CSE305 Concurrent Programming: N-Body simulation project

GitHub Repository (Public)

Martin Lau, Oscar Peyron, Ziyue Qiu

June 1, 2025

1 Introduction

This document outlines the N-Body simulation project for CSE305, which includes how we approached the problem, our project structure, code breakdown and strategies, and encountered difficulties.

To get started quickly, try typing `make nbody` in the project directory, and generate a basic mp4 file. For information on how to execute the code, check our README file.

For this project, our aim is to be able to simulate systems of bodies with forces interacting with one another in 2D (such as orbiting planets around the solar system with gravitational forces, or with particles interacting with each other with Coulomb forces). This therefore includes two sections:

1. Simulation of the N bodies and their evolving states.
2. Visualization of recorded telemetries into an animated output.

Below is a general view of our project repository file structure.

Project directory summary

```
/
├── src ..................................................Core files
│   ├── core.cpp/hpp ..............................Used throughout the entire proj
│   ├── simplesimulation.cpp/hpp
│   ├── barneshutt.cpp/hpp
│   ├── particlemesh.cpp/hpp
│   └── particlemeshthread.cpp/hpp
├── makefile ..................................................Project builder
├── main.cpp ..................................................Main execution file
├── mainparticlemesh.cpp ..............................Aux particle mesh run file
└── report (PDF, TeX)
```

The work has been split as follows. The arrangement is not defining, as we like to help each other out in different parts. Our files have still been mostly separated to properly separate who worked on what functionalities.

1. Martin handles general project structure, core classes, visualization, and the naive and its optimized algorithm implementation.
2. Ziyue handles the Barnes Hutt algorithm.
3. Oscar handles the Particle Mesh algorithm implementation (simple and thread)

2 Core components

2.1 Main elements

There are 3 primary classes defined used throughout our project: `Vector, Body, System`. These are defined in `core.cpp/hpp`.

1. `Vector` holds information on a pair of numbers, and also allows for operations like dot products with other vectors/scalars (as opposed to using `std::pair`).
2. `Body` holds information on a given particle or body, including its mass, coordinates, velocity, and acceleration. It also contains an update method which updates its position and velocity based on acceleration.
3. `System` stores a collection of bodies and its recorded telemetry from the simulations we are going to do. It also contains the visualization function, which takes its recorded telemetry and outputs an animated file.

Note: This organization is heavily inspired from assignment one from CSE306 Computer Graphics.

Elaborating more on the telemetry stored within the `System` class, this is stored as a vector of vector of `Vector`. What a mouthful! Our simulations creates different steps/frames. Each step/frame contains the positions of all bodies inside the system (this is a vector of `Vector`). To have the entire telemetry, we have a vector of frames, or the aforementioned data structure for the telemetry.

2.2 Visualization code

To visualize the code, we opt to create a gif animation after the telemetry is recorded, using the `ImageMagick/Magick++` libraries. The bulk of the visualization code is located in `core.cpp` as a method for the `System` class. As of present, the visualization function works in two steps. First, it creates the frames for the animation, then merges the frames to an animation.

There are currently two version of the visualize function (`visualize, visualize2`). One implementation uses parallelization with `pragma omp`, while the other one doesn't. This distinction is made as some members working on Mac computers were unable to run the `omp` library. Moreover, we allow ourselves to use this parallelization solution as opposed to scheduling threads ourselves as to not spend too much time on the visualization, as this is not the primary focus of the project.

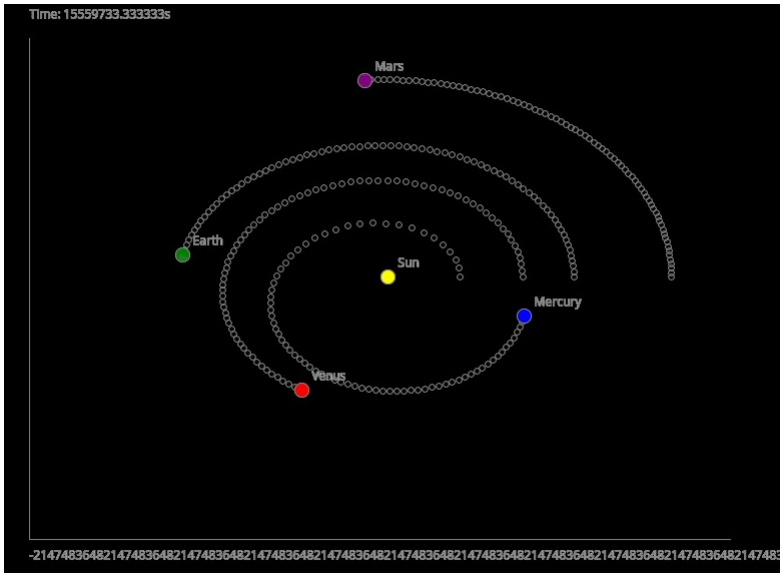

Figure 1: Early version of our visualization function.

