Assignment1: ECE1747

Akhil Jarodia
Student Number: 1010169899

November 2, 2024

Introduction

1 Hardware specification

For this assignment, I was using the ECF machines at UofT. Following are some of the specifications: **CPU model**:Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz **Number of threads**: 80 **Cores per socket**: 20 **Sockets** : 2 **Threads per socket**: 2 **Memory** : 754 GB(Avg free memory: 201GB) **OS** : Alma Linux

Program Architecture

1.1 Algorithm Used

For calculating the force, initially I used the brute force method. However, the runtime was very high and was not feasible to use it for covering different values of cutoff radius. Thereby, I'm using KD Tree here for storing and exploring neighbours in a given range.

A KD-Tree (k-dimensional tree) is a binary tree data structure used for organizing points in k-dimensional space, allowing for efficient range and nearest neighbor searches. It is built recursively by selecting a median point along alternating axes at each depth level, ensuring a balanced structure. The query operation traverses the tree based on the target's coordinates, utilizing bounding boxes to minimize distance calculations. This approach reduces the search space significantly, making it faster than a brute-force search.

1.2 Algorithm Flow

1 explains how the mode 1, 2 and mode 3 works. K-D tree helps us in finding the neighbor efficiently while pthread helps in distributing the workload among various threads.

1.2.1 Mode 1: Sequential Approach

In the sequential approach, the algorithm processes each particle one by one in a single thread of execution. The key steps involved in this mode are as follows:

1. **Loading Particle Data**: The program reads particle information, including their positions and charges, from a file.
2. **Building the KD-Tree**: A KD-Tree is constructed using the particle positions. This data structure is used to efficiently query particles within a certain radius, optimizing the neighbor search process.
3. **Calculating Net Forces**: For each particle, the algorithm queries the KD-Tree to find neighboring particles within a specified cutoff radius. The Coulomb force between the target particle and each neighbor is then computed and summed up to get the net force on that particle.
4. **Writing Results to File**: Once all particles have been processed, the net forces are written to a file.

The sequential nature of this approach ensures simplicity in implementation but can be slow when handling a large number of particles, as each particle is processed one after another without parallelism.

1.2.2 Mode 2: Pthread Approach

The Pthread approach leverages multi-threading to speed up the computation by dividing the workload among multiple threads. The key steps involved in this mode are as follows:

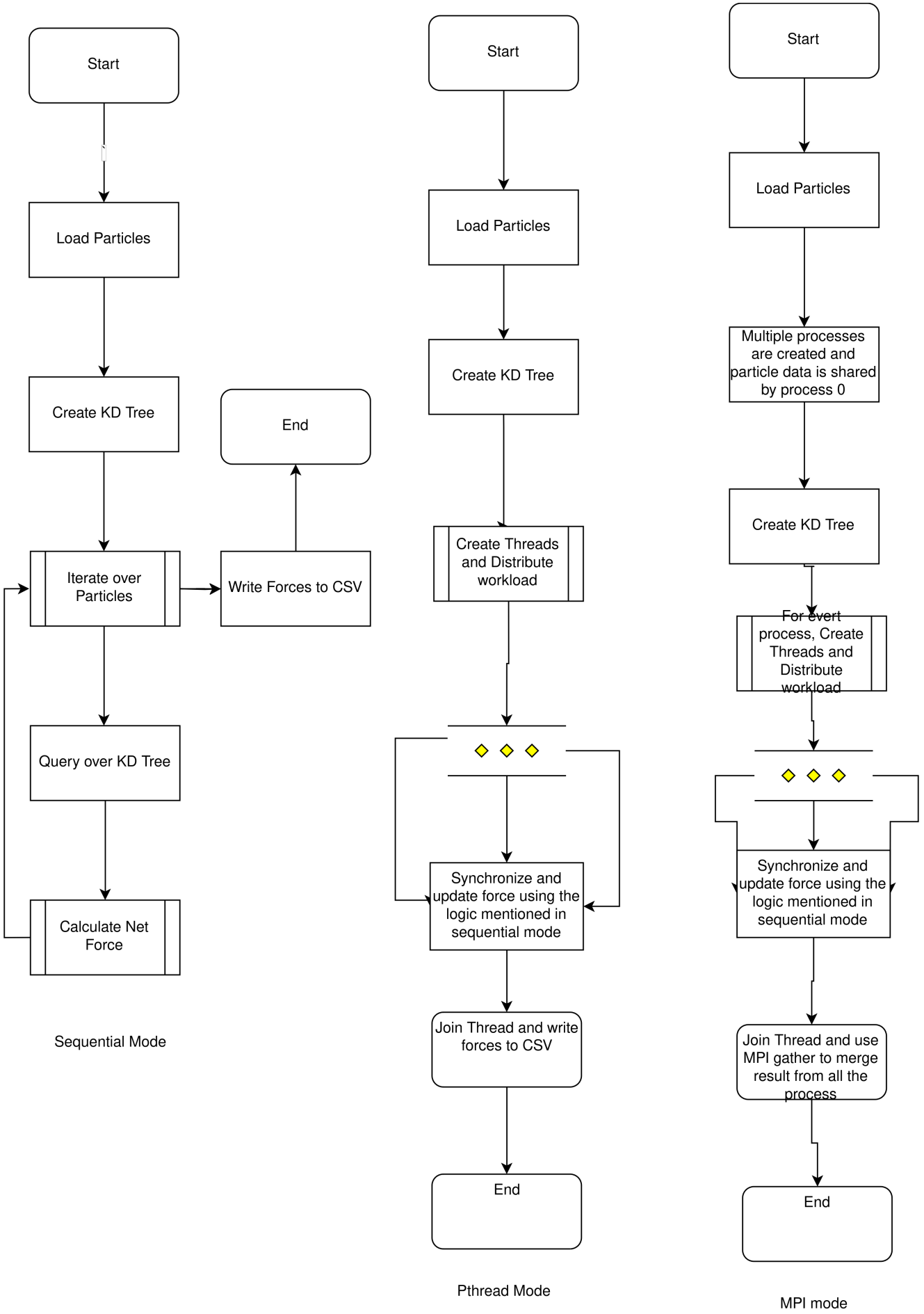

Figure 1: Flow diagram for mode1, 2 and 3

1. **Creating Threads for Parallel Execution**: Multiple threads are spawned, with each thread responsible for processing a subset of the particles.
2. **Parallel Calculation of Net Forces**: Each thread independently queries the KD-Tree to find neighbors for its assigned particles and calculates the net forces concurrently. A mutex is used to manage shared data access safely.
3. **Joining Threads and Writing Results**: After all threads complete their execution, the main program collects the results and writes the net forces to a file.

By using multi-threading, the Pthread approach can significantly reduce execution time, especially for large datasets. However, the performance gain depends on the number of particles, the number of available CPU cores, and the overhead of thread management.

1.3 Mode 3 algorithm

This mode utilizes MPI (Message Passing Interface) for distributed computing, pthreads for multithreading, and a KD-tree data structure to optimize spatial queries.

Key components of the program include:

- MPI is employed to distribute particles and workload across multiple processes. Each process builds the KD-tree locally and calculates forces for a portion of the particles.
- Pthreads are used within each MPI process to further parallelize the computation by processing tasks concurrently.
- Results are gathered from all processes, and the final net forces are outputted to a CSV file.

1.4 Testing architecture

In order to test if the output is matching our input, I've created a python script that compares the value of all the elements and counts the number of variables that are under 5%. My aim was to achieve more than 95% of the elements fall under the 5% error bracket.

2 Mode 1: Sequential Computation Analysis

As mentioned in the previous section, I'm using a KD tree for storing the points. While this approach is efficient for smaller radius, with increasing distance, the number of search increases drastically leading to perform equivalent to a boot-force implementation.

For higher value than 10000, I was facing CPU time limit exceeded (core dumped) error thereby it is the best cutoff using sequential mode. Refer to Table 1 for more details.

With increasing value of the cutoff radius, the accuracy increases. However the processing time increases exponentially.

3 Mode 2: Evenly distributed Parallel Computation Analysis

I'm using a UofT server for this assignment which has 80 cores which is way more than any personal computer. Thereby here I'm using a multiple of 10. Following are the results and comparison. There is a substantial performance improvement up to 80 threads, after which the benefits plateau. Beyond this range, additional threads contribute to higher overhead without proportional gains in speed, indicating that the system's efficiency decreases with excessive parallelism.

In comparison to the mode 1, pthread based option performs way better with the best model giving the output in 1% of the sequential time. Refer to Table 2 for more details.

Radius	Processing Time(s)	Accuracy
6	1.585	50%
1000	23.65	77.2%
2000	79.0851	85.31%
3000	160.45	89.47%
4000	312.187	91.81%
5000	370.64	93.33%
6000	576.235	94.388%
7000	851.704	95.15%
8000	1000.91	95.71%
9000	1210.08	96.13%
10000	1380.18	96.50%

Table 1: Sequential Processing Time & Accuracy comparison

Table 2: Pthread Performance Summary(80 Cores)

Threads	**Parallel Duration (s)**	**Thread Creation Time (s)**	**Total Threads Execution Time (s)**
10	55.9548	0.0026	526.496
20	26.8185	0.0029	526.374
30	18.8858	0.0059	549.868
40	15.0401	0.0081	570.273
50	16.5140	0.0119	681.521
60	14.9649	0.0147	781.179
70	13.9730	0.0173	903.585
80	13.2505	0.0188	1021.450
90	13.3869	0.0559	1130.310
100	13.4557	0.2112	1239.090
120	13.2938	0.0624	1492.540
130	13.3432	0.2677	1641.550
140	13.3432	0.2677	1758.570
150	13.2143	0.0231	1874.590
160	13.1825	0.0716	2006.920

4 Mode 2: Parameter tuning

From this experiment, we'll continue ahead with 80 threads, 10000 cutoff radius and further try to improve our results.

Given that the execution time have reduced, we now use a bigger cutoff radius. Given the performance is not improving if I increase my number of threads further to 80, I'll focus on cutoff radius with 80 threads.

