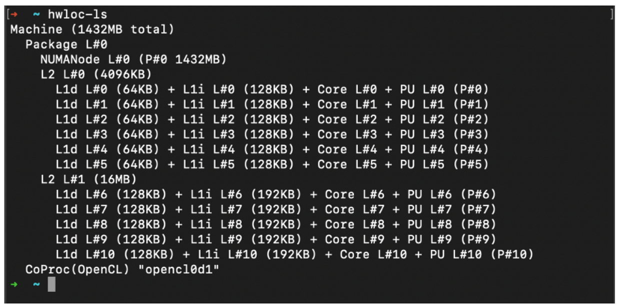
Parallel Computation in N-Particle Electrostatic Simulations using Pthreads and MPI

Xiangyu Liu (1006743179)

October 31, 2024

1 Hardware Resources

The hardware used for this project is a MacBook equipped with an Apple Silicon processor. It features a single CPU package with 11 cores organized in two clusters. Cores 0-5 are efficiency cores, each with a dedicated 64KB L1 data cache and 128KB L1 instruction cache, and they share a 4096KB L2 cache. Cores 6-10 are performance cores, each with a 128KB L1 data cache and 192KB L1 instruction cache, sharing a larger 16MB L2 cache. There is no L3 cache in this architecture. Additionally, an integrated OpenCL-compatible coprocessor is available for parallel processing tasks. The total system memory available is 1.4GB.



```
[➜  ~ hwloc-ls                                                  ]
Machine (1432MB total)
  Package L#0
    NUMANode L#0 (P#0 1432MB)
    L2 L#0 (4096KB)
      L1d L#0 (64KB) + L1i L#0 (128KB) + Core L#0 + PU L#0 (P#0)
      L1d L#1 (64KB) + L1i L#1 (128KB) + Core L#1 + PU L#1 (P#1)
      L1d L#2 (64KB) + L1i L#2 (128KB) + Core L#2 + PU L#2 (P#2)
      L1d L#3 (64KB) + L1i L#3 (128KB) + Core L#3 + PU L#3 (P#3)
      L1d L#4 (64KB) + L1i L#4 (128KB) + Core L#4 + PU L#4 (P#4)
      L1d L#5 (64KB) + L1i L#5 (128KB) + Core L#5 + PU L#5 (P#5)
    L2 L#1 (16MB)
      L1d L#6 (128KB) + L1i L#6 (192KB) + Core L#6 + PU L#6 (P#6)
      L1d L#7 (128KB) + L1i L#7 (192KB) + Core L#7 + PU L#7 (P#7)
      L1d L#8 (128KB) + L1i L#8 (192KB) + Core L#8 + PU L#8 (P#8)
      L1d L#9 (128KB) + L1i L#9 (192KB) + Core L#9 + PU L#9 (P#9)
      L1d L#10 (128KB) + L1i L#10 (192KB) + Core L#10 + PU L#10 (P#10)
  CoProc(OpenCL) "opencl0d1"
➜  ~ 
```

Figure 1: Hardware layout of the Apple Silicon processor as reported by `hwloc-ls`.

2 Program Architecture

This section describes the architectural choices and partitioning strategies implemented for each computation mode. I employ a spatial partitioning approach where particles are divided based on a grid structure. The grid is organized by a specified cutoff radius, ensuring that each particle only interacts with neighboring particles within this radius. In Modes 2 and 3, parallelism is introduced by dividing tasks among multiple threads and processes, effectively distributing the workload across hardware resources.

2.1 Mode 1: Sequential Computation

In Mode 1, all particles are processed in a single thread without parallelism. The particles are divided into a grid, with each grid cell spanning the specified cutoff radius. For each particle, only the particles within the current cell and its 8 neighboring cells need to be considered for interaction calculations. This approach significantly reduces the number of pairwise calculations compared to checking every other particle in the system.

