MultiCore Programming
Final Project

N-Body Problem

Tahmine Tavakoli   Elham Armin   Seyed Alireza Amini
9912762267   400126202   9912762610

About the problem

In physics, the n-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally. Solving this problem has been motivated by the desire to understand the motions of the Sun, Moon, planets, and visible stars. In the 20th century, understanding the dynamics of globular cluster star systems became an important n-body problem. The n-body problem in general relativity is considerably more difficult to solve due to additional factors like time and space distortions.

General formulation

The n-body problem considers n point masses mi (i = 1, 2, …, n), in an inertial reference frame in three dimensional space ℝ3, moving under the influence of mutual gravitational attraction. Each mass mi has a position vector qi. Newton's law of gravity says that the gravitational force felt on mass mi by a single mass mj is given by equation 1.

$$\mathbf{F}_{ij} = \frac{G m_i m_j}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^2} \cdot \frac{\left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|} = \frac{G m_i m_j \left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^3}$$

Equation 1.

In equation 1, G is the gravitational constant and ‖qj − qi‖ is the magnitude of the distance between qi and qj.
With every time step, positions of the bodies change (net force) due to other bodies.

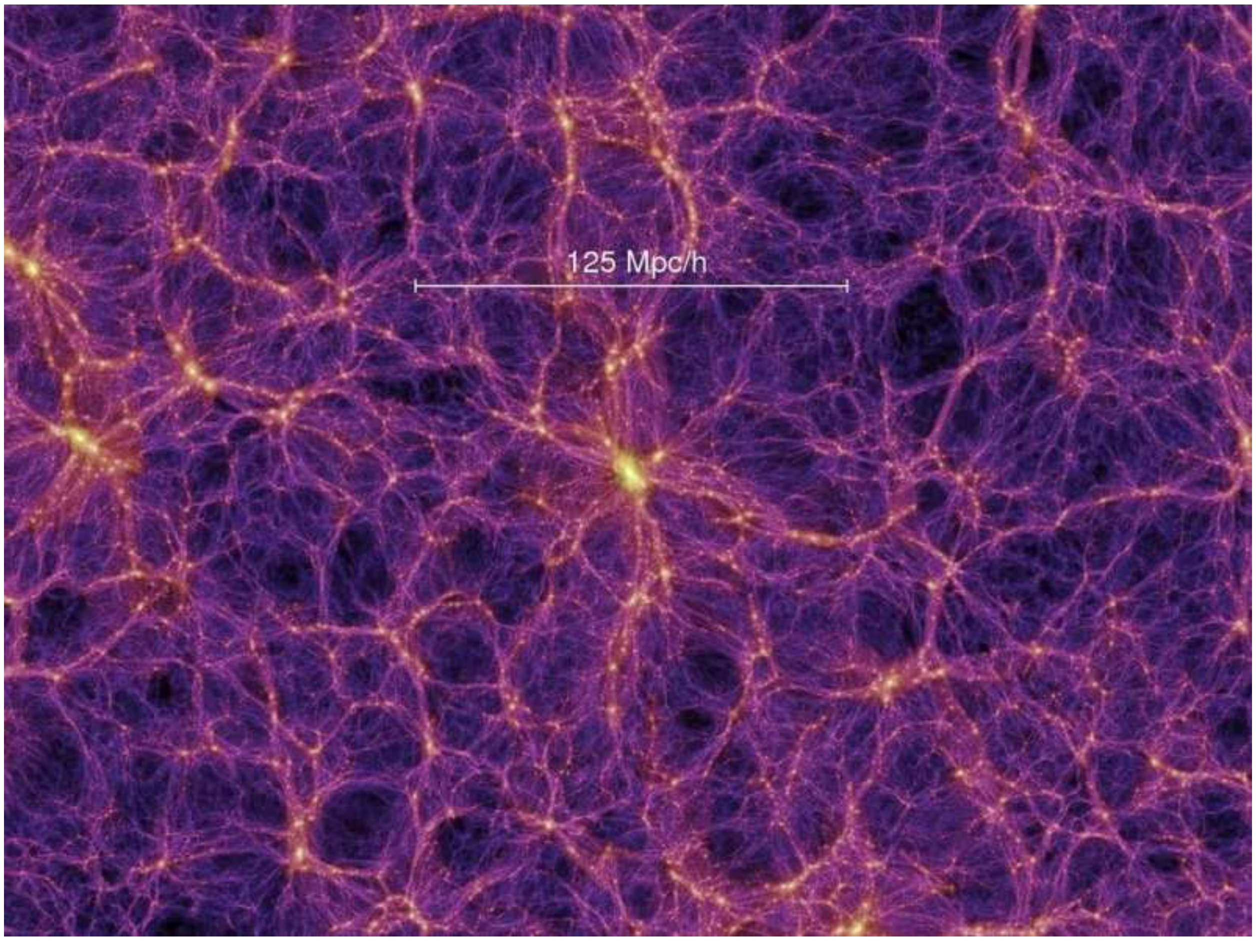


Figure 1. This image represents approximately 10 billion bodies.

Brute force approach demands huge computational power and the complexity is O(n^2). The serial approach works well for a small number of bodies but when we have a big number, it simply fails. Thus we need a parallel approach to solve the problem taking advantage of the computational resources we have.

Techniques to solve the problem

1. All pairs

   We calculate all possible combinations of interactive forces between the bodies.
   - Most accurate solution
   - Computationally intensive
   - Choose variable time step schemes to calculate interaction
   - Computational power and the complexity is O(n^2)

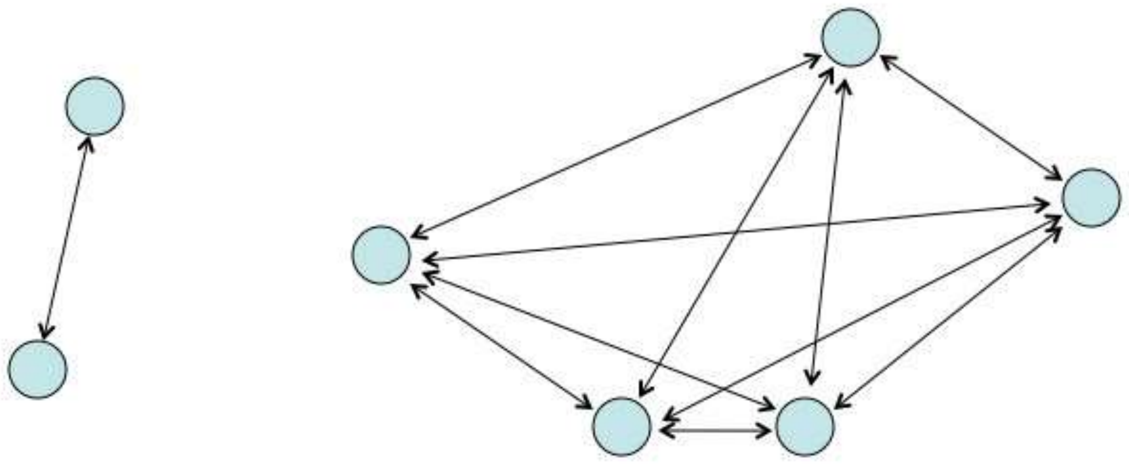


Figure 2. N-Body Simulation

2. Barnes Hut Algorithms

Barnes-Hut is an approximation technique used. The idea behind this is that for bodies which are farther away, instead of calculating all pairs interactive forces, the center of mass is calculated.

Barnes-Hut algorithm recursively divides the n bodies into groups by storing them in an octree (or a quad-tree in a 2D simulation). Each node in this tree represents a region of the three-dimensional space. The topmost node represents the whole space, and its eight children represent the eight octants of the space.

The space is recursively subdivided into octants until each subdivision contains 0 or 1 bodies (some regions do not have bodies in all of their octants).

There are two types of nodes in the octree: internal and external nodes. An external node has no children and is either empty or represents a single body. Each internal node represents the group of bodies beneath it, and stores the center of mass and the total mass of all its children bodies.

To calculate the net force on a particular body, the nodes of the tree are traversed, starting from the root. If the center of mass of an internal node is sufficiently far from the body, the bodies contained in that part of the tree are treated as a single particle whose position and mass is respectively the center of mass and total mass of the internal node. If the internal node is sufficiently close to the body, the process is repeated for each of its children.

Computational power and the efficiency is O(n log n).

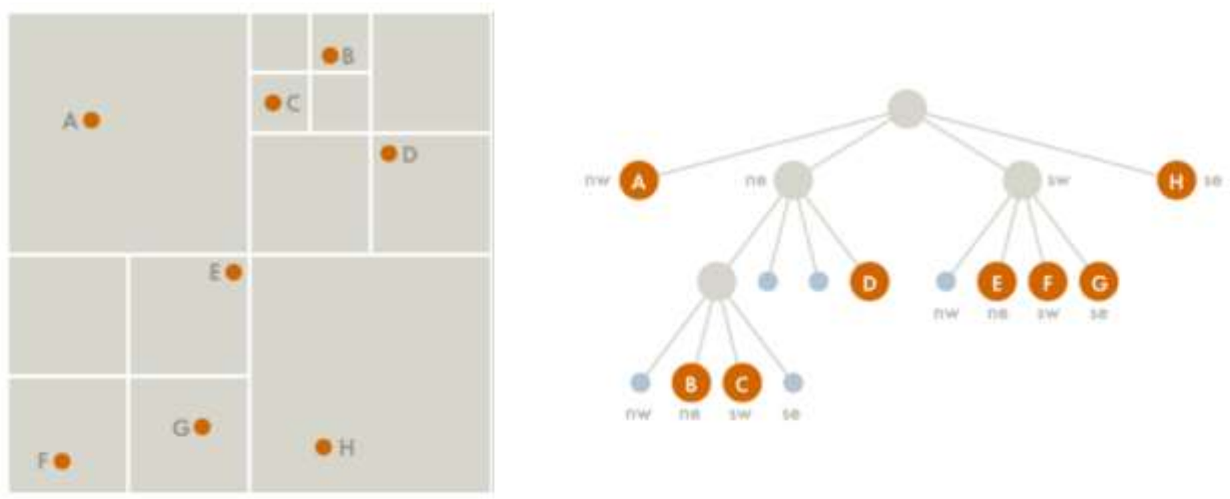


Figure 3. Barnes-Hut Algorithm

Serial Implementation

The N-body problem in physics and astrophysics involves predicting the individual motions of a group of celestial objects interacting with each other gravitationally. A serial implementation of the N-body problem in a programming context involves calculating the forces acting between each pair of objects and updating their positions and velocities over time.
Steps in Serial Implementation of the N-Body Problem:

1. Initialization
   Define the positions, velocities, and masses of the N bodies.
   Choose a time step for the simulation.

