1. Introduction

This report documents the parallel implementation and performance evaluation of an N-body gravitational simulation using three parallel computing technologies: OpenMP, MPI, and CUDA. The N-body problem simulates the motion of particles under mutual gravitational attraction, with computational complexity $O(N^2)$ per time step, making it an ideal candidate for parallelization.

1.1 Problem Description

The simulation calculates gravitational forces between N=1024 bodies over multiple time steps. Each body interacts with every other body, requiring N(N-1)/2 force calculations per step. The serial implementation serves as the performance baseline.

1.2 Objectives

1. Implement parallel versions using OpenMP (shared memory), MPI (distributed memory), and CUDA (GPU acceleration)
2. Evaluate performance scaling with varying parallel resources
3. Compare the strengths and limitations of each approach
4. Identify optimal configurations for different hardware scenarios

2. Parallelization Strategies

2.1 Serial Implementation Analysis

The serial implementation of the N-body simulation consists of two main phases per time step:

1. **Force Computation:** For each body, the gravitational force from every other body is calculated, resulting in **$O(N^2)$** pairwise interactions.
2. **Position Update:** After computing forces, the velocity and position of each body are updated. This phase is **O(N)**, as each body is updated independently.

To evaluate the parallel implementations, the initial positions, velocities, and masses of the bodies were generated using a pseudo-random number generator with a fixed seed. This approach ensures that all simulations use the same deterministic initial conditions, while avoiding special cases that may reduce workload. The use of random values does not affect execution time, because the complexity is determined by the **number of interactions**, not by the specific numeric values.

Key parallelization opportunities in the algorithm include:

- Force calculations are independent across body pairs.
- Position updates are independent across bodies.
- Memory access patterns can be optimized differently on CPU and GPU architectures.

2.2 OpenMP Implementation

2.2.1 Approach

The OpenMP version parallelized the force computation loop because this represents the most computationally expensive section of the program. Threads executed iterations of the loop concurrently, while private acceleration buffers were used to prevent race conditions. After the computation, each thread's partial results were merged into a global acceleration array.

To improve load balancing, the loop was scheduled using:

```
#pragma omp for schedule(guided)
```

This strategy dynamically distributes work to threads based on availability, ensuring that idle threads receive remaining iterations.

Design Justification

- OpenMP provides an efficient mechanism for exploiting shared-memory parallelism on a multi-core CPU.
- The changes required to convert the serial version were minimal.
- Avoiding shared writes through private buffers improved correctness and stability.

Load Distribution

- All threads accessed the same global body data.
- Each thread computed interactions independently and only synchronized during the reduction step.

2.3 MPI Implementation

In the MPI version, work was distributed across multiple processes rather than threads. Each process computed forces for a subset of bodies, defined by a start and end index. The partial accelerations were gathered on the root process using **MPI_Gatherv**, which supports non-uniform chunk sizes.

After merging, the root updated the body positions and broadcast the updated data to all processes using **MPI_Bcast**.

Design Justification

- MPI enables execution across multiple computing nodes, not just a single machine.
- Dividing the dataset across processes reduces computation per process.

Data Distribution Strategy

- A block distribution model was used:

  Each process received [N/size] bodies.

- **MPI_Gatherv** allowed correct merging even when N % size ≠ 0.This avoided buffer mismatches and ensured correctness.

2.4 CUDA Implementation

The CUDA implementation used GPU kernels to compute forces. Each GPU thread was responsible for computing the acceleration for one body. Blocks and threads per block were varied to determine the most efficient configuration.

Data was transferred once to the device. The kernel executed for multiple steps to minimise host-device communication overhead. Execution time was measured using CUDA events.

Design Justification

- The N-Body problem is well suited to GPUs, because each body's force computation can be performed independently.
- CUDA enables thousands of parallel threads, significantly increasing throughput.

