Report: Range-Limited Electrostatic N-Particle Simulation with Pthreads and MPI

Ke Li

damian.li@mail.utoronto.ca

1. Hardware Resources

The program was executed on a Mac system running macOS Sonoma Version 14.3.1. The hardware specifications of the machine used are as follows:

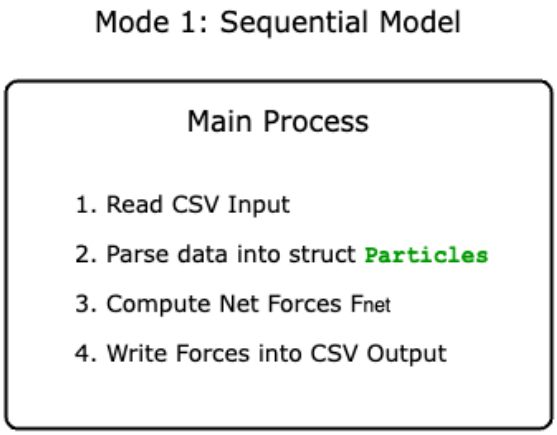
- Chip: Apple M3 Max
- CPU: 16-core, with a memory bandwidth of 409.6 GB/s
- GPU: 40-core GPU, with a memory bandwidth of 409.6 GB/s (not utilized for computations in this project).
- Memory: 48 GB

The configuration of this hardware was leveraged to execute the particle simulation efficiently, utilizing the multiple CPU cores to parallelize tasks and improve performance. The high memory bandwidth contributed to faster data handling, especially for tasks involving high data throughput. The GPU was not utilized in this project, as the computations were performed entirely on the CPU using C++ std::thread library and Open MPI.

2. Program Architecture

In this section, I describe the program architecture for the three different modes: Mode 1 (Sequential), Mode 2 (Evenly-Distributed Parallel Computation), and Mode 3 (Load-Balanced, Leader-Based Parallel Computation). Each mode has a distinct approach and the figures provided illustrate how threads and processes are involved in executing the calculations.

Mode 1: Sequential Model

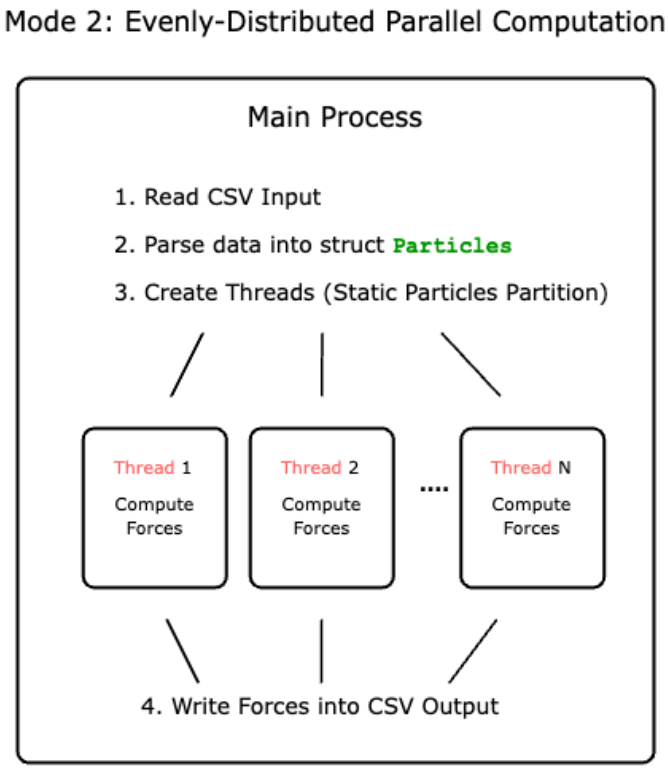


Mode 1 is a purely sequential model, in which a single process handles all tasks from start to finish. The main process performs the following steps sequentially:

- Read CSV Input: The program reads the particle data from the CSV file.
- Parse Particles: The input data is parsed into a self-defined particle structure.
- Compute Forces: The forces acting on each particle are computed.

