CS3210 Parallel Computing

Assignment 1A Report

Discrete Particle Simulation with OpenMP

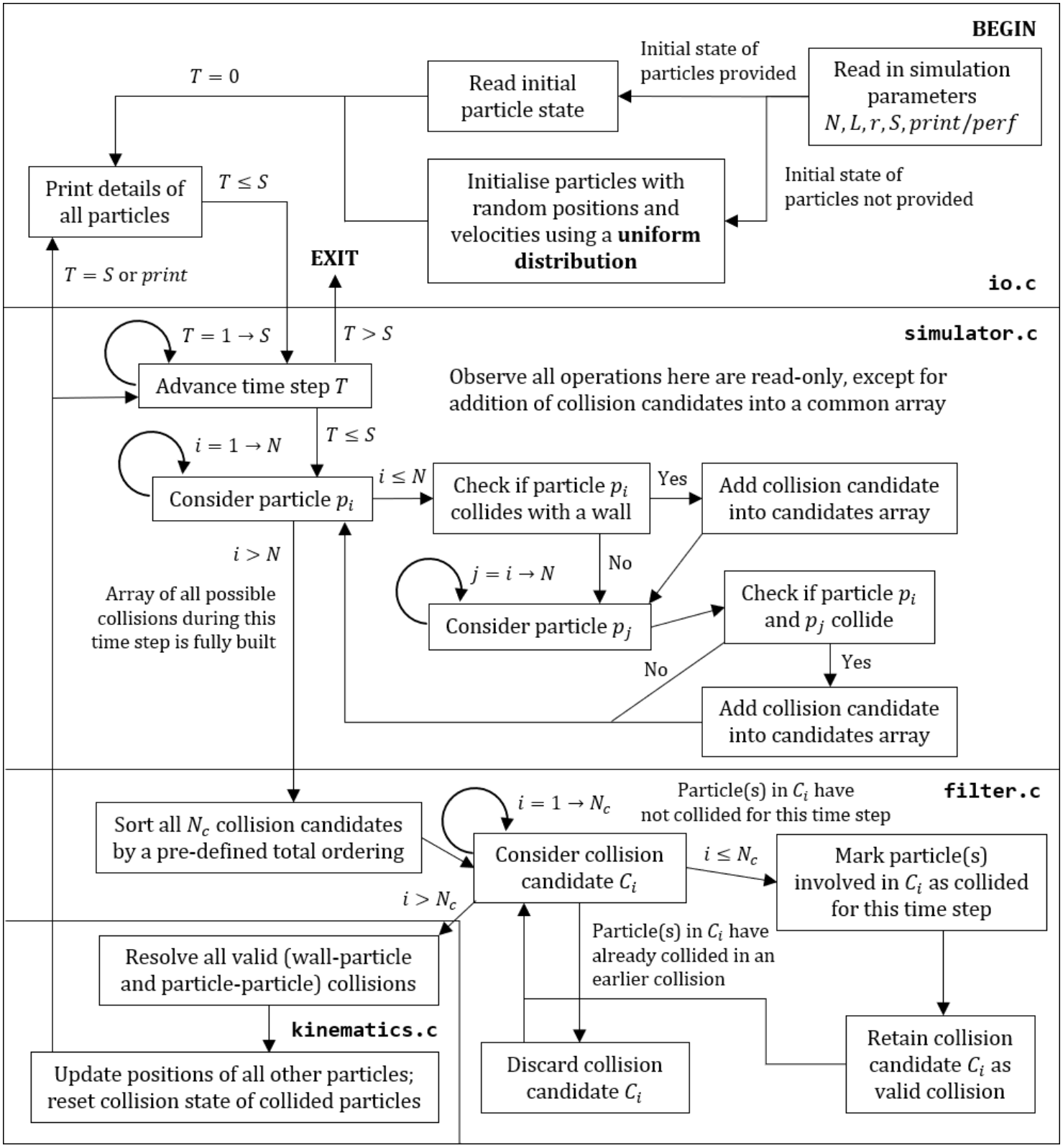
30 September 2018

Lee Yong Jie, Richard (A0170235N)
Keven Loo Yuquan (A0183383Y)

Table of Contents

1 Program Design

The discrete particle simulation is implemented fully in C and OpenMP. The overall architecture of the simulator is summarised in the diagram below.



2 Implementation Assumptions and Details

2.1 Assumptions

We make the following assumptions in our simulation.

1. Even though the maximum initial velocity limit is $L/4$, we assume that the typical velocity of a particle is much less than this ⇒ particles move only a small amount for each time step, hence for each particle p_i and particle p_j, the probability of collision $P(p_i, p_j\ collide) \ll 1$ ⇒ the total number of collisions per step $N_{collisions} = \Theta(N)$.

2. A particle is involved in no more than one collision per time step.

    - **Implications**
        i. If a particle collides with a wall after any collision, it is placed at the wall at the end of that time step, ready to collide with the wall the next time step
        ii. If particle P collides with a particle Q after any collision, it will phase through the particle Q for the remainder of this time step, or will ignore Q the next time step if they happen to overlap

3. All collisions between particles and with the wall are elastic (kinetic energy and momentum are conserved).

4. It is possible to fit all N particles of radius r into the square of length L without overlapping.

    - **Implication**
        i. The simulation exits if one of these two conditions fail: $L < 2r$ (not possible to fit a single particle) or $Nr^2 > L^2$ (not possible to pack N particles in a grid in a square with area L^2)

5. The set of possible collisions C satisfies a **total ordering**.

    - **Implication**
        i. For each time step T, it is possible to sort all collision candidates by this total ordering to determine which collisions should be prioritised over others.