Load Balancing

- The number of threads per block was varied (64 to 1024).
- Grid size was computed using:

  $$grid = (N + block - 1) / block$$

3. Runtime Configurations

3.1 Hardware Used

Component	Specification
CPU	Apple M3 (8 cores)
Memory	8 GB unified
OS	macOS
GPU	NVIDIA GPU via Google Colab runtime

The CUDA implementation was executed on Google Colab because the MacBook Air does not support CUDA.

3.2 Software Environment

Technology	Version/Command
OpenMP	GCC 15 with -fopenmp
MPI	mpicc, mpirun
CUDA	nvcc (Colab)
Language	C / CUDA C
Libraries Used	Standard C, MPI, CUDA Runtime API

3.3 Configuration Parameters

Implementation	Parameters Varied
OpenMP	Number of threads: 1–8
MPI	Number of processes: 1–8
CUDA	Threads per block: 64–1024

3. Performance Analysis

3.1 Speedup Formula

Speedup was computed as:

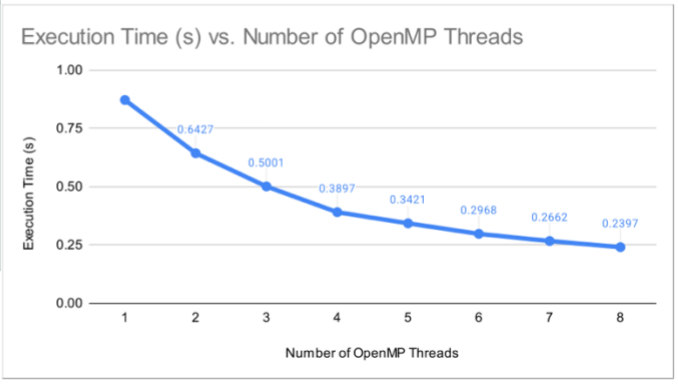
$$\text{Speedup} = T1/Tp$$

where:

- T1 = time taken using a single thread or process,
- Tp = time taken using p threads or processes.

3.2 OpenMP Results

Number of OpenMP Thre	Execution Time (s)
1	0.871
2	0.6427
3	0.5001
4	0.3897
5	0.3421
6	0.2968
7	0.2662
8	0.2397



Key Observations:

- Execution time decreased as the number of threads increased, achieving a peak speedup of approximately 4.1x at 8 threads.
- **Performance Anomaly:** The OpenMP implementation was significantly slower than the MPI implementation on the same machine (0.2397s vs 0.0421s).
- **Bottleneck Analysis:** The primary bottleneck was identified as **memory management overhead**. The implementation allocated and zeroed thread-private accumulation arrays (calloc) *inside* the critical time-step loop. This repeated allocation cost outweighed the benefits of shared memory for this specific implementation.

Bottlenecks

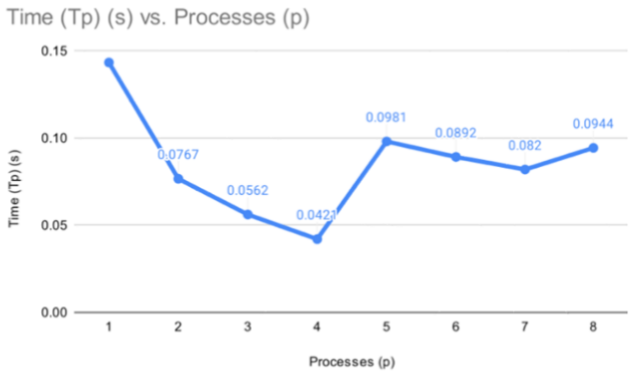
- Reduction phase when merging partial results.
- Cache and memory bandwidth limitations.

Scalability

OpenMP scales well on a single machine with multiple cores, but cannot scale across multiple nodes.