Each particle is assigned to a grid cell based on its position, calculated using the cutoff radius, with a hash map used to store each grid cell and its associated particles for efficient access. For example, consider a particle with coordinates $(x, y) = (6, 9)$ and a cutoff radius of 2 units. The grid cell for this particle is determined by dividing each coordinate by the cutoff radius and taking the floor of the result. Thus, the particle is assigned to the grid cell $\left(\left\lfloor \frac{6}{2} \right\rfloor, \left\lfloor \frac{9}{2} \right\rfloor\right) = (3, 4)$. This hash map structure enables quick access to particles within relevant neighboring cells.

A single loop iterates over all particles to compute forces. Since there is no parallelism, this computation executes serially, with only one particle's interactions processed at a time.

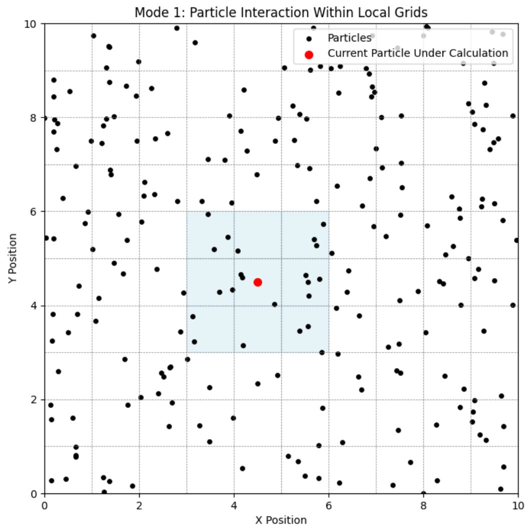


Figure 2: Mode 1: Particle Interaction Within Local Grids

2.2 Mode 2: Evenly-Distributed Parallel Computation

In Mode 2, multi-threading is introduced to divide the workload across multiple threads, utilizing hardware concurrency. This mode does not involve multiple processes, so all threads share access to the full set of particles.

The particles are split evenly among threads. For instance, if there are 1000 particles and 4 threads, each thread handles 250 particles. Specifically, thread 0 handles particles 0-249, thread 1 handles particles 250-499, thread 2 handles particles 500-749, and thread 3 handles particles 750-999.

Each thread independently processes its assigned particles. Once a thread completes its portion of work, it becomes idle until all other threads finish their respective tasks. The entire system returns when all threads have completed their calculations.

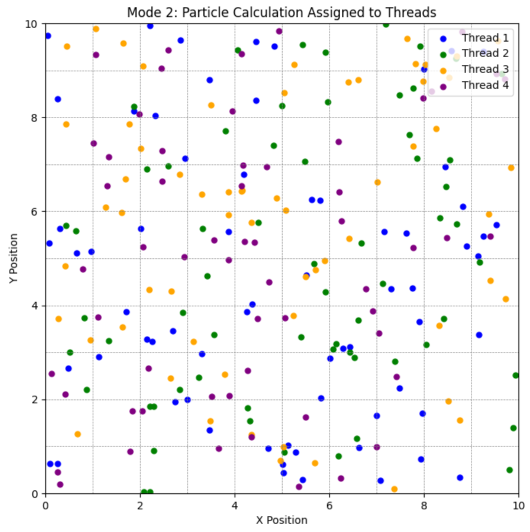


Figure 3: Mode 2: Particle Calculation Assigned to Threads

2.3 Mode 3: Load-Balanced, Leader-Based Parallel Computation

In Mode 3, both multiple processes and threads within each process are used to maximize parallelism. Each process receives a copy of all particles to ensure that boundary interactions between partitions are accounted for.

To evenly divide the full set of particles among multiple processes as required by the instruction, I first apply a similar strategy as in Mode 2. For instance, with 1000 particles and 4 processes, the particles are divided as follows: Process 0 handles particles 0-249, Process 1 handles particles 250-499, and so on. However, in Mode 3, the full set of particles is copied to each process. Each process is responsible for calculating the interactions only for its assigned subset of particles. Within each process, I further partition the particles into smaller tasks based on their grid locations. Each grid cell represents a task containing the particles within that cell. These grid-based tasks are then stored in a queue and executed by worker threads within the process. This grid-based partitioning within each process improves data locality, as particles in the same grid cell and neighboring cells are often accessed together and have a high likelihood of being reused in subsequent computations.