2. Force Calculation
   For each body, calculate the gravitational force exerted by every other body.
   Use Newton’s law of gravitation

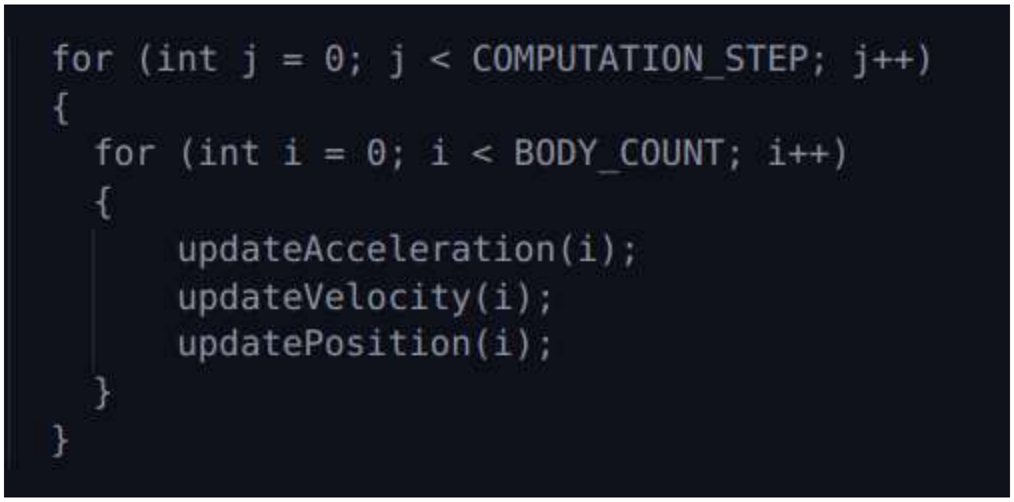
$$F = G\frac{m_1 m_2}{r^2}$$

Equation 2.
Where G is the gravitational constant, m_1 and m_2 are the masses of the bodies, and r is the distance between bodies 1 and 2.

3. Update Velocities
   Use the forces calculated to update the velocities of the bodies.
   Apply Newton's second law to compute the accelerations and velocities.

4. Update Positions
   Update the positions of the bodies using the new velocities.

5. Repeat
   Repeat the force calculation and update steps for each time step until the desired simulation time is reached.



```
for (int j = 0; j < COMPUTATION_STEP; j++)
{
  for (int i = 0; i < BODY_COUNT; i++)
  {
      updateAcceleration(i);
      updateVelocity(i);
      updatePosition(i);
  }
}
```

Algorithm 1. serial implementation of the N body problem

Parallel OpenMP Implementation

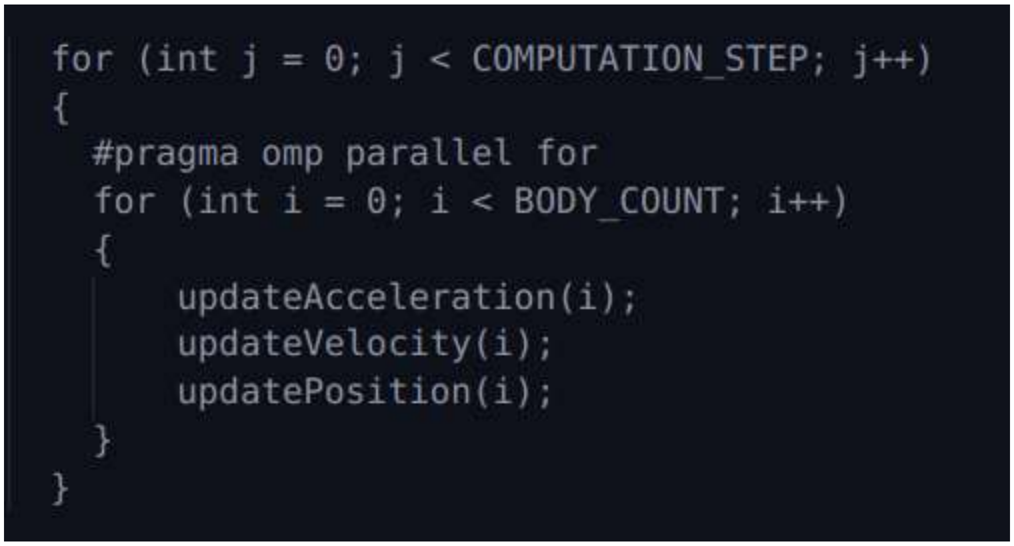
The parallel implementation of the N-body problem aims to accelerate the computation by distributing the workload across multiple processors. This can significantly reduce the time required for force calculations, which are O(N^2) in complexity.
Parallel Implementation Strategies:

- Domain Decomposition
  Divide the space into subdomains and assign each subdomain to a different processor.
  Each processor is responsible for calculating the interactions of bodies within its subdomain.

- Force Calculation
  Each processor calculates the forces between bodies in its own subdomain.
  Communication between processors is necessary to account for interactions between bodies in different subdomains.

- Data Parallelism
  Distribute the bodies among processors so that each processor calculates the forces on a subset of bodies.

Each processor may need to know the positions of all bodies, leading to communication overhead.
We have implemented the parallelized version of the N-body problem using OpenMP in C++.



```
for (int j = 0; j < COMPUTATION_STEP; j++)
{
  #pragma omp parallel for
  for (int i = 0; i < BODY_COUNT; i++)
  {
      updateAcceleration(i);
      updateVelocity(i);
      updatePosition(i);
  }
}
```

Algorithm 2. Parallel OpenMp implementation of the N body problem

Parallel Cuda Implementation

The CUDA implementation of the N-body problem takes advantage of the GPU's ability to handle thousands of threads concurrently, making it well-suited for computations involving large numbers of particles. The primary goal is to reduce computation time by exploiting the parallel nature of GPUs.

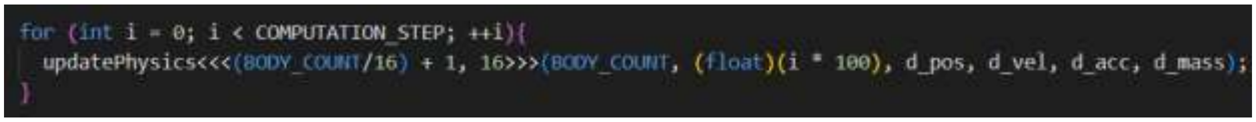
Key Strategies in CUDA:

- **Grid and Block Decomposition**
  - The problem space is divided into a grid of thread blocks, each containing multiple threads.
  - Each thread within a block is responsible for computing the forces, velocities, and positions for a subset of bodies.
- **Force Calculation with CUDA Kernels**
  - The updatePhysics kernel is launched with a grid configuration that ensures each body is processed by a separate thread.
  - Each thread calculates the net force acting on its assigned body by iterating over all other bodies.
- **Memory Management and Data Transfer**
  - Memory for positions, velocities, accelerations, and masses of bodies is allocated on both the host (CPU) and the device (GPU).
  - Data is transferred from the host to the device before computation and back to the host after computation.
  - Efficient memory access patterns and coalesced memory access are used to optimize performance.
- **Parallelism and Synchronization:**
  - Each thread operates independently, allowing for massive parallelism.

  - Synchronization within blocks is minimized by ensuring that each thread works on separate data.

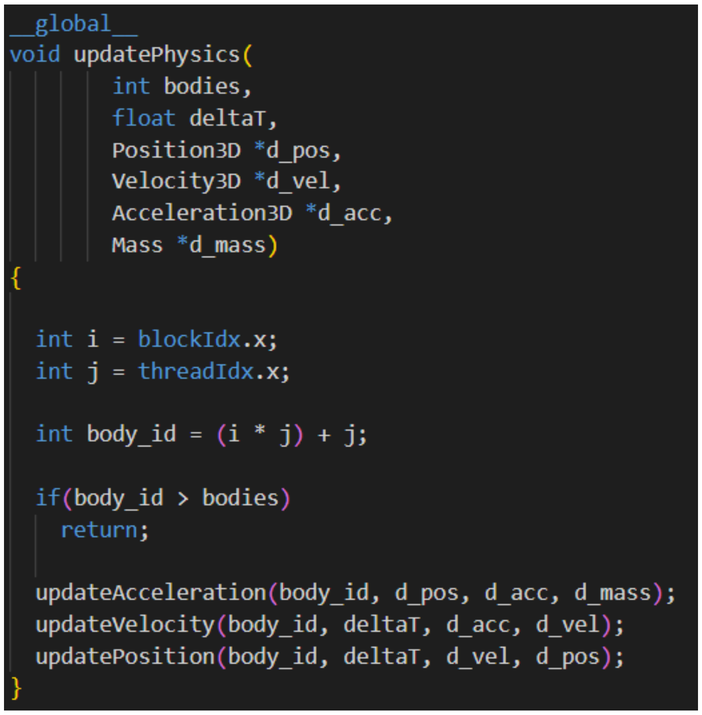
CUDA Implementation Functions:

- **Host Functions:**
  - compute: Manages memory allocation and data transfer, and launches the updatePhysics kernel iteratively to simulate multiple time steps.
- **Kernel Function:**
  - updatePhysics: A CUDA kernel that calls the above device functions for each body. Each thread computes the physics for a single body.
- **Device Function:**
  - updateAcceleration: Computes the gravitational forces and updates the acceleration of a body.
  - updateVelocity: Updates the velocity of a body based on its acceleration.
  - updatePosition: Updates the position of a body based on its velocity.



```
for (int i = 0; i < COMPUTATION_STEP; ++i){
  updatePhysics<<<(BODY_COUNT/16) + 1, 16>>>(BODY_COUNT, (float)(i * 100), d_pos, d_vel, d_acc, d_mass);
}
```

Algorithm 3. Host Function of the parallel Cuda implementation of the N body problem



```
__global__
void updatePhysics(
        int bodies,
        float deltaT,
        Position3D *d_pos,
        Velocity3D *d_vel,
        Acceleration3D *d_acc,
        Mass *d_mass)
{

  int i = blockIdx.x;
  int j = threadIdx.x;

  int body_id = (i * j) + j;

  if(body_id > bodies)
    return;

  updateAcceleration(body_id, d_pos, d_acc, d_mass);
  updateVelocity(body_id, deltaT, d_acc, d_vel);
  updatePosition(body_id, deltaT, d_vel, d_pos);
}
```

Algorithm 4. Kernel Function of the parallel Cuda implementation of the N body problem

Tiling: Tiling is a common optimization technique in parallel computing, where data is divided into smaller chunks (tiles) that fit into faster memory (like shared memory in CUDA) to reduce memory access times. However, tiling is not particularly effective for the N-body problem due to its specific computational characteristics:

1. All-to-All Interaction:
- In the N-body problem, each body interacts with every other body, resulting in an all-to-all communication pattern. This means that each thread needs to access data from all other bodies to compute the forces accurately. Tiling is most effective when there's a way to limit data access to a small subset of data (a tile), but in the N-body problem, each body needs access to the entire dataset.