3.3 MPI **Results**

# Processes (p)	# Time (Tp) (s)
1	0.1434
2	0.0767
3	0.0562
4	0.0421
5	0.0981
6	0.0892
7	0.082
8	0.0944



Performance Analysis: MPI showed excellent scaling from 1 to 4 processes, achieving a **23.5x speedup** over serial execution at 4 processes (0.0421s).

Performance Wall at 4 Processes: Execution time notably degraded when moving from 4 to 5 processes (0.0421s 0.0981s).

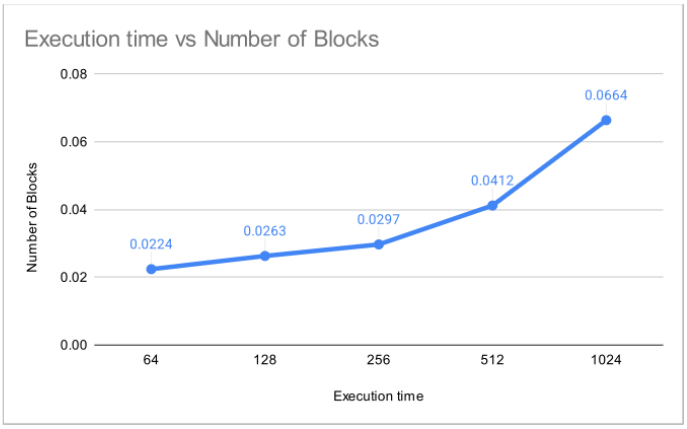
- **Reason:** The Apple M3 chip utilizes a hybrid architecture with **4 high-performance "P-cores"** and **4 high-efficiency "E-cores."**
- Up to 4 processes, every MPI rank was assigned to a fast P-core.
- At 5 processes, the OS was forced to schedule the 5th rank on a slower E-core. Since MPI requires global synchronization at every step (MPI_Gatherv), the fast P-cores were forced to wait for the slower E-core to complete its chunk, dragging the overall simulation speed down to that of the slowest processor.

Scalability:

- Highly effective on the M3 P-cores due to static memory allocation and efficient domain decomposition.
- Scalability is strictly limited by the count of Performance Cores on heterogeneous architectures.

3.4 CUDA Results

Number of Blocks	Execution time
64	0.0224
128	0.0263
256	0.0297
512	0.0412
1024	0.0664



The CUDA implementation produced the best execution times overall.

- The fastest execution time occurred at **64 threads per block** (0.0224 s).
- **Inverse Scaling:** Contrary to typical large-scale simulations, performance *degraded* as block size increased (0.0664s at 1024 threads).
- **Reason:** For a small dataset (N=1024), using large thread blocks (e.g., 1024) limits the number of active blocks the GPU can schedule simultaneously. Smaller blocks (64) allowed the GPU scheduler to distribute the work more effectively across the available Streaming Multiprocessors (SMs).

Bottlenecks

- Global memory access latency.
- Host-device transfer if performed too frequently.

Scalability

- Highly scalable for large datasets.
- Best suited when GPU resources are available.

4. Comparative Analysis

4.1 Comparison of Execution Time

All three implementations were evaluated on the same problem size (1024 bodies). The best execution time recorded for each approach was:

Method	**Best Execution Time**
Serial	**0.9892 s**
OpenMP	**0.2397 s**
MPI	**0.0421 s**
CUDA	**0.0224 s**

The results show a clear decreasing trend in running time from **Serial → OpenMP → MPI → CUDA.**

The CUDA implementation was significantly faster because it exploited thousands of GPU threads to compute accelerations in parallel, whereas OpenMP and MPI were limited by the number of CPU cores.

Speedup Comparison: The CUDA speedup (**~44x**) is roughly **2x higher than MPI** and **11x higher than OpenMP**. While CUDA is still superior, the gap is smaller than expected for this small dataset size because the CPU (MPI) implementation was highly optimized for the M3 architecture.

