CSE305 Concurrent Programming: N-Body simulation project

Martin Lau, Oscar Peyron, Ziyue Qiu

May 30, 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 gif 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

```
/
├─ output ............................................Output gif and csv files
│  ├─ telemetry.csv
│  └─ out.gif
├─ src ...........................................................Core files
│  ├─ core.cpp/hpp ...........................Used throughout the entire proj
│  ├─ simplesimulation.cpp/hpp
│  ├─ barneshutt.cpp/hpp
│  ├─ particlemesh.cpp/hpp
│  └─ particlemeshthread.cpp/hpp .2 makefile .................Project builder
├─ main.cpp ..............................................Main execution file
├─ mainparticlemesh.cpp ...........................Aux particle mesh run file
├─ test_telemetry.cpp ....................................Telemetry test 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, thread based and cuda implementation)

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 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 one assignment from CSE306 Computer Graphics.

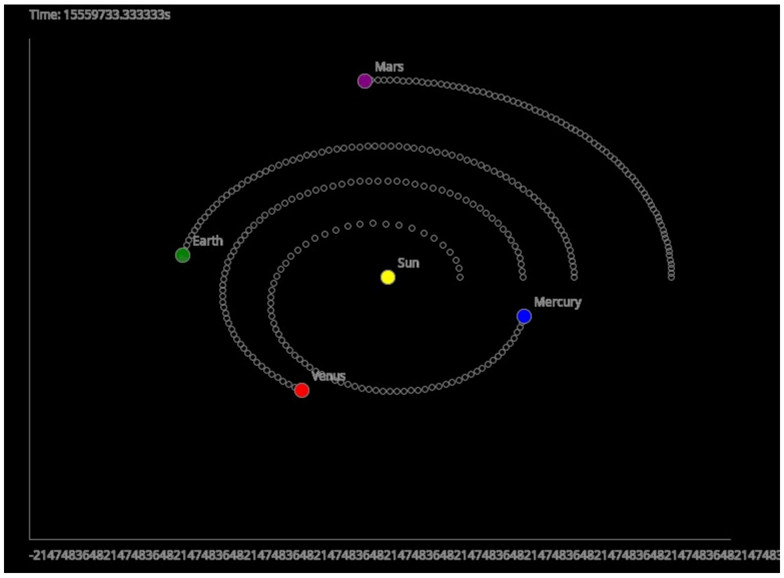
Elaborating more on the telemetry stored within the `System` class, this is stored as a vector of vector of `Vector`. What a mouthful! Our simulations creates different steps/frames. Each step/frame contains the positions of all bodies inside the system (this is a vector of `Vector`). To have the entire 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 writes the frames to a gif file. The frame creation is relatively quick, and most of the visualization time is actually spent in one line (`writeImages(frames.begin(), frames.end(), name)`).

We will look into further solutions to try and decrease the time taken to create the visualization gifs, like reducing image quality, or the number of frames.

For testing purposes, there is also a visualizer in `visualizer.py` which creates a animation using Python's `matplotlib` library, working significantly faster, allowing us to test the correctness of our telemetries.



3 Naive algorithm

So far, we have a basic implementation of the naive algorithm without multi threading (most of my time was taken bugfixing core functionalities and getting visualization to work).

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 $\Delta t$, number of steps $N$
  telemetry ← $\emptyset$
  telemetry.append(initial positions)
  for step ← 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
      telemetry.append(current positions)
  end for
Ensure: Position history for all bodies stored in telemetry
```

Other aspects (parallelizing the update steps, parallelizing forces computations, avoiding race conditions) will be implemented later on.

4 Particle Mesh based algorithm

I have implemented a sequential Particle mesh algorithm, as well as a thread based.

Here below is the pseudo-code for the sequential implementation as well as for the thread-based implementation