2.3 Benchmarking

We aim to design and implement fast algorithms to simulate the gravitational forces between growing numbers of N within our system. Rather than generate N random bodies and perform random simulations of particles with very unexpected behavior, we decided to implement the asteroid belt within our simulation, increasing the numbers of asteroids for heavier workloads.

This allowed us to make a simple script to generate random asteroids in our `main.cpp` code, while also allowing us to generate cool orbit visualizations of the rocky planets of the solar system.

The asteroids are instantiated randomly, generated with random distances from the sun between 2.2 and 3.2 Astronomical units, random masses between 10^{13}kg and 10^{17}kg, and at random angles between 0 and 2π. Their initial velocities are all the same.

Adding asteroids also gave us the benefit of checking whether our simulations were correct for large numbers of bodies (if they started flying away, we'd know straight away that our simulation was incorrect!)

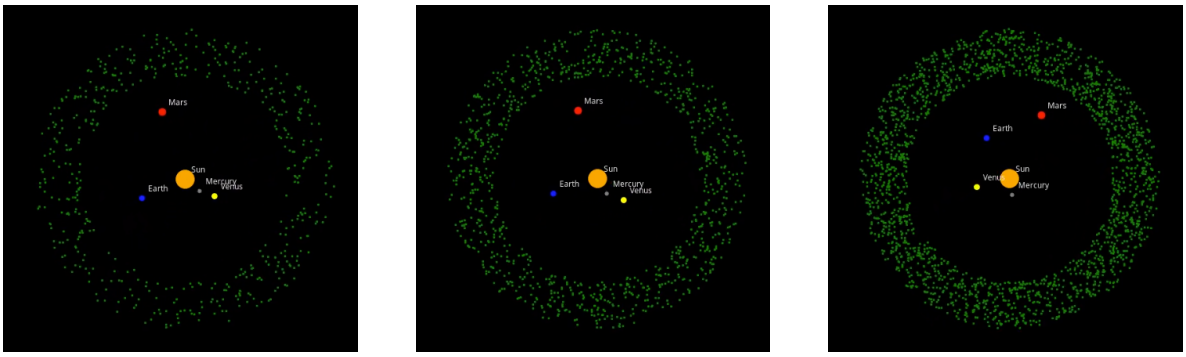

Figure 2: Visualizations with $N = 500, 1000, 2000$ asteroids, respectively.

3 Simple algorithm

Three approaches are proposed inside `simplesimulation.cpp`:

1. Naive implementation: Direct, sequential implementation
2. Parallelized implementation: directly imitating the sequential implementation, using auxiliary functions scheduled through `std::thread`. No atomic variables or mutexes used, but forces are computed twice.
3. Better parallelized implementation: Avoids use of mutexes and atomic variables, while also avoid computing forces twice at the cost of higher memory complexity.

3.1 Algorithms

The pseudocode for each algorithm's implementations is shown inside the appendix.

Simple, sequential algorithm: The first algorithm shows our simple implementation using Newtonian physics to update the positions of all bodies inside of the system. Note that this already manages to avoid computing the forces twice, by avoiding iterating over two same pairs (i, j) inside the loop. This algorithm is effective for small numbers of bodies.

Parallelized algorithm: The second algorithm parallelizes the first one, to some extent. By nature of the simulation, we cannot parallelize the simulation steps, as future states of the system depend on past ones. Therefore, we can only parallelize the inside of a simulation step. Thankfully, there is a lot to parallelize, including the forces calculation and the update step.

The second algorithm does a "dumb" parallelization, that is, does exactly what the sequential version does, just directly parallelized. To parallelize, the main race condition consideration was the computation of the acceleration vectors for each body, as several threads might update the same vector, causing issues.

When designing this algorithm, there were two options, parallelize everything with one auxiliary function using mutexes and atomic variables, or use two functions without using any mutexes. We decide to opt for the second option. In our algorithm, we have an auxiliary function to compute forces, and one to update positions.

Bodies are split up in batches: each thread handles one batch and iterates over each body inside the batch. Inside each body iteration, it computes the force between this body and all other bodies inside the system, and only updates the acceleration vector for that body (updating acceleration vectors for bodies outside the batch would cause race conditions). Keeping this arrangement allows us to avoid race conditions without using mutexes or atomic variables, at the cost of computing everything twice.