Table 3: Performance Summary for 80 Threads with Accuracy

Cutoff	**Parallel Duration (s)**	**Total Threads Execution Time (s)**	**Accuracy (%)**
10,000	13.2505	1021.450	96.50
12,000	19.4130	1503.930	97.05
13,000	22.0660	1716.410	97.283
14,000	25.3164	1960.870	97.4811

From these four experiments, we can see that the accuracy increased by 1% but the computation time was double. Therefore, I'll do a slight tradeoff and prefer to go with the 10,000 cutoff as provides with mostly accurate values in the least amount of time.

5 Mode 3: Load balanced, leader based parallel computation Analysis

We carried out multiple experiments while changing these values. Following is the summary of the same:

Cutoff Radius	Number of Threads	Number of Processes	Total Comp Time
10000	10	4	60.4076
10000	40	5	52.6172
10000	50	4	44.6282
10000	60	4	45.0061
10000	70	4	44.4129
10000	80	4	44.6167
10000	50	5	52.6258
12000	50	4	60.6630
13000	50	4	68.8163

Table 4: MPI Computation for Various Cutoff Radii and Number of Threads

The results show that increasing the cutoff radius leads to higher computation times, indicating more complex calculations. More processes don't always improve performance due to potential overhead, and there seems to be an optimal thread count for best efficiency. Refer table 4.

When comparing the use of 4 and 5 processes (e.g., for a cutoff radius of 10,000 with 50 threads: 44.6282 seconds for 4 processes versus 52.6258 seconds for 5 processes), it can be observed that adding more processes doesn't always reduce computation time.

With a fixed cutoff radius (e.g., 10,000), changing the number of threads from 10 to 80 shows reduction in computation time.

For a cutoff radius of 10,000, the configuration with 4 processes and 50 threads resulted in the shortest time (44.6282 seconds).

However, these values are way more than the time required by the pthread implementation. Possibly due to increase in the communication and data locality.

6 Execution Time vs. Mode

For our comparison, I'm taking the best performing parameters from each of the stage. Following is the list of the same.

- Mode 1: cutoff Radius: 1000m, Processing Time: 1380s
- Mode 2: cutoff Radius: 1000m, 80 threads, Processing Time:13.2505s
- Mode 3: cutoff Radius: 1000m, 50 threads and 4 processes. Processing Time: 44.6282s

Between mode 2 and 3, following is the comparison:

Mode 2 is the fastest among all of them and Mode 1 is the slowest. Increasing the number of threads in Mode 2 and using processes in Mode 3 allow for better utilization of CPU resources, leading to reduced execution times. The cutoff radius affects the computational complexity. In Mode 1, as the radius increases, the processing time grows significantly because all calculations are done sequentially. In contrast, modes 2 and 3 can distribute the workload more effectively, mitigating the impact of the increased complexity.

Figure 2: Execution times for different processing modes

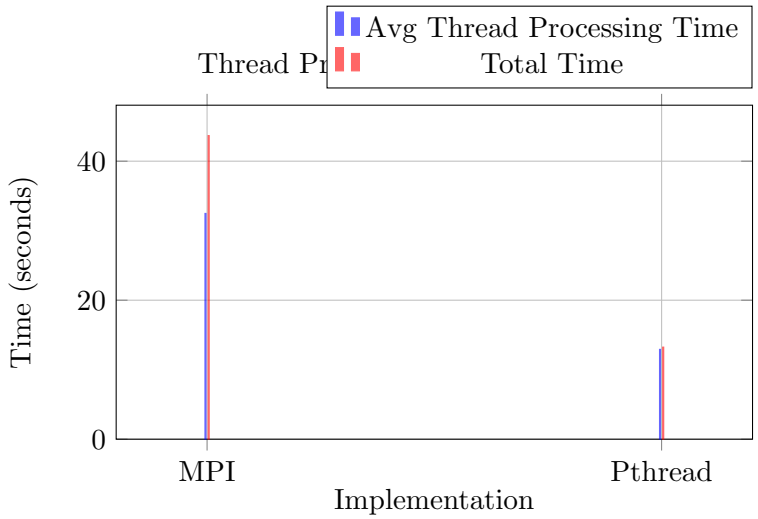

Figure 3: Comparison of Average Thread Processing Time and Total Time for MPI and Pthread

Mode 3 is slightly slower because it involves sending messages across different leaders which is an additional overhead. For some algorithms, it can be a better alternative. Figure 3 shows us that there's a big difference between the total processing time and the thread processing time which is not the case with pthread.

7 Speedup

Superlinear speedup is achieved when the factor between sequential and parallel time is greater than the number of cores in the system.

$$SpeedupFactor = \frac{T_{Seq}}{T_{parallel}} = \frac{1380}{13} = 104$$

In our case we achieved this in case of pthread case where the speedup is 104 which is greater than the number of cores in the system(80). I think improved cache utilization and reduced overhead lead to this

situation.

However we did not achieved this in MPI.

8 Reusability

Following are the list of items that are computed only once:

- KDTree Construction(where all the particles are stored)
- Particle Data Vector

These values are computed and shared across all the processes/threads where they are required.