Algorithm 2 Particle-Mesh Simulation, using Nearest-Grid-Point (NGP)

```
Require: System $universe$, time step $\Delta t$, grid size $N$, spatial extent $R$
 1: telemetry $\leftarrow \emptyset$
 2: telemetry.append(initial positions)
 3: boundaries $\leftarrow [-R, R]$
 4: Compute cell size $h \leftarrow \frac{2R}{N}$
 5: Initialize mass density grid $M[G][G] \leftarrow 0$
 6: Initialize potential grid $\Phi[G][G] \leftarrow 0$
 7: Initialize FFTW input/output arrays and plans
 8: for each time step $s = 1$ to $S$ do
 9:     Clear mass density grid $M$
10:     for each body b in $universe$ do
11:         $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
12:         $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
13:         if $(i, j)$ in bounds then
14:             $M[i][j] \leftarrow M[i][j] + \text{body.mass}$
15:         end if
16:     end for
17:     Copy grid mass to FFTW input array
18:     Compute FFT of mass density using forward FFT
19:     for each $(i, j)$ in frequency domain do
20:         Compute wave numbers $(k_x, k_y)$
21:         Compute $k^2 \leftarrow k_x^2 + k_y^2$
22:         if $k^2 > 0$ then
23:             Multiply by $-\frac{G}{k^2}$ in frequency domain
24:         else
25:             Set value to zero
26:         end if
27:     end for
28:     Compute inverse FFT to obtain gravitational potential
29:     Normalize inverse FFT result and store in potential grid $\Phi$
30:     for each body b in $universe$ do
31:         $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
32:         $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
33:         if $(i, j)$ is valid and not at boundary then
34:             Compute force via central difference of potential:
35:             $\vec{a}_i \leftarrow -\left( \frac{\Phi[i+1][j] - \Phi[i-1][j]}{2h}, \frac{\Phi[i][j+1] - \Phi[i][j-1]}{2h} \right)$
36:         else
37:             $\vec{a}_i \leftarrow (0, 0)$
38:         end if
39:     end for
40:     for each body $i$ do
41:         $\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \Delta t$                ▷ Update velocity
42:         $\vec{r}_i \leftarrow \vec{r}_i + \vec{v}_i \Delta t$                ▷ Update position
43:     end for
44:     telemetry.append(current positions)
45: end for
46: Cleanup FFTW plans and memory
```

Algorithm 3 Parallel Particle-Mesh Simulation using Nearest-Grid-Point (NGP)

```
Require: System $universe$, time step $\Delta t$, grid size $N$, spatial extent $R$, number of threads $T$
 1: telemetry $\leftarrow \emptyset$
 2: telemetry.append(initial positions)
 3: boundaries $\leftarrow [-R, R]$
 4: Compute cell size $h \leftarrow \frac{2R}{N}$
 5: Initialize mass density grid $M[N][N] \leftarrow 0$
 6: Initialize potential grid $\Phi[N][N] \leftarrow 0$
 7: Initialize FFTW input/output arrays and plans
 8: for each time step $s = 1$ to $S$ do
 9:     Clear mass density grid $M$
10:     Parallel for each thread $t = 1$ to $T$
11:        Assign a chunk of bodies to thread $t$
12:     for each body $b$ assigned to thread $t$ do
13:         $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
14:         $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
15:         if $(i, j)$ in bounds then
16:             Lock $M[i][j]$
17:             $M[i][j] \leftarrow M[i][j] +$ body.mass
18:             Unlock $M[i][j]$
19:         end if
20:     end for
21:     End parallel for
22:     Copy mass grid to FFTW input array
23:     Compute FFT of mass density using forward FFT
24:     for each $(i, j)$ in frequency domain do
25:         Compute wave numbers $(k_x, k_y)$
26:         $k^2 \leftarrow k_x^2 + k_y^2$
27:         if $k^2 > 0$ then
28:             Multiply by $-\frac{G}{k^2}$ in frequency domain
29:         else
30:             Set value to zero
31:         end if
32:     end for
33:     Compute inverse FFT to obtain gravitational potential
34:     Normalize result and store in potential grid $\Phi$
35:     Parallel for each thread $t = 1$ to $T$
36:        Assign a chunk of bodies to thread $t$
37:     for each body $b$ assigned to thread $t$ do
38:         $i \leftarrow \left\lfloor \frac{x_b - \min_x}{h} \right\rfloor$
39:         $j \leftarrow \left\lfloor \frac{y_b - \min_y}{h} \right\rfloor$
40:         if $(i, j)$ valid and not at boundary then
41:             Compute force via central difference:
42:             $\vec{a}_b \leftarrow -\left( \frac{\Phi[i+1][j] - \Phi[i-1][j]}{2h}, \frac{\Phi[i][j+1] - \Phi[i][j-1]}{2h} \right)$
43:         else
44:             $\vec{a}_b \leftarrow (0, 0)$
45:         end if
46:     end for
47:     End parallel for
48:     Parallel for each thread $t = 1$ to $T$
49:        Assign a chunk of bodies $t = 1$ to $T$
50:     for each body $b$ assigned to thread $t$ do
51:         $\vec{v}_b \leftarrow \vec{v}_b + \vec{a}_b \Delta t$
52:         $\vec{r}_b \leftarrow \vec{r}_b + \vec{v}_b \Delta t$
53:     end for
54:     End parallel for
55:     telemetry.append(current positions)
56: end for
57: Cleanup FFTW plans and memory
```

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

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.

Parallel computation was done through parallelizing via the bodies. 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.