This mode does not involve any parallelism; there is only one process and one thread, which performs all computations serially. This makes Mode 1 a straightforward implementation with no load balancing or parallel optimization.

Mode 2: Evenly-Distributed Parallel Computation



Mode 2 introduces parallelism by creating multiple threads to distribute the computation evenly. The key characteristics of Mode 2 are as follows:

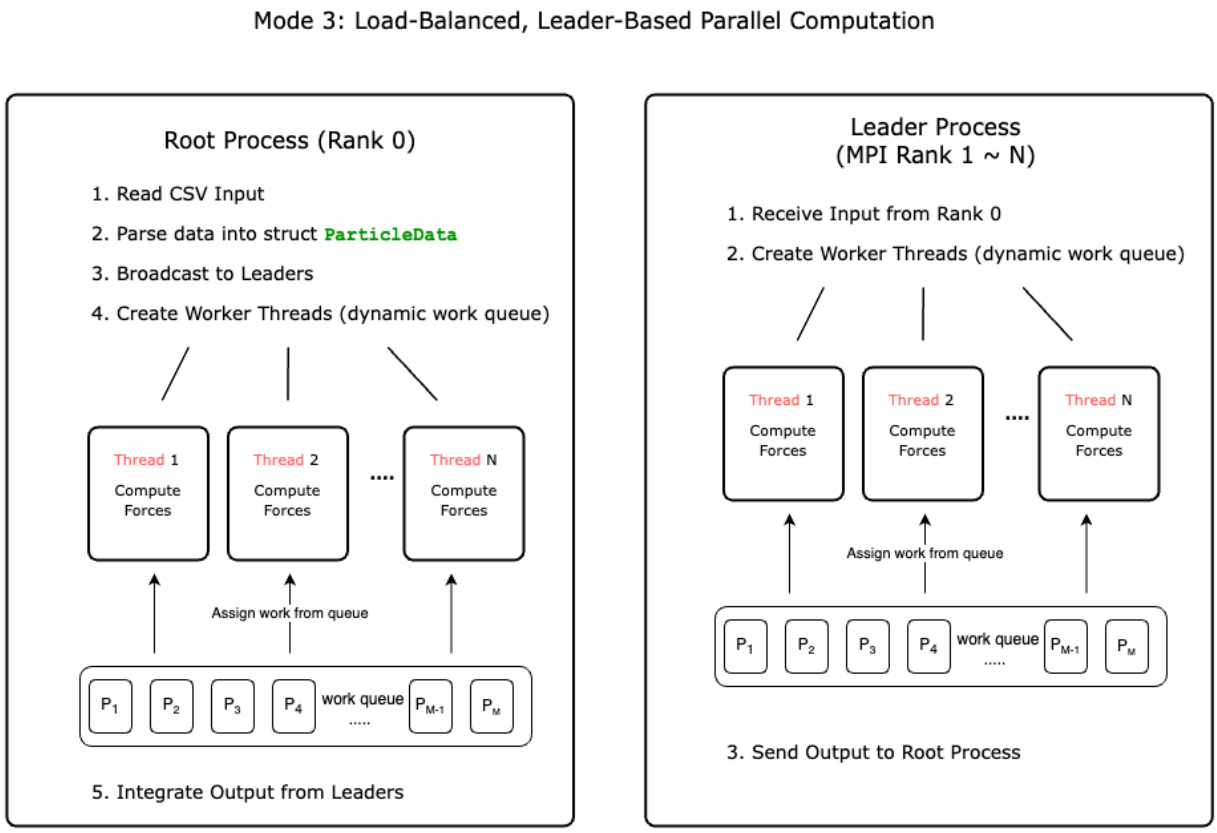
- Single Process, Multiple Threads: A single process runs the entire program, but it creates multiple threads using `std::thread` to parallelize the computation of particle forces.

- Static Work Assignment: Particles are distributed evenly across threads at the time of thread creation. Each thread is assigned a fixed subset of particles to process, ensuring that the workload is as evenly distributed as possible.