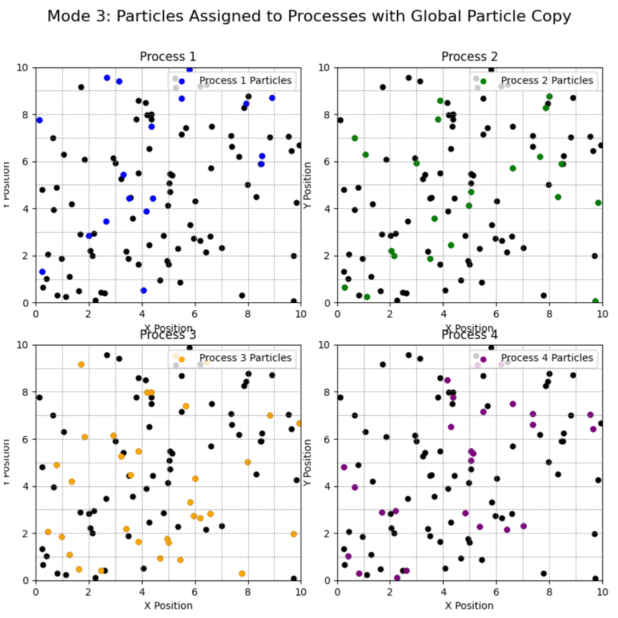


Figure 4: Mode 3: Particles Assigned to Processes with Global Particle Copy

To conclude, the workflow involves both serial and parallelized operations. Initially, reading data from the `particles.csv` file and assigning each particle to its corresponding grid based on its position are handled as serial tasks, as this setup phase requires coordinating the data structure and preparing the workload for parallel execution. Additionally, assigning tasks to each thread is a serial operation, where tasks are distributed based on the partitioning strategy. Once the setup is complete, the real computation begins, which is highly parallelized. Calculations such as computing distances between particles, determining the forces between them, summing up the net forces for each particle, and ultimately writing the computed results to `outcome.csv` are all executed in parallel across multiple threads or processes.

3 Mode 1: Sequential Computation

In this mode, the cutoff radius was varied to observe its effect on both the execution time and the error percentage (accuracy) of the computed forces. The error percentage is calculated by the following formula:

$$\text{Error Percentage} = \left(\frac{1}{N} \sum_{i=1}^{N} \frac{|F_{\text{computed},i} - F_{\text{oracle},i}|}{|F_{\text{oracle},i}|} \right) \times 100\%$$

Cutoff Radius	Time Taken (s)	Error (%)
1	1.27	99.99
10	17.45	99.24
100	202.15	151.43
1000	7.71	80.66
10000	78.03	11.07
17500	231.66	4.85
18000	247.17	4.70

Table 1: Mode 1: Sequential Computation - Time Taken and Error Percentage for Varying Cutoff Radii

As the cutoff radius increases, the error percentage generally decreases, indicating improved accuracy in force calculations due to the inclusion of interactions from particles located further away. However, the execution time also increases with larger cutoff radii, except for an anomaly at radius 1000, where a significant drop in time is observed. This anomaly suggests a tradeoff between accuracy and execution time based on the cutoff radius, where optimal performance may be achieved by balancing the radius to minimize error without excessively increasing computation time.

4 Mode 2: Evenly-Distributed Parallel Computation

In this section, I evaluate the performance of the program in Mode 2 by analyzing execution time and speedup as the number of threads increases. By setting the cutoff radius to 18000, it results in an error rate of 4.70%. Starting from 1 thread, I incrementally increase the number of threads up to 22 (twice the core count of the hardware) to observe the program's scalability and efficiency.