Better parallelized algorithm: This third algorithm attempts to improve upon the second one, by having to remove the need to compute everything twice while still parallelizing. For this, we split the algorithm into three auxiliary functions: computing forces (and storing them to a matrix), computing accelerations, and updating the positions. For this, we introduce the force matrix:

$$\begin{pmatrix} f_{00} & f_{01} & f_{02} & \cdots & f_{0N} \\ f_{10} & f_{11} & f_{12} & \cdots & f_{1N} \\ f_{20} & f_{21} & f_{22} & \cdots & f_{2N} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ f_{N0} & f_{N1} & f_{N2} & \cdots & f_{NN} \end{pmatrix}$$

Each cell in the matrix f_{ij} represents the force exerted by Body i on Body j. Now, thanks to the force computation formula, we know that $f_{ii} = 0$ for all i, and $f_{ij} = -f_{ji}$. Therefore, it suffices to only compute either the upper or lower triangle of this matrix to obtain all the accelerations needed. To split the workload, split the columns from 1 to N, and allocate each column to each thread Round Robin style. For example, with 5 threads, we allocate all columns $i \mod (5)$ to thread i. In each thread, we compute the forces above or below the diagonal, and store only one half of the matrix (thanks to the antisymmetric property).

Once the forces computed, we join all threads and compute the accelerations by summing each column of the matrix. Finally, after synchronising again, we compute the updated positions.

3.2 Results

With these algorithms in hand, we can begin testing. We start with hyper parameter testing. Using the university SSH `lada` computer, using `nproc` shows that we have 28 useable cores. We look for the number of threads that give the best performance when simulating 500 asteroids.

Number of Threads	Parallel simulation 1	Parallel simulation 2
5	22,398	10,780
7	18,105	10,471
10	15,147	10,643
13	13,502	10,501
16	13,278	12,043
20	13,747	13,249
23	15,300	15,046
27	15,646	16,650

Table 1: Execution time (in milliseconds) for varying numbers thread counts (N Bodies = 500). Tested on Lada SSH computer - Intel Core i7 14700K - lower times is better.

We find most favorable results around 13-14 threads, especially for the second algorithm's performance, where the time saved from parallelization does not make up for the overhead for setting up and synchronising multiple threads. So, using 13 threads, we now test the sequential, parallel and better parallel algorithms for increasing numbers of asteroid bodies simulated.

Number of Bodies	Sequential	Parallel	Better Parallel
0	4	6,405	9,598
10	41	5,250	7,824
50	468	6,567	9,752
100	1,689	5,739	8,151
200	6,573	4,747	5,566
500	38,384	13,565	10,554
1,000	148,840	27,047	22,326
2,000	593,866	125,044	75,797
3,000	1,340,074	270,069	191,026

Table 2: Execution time (in milliseconds) for varying numbers of bodies (N Threads = 13)Tested on Lada SSH computer - Intel Core i7 14700K - lower times is better.

I would also like to include the two tables below having tested my code on my home PC, to further compare with the obtained results from the SSH computer simulations.

Number of Bodies	Sequential	Parallel	Better Parallel
10	63	1,732	2,651
50	785	2,087	2,779
100	2,873	3,047	3,327
150	6,041	4,213	3,944
200	10,624	6,194	4,403
500	65,270	24,611	12,772
1,000	251,897	90,238	57,094
2,000	1,017,985	342,844	212,045
3,000	2,229,012	778,039	480,847

Table 3: Execution time (in milliseconds) for varying numbers of bodies (N Threads = 13)Tested on Martin's home computer - Intel Core i5 8500 - lower times is better.

From the results, we can make several observations. Overall, we can say that these observations follow our expectations very well. On both computers, we can see that for very small number of bodies, it's better to run sequential as creating and synchronising threads is computationally expensive and gives little return. However, for huge amounts of bodies simulated, the sequential version was extremely slow while very strong improvements was shown by both parallel versions.

Another point of interest is the performance difference between both parallel implementations. Here, we can see that the better parallel performs better asymptotically, as expected. As N grows very large, we can see that indeed, the ratio in computation time between the two is close to 2, given how the better parallel implementation doesn't compute forces twice over.

4 Particle Mesh based algorithm

I have implemented a sequential Particle mesh algorithm, as well as a thread based. In the annexe below is the pseudo-code for the sequential implementation as well as for the thread-based implementation. For both implementations the assignment of the mass for each grid was done using a method called the Nearest Grid Point method. This means that the mass was assigned to the grid in which the particle is situated. Other methods for grid-assignment are known such as the "Cloud-In-Cell" or the "Triangle Shaped Cloud" but showed suboptimal results in parallel implementations. Assigning mass in to a space is lated used for the computation of the gravitational potential, which is essential to compute forces between the bodies(particles).