2. High Memory Access Overhead:
- Even if you use tiling, each tile still needs to load data from global memory multiple times because each body needs data from all other bodies. This results in high memory access overhead, diminishing the benefits of tiling.

3. Complex Data Dependencies:
- The interactions between bodies create complex data dependencies, making it difficult to organize data into independent tiles that can be processed in parallel without frequent synchronization and data exchange between tiles.

4. Shared Memory Limitations:
- The amount of shared memory available per block is limited. Given that each thread requires data from all bodies, it's challenging to fit all the required data into the available shared memory. This limitation reduces the effectiveness of tiling, as only a small portion of the dataset can be loaded into shared memory at a time.

5. Synchronization Overhead:
- Implementing tiling in the N-body problem would require frequent synchronization between threads to ensure that all necessary data is available when needed. This synchronization can introduce significant overhead, negating the performance benefits of using shared memory.

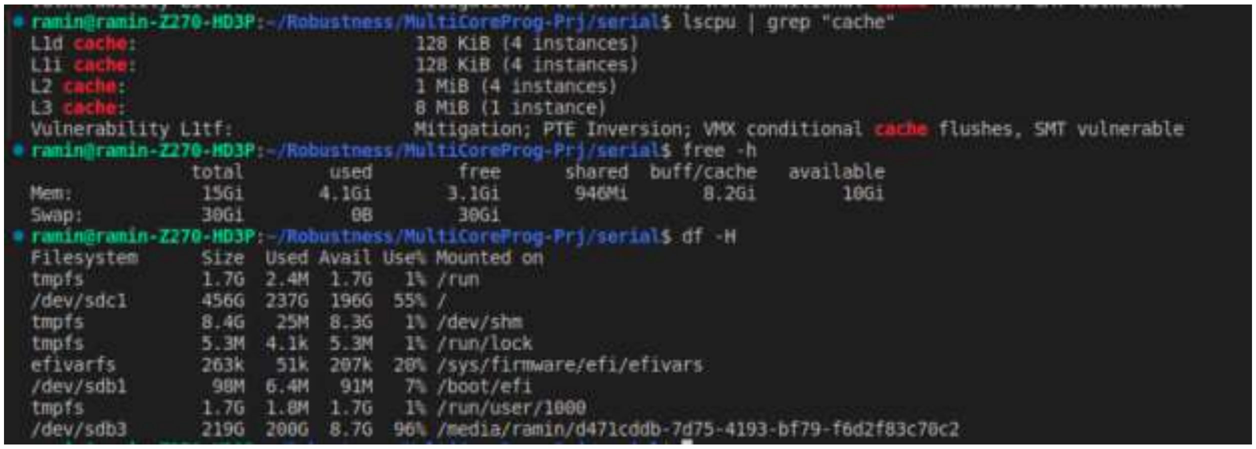
Experimental results

Below are the results of the serial and parallel versions of the algorithm with different variations. The time step for the simulation is set to 100 for all experiments.

Table 1. Serial version with different number of bodies

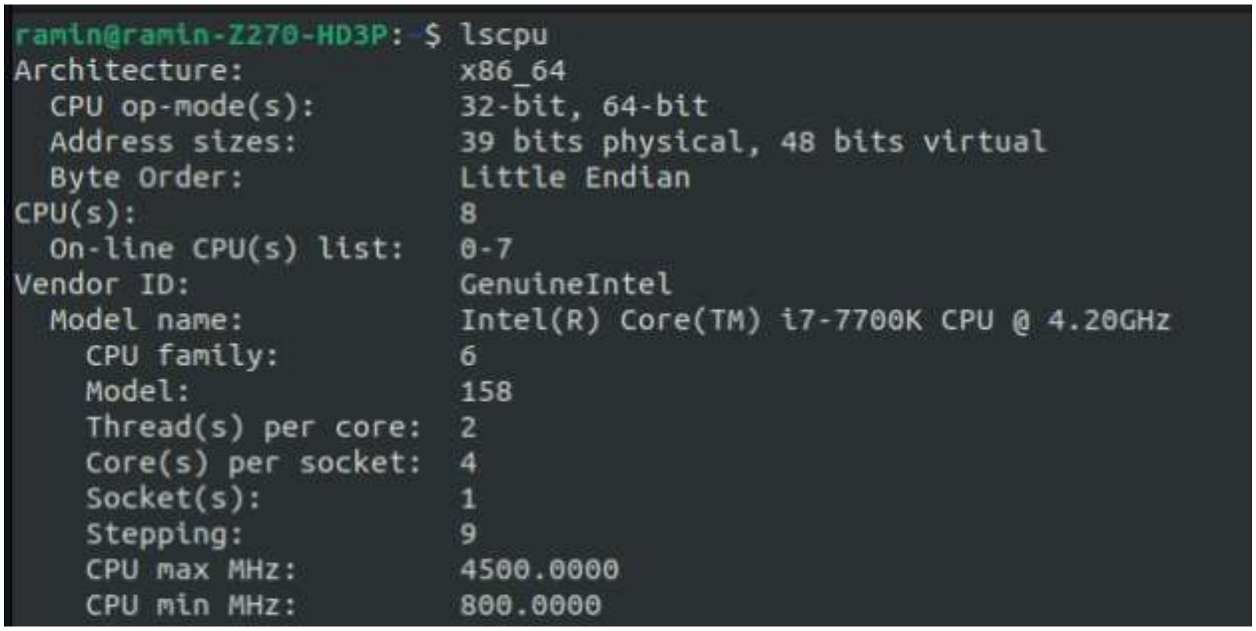
Experiment	Number of Bodies	Size (kB)	Time (ms)
Serial	600	65	1,082.197
Serial	960	103	2,766.315
Serial	1440	153	6,229.803
Serial	2880	306	24,952.521
Serial	5760	610	99,625.237
Serial	11520	1247	402,476.433
Serial	23040	2495	1608,650.585 ≈ 27 min

Table 1 represents the results of running the serial implementation of the code on a different number of bodies. Based on these results, the computation time increases as the number of bodies grows.



```
ramin@ramin-Z270-MD3P:~/Robustness/MultiCoreProg-Prj/serial$ lscpu | grep "cache"
  L1d cache:                          128 KiB (4 instances)
  L1i cache:                          128 KiB (4 instances)
  L2 cache:                           1 MiB (4 instances)
  L3 cache:                           8 MiB (1 instance)
  Vulnerability L1tf:                 Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
ramin@ramin-Z270-MD3P:~/Robustness/MultiCoreProg-Prj/serial$ free -h
                 total        used        free      shared  buff/cache   available
  Mem:            15Gi       4.1Gi       3.1Gi       946Mi       8.2Gi        10Gi
  Swap:           30Gi          0B        30Gi
ramin@ramin-Z270-MD3P:~/Robustness/MultiCoreProg-Prj/serial$ df -H
  Filesystem      Size  Used Avail Use% Mounted on
  tmpfs           1.7G  2.4M  1.7G   1% /run
  /dev/sdc1       456G  237G  196G  55% /
  tmpfs           8.4G   25M  8.3G   1% /dev/shm
  tmpfs           5.3M  4.1k  5.3M   1% /run/lock
  efivarfs        263k   51k  207k  20% /sys/firmware/efi/efivars
  /dev/sdb1        98M  6.4M   91M   7% /boot/efi
  tmpfs           1.7G  1.8M  1.7G   1% /run/user/1000
  /dev/sdb3       219G  200G  8.7G  96% /media/ramin/d471cddb-7d75-4193-bf79-f6d2f83c70c2
```

Figure 4. System properties



```
ramin@ramin-Z270-HD3P:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
    CPU family:          6
    Model:               158
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            9
    CPU max MHz:         4500.0000
    CPU min MHz:         800.0000
```

Figure 5. CPU architecture

Table 2. Parallel OpenMP version with different number of threads (# body = 600)

Experiment	Number of Threads	Number of Bodies	Time (ms)
OMP	2	600	541.575934
OMP	3	600	373.733389
OMP	4	600	302.805393
OMP	7	600	261.88397
OMP	15	600	249.817228
OMP	16	600	**238.701163**
OMP	20	600	279.400413
OMP	30	600	250.199123
OMP	32	600	252.246021

Table 2 represents the results of running the parallel OpenMP implementation of the code on a different number of threads. The number of bodies is fixed to 600. The best computation time for 600 bodies is 238 ms, which is reached with 16 threads.
According to the results, the computation time decreases as the number of threads increases. This happens until we reach 16 threads. Afterwards, increasing the number of threads increases the computation time.
This behavior is quite common in parallel computing and can be attributed to several factors:

- Overhead of Thread Management
  As the number of threads increases, the overhead associated with managing these threads

also increases. This includes the time taken to create, destroy, and synchronize threads. Beyond a certain point, the overhead can outweigh the benefits of parallelization.

- Resource Contention

With more threads, there is increased contention for shared resources such as memory bandwidth, cache, and I/O. This contention can lead to bottlenecks, reducing the efficiency of parallel execution.

- False Sharing

When multiple threads modify variables that reside on the same cache line, cache coherence protocols can cause delays. This issue, known as false sharing, becomes more pronounced with more threads.