4.1 Execution Time and Speedup Analysis

Table 2 provides the execution time for each configuration of threads. Figure 5 illustrates the effect of increasing the number of threads on both the execution time and speedup compared to Mode 1.

Threads	Time (s)	Speedup
1	244.412	1.01x
2	123.571	2.00x
3	82.9421	2.98x
4	62.3182	3.97x
5	51.2583	4.82x
6	46.806	5.28x
7	41.6998	5.93x
8	36.9282	6.69x
9	33.8573	7.30x
10	32.2985	7.65x
11	30.9723	7.98x
12	29.8001	8.29x
13	30.1844	8.19x
14	29.8798	8.27x
15	29.524	8.37x
16	29.5036	8.37x
17	31.4519	7.86x
18	31.481	7.85x
19	30.0445	8.23x
20	29.9204	8.26x
21	29.6652	8.33x
22	30.7813	8.03x

Table 2: Execution Time and Speedup for Mode 2 with Varying Number of Threads (Cutoff Radius = 18000, Error Percentage: 4.69969%), Speedup = (mode1 / mode2)

As shown in Table 2 and Figure 5, the execution time significantly decreases as the number of threads increases, demonstrating the effectiveness of parallel computation in Mode 2. The speedup compared to Mode 1 improves notably, reaching a peak at approximately 8.37x speedup with 16 threads.

However, as the number of threads increases, the improvement in execution time begins to plateau, particularly between 12 and 22 threads. This stabilization is likely due to factors such as thread

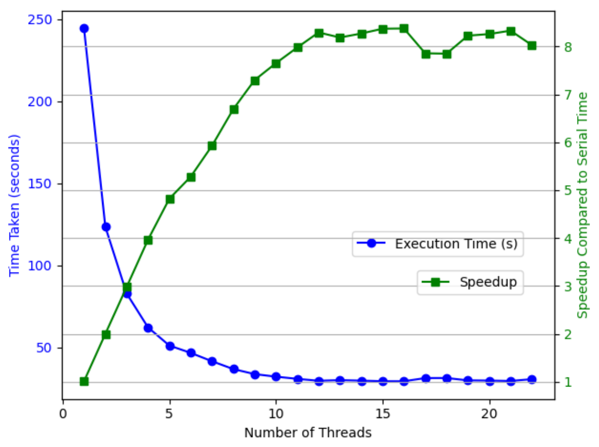


Figure 5: Execution Time and Speedup for Mode 2 with Varying Number of Threads (Cutoff Radius = 18000, Error Percentage = 4.69969%)

management overhead and contention for shared resources. Optimal performance is observed around 12-16 threads, after which additional threads provide minimal gains, with speedup even decreasing at 17 and 18 threads. This reflects diminishing returns in parallel efficiency, primarily due to the limitations of CPU resources as more threads compete for processing power.

Overall, Mode 2 achieves significant speedup compared to Mode 1 by leveraging multi-threading, showcasing the scalability of the approach up to a certain point.

4.2 Optimizing Cutoff Radius and Threads

To achieve the fastest execution time with an error margin below 5%, I experimented further with a cutoff radius of 17500, varying the number of threads from 8 to 22. This range was chosen based on previous findings with a cutoff radius of 18000, where optimal performance was observed only when the thread count exceeded 8. Table 3 presents the execution times for a cutoff radius of 17500 across different thread counts, while Figure 6 shows a comparison of execution times for both cutoff values.

As shown in Table 3 and Figure 6, the best performance was achieved with a cutoff radius of 17500 and 16 threads, yielding an execution time of 28.0981 seconds and a speedup of 8.24x over the sequential mode with the same cutoff.