For parallelization and serialization, the input read and parsing of particles are done serially by the main process in mode 2; Apart from that, the force computation is parallelized across the threads, with each thread computing forces for its assigned subset of particles. The work assignment is static, meaning there is no dynamic load balancing between threads once they start execution.

This mode is useful for evenly distributing the computational load, but it does not handle dynamic imbalances that may arise due to varying computational complexity among different particles.

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



Mode 3 is using both processes and threads to achieve dynamic load balancing and maximize efficiency. The key components are as follows:

- Multiple Processes (Leaders): Mode 3 employs multiple MPI processes (referred to as "leaders"), where each process runs a separate instance of the computation. The root/master process (rank 0) also plays a special role in managing input and output.

- Root Process Responsibilities: The root process reads the input data, broadcasts it to all other leader processes, and participates in the computation along with other leaders. It also integrates the results from all leaders to generate the final output.

- Multiple Threads per Leader: Each leader process creates multiple threads to further parallelize the computation. The threads dynamically fetch work from a shared queue, which ensures that any thread that becomes available can take on the next task, allowing for dynamic load balancing.

For Parallelization and Serialization:

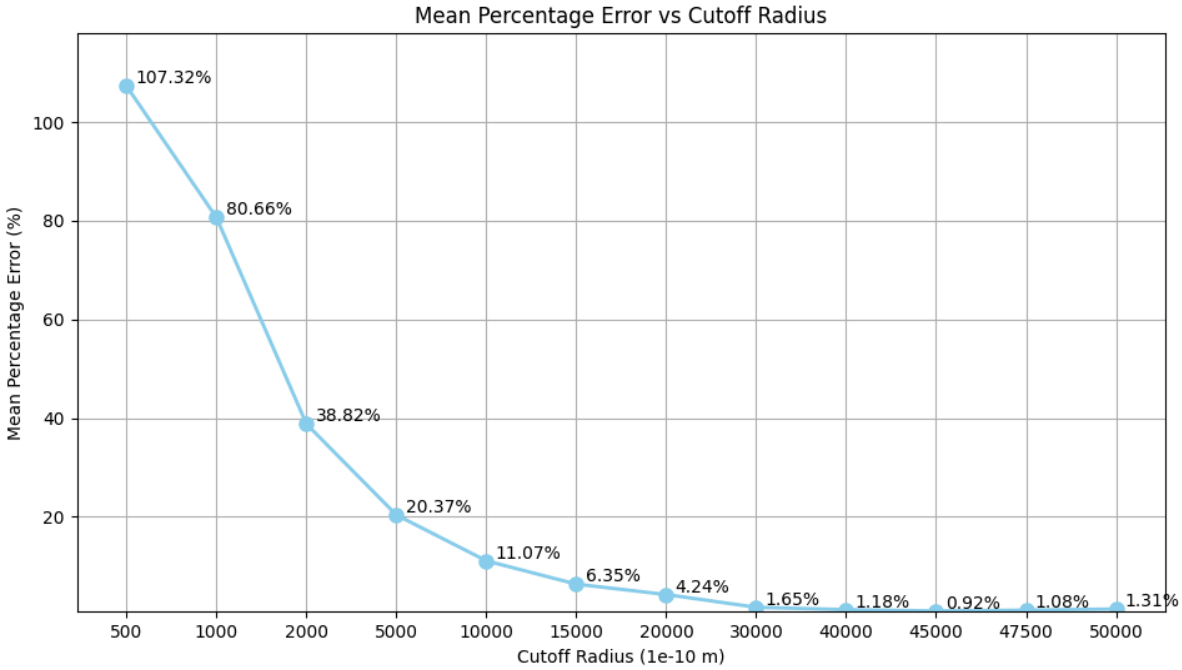
- Input Reading: The root process reads the CSV input file and broadcasts the parsed particle data to all leaders. This part is serial with respect to the root process.

- Work Distribution: After broadcasting, each leader process (including the root) creates threads that dynamically fetch tasks from a shared queue to compute the

particle forces. This ensures efficient utilization of available threads and helps to avoid imbalances.