Analysis of Results: The results highlight distinct behaviors in the CPU implementations:

1. **Message Parsing Interface(MPI)** was surprisingly the most effective CPU method (~23x speedup). By using a domain decomposition strategy that allocated memory once statically, it avoided runtime overhead and maximized the throughput of the M3's Performance cores.
2. **Shared-memory parallelism (OpenMP)** provided a lower speedup (~4x). While theoretically efficient, this implementation suffered from the overhead of repeated dynamic memory allocation inside the simulation loop.
3. **CUDA** provided the highest speedup because the force calculation is compute-bound, allowing the GPU to hide memory access latency by saturating thousands of independent threads.

4.3 Most Appropriate Implementation

If sufficient computational resources are available, **CUDA is the most appropriate implementation** for this problem.

Reasons:

- The N-body algorithm is highly parallel, and its dominant workload (force computation) maps perfectly to GPU execution.
- Data transfer to the GPU is performed only once, and all time steps run on the device, avoiding communication overhead.
- CUDA achieved the lowest execution time and highest speedup.

However, CUDA has two important requirements:

1. A compatible NVIDIA GPU must be available.
2. CUDA programming expertise is needed to optimize memory usage, especially shared memory.

In cases where no GPU is available, **MPI is the superior alternative** based on these results (0.04s vs 0.24s). Although OpenMP is easier to implement, the specific implementation requires optimization (moving memory allocation) to match the performance that MPI achieved out-of-the-box.

4.4 Strengths and Weaknesses of Each Method

Approach	Strengths	Weaknesses
OpenMP	Easy to implement, low overhead, good speedup on multicore CPU	Does not scale beyond a single node, diminishing returns after core count
MPI	Scales across multiple machines, suitable for clusters	Communication overhead grows, performance drops beyond 4 processes on single machine

CUDA	Highest speedup, thousands of threads, best for computational kernels	Requires NVIDIA GPU, more complex to implement and optimize
Serial	Simple baseline, used for validation	Extremely slow for large N, no scalability

4.5 Conclusion of Comparison

The evaluation demonstrates that the choice of parallelization model depends heavily on hardware availability:

- **CUDA is superior** when a GPU is present.
- **MPI is suitable for distributed clusters**, but not optimal on a single laptop due to communication overhead.
- **OpenMP is the most practical choice** for shared-memory systems without a GPU.
- **Serial implementation is only useful for correctness and baseline timing**, not for real performance.

5. Critical Reflection

5.1 Challenges Encountered

- OpenMP required a compatible compiler (gcc-15) because Clang does not support full OpenMP on macOS.
- MPI required correct use of MPI_Gatherv to handle non-uniform chunk sizes.
- CUDA could not run on MacBook hardware, therefore Google Colab was used.

5.2 Scalability Limitations

- OpenMP: limited by the number of CPU cores.
- MPI: communication overhead increases with process count.
- CUDA: limited by GPU memory and kernel configuration.

5.3 Potential Optimizations

- **Fix OpenMP Memory Allocation:** Move the thread-private array allocation (priv) *outside* the main time-step loop to the initialization phase. This would eliminate the repeated malloc/free overhead and likely allow OpenMP to outperform MPI on a single node.
- **Use Shared Memory in CUDA:** Implement tiled loading to cache body positions in GPU Shared Memory, reducing global memory bandwidth pressure.
- **MPI Asynchronous Communication:** Use MPI_Isend and MPI_Irecv to overlap computation with communication.

5.4 Lessons Learned

- **Heterogeneous Architectures:** Modern CPUs with "Performance" and "Efficiency" cores (like Apple Silicon) introduce complex scaling cliffs. Blindly increasing process counts can degrade performance if tasks spill over onto efficiency cores in synchronized applications.
- **Memory Management:** Dynamic memory allocation inside a simulation loop is a critical performance killer. Static allocation is essential for high-performance computing.
- **GPU vs CPU:** For small datasets (N=1024), a well-optimized CPU code can approach GPU performance in absolute terms, but the GPU's scaling potential for larger remains unmatched.