- Non-Uniform Memory Access (NUMA)

In systems with NUMA architecture, memory access times vary depending on the memory location relative to the processor. More threads can lead to less efficient memory access patterns, especially if threads are spread across multiple NUMA nodes.

- Diminishing Returns

The speedup from parallelization follows Amdahl's Law, which states that the maximum improvement to an overall system when only part of the system is improved is limited. If the serial portion of your code is significant, adding more threads yields diminishing returns.

- System Limits

The hardware itself may have limitations. For example, a CPU with 16 physical cores (without Hyper-Threading) might not benefit from more than 16 threads, as additional threads would be competing for the same physical cores.

Table 3 represents the results of running the parallel OpenMP implementation of the code with different setups. For each number of bodies, we run the program with a different number of threads. After choosing the best number of threads based on the computation time, we repeat that experiment with binding. The best computation time for each number of bodies is represented in bold.

Table 4 is the final comparison between the serial and the parallel OpenMP version. For each number of bodies, the setup for the parallel version is chosen from table 3.
In conclusion, the parallel OpenMP version can make the serial version more than 4.5 times faster.

Table 3. Parallel OpenMP version with different setups

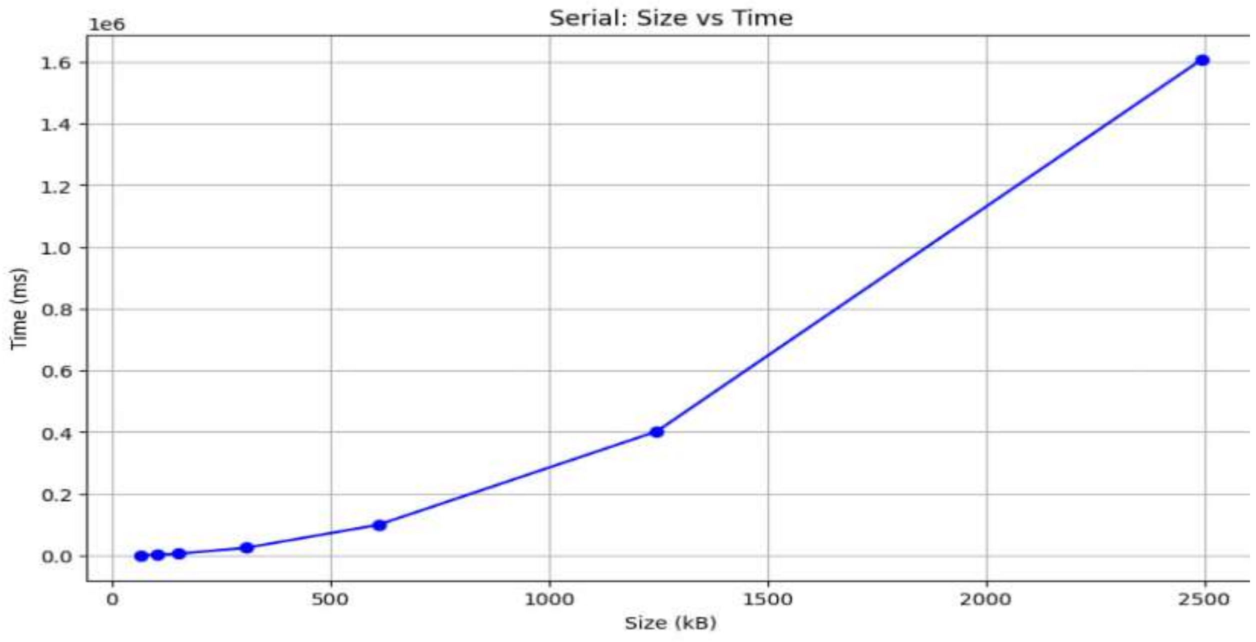
Experiment	setup	Number of bodies	Time (ms)
OMP	7 threads	600	261.883
OMP	16 threads	600	**238.701**
OMP	32 threads	600	252.246
OMP	16 threads - binding	600	255.919

OMP	7 threads	2880	6,022.407
OMP	16 threads	2880	5,447.353
OMP	32 threads	2880	5,605.822
OMP	16 threads - binding	2880	**5,287.960**
OMP	7 threads	11520	94,969.562
OMP	16 threads	11520	85,055.596
OMP	32 threads	11520	84,112.597
OMP	32 threads - binding	11520	84,631.150
OMP	Dynamic Schedule 32 Threads– 1 Chunk	11520	84,676.183
OMP	Dynamic Schedule 32 Threads – 2 Chunk	11520	84,472.529
OMP	Dynamic Schedule 32 Threads – 4 Chunk	11520	84,706.426
OMP	Dynamic Schedule 32 Threads – 10 Chunk	11520	84,167.087
OMP	Dynamic Schedule 32 Threads – 20 Chunk	11520	**83,600.308**
OMP	Dynamic Schedule 32 Threads – 30 Chunk	11520	84,165.227
OMP	Dynamic Schedule 32 Threads – 40 Chunk	11520	83,659.974
OMP	Dynamic Schedule 32 Threads – 80 Chunk	11520	83,955.684
OMP	Dynamic Schedule 32 Threads – 128 Chunk	11520	84,051.235
OMP	Dynamic Schedule 32 Threads – 256 Chunk	11520	85,165.031
OMP	7 threads	23040	369,521.086
OMP	16 threads	23040	335,518.742
OMP	32 threads	23040	336,101.978
OMP	Dynamic Schedule 16 Threads – 2 Chunk	23040	**333,118.766**

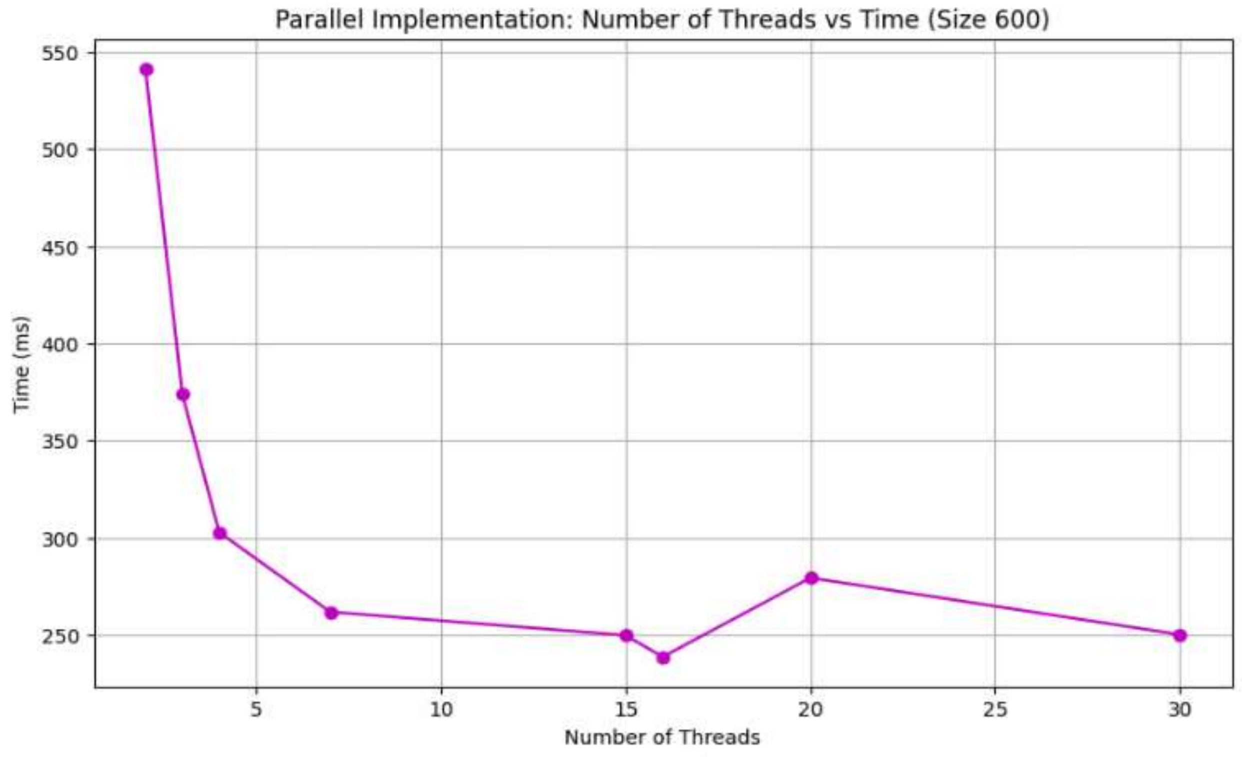
Table 4. Serial vs. parallel OpenMP

Number of Bodies	Size (kB)	Serial Time (ms)	Best OMP Time (ms)	Speed Up
600	65	1,082.197	238.701	4.533
2880	306	24,952.521	5,287.960	4.718
11520	1247	402,476.433	83,600.308	4.784
23040	2495	1608,650.585	335,518.742	4.794