The tuning process involved balancing the cut-

Threads	Time (s)	Speedup
8	37.2807	6.21x
9	35.8547	6.46x
10	32.2724	7.18x
11	32.0240	7.23x
12	31.4691	7.36x
13	28.3081	8.18x
14	32.0610	7.23x
15	28.1769	8.22x
16	28.0981	8.24x
17	28.1568	8.23x
18	28.5388	8.12x
19	28.2455	8.20x
20	31.3836	7.38x
21	32.0015	7.24x
22	32.9094	7.04x

Table 3: Execution Time and Speedup for Mode 2 with Varying Number of Threads (Cutoff Radius = 17500, Error = 4.8533%), Speedup = (mode1 / mode2)

off radius and thread count to minimize execution time while maintaining an acceptable error margin. Through experimentation, we can observe that reducing the cutoff radius to 17500 results in a slight decrease in execution time. Increasing the thread count improved performance up to 16 threads, beyond which the benefit from additional threads diminished due to factors such as contention for CPU resources.

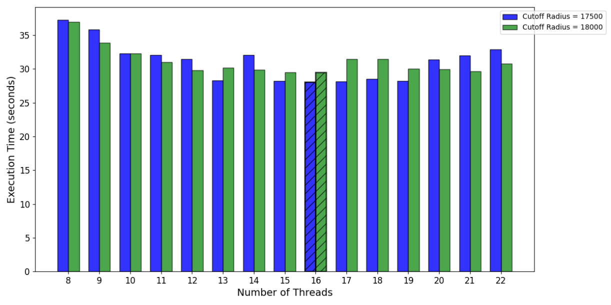


Figure 6: Comparison of Execution Times for Cutoff Radius 17500 and 18000 with Varying Thread Counts (Both Error Percentage < 5%)

5 Mode 3: Load-Balanced, Leader-Based Parallel Computation

In this section, I extend the analysis from Mode 2 by introducing multiple leader processes and worker threads. Based on prior experiments conducted in Mode 2, where a cutoff radius of 17500 yielded faster results than 18000 while maintaining an error margin below 5%, we opted to retain the cutoff value at 17500 for Mode 3.

The experiments were conducted with configurations of 1, 2, and 4 processes and varying numbers of threads. For comparison, I excluded configurations with a single process, as they are functionally similar to Mode 2, which uses only one process but multiple threads. Consequently, to assess the unique impact of Mode 3, let's focus on configurations with two or more processes. Among the configurations tested, the combination of 2 processes and 11 threads demonstrated the best performance, with an execution time of approximately 90.854 seconds, making it the optimal configuration.

This superior performance with 2 processes and 11 threads can be attributed to a well-balanced distribution of computational workload across processes and threads. By using two processes, the workload is divided efficiently without causing excessive memory usage or computational overhead. This configuration may be also well-suited for this hardware, which has only two CPU clusters, allowing each process to take advantage of a dedicated cluster.

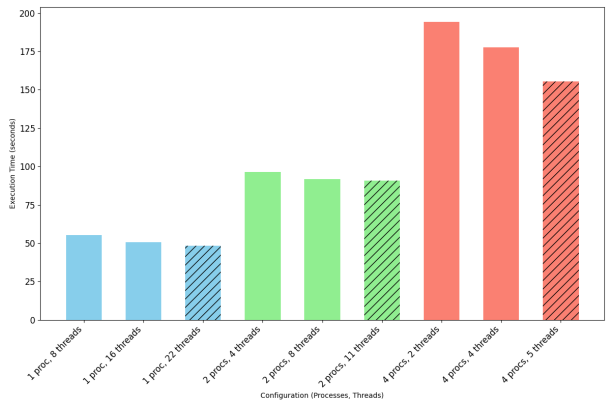


Figure 7: Execution Time for Different MPI Configurations (Cutoff Radius = 17500, Error Percentage = 4.8533%)

Despite these optimizations, Mode 3 generally underperformed compared to Mode 2. Several factors contribute to this observation:

- **Algorithmic Limitations**: In Mode 3, as required by the instruction, each leader process must be assigned an equal partition of the dataset. For simplicity I apply it based on the order in which particles are saved in `particles.csv` (e.g., with 1,000,000 particles and 4 processes, process 0 is assigned particles 0-249999, and so on). This partitioning is based on data order rather than physical location (cells), meaning particles assigned to a process can exist anywhere within the dataset. Consequently, to ensure that each particle can find others within the cutoff radius, all one million particles are copied to each process. This replication increases memory usage proportionally to the number of processes, resulting in $n \times 1,000,000$ particles for n processes, which significantly consumes memory resources. Furthermore, using MPI to create multiple processes adds initialization and communication overhead, which can offset any potential performance gains, especially as the number of processes increases.