- Result Integration: Once all threads complete their computations, each leader sends its results back to the root process which is rank 0 process when in MPI. The root process integrates these results, which represents a combination of serial and parallel operations.

3. Mode 1: Sequential Computation



In Mode 1, the cutoff radius was varied from 5e-8 (m) until the performance of the system reached a plateau, with the objective of evaluating the accuracy of the numerical approximation compared to the oracle (the exact solution). This is the command I used in Mode 1 (`{%d}` from 500 to 50000):

```
./nParticleSim --mode=1 --cutoff_radius={%d}
--input=../dataset/particles.csv
```

The cutoff radius values used were measured in units of meters, as shown on the x-axis of the figure above. The y-axis represents the mean percentage error, compared with the exact solution for different cutoff radius.

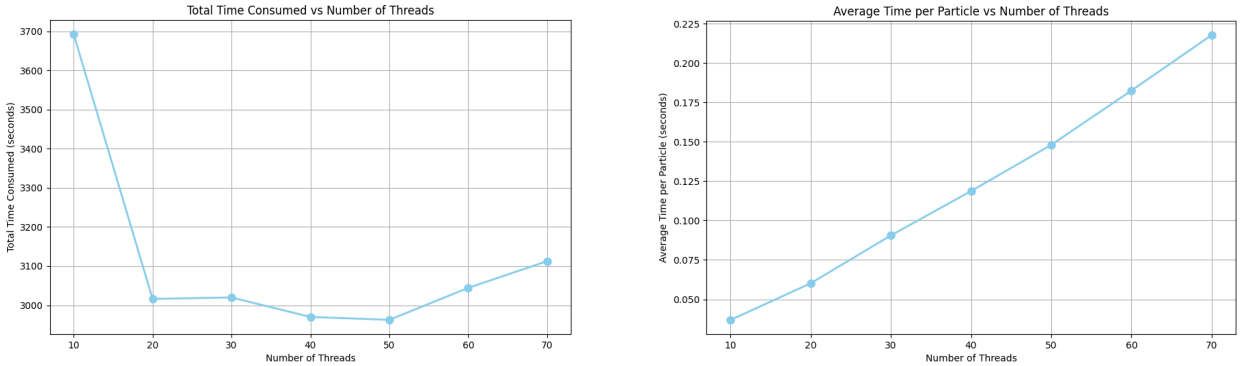
From the graph, it illustrates that the mean percentage error decreases significantly as the cutoff radius increases, showing improved accuracy in the approximation. More specifically, at the smallest cutoff radius I choose, 5e-8 (m), the error was relatively high at 107.32%; As the cutoff radius increased, the mean error percentage decreased, reaching

0.92% at a cutoff radius of 4.5e-6 (m). This value was identified as yielding the smallest error.

Moreover, when the cutoff radius was further increased to 4.75e-6 (m), the mean error rate slightly increased to 1.08%, indicating that increasing the cutoff radius beyond a certain point does not necessarily lead to better accuracy and may introduce numerical instability.

Based on these results, a cutoff radius of 4.5e-6 (m). was selected for subsequent computations in Mode 2 and Mode 3 to ensure a balance between computational performance and accuracy.

4. Mode 2: Evenly-Distributed Parallel Computation



In Mode 2, the cutoff radius of 4.5e-6 (m) was chosen, which, as discussed in the Mode 1 section, yielded a mean error of 0.92% compared to the oracle solution. The command used for this mode was as follows:

```
./nParticleSim --mode=2 --cutoff_radius=45000 --threads={%d}
--input=../dataset/particles.csv
```

The number of threads (`{%d}`) varied from 10 to 70, incrementing by 10 each time. The average time for reading particles into struct `Particles` is 0.924 seconds. In Mode 2, all the particle reading was done before emplace threads, so the time spent for that remains consistent when the number of threads varies. The two graphs provided represent the Total Time Consumed vs Number of Threads (left) and Average Time per Particle vs Number of Threads (right).