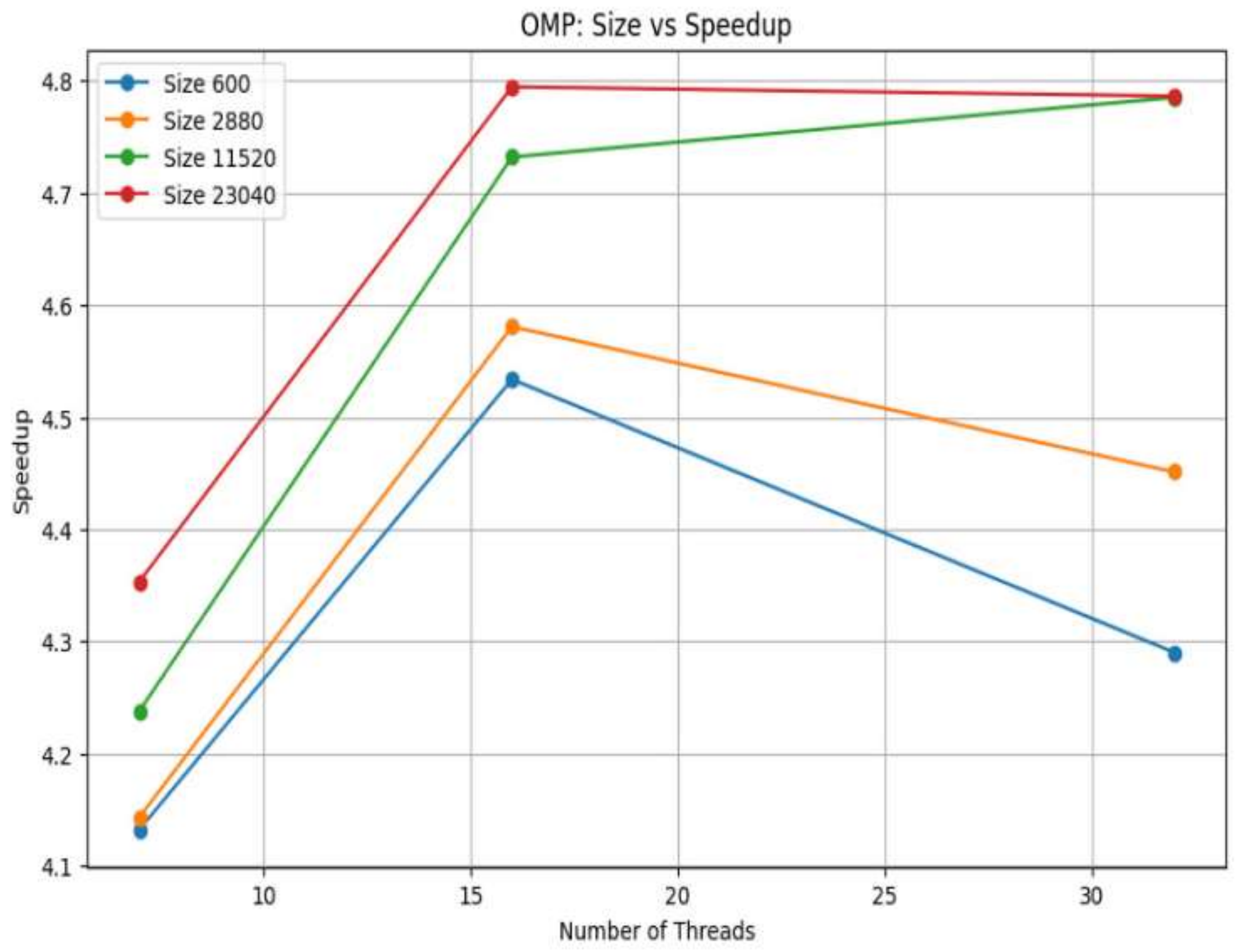
Plot 1.



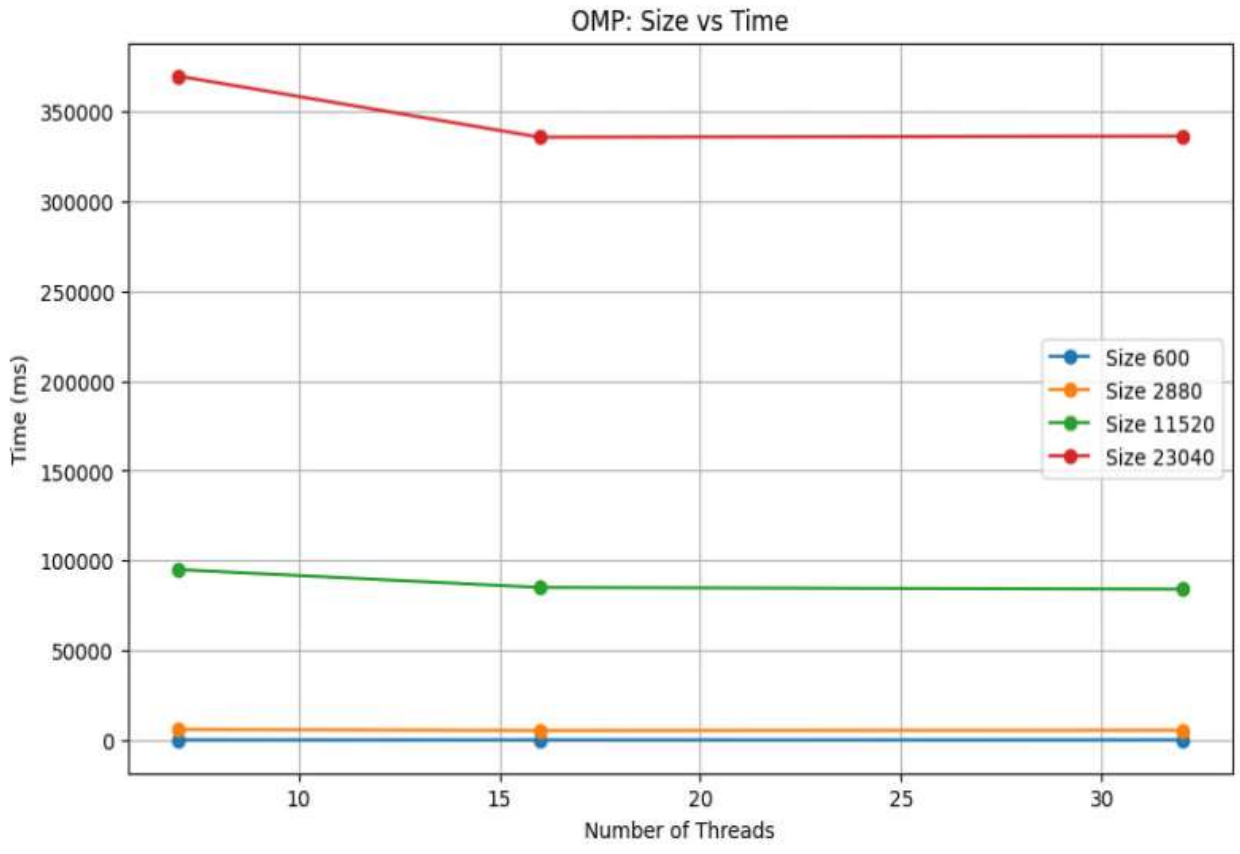
Plot 2.


Parallel Implementation: Number of Threads vs Time (Size 600)
Time (ms)
550
500
450
400
350
300
250
5
10
15
20
25
30
Number of Threads

Plot 3.

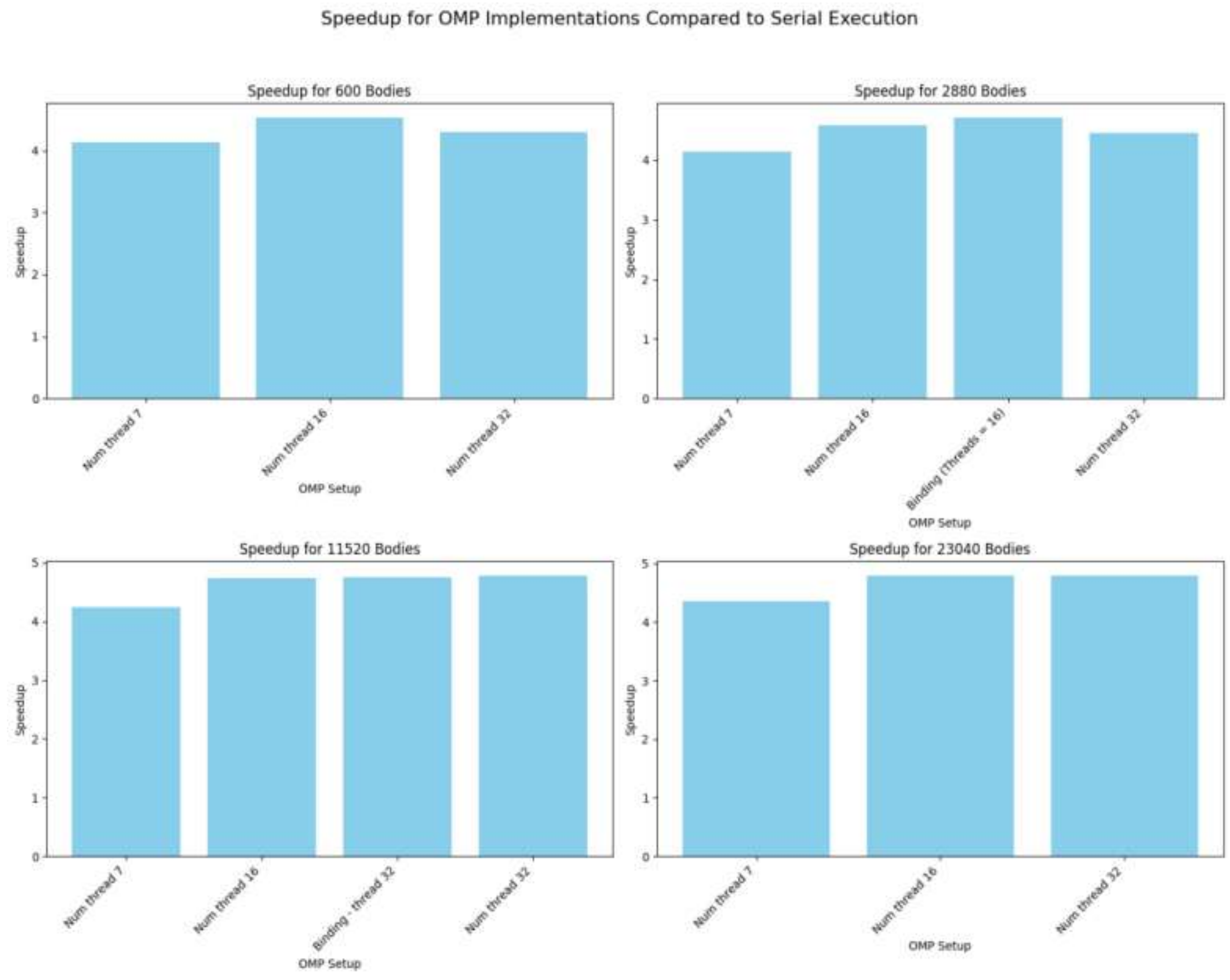

OMP: Size vs Speedup
Size 600
Size 2880
Size 11520
Size 23040
Speedup
Number of Threads
4.1
4.2
4.3
4.4
4.5
4.6
4.7
4.8
10
15
20
25
30

Plot 4.