- **Hardware Constraints**: Mode 3's performance limitations can also be attributed to the hardware architecture of the testing environment. The experiments were conducted on a laptop with only two CPU clusters, restricting the effective scalability of multi-process configurations. As a result, increasing the number of processes beyond the available clusters does not yield significant performance gains. This limitation amplifies the communication overhead, particularly when multiple leader processes are attempting to share and process data concurrently across limited hardware resources.

6 Execution Time vs. Mode

In this experiment, I evaluated the performance of three computation modes for calculating particle forces within a spatial grid. A cutoff radius of 17500 was chosen again based on previous tests. Execution time was recorded across three stages for each mode: data parsing, data partitioning, and force calculation. Based on prior experiments in Modes 2 and 3, the optimal configuration was identified as 16 threads for Mode 2 and 2 processes with 11 threads each for Mode 3. Consequently, these configurations were retained for this analysis.

By examining the execution time for each stage (data parsing, data partitioning, and force calculation), we can observe that the force calculation stage dominate the overall execution time, with the other two stages contributing minimally. Data parsing took approximately 0.25 seconds, and data partitioning required around 0.34 seconds in average, showing consistency and minimal impact on overall performance. The significant variation between modes occurred primarily in the force calculation stage.

As shown in Table 4, among the modes, Mode 2 with 16 threads was the fastest, demonstrating a substantial reduction in force calculation time due to its efficient use of parallel threads, which maximized computational throughput. In contrast, Mode 1 was the slowest, as it processed all calculations sequentially without any parallelization.

Mode 3 underperformed compared to Mode 2, despite using multiple processes. The need to replicate the entire dataset across processes in Mode 3 increased memory usage and introduced inefficiencies due to redundant data. Additionally, hardware limitations (two CPU clusters) restricted the potential benefits of the multi-process setup, as the available cores were not fully optimized for this structure.

In summary, Mode 2 outperformed the other configurations due to its straightforward, efficient threading model, while Mode 1 served as a useful baseline for sequential execution. Mode 3, while not as fast as Mode 2, still showed decent performance.

7 Speedup

Superlinear speedup occurs when the parallel execution of a task yields a speedup greater than the number of processing units used. For instance, if doubling the number of processors results in more than twice the speedup, it is considered superlinear. This phenomenon typically arises due to effects such as improved cache usage, reduced memory access times, or task scheduling efficiencies that are not achievable in serial execution.

In this experiment, superlinear speedup was not observed. Based on the data observed in Table 2 (Mode 2), the speedup achieved is predominantly linear. For instance, with 2, 3, 4 and 5 threads, the speedup values align closely with the number of threads, indicating a near-linear relationship between thread count and performance gains. This linear speedup suggests that the parallelization in this setup effectively distributes the workload among threads without achieving additional efficiencies such as cache optimization, which would be required for superlinear speedup. However, near-linear speedup is also a good outcome, demonstrating efficient parallel performance for this configuration.

Mode	Configuration	Parsing Data Stage (s)	Data Partition Stage (s)	Force Calculation Stage (s)
Mode 1	Single-threaded	0.189704	N/A	231.473
Mode 2	16 threads	0.190051	0.0302328	27.8778
Mode 2	22 threads	0.181878	0.0313978	32.6961
Mode 3	2 processes, 8 threads each	0.352092	0.0418883	91.8289
Mode 3	2 processes, 11 threads each	0.344028	0.0352039	90.8541

Table 4: Execution Time Breakdown for Different Modes (Cutoff Radius = 17500, Error = 4.8533%)

8 Re-usability

In the code, several parts were optimized to improve reusability. The initial parsing of the particle data from the input CSV file was designed to occur only once. After loading, this data was broadcasted to all processes in parallel configurations, allowing each process to access the complete dataset without redundant file reads. Additionally, spatial partitioning of the dataset into cells was also computed only once and stored in a shared structure, allowing individual threads to focus on their assigned tasks without recalculating partitioning. These optimizations reduced redundant computations and minimized memory usage, enabling each process and thread to utilize the precomputed data efficiently.