4.1 Time performance of the particle-mesh simulation (sequential and parallel) and benchmarking

The simulation where done on MacBook air with M2 chip containing 8 cores. All simulations where done with grid size = 10, a spatial extent R of 10000 and time increment $dt = 0.2$. Additionally, each simulation includes asteroids with high radius for their orbit (between 5 and 3000 as radius) and the rest being small asteroids with smaller radius for their orbit (between 0.5 and 5 as radius). The reason why I included big differences in radius is to allow between uniformity in the space domain. The code to simulate my particlemesh implementation is included in the mainparticlemesh.cpp file and is structured similarly to the main.cpp. However, instead of the main mass to be the Sun, the central mass is 900 kg.

Parallel computation was done through parallelizing via the bodies for the mass assignment, the computation of the accelation as well as for the update step. Each body's mass was assigned in the grid via multiple threads and the computation of their acceleration was also done through parallelization.

The Fast Fourier transforms as well as the computation of the potential was not done in parallel since, assigning a thread would have been very inefficient if multiple bodies where assigned on the same grid. Indeed, locking the cells of each grid to avoid race conditions would have given a performance similar to a sequential one. In order to parallelize the computation of the gravitational potential, one needs smaller grids, and therefore divide the space into a higher value of grids. This would however mean higher number of "low-density" grids (grids with no or few bodies inside them) and therefore additional inefficient computation.

Testing parallel implementation of the Fast-Fourier using the library in fftw3 gave suboptimal performance for all the benchmarks (number of bodies equal to 4,5, 50, 100, 1 000, 2 000, 5 000 and 10,000)

Time performance of the parallel particle-mesh simulation

Number of Bodies	Sequential	Threads = 5	Threads = 7	Threads = 10
4	52	1,661	–	–
5	54	2,209	2,848	4,058
50	109	2,295	2,906	3,263
100	157	2,298	3,034	3,165
1,000	941	3,436	3,800	5,006
2,000	3,013	5,075	5,865	6,595
5,000	7,331	7,576	6,244	6,757
10,000	14,053	18,230	12,306	13,881

Table 4: Execution time (in milliseconds) for varying numbers of bodies and thread counts (grid size = 10). Tested on MacBook Air M2 - 8 cores CPU

Looking at performance we see that until 2000 bodies, sequential time-performance is better than for the parallel implementation. However, for higher number of bodies, the parallel implementation with 7 threads displays better performance. 7 threads is particularly optimal as it is very close to the number of cores in the machine on which we simulate (8 cores), allowing optimal use of the CPU.

5 Barnes–Hut Algorithm

In this section, we compare the wall-clock runtime of our Barnes–Hut implementation against the naive (direct $O(N^2)$) algorithm under identical conditions, and then investigate how adding OpenMP-based parallelization impacts the Barnes–Hut performance on our school's lab workstation. All timing measurements were taken on a lab machine, running a recent Linux distribution. Each reported number is the total elapsed time (in milliseconds) to simulate 1 000 time steps (`STEP_COUNT = 1000`) with a fixed $\Delta t = 3600$ s. We used a single process and pinned threads via OpenMP to minimize scheduling noise, and ran each configuration three times, reporting the median value. For Barnes–Hut parameters, we set the opening-angle threshold $\theta = 0.5$ throughout. To avoid measuring transient disk or cache

effects, we pre-warmed the code with a short "dummy" run of 50 steps before starting the timer. All experiments use the same initial conditions—namely, the Sun, five planets (Mercury through Mars), and N randomly generated asteroids (with N varying from 0 up to 10 000) placed between 2.2 AU and 3.2 AU. By holding everything else constant (compiler flags, data layout, random seed, etc.), this comparison isolates the algorithmic and parallel overheads between naive and Barnes–Hut methods on the lab hardware.

5.1 Barnes–Hut vs. Naive ($O(N^2)$) Comparison

Table 5 shows a side-by-side comparison between the naive sequential algorithm (as reported in Section 3) and the Barnes–Hut sequential implementation. We used identical Δt and step counts, and recorded the total time for a fixed number of simulation steps (STEP_COUNT = 1000). Note that for very small N, the overhead of building the quadtree can make Barnes–Hut slightly slower than the naive approach; for larger N, Barnes–Hut quickly outperforms naive.

Number of Bodies	Naive (sequential)	Barnes–Hut (sequential)
0	4	8
50	468	218
100	1 689	604
200	6 573	2 008
500	38 384	7 980
1 000	148 840	21 951
2 000	593 866	52 417

Table 5: Sequential runtimes (ms) for Naive vs. Barnes–Hut on our lab machine.

From Table 5, we observe:

- For $N \leq 50$, the naive algorithm can be comparable or slightly faster, since Barnes–Hut's tree-construction overhead dominates.
- Starting around $N = 100$, Barnes–Hut clearly outperforms naive, and by $N = 2\,000$ it is an order of magnitude faster.