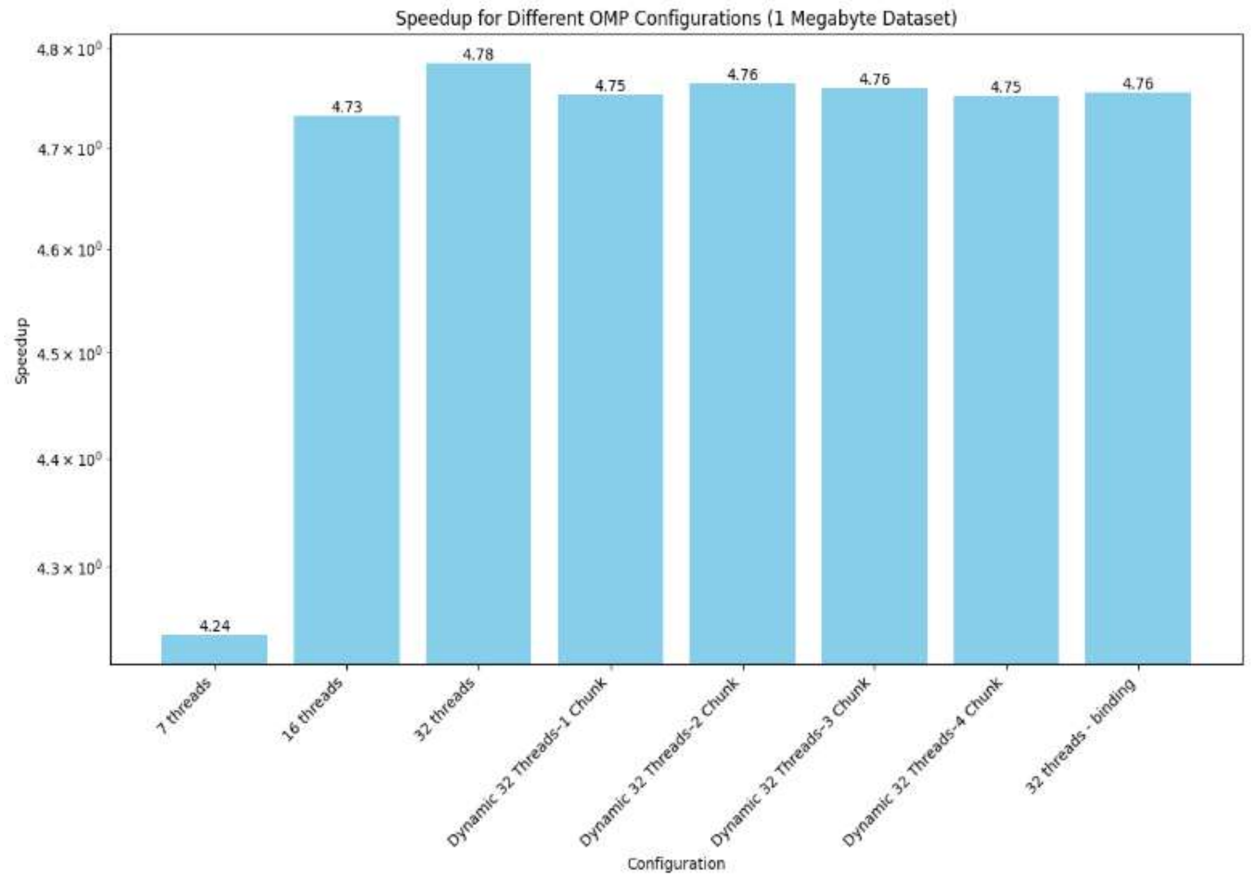


OMP: Size vs Time
Time (ms)
Number of Threads
Size 600
Size 2880
Size 11520
Size 23040
0
50000
100000
150000
200000
250000
300000
350000
10
15
20
25
30

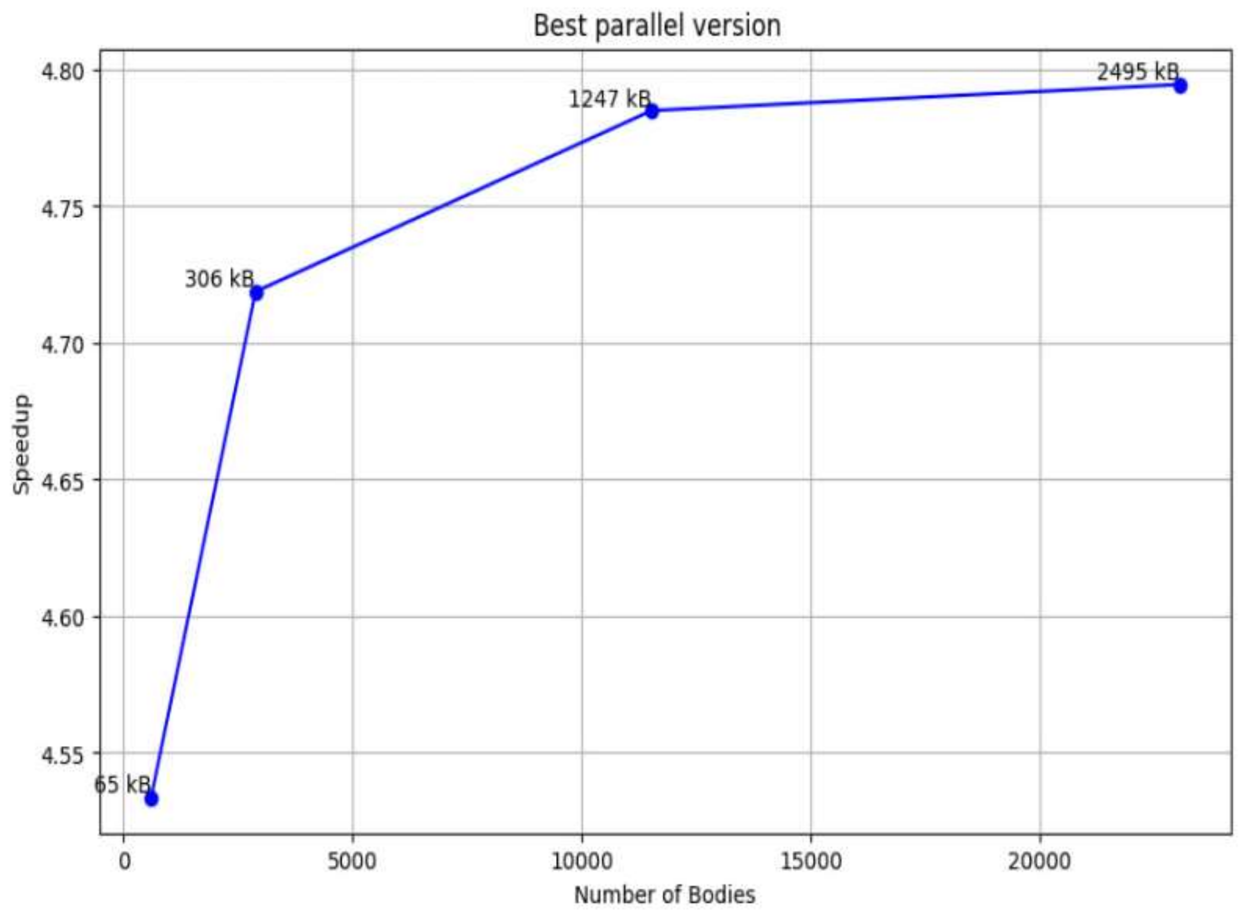
Plot 5.

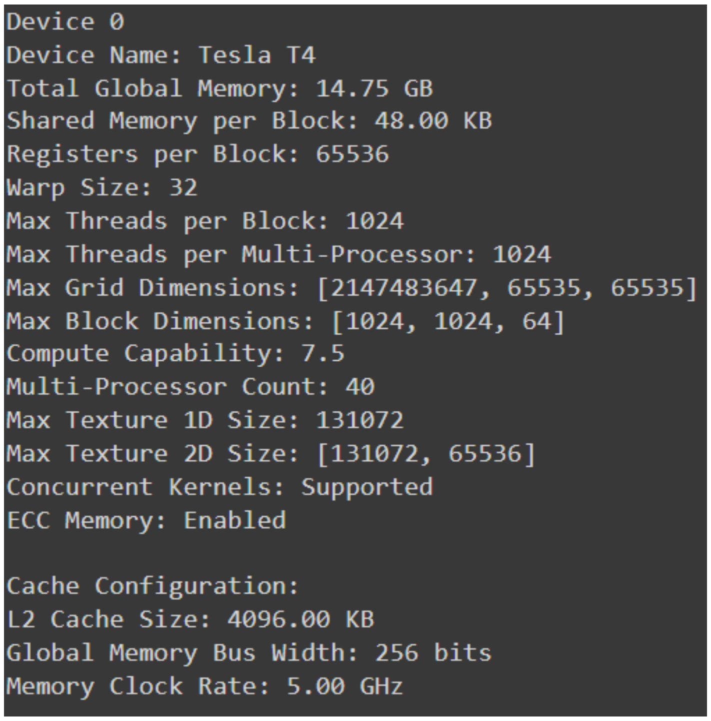

Speedup for OMP Implementations Compared to Serial Execution
Speedup for 600 Bodies
Speedup for 2880 Bodies
Speedup for 11520 Bodies
Speedup for 23040 Bodies
Speedup
OMP Setup
Num thread 7
Num thread 16
Num thread 32
Binding (Threads = 16)
Binding - thread 32

Plot 6.


Speedup for Different OMP Configurations (1 Megabyte Dataset)
Speedup
4.8×10^0
4.7×10^0
4.6×10^0
4.5×10^0
4.4×10^0
4.3×10^0
4.24
4.73
4.78
4.75
4.76
4.76
4.75
4.76
7 threads
16 threads
32 threads
Dynamic 32 Threads-1 Chunk
Dynamic 32 Threads-2 Chunk
Dynamic 32 Threads-3 Chunk
Dynamic 32 Threads-4 Chunk
32 threads - binding
Configuration

Plot 7.