The total computation time graph shows that as the number of threads increased from 10 to 50, the total time consumed decreased significantly, reaching a minimum. However, after 50 threads, the total time began to increase, indicating reduced efficiency. The potential reason for this behavior is related to the operating system's management of threads. On macOS, there is no direct mechanism to bind a specific thread to a specific core, which can lead to inefficiencies when too many threads are created. The increased time beyond 50 threads could be attributed to excessive context switching, where the operating system spends more time switching between threads than actually executing computations.

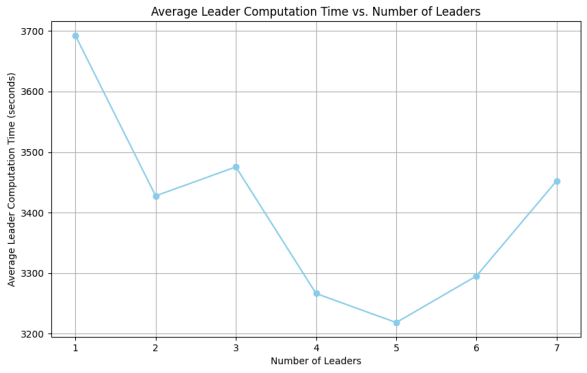
The second graph, Average Time per Particle vs Number of Threads, shows that the average time per particle increases linearly as the number of threads increases. This aligns with the earlier observation about context switching, where having more threads than the available cores results in each core managing multiple threads. The overhead of managing these threads appears to degrade performance.

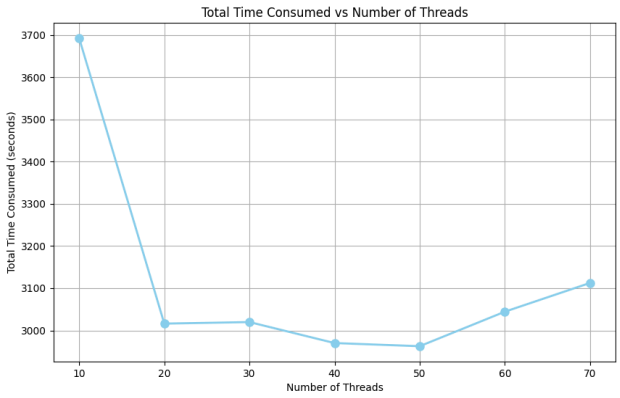
To quantify the average time per particle, I calculated the total time taken by each thread and divided it by the number of particles that the thread processed. The steady increase in time per particle suggests diminishing returns when attempting to use more threads than cores.

Overall, the optimal number of threads in Mode 2 was found to be 50 (among all the test cases I wrote for Mode 2), where the total computation time was minimized. Beyond this point, the overhead from thread management outweighed the potential benefits of parallelization. The tradeoff observed here is between parallelizing the workload across more threads and managing the resulting overhead due to limited hardware resources (i.e., 16-core CPU).

For the subsequent experiments in Modes 2 and 3, 50 threads were used to achieve the best performance, ensuring a balance between efficient computation and manageable system resource usage.

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





In Mode 3, the goal was to investigate the effect of varying the number of leader processes, worker threads, and the cutoff radius on computation performance.

Experiment 1: Varying the Number of Leader Processes

The first graph provided shows the Average Leader Computation Time vs. Number of Leaders. In this experiment, I fixed the number of worker threads per leader at 10 and varied the number of leader processes from 1 to 7. The cutoff radius was set to 4.5e-6 (m), which achieved an error margin below 5% as established in Mode 1.

The graph illustrates that increasing the number of leaders initially results in a significant reduction in the average leader computation time. However, beyond 5 leaders, the

average time begins to increase. The optimal number of leader processes was found to be 5, where the average computation time was minimized. Beyond this point, adding more leaders introduced additional communication overhead, which negated the benefits of parallelism.

Experiment 2: Varying the Number of Worker Threads

The second graph, Total Time Consumed vs Number of Threads, shows the performance when varying the number of worker threads per leader. The number of threads was varied from 10 to 70, incrementing by 10 each time, while keeping the number of leaders fixed at 5 (based on the previous experiment). The cutoff radius remained at 4.5e-6 (m).