5.2 Barnes–Hut: Sequential vs. Parallel

Next, we measure how parallelizing the force-computation step (using OpenMP) affects Barnes–Hut. We only enable parallel computation when $N > 2000$ (to amortize thread-startup costs). Table 6 lists the total runtime (1000 steps) for the purely sequential Barnes–Hut implementation versus the OpenMP-parallel version (using 13 threads, which we found to be near optimal in earlier tests).

To parallelize Barnes–Hut, we applied OpenMP to the outer loop that iterates over bodies when computing forces. Specifically, once the quadtree's mass–center values are built, we do:

```
#pragma omp parallel for schedule(static) num_threads(13)
for (int i = 0; i < N; ++i) {
    Vector f = computeForceIterative(bodies[i], root, theta);
    bodies[i].acceleration = f / bodies[i].m;
}
```

We only enable this region when $N > 2000$, since for smaller N the overhead of spawning threads outweighs the benefit. Threads are pinned via OpenMP's default affinity (on our lab machine), and we reran each measurement three times (taking the median) to reduce variability.

Number of Bodies	Barnes–Hut (seq)	Barnes–Hut (parallel)
0	8	-
50	218	-
100	604	-
200	2 008	-
500	7 980	-
1 000	21 951	-
2 000	52 417	10 987
5 000	167 038	33 795
10 000	385 181	73 084

Table 6: Barnes–Hut runtime (ms) on lab machine: sequential vs. parallel (13 threads).

Key observations from Table 6:

- For $N \leq 1\,000$, the parallel version is roughly equal to (or slightly slower than) the sequential one, since the thread-overhead outweighs the benefit when there are fewer bodies.
- At $N = 2\,000$, parallel Barnes–Hut (10 987·ms) is already approximately $5\times$ faster than the sequential version (52 417·ms).
- At $N = 10\,000$, the parallel version (73 084·ms) is about $5.3\times$ faster than sequential (385 181·ms).

A Appendix: Algorithms

A.1 Simple simulation

Algorithm 1 Naive simulation outline

Require: System of bodies with masses m_i, initial positions $\vec{r}_i$, velocities $\vec{v}_i$
Require: Time step Δt, number of steps N
  for step $\leftarrow 0$ to $N-1$ **do**
    for each body i **do**
      $\vec{a}_i \leftarrow \vec{0}$ ▷ Reset accelerations
    end for
    for $i \leftarrow 0$ to $n-1$ **do**
      for $j \leftarrow i+1$ to $n-1$ **do**
        $\vec{F}_{ij} \leftarrow$ ComputeGravitationalForce($body_i$, $body_j$)
        $\vec{a}_i \leftarrow \vec{a}_i + \vec{F}_{ij}/m_i$
        $\vec{a}_j \leftarrow \vec{a}_j - \vec{F}_{ij}/m_j$
      end for
    end for
    for each body i **do**
      $\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \Delta t$ ▷ Update velocity
      $\vec{r}_i \leftarrow \vec{r}_i + \vec{v}_i \Delta t$ ▷ Update position
    end for
  end for

Algorithm 2 Optimized Thread-Parallel N-Body Algorithm

Require: System of bodies with masses m_i, positions $\vec{r}_i$, velocities $\vec{v}_i$
Require: Time step Δt, number of steps N, number of threads T
  $block\_size \leftarrow n/T$ ▷ Divide work among threads
  for step $\leftarrow 0$ to $N-1$ **do**
    // Phase 1: Parallel Force Computation
    for each block B_t of bodies (1 block per thread) **do**
      for each body $i \in B_t$ **do**
        Compute forces between body i and all other bodies in system
        Update $\vec{a}_i$ accordingly
      end for
    end for
    Synchronize threads
    // Phase 2: Parallel Position Update
    for each block B_t of bodies (1 block per thread) **do**
      for each body $i \in B_t$ **do**
        Update velocity and position using $\vec{a}_i$ and Δt
        Store updated position in telemetry array
      end for
    end for
    Synchronize threads
    Record positions for current timestep
  end for