Best parallel version
Speedup
Number of Bodies
4.80
4.75
4.70
4.65
4.60
4.55
0
5000
10000
15000
20000
65 kB
306 kB
1247 kB
2495 kB



```
Device 0
Device Name: Tesla T4
Total Global Memory: 14.75 GB
Shared Memory per Block: 48.00 KB
Registers per Block: 65536
Warp Size: 32
Max Threads per Block: 1024
Max Threads per Multi-Processor: 1024
Max Grid Dimensions: [2147483647, 65535, 65535]
Max Block Dimensions: [1024, 1024, 64]
Compute Capability: 7.5
Multi-Processor Count: 40
Max Texture 1D Size: 131072
Max Texture 2D Size: [131072, 65536]
Concurrent Kernels: Supported
ECC Memory: Enabled

Cache Configuration:
L2 Cache Size: 4096.00 KB
Global Memory Bus Width: 256 bits
Memory Clock Rate: 5.00 GHz
```

Figure 6. GPU Architecture

Table 5. Parallel CUDA version with different setups

Experiment	setup	Number of bodies	Time (ms)
CUDA	Block Size 16	600	673.287
CUDA	Block Size 32	600	**511.666**
CUDA	Block Size 64	600	555.038
CUDA	Block Size 128	600	596.572
CUDA	Block Size 16	2880	2,484.351
CUDA	Block Size 32	2880	**1,627.396**
CUDA	Block Size 64	2880	1,904.166
CUDA	Block Size 128	2880	1,864.272
CUDA	Block Size 16	11520	30,405.216
CUDA	Block Size 32	11520	**15,161.601**
CUDA	Block Size 64	11520	15,208.548
CUDA	Block Size 128	11520	18,241.501

CUDA	Block Size 16	23040	115,496.223
CUDA	Block Size 32	23040	61,749.649
CUDA	Block Size 64	23040	**60,048.935**
CUDA	Block Size 128	23040	60,095.977

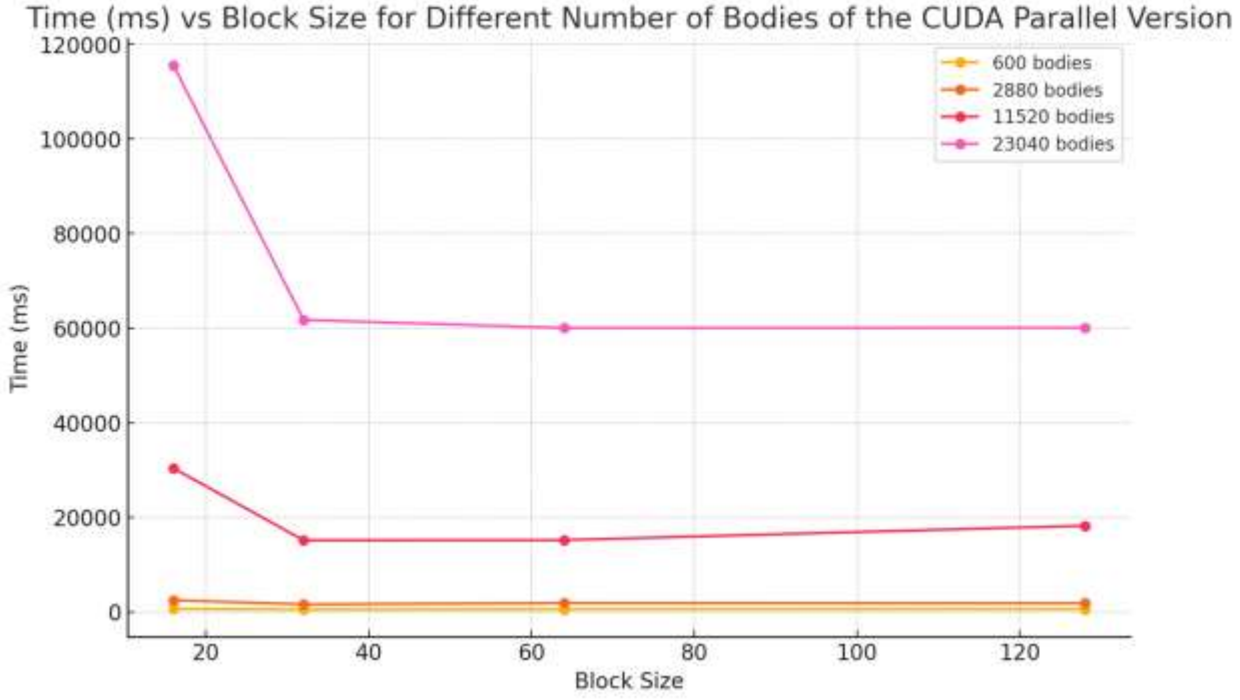
Table 6: Serial vs. OMP vs. CUDA

Number of Bodies	Size (kB)	Serial Time (ms)	Best CUDA Time (ms)	Serial Speed Up	OMP Speed Up
600	65	1,082.197	511.666	2.115	0.466
2880	306	24,952.521	1,627.396	15.332	3.249
11520	1247	402,476.433	15,161.601	26.545	5.513
23040	2495	1,608,650.585	60,048.935	26.788	5.587

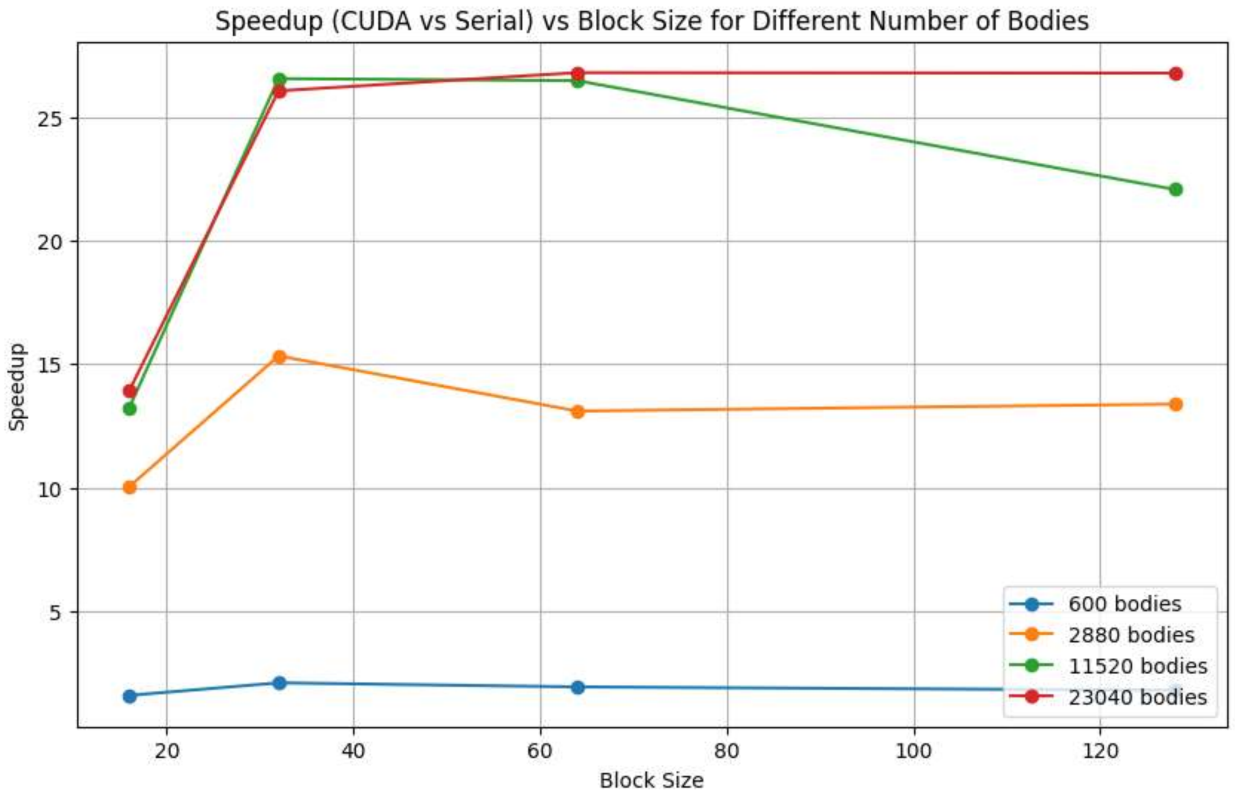
Table 5 represents the results of running the parallel CUDA implementation of the code with different setups. For each number of bodies, we ran the program with different block sizes such as 16, 32, 64, and 128. The best computation time for each number of bodies is represented in bold. According to the results, for 600, 2880, and 11520 bodies, blocks with a size of 32 have the best performance among the other sizes. For 23040 bodies, kernels with blocks that contain 64 threads have the best computation time among the other sizes. So it can be concluded that by increasing the number of bodies, larger block sizes should be chosen.

Table 4 is the final comparison between the serial, OpenMP parallel, and CUDA parallel versions. For each number of bodies, the best setup of the CUDA parallel version is chosen from Table 5. The speedup of the CUDA version compared to both the serial and OpenMP versions has been calculated for each number of bodies. According to the results, for every number of bodies, the speedups of the CUDA version compared to the serial version are more than 1, indicating that the performance of the CUDA version is much better than the serial version. The computation time of the CUDA version is also better than the computation time of the OpenMP version except for the 600 bodies. Another point that can be concluded from the results is that by increasing the number of bodies, the speedup of the CUDA version compared to both serial and OpenMP versions increases, and the CUDA version has better efficiency for larger datasets.

