Hut; worked on paper; recorded benchmarks and video.
Rafay: Implemented Morton's ordering and handled load imbalance for Barnes-Hut, designed ways to test correctness of code, conducted literature review
Sezim: Planned and led the development of the alternative Barnes-Hut method; explored the ways to implement the shared memory parallelism of the quadratic approach.

References

[1] Bo Peng, Rowland W Pettit, Christopher I Amos, Population simulations of COVID-19 outbreaks provide tools for risk assessment and continuity planning, JAMIA Open, Volume 4, Issue 3, July 2021, ooaa074, https://doi.org/10.1093/jamiaopen/ooaa074

[2] Sander Vandenhaute, Sven M. J. Rogge and Veronique Van Speybroeck. Large-Scale Molecular Dynamics Simulations Reveal New Insights Into the Phase Transition Mechanisms in MIL-53(Al), Frontiers in Chemistry, Volume 9, 2021, https://www.frontiersin.org/articles/10.3389/fchem.2021.718920/full

[3] Alexander Brandt, On Distributed Gravitational N-Body Simulations, arXiv (Cornell University), 2022, https://doi.org/10.48550/arxiv.2203.08966.