9 Optimizing for Skewed Dataset

To optimize the code for the skewed dataset, I leveraged some observations and insights from the experience with previous modes. Upon examining the skewed dataset, we can find that the range of particle locations is much larger, spanning from approximately $(0, 0)$ to $(1000000, 1000000)$, compared to the original dataset's range of $(-100000, -100000)$ to $(100000, 100000)$. Additionally, all particles are clustered in the positive $(+, +)$ quadrant, making the distribution sparser with fewer particles per cell. To achieve an error of less than 5%, a higher cutoff radius is required for this dataset. Through testing, a cutoff of 26000 produced an error of 4.84679%, whereas a cutoff of 17500 yielded a similar error of 4.8533% in the original dataset.

Considering these differences, I optimized my code by combining the best features of Mode 2 and Mode 3. Instead of partitioning data by particle index as in Mode 2, I implemented a queue-based task system inspired by Mode 3. In this approach, each cell became a task, and all tasks were stored in a queue. Threads then continuously consumed tasks from this queue. This setup enhances data locality because particles within the same cell are likely to interact, allowing threads to reuse data more effectively. In contrast to Mode 2's approach, where particles are divided by index (resulting in particles assigned to threads being scattered throughout the system), this task-based method achieves a more natural spatial locality. Additionally, I chose to forego the use of multiple leader processes and utilized a single process, as in Mode 2. Figure 7 illustrates that increasing the number of processes for this task

Threads	Time (s)	Speedup
1	74.1373	1.00x
2	41.0928	1.80x
3	28.7357	2.58x
4	23.4390	3.16x
5	17.9526	4.13x
6	17.3460	4.27x
7	16.1994	4.58x
8	13.5486	5.47x
9	13.1358	5.64x
10	12.6742	5.85x
11	11.4734	6.46x
12	12.6106	5.88x
13	10.9929	6.75x
14	11.1599	6.64x
15	10.8976	6.80x
16	10.8560	6.83x
17	10.8902	6.81x
18	10.2536	7.23x
19	10.8198	6.85x
20	10.5116	7.05x
21	10.4474	7.10x
22	10.7733	6.88x

Table 5: Execution Time and Speedup for Skewed Dataset with Varying Number of Threads (Cutoff Radius = 26000, Error Percentage = 4.84679%)

actually increases the execution time, which aligns with the analysis presented in Section 5, where I discuss how adding processes introduces overhead and decreases efficiency. Thus, my optimization combines the single-process advantage of Mode 2 with the queue-based task structure of Mode 3, resulting in a balance of data locality and minimized overhead for the skewed dataset.

To further optimize, I adopted a two-step process for checking distances between particles. For each particle, first check if another particle is within the range $(x-r, x+r)$ and $(y-r, y+r)$, where r is the cutoff distance. If a particle passes this pre-screening, it then calculates the Euclidean distance to determine if it is within the cutoff. This optimization significantly reduces computation time, as the conditional check is much faster than calculating the Euclidean distance, making it especially beneficial for very large datasets. It allows us to skip unnecessary distance calculations for particles that are guaranteed to be out of range.

The optimized approach yielded excellent results for the skewed dataset. The optimal configuration, as observed from execution times in Table 5, resulted in just 10.2536 seconds for 18 threads to compute the net force for one million particles, a substantial improvement over the original dataset, where the same number of particles took 28.0981 seconds. Despite the optimizations, it did not achieve superlinear speedup; however, the performance gains were significant, producing highly efficient execution times even without surpassing linear speedup.