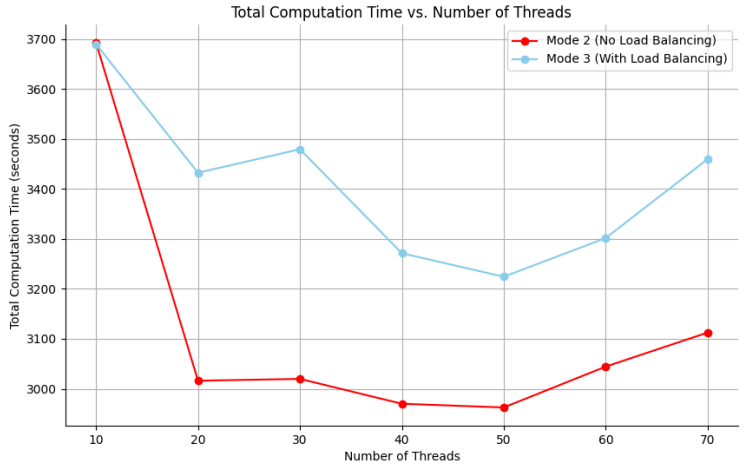
The graph demonstrates that as the number of threads increased from 10 to 50, the total time consumed decreased, reaching a minimum. Beyond 50 threads, the total time began to increase, similar to the observations in Mode 2. This behavior can be attributed to the operating system's thread management inefficiencies when the number of threads exceeds the available CPU cores just like what I encountered in Mode 2.

A larger cutoff radius increases the number of particle interactions that must be computed for each particle's net force. As a result, the computation time increases as the cutoff radius becomes larger. The cutoff radius was fixed at 4.5e-6 (m) to maintain a balance between accuracy (error margin = 0.92%) and performance.

For the worker threads, the optimal number of worker threads per leader was 50. Beyond this point, the increased context switching overhead reduced the performance gains from adding more threads. This result is consistent with the findings in Mode 2, where 50 threads provided the best performance.

And lastly for the leader processes, the optimal number of leaders was found to be 5. Adding more leaders beyond this point increased the communication overhead, which reduced the overall performance benefits. The average leader computation time was minimized at 5 leaders.

6. Execution Time vs. Mode



In this section, we compare the execution times of Mode 2 and Mode 3 under different configurations to evaluate their performance in various stages of computation. Specifically, we investigate the Total Computation Time vs. Number of Threads for both modes, as illustrated in the graph provided. The graph compares Mode 2 (in red) and Mode 3 (in blue).

Experimental Setup:

- Number of Threads: In Mode 2, a single process was used with multiple threads, ranging from 10 to 70, incrementing by 10 each time. In Mode 3, multiple processes (leaders) were used, each with fixed 10 worker threads per leader, ensuring consistency in the total number of threads across both modes. For instance, when using 30 threads in Mode 3, 3 leader processes with 10 threads per leader were utilized.
- Cutoff Radius: The cutoff radius was fixed at 4.5e-6 (m), which provided an error margin below 5% based on the results from Mode 1.
- Stages Measured: The comparison focuses on the Total Computation Time, which includes parsing data, data partitioning, and force calculation. The parsing data stage was consistent across both modes, as the input was parsed by a single process before threads or processes were initialized.

Analysis of Results:

- Mode 2 (No Load Balancing): In Mode 2, the computation involves a single process with multiple threads. The red line in the graph shows that as the number of threads increased from 10 to 50, the total computation time decreased significantly, reaching a minimum at 50 threads. However, beyond 50 threads, the performance degraded due to increased context switching overhead.
- Mode 3 (With Load Balancing): In Mode 3, multiple leader processes were employed, each with multiple threads (10 per leader). The blue line shows that the total computation time initially decreased as the number of leader processes increased, but the performance plateaued and eventually worsened beyond a certain point. The reason for this performance degradation is attributed to the inter-process communication overhead. In Mode 3, data must be broadcast to all leader processes, and each leader must synchronize with others. This introduces additional latency, which grows as the number of leaders increases.
- Performance Comparison: From the graph, it is evident that Mode 2 outperforms Mode 3 in terms of total computation time. The key reason is the absence of inter-process communication overhead in Mode 2. In Mode 3, while dynamic load balancing is beneficial, the cost of communication across multiple processes offsets the gains from load distribution. As a result, Mode 3 takes longer to execute compared to the single-process, multi-thread approach of Mode 2.