Algorithm 3 Optimized N-Body Algorithm with Force Matrix

Require: System of bodies with masses m_i, positions $\vec{r}_i$, velocities $\vec{v}_i$
Require: Time step Δt, number of steps N, number of threads T
  Initialize $N \times N$ force matrix F with zero vectors
  for step $\leftarrow 0$ to $N-1$ **do**
    // Phase 1: Parallel Force Matrix Computation
    for each thread t **do**
      for body $i \leftarrow t$ to $N-1$ with stride T **do** ▷ Round-robin distribution
        for body $j \leftarrow 0$ to $i-1$ **do**
          $F_{ij} \leftarrow$ ComputeGravitationalForce($body_i$, $body_j$)
        end for
      end for
    end for
    Synchronize threads
    // Phase 2: Parallel Acceleration Computation
    for each block B_t of bodies **do**
      for each body $i \in B_t$ **do**
        Sum forces from upper triangle: $\vec{a}_i \leftarrow \sum_{j<i} F_{ij}$
        Sum forces from lower triangle: $\vec{a}_i \leftarrow \vec{a}_i - \sum_{j>i} F_{ji}$
        $\vec{a}_i \leftarrow \vec{a}_i / m_i$
      end for
    end for
    Synchronize threads
    // Phase 3: Parallel Position Update
    for each block B_t of bodies **do**
      for each body $i \in B_t$ **do**
        Update velocity and position using $\vec{a}_i$ and Δt
        Store updated position in telemetry array
      end for
    end for
    Synchronize threads
    Record positions for current timestep
  end for

A.2 Barnes Hutt Algorithms

Algorithm 4 Barnes–Hut Single-Step (iterative, with pool)

Require: `bodies`: array of N Body objects
Require: Δt: time step, θ: opening angle, `useParallel`: {true/false}

`poolIndex` $\leftarrow 0$
`nodePool.clear()`
`nodePool.reserve`$(4N + 1)$ ▷ Preallocate $\approx 4N$ nodes

`bounds` $\leftarrow$ `computeBounds`(`bodies`)
`root` $\leftarrow$ `allocateNode`(`bounds`)
for each Body $b \in$ `bodies` **do**
    `insertBody`(`root`, b)
end for
`computeMassDistribution`(`root`)

`n` $\leftarrow N$
if `useParallel` $\wedge$ `n` > 2000 **then**
    parallel for $i = 0$ to $n - 1$
        $\mathbf{F} \leftarrow$ `computeForceIterative`(`bodies`$[i]$, `root`, θ)
        `bodies`$[i].a \leftarrow \mathbf{F}/$`bodies`$[i].m$
    end parallel for
else
    for $i = 0$ to $n - 1$ **do**
        $\mathbf{F} \leftarrow$ `computeForceIterative`(`bodies`$[i]$, `root`, θ)
        `bodies`$[i].a \leftarrow \mathbf{F}/$`bodies`$[i].m$
    end for
end if

for each Body $b \in$ `bodies` **do**
    $b.v_x \mathrel{+}= b.a_x \times \Delta t$
    $b.v_y \mathrel{+}= b.a_y \times \Delta t$
    $b.x \mathrel{+}= b.v_x \times \Delta t$
    $b.y \mathrel{+}= b.v_y \times \Delta t$
end for

return `bodies` ▷ (Updated in place; tree nodes remain in pool for next step.)

Helper Routines

computeBounds(`bodies`):

1. If `bodies` is empty, return the square $[-1, 1] \times [-1, 1]$.
2. Otherwise, initialize
$$\min X = \max X = \texttt{bodies}[0].x,$$
$$\min Y = \max Y = \texttt{bodies}[0].y.$$
3. For each body b in `bodies`:
$$\min X \leftarrow \min(\min X,\, b.x),$$
$$\max X \leftarrow \max(\max X,\, b.x),$$
$$\min Y \leftarrow \min(\min Y,\, b.y),$$
$$\max Y \leftarrow \max(\max Y,\, b.y).$$
4. Let
$$d = \max(\max X - \min X,\ \max Y - \min Y),$$
$$c_x = \tfrac{\min X + \max X}{2}, \quad c_y = \tfrac{\min Y + \max Y}{2}.$$

5. Return the square
$$[\,c_x - d/2,\ c_y - d/2,\ c_x + d/2,\ c_y + d/2\,].$$

allocateNode(`region`**)**:

1. If `poolIndex` $<$ `nodePool.size()`, set `node` $=$ &`nodePool[poolIndex]`.
2. Otherwise, append a new `QuadNode(region)` to `nodePool` and set `node` $=$ &`nodePool.back()`.
3. Reset `node`:
$$\begin{aligned}&\texttt{node->region} = \texttt{region},\\ &\texttt{node->totalMass} = 0,\\ &\texttt{node->centerOfMass} = (0,0),\\ &\texttt{node->singleBody} = \texttt{nullptr},\\ &\texttt{node->children}[0..3] = \texttt{nullptr}.\end{aligned}$$
4. Increment `poolIndex`. Return `node`.

insertBody(`node`, b**)**:

