Parallel Gravitational N-Body (Planetary) Simulation: Performance Comparison of Sequential, OpenMP, and CUDA Backends

---

1. Team Members

Jack Herrington

---

2. Project Overview

This project implements an **N-body gravitational simulation** for planetary systems using three computational backends:

1. **Sequential (C++17 baseline)**
2. **OpenMP (multicore CPU parallelism)**
3. **CUDA (GPU parallelism)**

The simulation models Newtonian gravitational interactions between celestial bodies and compares the computational performance of each backend under identical initial conditions.

The focus is on **parallel performance analysis** rather than numerical accuracy of trajectories. The project aims to quantify how computational resources—CPU threads versus GPU cores—affect runtime, throughput, and scalability.

---

3. Objectives

- Implement a shared codebase supporting three interchangeable computational backends.
- Compare **runtime per time step**, **throughput (interactions per second)**, and **scaling performance** across CPU threads and GPU configurations.
- Read all input data (planetary positions, velocities, and masses) from a file.
- Produce timing logs and plots for empirical performance comparison.

- Demonstrate practical understanding of **data-parallel** and **task-parallel** design in scientific computation.

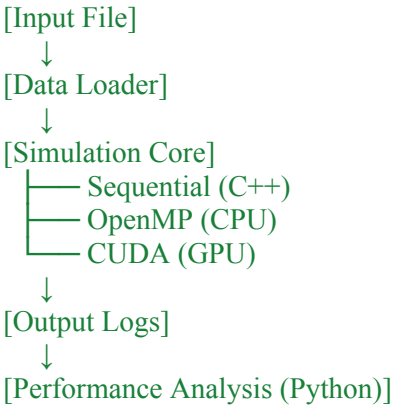
---

4. System Design Overview

4.1 Core Components

Module	Description
main.cpp	Initializes data, selects backend, and manages the simulation loop.
nbody_common.hpp	Defines shared structures for bodies, constants (G), and file I/O.
integrator.cpp	Time integration (Velocity Verlet or Euler method).
backend_seq.cpp	Sequential CPU implementation.
backend_omp.cpp	OpenMP implementation (multithreaded loops).
backend_cuda.cu	CUDA GPU implementation using kernels.
analysis.py	Python script (Matplotlib) to generate performance plots.

4.2 Data Flow Diagram



---

5. Algorithm Design

5.1 N-Body Force Computation

Each particle *i* experiences acceleration due to every other particle *j* according to Newton's Law of Gravitation:

$$\vec{a}_i = G \sum_{j \neq i} m_j \frac{(\vec{r}_j - \vec{r}_i)}{|\vec{r}_j - \vec{r}_i|^3}$$

This computation exhibits **O(N²)** complexity and is ideal for evaluating the benefits of parallelization.

5.2 N Body Implementation

While there exist many formulas and algorithms to simulate an N-body problem, the implementation of choice for this simulation is the basic Brute Force Approach. This means we approach the problem in the most basic way and compute all pairwise gravitational forces between particles. This is the most accurate and basic way to complete the computation. This computation exhibits **O(N²)** complexity.

5.3 Integration Loop

At each time step:

1. Compute accelerations on all particles.
2. Update velocities and positions.
3. Record timing metrics and iteration data.

A simple **Velocity Verlet** or **Euler** integrator will be used to advance the system state.

---

6. Pseudocode

Sequential Baseline

```
for step in 0..num_steps:
   for i in 0..N:
      ax[i] = ay[i] = az[i] = 0
      for j in 0..N:
```

```
        if i != j:
            compute_force(i, j)
    update_positions_and_velocities()
```

OpenMP Backend

```
#pragma omp parallel for schedule(static)
for i in 0..N:
    for j in 0..N:
        if (i != j)
            compute_force(i, j);
```

CUDA Backend

```
__global__ void compute_forces(Body *bodies, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        float3 acc = {0, 0, 0};
        for (int j = 0; j < N; j++) {
            if (i != j) acc += compute_pairwise(bodies[i], bodies[j]);
        }
        bodies[i].acc = acc;
    }
}
```

---

7. Software Tools and Environment

- **Languages:** C++17 and CUDA C
- **Libraries:**
    - OpenMP for CPU parallel loops
    - CUDA Runtime API for GPU acceleration
    - Python (Matplotlib) for performance visualization
- **Development Platform:** Gigabyte Aorus 15P KD-72US223SH, 15.6" FHD 240Hz, Core i7, GeForce RTX 3060 Graphics Gaming Laptop and/or Thor Cluster
- **Build Tools:** GCC/G++, NVCC

---

8. Testing and Evaluation Plan

8.1 Test Inputs

Simulation input files will contain initial positions, velocities, and masses for different system sizes:

- **Small:** N = 1,000
- **Medium:** N = 10,000
- **Large:** N = 100,000

Each dataset will be read from a .txt or .csv file.

8.2 Planned Experiments

Experiment	Description	Performance Metric
Problem Size Scaling	Vary N to observe growth and computational scaling.	Runtime per step
CPU Thread Scaling	Test OpenMP using 1–16 threads.	Speedup, efficiency
Backend Comparison	Compare best OpenMP vs CUDA configurations.	Throughput (interactions/s)

8.3 Performance Metrics

- **Runtime per iteration**
- **Speedup:** $S=T_{seq}/T_p$
- **Efficiency:** $E=S/p$
- **Throughput:** Total interactions per second

---

9. Expected Results

- The **CUDA** implementation is expected to outperform OpenMP for large-scale N due to massive GPU parallelism.
- The **OpenMP** backend should show near-linear speedup for small-to-medium problem sizes before plateauing due to thread contention.
- The **Sequential** version serves as a correctness and timing baseline.
- Visualization will display performance crossovers between CPU and GPU backends.

---

10. Risks and Mitigation Strategies

Risk	Mitigation Strategy
Numerical instability due to large time steps	Use adaptive or sufficiently small Δt and double precision arithmetic.
GPU memory limitations	Implement chunked/batched computation for large N.
OpenMP thread synchronization overhead	Optimize scheduling, memory access patterns, and cache usage.
File I/O bottlenecks	Use binary or preloaded memory buffers to reduce read latency.

---

11. References

- Peter Pacheco, *An Introduction to Parallel Programming* (Chapters 5 and 6).
- NVIDIA Corporation, *CUDA Programming Guide*.
- OpenMP Architecture Review Board, *OpenMP API Specification v5.0*.
- Parallel Computing course materials (CS 5260, Fall 2025).