However, one can still implement fast-fourier transform in parallel within the library fftw3. Unfortunately, I was not able to make it work on my machine.

Time performance of the sequential particle-mesh simulation

Number of Bodies	Execution Time (ms)
4	52
5	54
50	109
100	157
1,000	941
2,000	3,013
5,000	7331
10,000	14 053

Table 1: Execution time of the particle-mesh simulation for varying numbers of bodies (grid size = 10)

Time performance of the parallel particle-mesh simulation

Number of Bodies	Threads = 5	Threads = 7	Threads = 10
4	1661	–	–
5	2209	2848	4058
50	2295	2906	3263
100	2298	3034	3165
1000	3436	3800	5006
2000	5075	5865	6595
5000	7576	6244	6757
10000	18230	12306	13881

Table 2: Execution time (in milliseconds) for varying numbers of bodies and thread counts (grid size = 10).

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

The Barnes–Hut algorithm reduces the complexity of the classical $O(N^2)$ N-body force computation to approximately $O(N \log N)$ by hierarchically clustering distant bodies. Our implementation follows these main stages:

Algorithm 4 Barnes–Hut Simulation Outline

```
Require: System of bodies with masses $m_i$, positions $\vec{r}_i$, velocities $\vec{v}_i$
Require: Time step $\Delta t$, number of steps $N$, opening angle $\theta$
 1: telemetry ← $\emptyset$
 2: telemetry.append(initial positions)
 3: for step ← 0 to $N - 1$ do
 4:     [minB, maxB] ← computeBounds(universe)
 5:     root ← createRootNode([minB, maxB])
 6:     for each body $b_i$ do
 7:         insertBody(root, $b_i$)
 8:     end for
 9:     computeMassDistribution(root)
10:     for each body $b_i$ do
11:         $\vec{a}_i$ ← forceOnBody($b_i$, root, $\theta$)/$m_i$
12:     end for
13:     updateBodies(universe, $\Delta t$)
14:     freeQuadTree(root)
15:     telemetry.append(current positions)
16: end for
Ensure: Position history for all bodies stored in telemetry
```

1. **Compute Simulation Bounds.** We first call
   - `computeBounds(const System&)` to find an axis-aligned square enclosing all bodies (with a small padding).
2. **Quadtree Construction.** We represent space by a pointer-based quadtree of `QuadNode` objects:
   - `createRootNode(const Bounds&)` allocates the root covering the full region.
   - `insertBody(QuadNode *, Body&)` recursively subdivides nodes so that each leaf contains at most one body.
3. **Mass–Center Distribution.** A post-order traversal aggregates mass and center-of-mass at every internal node:
   - `computeMassDistribution(QuadNode *)` computes `totalMass` and `centerOfMass`.
4. **Force Computation.** For each body b_i, we traverse the tree and apply the opening-angle criterion θ:
   - `forceOnBody(const Body&, QuadNode, double theta)` approximates distant clusters as single masses, or recurses into children when closer.
5. **Parallelization (Shared-Memory).** To exploit multicore CPUs, we compute forces in parallel:
   - `computeForcesParallel(System&, QuadNode, double theta)` spawns a pool of `std::threads`, partitions the bodies into chunks, and each thread calls `forceOnBody` on its subset.
   - Accelerations are stored per-thread and then written back, avoiding fine-grained locks.
6. **Time Integration.** Once all accelerations are known, we update velocities and positions in one pass:
   - `updateBodies(System&, double dt)` applies

$$\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a}\,\Delta t, \quad \mathbf{x} \leftarrow \mathbf{x} + \mathbf{v}\,\Delta t.$$

7. **Cleanup.** The quadtree is deallocated to avoid memory leaks:
    - `freeQuadTree(QuadNode *)` recursively `delete`s all nodes.

Concurrency Aspects

- **Multithreading in C++:** we illustrate basic shared-memory concurrency by partitioning the force computation across threads. This uses standard `std::thread` and per-thread buffers, avoiding complex locking.
- **Concurrent Data Structures:** insertion into the quadtree could be parallelized by feeding bodies into a thread-safe queue; our current version remains serial for clarity but is structured to allow a concurrent `insertBody` using a mutex per node or a lock-free pointer array.
- **GPU / PRAM Illustration (Optional):** under `USE_CUDA`, we provide `simulateBruteForceGPU(System&,double)` which launches an $O(N^2)$ CUDA kernel. Each GPU thread computes one body's net force, demonstrating the PRAM model in practice.
- **Correctness & Testing:** we compare parallel results to a single-threaded reference on small N, and use tools like ThreadSanitizer to detect data races. Telemetry outputs (positions over time) are also verified for physical invariants (e.g. center-of-mass motion).