1. If `node->singleBody` $=$ `nullptr` and `node->children`$[0]$ $=$ `nullptr`, then store `node->singleBody` $\leftarrow$ &b. Return.
2. If `node->children`$[0]$ $=$ `nullptr` but `node->singleBody` $\neq$ `nullptr`, split:
$$\begin{aligned}&\texttt{old} = \texttt{node->singleBody},\\ &\texttt{node->singleBody} = \texttt{nullptr},\\ &\forall\, i \in \{0,1,2,3\},\quad \texttt{node->children}[i] = \texttt{allocateNode}\big(\texttt{childBounds}(\texttt{node->region}, i)\big),\\ &\texttt{quadOld} = \texttt{getQuadrant}(\texttt{node->region}, \texttt{old->coordinates}),\\ &\texttt{node->children[quadOld]->singleBody} = \texttt{old}.\end{aligned}$$
3. Compute `quadNew` $=$ `getQuadrant(node->region,` b`.coordinates)`. Recurse:
$$\texttt{insertBody}\big(\texttt{node->children[quadNew]},\ b\big).$$

computeMassDistribution(`node`**)**:

1. If `node` $=$ `nullptr`, return.
2. If `node->children`$[0]$ $=$ `nullptr` (leaf):
$$\begin{aligned}&\text{If } \texttt{node->singleBody} \neq \texttt{nullptr}:\\ &\quad \texttt{node->totalMass} = \texttt{node->singleBody->m},\\ &\quad \texttt{node->centerOfMass} = \texttt{node->singleBody->coordinates}.\end{aligned}$$
Return.
3. Otherwise (internal node):
$$\begin{aligned}&msum = 0,\quad WeightedSum = (0,0),\\ &\text{for } i = 0\ldots 3:\\ &\quad \texttt{computeMassDistribution(node->children}[i]);\\ &\quad \text{if } \texttt{node->children}[i] \neq \texttt{nullptr} \text{ and } \texttt{children}[i]{-}{>}\texttt{totalMass} > 0, \text{ then}\\ &\qquad msum \mathrel{+}= \texttt{children}[i]{-}{>}\texttt{totalMass},\\ &\qquad WeightedSum \mathrel{+}= \big(\texttt{children}[i]{-}{>}\texttt{centerOfMass}\big) \times \texttt{children}[i]{-}{>}\texttt{totalMass},\\ &\texttt{node->totalMass} = msum,\\ &\text{if } msum > 0:\quad \texttt{node->centerOfMass} = \tfrac{WeightedSum}{msum}.\end{aligned}$$

computeForceIterative(`bi`, `root`, θ**)**:

1. Initialize `totalForce` $= (0,0)$.

2. If `root` = `nullptr` or `root->totalMass` = 0, return `totalForce`.
3. Create a stack S. Push `root` onto S.
4. While S is not empty:
   (a) Pop `node` = S.`back`(); S.`pop_back`().
   (b) If `node` = `nullptr` or `node->totalMass` = 0, **continue**.
   (c) If `node->singleBody` = &`bi` and `node->children`[0] = `nullptr`, **continue**.
   (d) Compute

$$\begin{aligned}
&\Delta x = \texttt{node->centerOfMass}.x - \texttt{bi}.x,\\
&\Delta y = \texttt{node->centerOfMass}.y - \texttt{bi}.y,\\
&r^2 = \Delta x^2 + \Delta y^2 + 10^{-12},\\
&r = \sqrt{r^2}, \quad \texttt{size} = \texttt{node->region.maxX} - \texttt{node->region.minX}.
\end{aligned}$$

   (e) If `node->children`[0] = `nullptr` (leaf) or $\frac{\texttt{size}}{r} < \theta$ (far enough), then:

$$\begin{aligned}
&F_{\text{mag}} = \frac{BH_G \times \texttt{bi}.m \times \texttt{node->totalMass}}{r^2},\\
&\texttt{totalForce}.x \mathrel{+}= (\Delta x/r) \times F_{\text{mag}},\\
&\texttt{totalForce}.y \mathrel{+}= (\Delta y/r) \times F_{\text{mag}}.
\end{aligned}$$

   (f) Otherwise (too close), for $i = 0 \ldots 3$: if `node->children`[i] $\neq$ `nullptr`, push `node->children`[i] onto S.
5. Return `totalForce`.

Discussion of the Appendix Algorithm

- We use an *explicit stack* instead of recursive calls in `computeForceIterative` to avoid deep recursion when the quadtree is very unbalanced.
- The θ-criterion $\left(\frac{\text{cell\_size}}{r} < \theta\right)$ determines if a node can be approximated as a single mass. We set $\theta = 0.5$ for all experiments.
- In each iteration, we rebuild the entire quadtree from scratch (clearing the pool at step 1). This remains $O(N \log N)$ per step, and in practice performs very well for $N > 200$.
- The `poolIndex`/`nodePool` strategy avoids repeated `new`/`delete` overhead and ensures all `QuadNode` pointers remain valid during the force-computation phase.
- Parallelization is applied only to lines 15–18 (computing forces for each body) when $N > 2000$. We spawn an OpenMP thread pool once per step, and each thread processes a contiguous block of bodies to avoid false sharing.