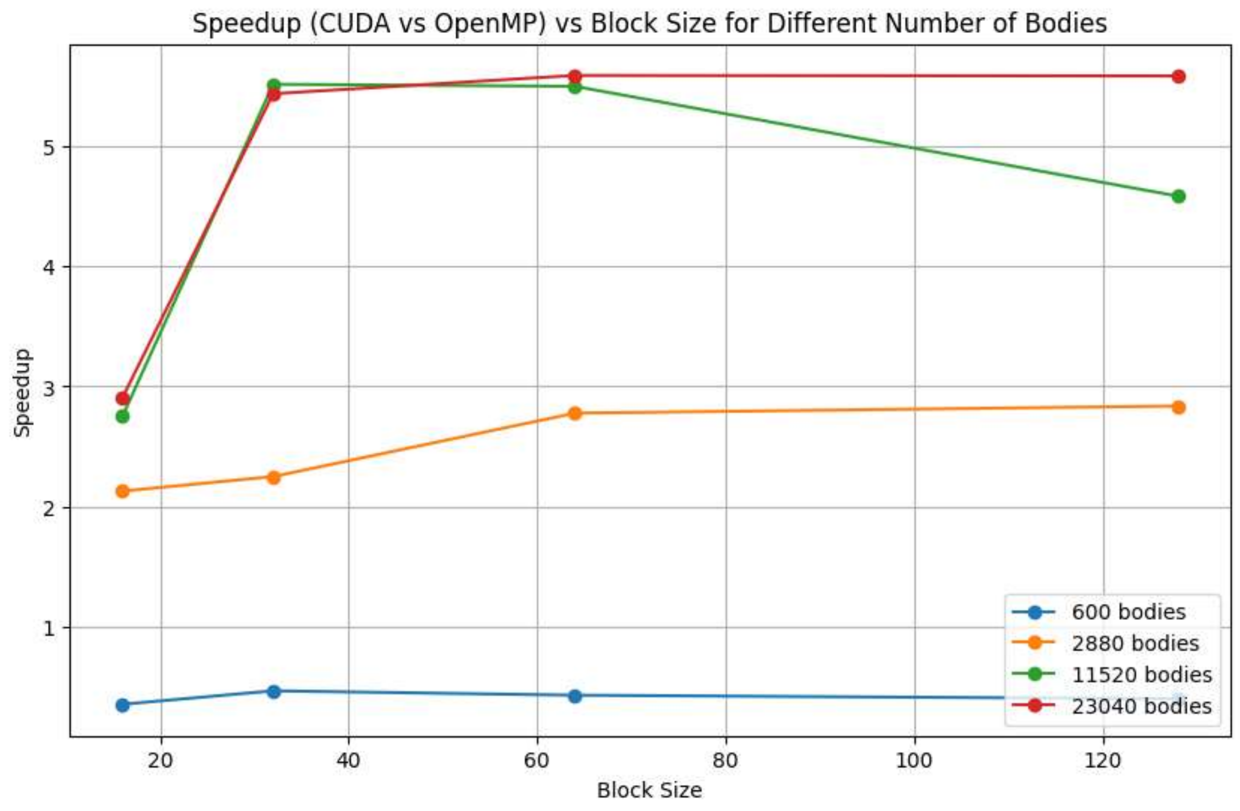
Plot 8.


Time (ms) vs Block Size for Different Number of Bodies of the CUDA Parallel Version
Time (ms)
Block Size
600 bodies
2880 bodies
11520 bodies
23040 bodies
0
20000
40000
60000
80000
100000
120000
20
40
60
80
100
120

Plot 9.

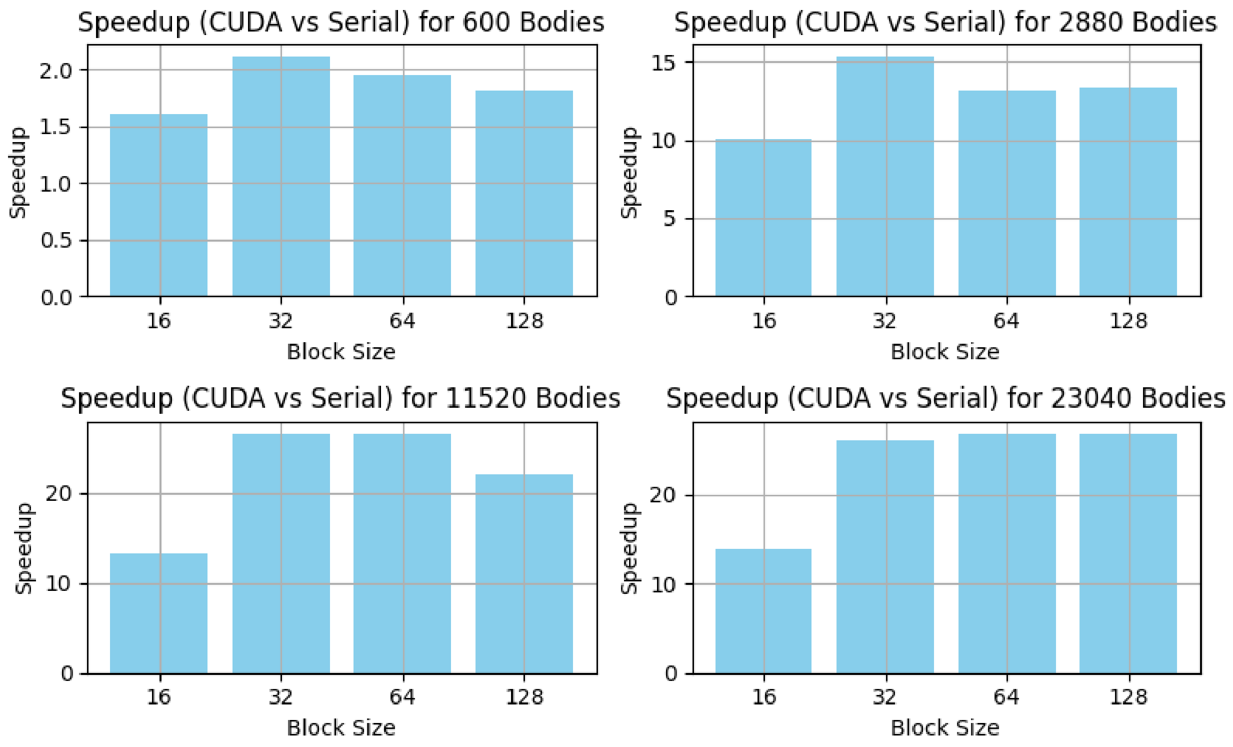

Speedup (CUDA vs Serial) vs Block Size for Different Number of Bodies
Speedup
Block Size
600 bodies
2880 bodies
11520 bodies
23040 bodies
5
10
15
20
25
20
40
60
80
100
120

Plot 10.

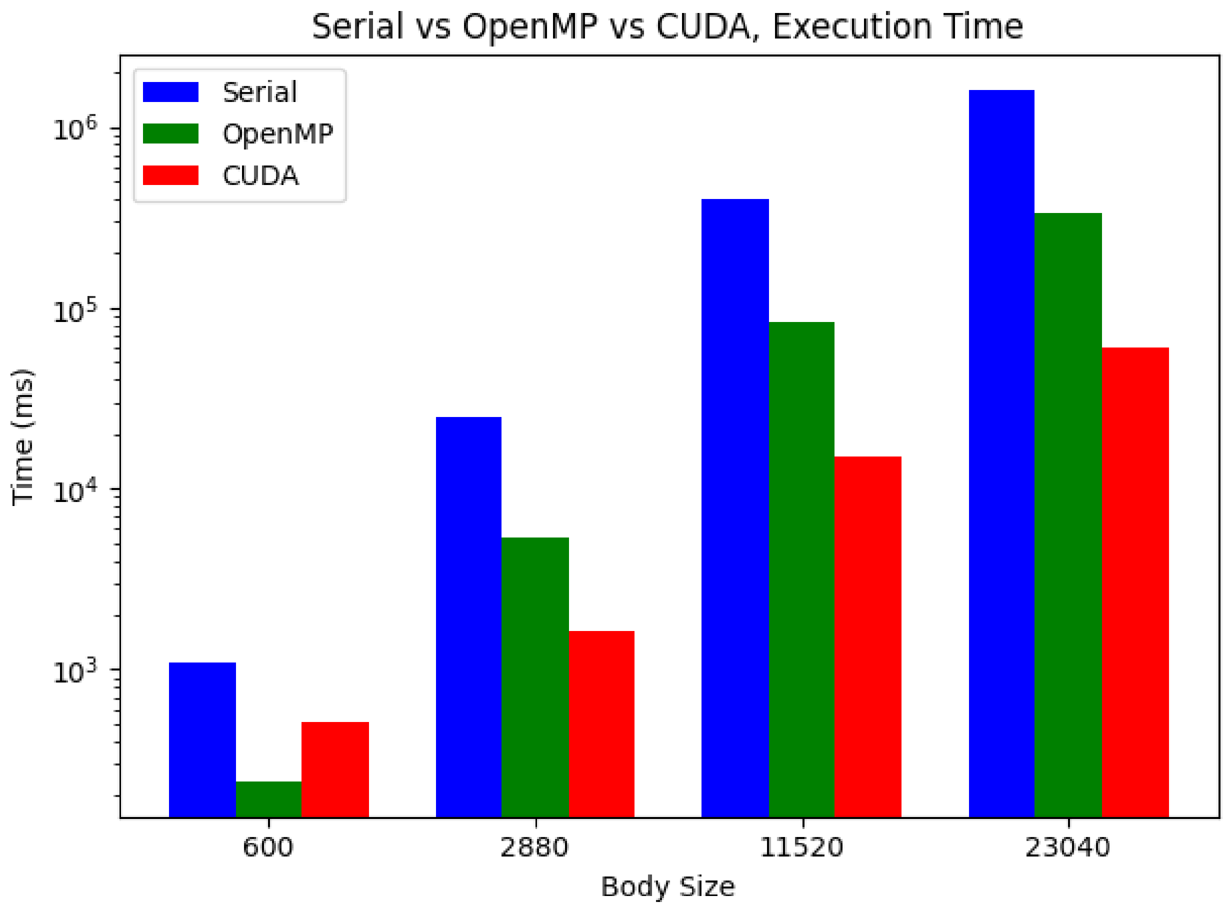


Speedup (CUDA vs OpenMP) vs Block Size for Different Number of Bodies
Speedup
Block Size
600 bodies
2880 bodies
11520 bodies
23040 bodies

Plot 11.


Speedup (CUDA vs Serial) for 600 Bodies
Speedup
Block Size
16
32
64
128
0.0
0.5
1.0
1.5
2.0
Speedup (CUDA vs Serial) for 2880 Bodies
0
5
10
15
Speedup (CUDA vs Serial) for 11520 Bodies
0
10
20
Speedup (CUDA vs Serial) for 23040 Bodies

Plot 12.



In conclusion, the parallel CUDA version can make the serial version more than 26.5 times faster (for large datasets), and its computation time is much better than the OpenMP parallel version. According to Plot 12, the OpenMP version is a good solution for small datasets and can significantly increase efficiency. However, for larger datasets, the CUDA version achieves better computation times than the OpenMP version and is the best solution for achieving parallelism.

Running the code

- To run the Serial version

```
$ cd serial/
$ g++ -o nbody_serial n_body.cpp -lm
$ ./nbody_serial
```

- To run the openMP version:

Pre-requisites: CUDA, openMP and openGL libraries should be present

```
$ cd openMp/
$ g++ -o nbody_omp n_body_omp.cpp -fopenmp
$ ./nbody_omp
```

- To run the CUDA version:

```
$ cd cuda/
$ nvcc -o nbody_cuda n_body_cuda.cu
$ ./nbody_cuda
```