7. Speedup

In the context of my experiments, superlinear speedup was observed in certain situations for both Mode 2 and Mode 3 compared to Mode 1 (Sequential Model). From the experimental results, both Mode 2 (Evenly-Distributed Parallel Computation) and Mode 3 (Load-Balanced, Leader-Based Parallel Computation) demonstrated significant efficiency improvements over Mode 1.

Mode 2: Thread-Based Parallelism

In Mode 2, the performance did not consistently improve with an increasing number of threads. The maximum efficiency was found to be hardware-dependent, and adding more threads beyond a certain point resulted in diminishing returns. This behavior is partly due to limitations specific to macOS, where, unlike Linux, it is not straightforward to bind threads to specific CPU cores, leading to increased context switching and thread scheduling overhead.

However, for certain configurations (such as 50 threads), superlinear speedup was observed due to better cache utilization and the efficient parallel distribution of work. The parallel execution allowed for CPU to effectively caching of frequently accessed data, which reduced memory latency and improved performance beyond what would be expected from a linear increase in processing units.

Mode 3: Process and Thread-Based Parallelism

Mode 3 introduced leader-based parallelism, where multiple MPI processes (leaders) each had multiple worker threads. Compared to Mode 2, Mode 3 added complexity in the form of inter-process communication. While Mode 3 still achieved performance improvements over Mode 1, the benefits were limited by the communication cost between processes. This communication overhead often negated the benefits of adding more leaders, especially when the data size was not large enough to justify distributed communication.

In some scenarios, superlinear speedup was achievable in Mode 3 when the number of leaders and threads was balanced appropriately. This was primarily due to the efficient distribution of work across processes, leading to better utilization of hardware resources. However, as the number of leaders increased, the cost of broadcasting data and synchronizing processes introduced significant overhead, which limited further performance gains and made achieving superlinear speedup less consistent.

In summary, both Mode 2 and Mode 3 showed significant speedup compared to Mode 1. In some instances, superlinear speedup was achieved due to better cache utilization, reduced memory latency, and more efficient workload distribution. However, achieving superlinear speedup consistently is highly dependent on the hardware, operating system, and the characteristics of the workload. For Mode 2, the limitation was the inefficiency of thread scheduling beyond the optimal number of threads, while for Mode 3, the added cost of inter-process communication limited the achievable speedup.

8. Re-usability

Mode 3: Process and Thread-Based Parallelism

1. **Particle Data Reading**

   In Mode 3, the root process reads the particle data from the CSV file:

```
if (world_rank == 0) {
    // ...
    // Reads the particles from the file
    particles = reader.readParticles(input_file);
    num_particles = particles.size();
    // ...
}
```

   And the particle data is then broadcast to all other leader process using MPI:

```
// Broadcast the number of particles to all processes
MPI_Bcast(&num_particles, 1, MPI_UNSIGNED_LONG_LONG, 0,
MPI_COMM_WORLD);
// Broadcast the array of ParticleData to all processes
MPI_Bcast(particleDataArray.data(), num_particles, MPI_ParticleData,
0, MPI_COMM_WORLD);
```

   In this way, particle reading and struct construction are executed only once by the root process, and the results are distributed to all other processes, and the data were used many times afterwards.

2. **Work Queue Initialization**

   In Mode 3, I used a Work Queue to allow threads to dynamically fetch works to achieve load balancing and avoid waiting threads with no task to fetch. In Function `Leader::computeForces` in Leader.cpp:

```
{
    std::lock_guard<std::mutex> lock(queue_mutex);
    for (size_t i = start_index; i < end_index; ++i) {
        task_queue.push_back(i);
    }
}
```

   The initialization of the work queue is done once per leader, ensuring that all worker threads of a leader fetch tasks from the same shared queue. This avoids redundant work assignments and allows efficient reuse of the initial partitioning of work.