A.3 Particle Mesh Algorithm

Algorithm 5 Particle-Mesh Simulation, using Nearest-Grid-Point (NGP)

Require: System *universe*, time step Δt, grid size N, spatial extent R

telemetry $\leftarrow \emptyset$
telemetry.append(initial positions)
boundaries $\leftarrow [-R, R]$
Compute cell size $h \leftarrow \frac{2R}{N}$
Initialize mass density grid $M[G][G] \leftarrow 0$
Initialize potential grid $\Phi[G][G] \leftarrow 0$
Initialize FFTW input/output arrays and plans
for each time step $s = 1$ to S **do**
    Clear mass density grid M
    for each body b in *universe* **do**
        $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
        $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
        if (i, j) in bounds **then**
            $M[i][j] \leftarrow M[i][j] +$ body.mass
        end if
    end for
    Copy grid mass to FFTW input array
    Compute FFT of mass density using forward FFT
    for each (i, j) in frequency domain **do**
        Compute wave numbers (k_x, k_y)
        Compute $k^2 \leftarrow k_x^2 + k_y^2$
        if $k^2 > 0$ **then**
            Multiply by $-\frac{G}{k^2}$ in frequency domain
        else
            Set value to zero
        end if
    end for
    Compute inverse FFT to obtain gravitational potential
    Normalize inverse FFT result and store in potential grid Φ
    for each body b in *universe* **do**
        $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
        $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
        if (i, j) is valid and not at boundary **then**
            Compute force via central difference of potential:
            $\vec{a}_i \leftarrow -\left(\frac{\Phi[i+1][j] - \Phi[i-1][j]}{2h}, \frac{\Phi[i][j+1] - \Phi[i][j-1]}{2h} \right)$
        else
            $\vec{a}_i \leftarrow (0, 0)$
        end if
    end for
    for each body i **do**
        $\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \Delta t$ ▷ Update velocity
        $\vec{r}_i \leftarrow \vec{r}_i + \vec{v}_i \Delta t$ ▷ Update position
    end for
    telemetry.append(current positions)
end for
Cleanup FFTW plans and memory

Algorithm 6 Parallel Particle-Mesh Simulation using Nearest-Grid-Point (NGP)

Require: System *universe*, time step Δt, grid size N, spatial extent R, number of threads T

telemetry $\leftarrow \emptyset$
telemetry.append(initial positions)
boundaries $\leftarrow [-R, R]$
Compute cell size $h \leftarrow \frac{2R}{N}$
Initialize mass density grid $M[N][N] \leftarrow 0$
Initialize potential grid $\Phi[N][N] \leftarrow 0$
Initialize FFTW input/output arrays and plans
for each time step $s = 1$ to S **do**
    Clear mass density grid M
    Parallel for each thread $t = 1$ to T
        Assign a chunk of bodies to thread t
    for each body b assigned to thread t **do**
        $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
        $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
        if (i, j) in bounds **then**
            Lock $M[i][j]$
            $M[i][j] \leftarrow M[i][j] +$ body.mass
            Unlock $M[i][j]$
        end if
    end for
    End parallel for
    Copy mass grid to FFTW input array
    Compute FFT of mass density using forward FFT
    for each (i, j) in frequency domain **do**
        Compute wave numbers (k_x, k_y)
        $k^2 \leftarrow k_x^2 + k_y^2$
        if $k^2 > 0$ **then**
            Multiply by $-\frac{G}{k^2}$ in frequency domain
        else
            Set value to zero
        end if
    end for
    Compute inverse FFT to obtain gravitational potential
    Normalize result and store in potential grid Φ
    Parallel for each thread $t = 1$ to T
        Assign a chunk of bodies to thread t
    for each body b assigned to thread t **do**
        $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
        $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
        if (i, j) valid and not at boundary **then**
            Compute force via central difference:
            $\vec{a}_b \leftarrow -\left(\frac{\Phi[i+1][j] - \Phi[i-1][j]}{2h}, \frac{\Phi[i][j+1] - \Phi[i][j-1]}{2h} \right)$
        else
            $\vec{a}_b \leftarrow (0, 0)$
        end if
    end for
    End parallel for
    Parallel for each thread $t = 1$ to T
        Assign a chunk of bodies $t = 1$ to T
    for each body b assigned to thread t **do**
        $\vec{v}_b \leftarrow \vec{v}_b + \vec{a}_b \Delta t$
        $\vec{r}_b \leftarrow \vec{r}_b + \vec{v}_b \Delta t$
    end for
    End parallel for
    telemetry.append(current positions)
end for
Cleanup FFTW plans and memory