2.2 Implementation Details

1. If the initial state of particles are not provided, particles are generated with initial random positions and velocities using an **uniform distribution**.

    a. We use the pseudo-random number generator `rand` in the C library seeded with the number 3210.

    b. Particles are placed randomly in the square without overlapping.

2. Each particle keeps track of its own state: ID, x, y, v_x, v_y.

3. Our simulation has an additional parameter `SLOW_FACTOR` in `simulator.c` that increases the granularity of the simulation for greater accuracy.

    a. Setting `SLOW_FACTOR` to an integer > 1 slows the initial velocities of all particles by that factor. `SLOW_FACTOR` should be set to a power of two to avoid introducing additional floating-point errors when dividing the particles' velocities.

    b. The number of steps of the simulation is correspondingly multiplied by `SLOW_FACTOR`, i.e. each original step now corresponds to `SLOW_FACTOR` "micro-steps".

4. Collisions are of two types: either particle-wall collisions and particle-particle collisions. We describe a particle-wall collision as (P, `null`) and a particle-particle collision as (P, Q).

    a. As particle-particle collisions are symmetric (i.e. P collides with $Q \Leftrightarrow Q$ collides with P), we generate these collisions such that P is the particle with lower integer ID.

5. When sorting collision candidates with the C library function `qsort`, we enforce this total ordering in the function `cmpCollision`. For two collision candidates C_1 and C_2,

    - $C_1 < C_2$ if C_1 occurs before C_2
    - If C_1, C_2 occur at the same time
        - $C_1 < C_2$ if ID of P in C_1 < ID of P in C_2
        - If C_1, C_2 both involve the same particle P
            - $C_1 < C_2$ if C_1 is a wall collision
            - If C_1, C_2 are both particle-particle collisions
                - $C_1 < C_2$ if the ID of Q in C_1 < the ID of Q in C_2

6. For each particle in each time step, we check once if the particle collides with any of the four walls.

7. (**Sequential and parallel-1**) All possible collision pairs are checked for each time step. The total number of collision checks performed is thus

$$\begin{aligned} N_{potential\ collisions} &= N_{wall-particle} + N_{particle-particle} \\ &= N + \frac{N(N-1)}{2} \\ &= \Theta(N^2) \end{aligned}$$

8. Particle-wall collisions are checked by solving the trajectory equation of a particle and the positions equations of the wall. The equations of the walls are $x = 0, x = L, y = 0, y = L$.

9. Particle-particle collisions are checked by solving trajectory equations of two particles during the given time step.

   Consider two particles P, Q during a given fractional time step $0 \leq \Delta t \leq 1$. From Pythagoras' theorem, the distance between them is

$$d = \sqrt{\Big((x_Q + v_{xQ}\Delta t) - (x_P + v_{xP}\Delta t)\Big)^2 + \Big((y_Q + v_{yQ}\Delta t) - (y_P + v_{yP}\Delta t)\Big)^2}$$

   or re-written in terms of deltas (differences in state)

$$d = \sqrt{(\Delta x + \Delta v_x \Delta t)^2 + \big(\Delta y + \Delta v_y \Delta t\big)^2}$$

   The particles intersect when $d = 2r$, i.e. the particles touch at their circumference, hence by expanding and collecting terms we get the quadratic equation of form $A\Delta t^2 + B\Delta t + C = 0$,

$$\begin{aligned} (2r)^2 &= \Delta x^2 + 2\Delta x \Delta v_x \Delta t + \Delta v_x^2 \Delta t^2 + \Delta y^2 + 2\Delta y \Delta v_y \Delta t + \Delta v_y^2 \Delta t^2 \\ \Rightarrow \big(\Delta v_x^2 + \Delta v_y^2\big)\Delta t^2 &+ \big(2\Delta x \Delta v_x + 2\Delta y \Delta v_y\big)\Delta t + (\Delta x^2 + \Delta y^2 - 4r^2) = 0 \end{aligned}$$

   We observe that $A = \big(\Delta v_x^2 + \Delta v_y^2\big) > 0$ and thus the curve $y = d(\Delta t)$ is concave up. The discriminant for this quadratic equation, $B^2 - 4AC$, is

$$discriminant = \big(2\Delta x \Delta v_x + 2\Delta y \Delta v_y\big)^2 - 4\big(\Delta v_x^2 + \Delta v_y^2\big)(\Delta x^2 + \Delta y^2 - 4r^2)$$

   If this discriminant is ≥ 0, then the particles collide for some value of Δt, and we solve for this Δt. There are two possible roots,

$$\Delta t = \frac{-B \pm \sqrt{discriminant}}{2A}$$

Since the quadratic curve is concave up, we only compute and examine the first root (when the particles are approaching each other). This is

$$\Delta t = \frac{-B - \sqrt{discriminant}}{2A}$$

If $0 \leq \Delta t \leq 1$, then particles P, Q collide during this time step.

Note that, it is possible for $\Delta t < 0$ – this corresponds to cases where particles are overlapping from a previous step. We do not consider these as collisions in the current step and let these two particles phase through each other.

3 Parallelisation Strategy

A large part of the simulation is highly sequential in nature and not amenable to parallelisation. For each simulation step, all collision candidates have to be generated first before sorting is performed, then followed by filtering, before the collisions can be resolved in a parallel manner.

From our sequential implementation, there were some loops and independent function calls that were potentially amenable to parallelisation with OpenMP.

Decisions regarding parallelisation and synchronisation constructs are detailed below, in the order they are encountered in the diagram in Chapter 1.

File, Line	**random.c: 8-13**

Program Fragment	`for (int i = 0; i < n; i++) {`
	`  // Build N particle structs and update pointers`
	`}`

Decision	This loop can be safely parallelised with **`#pragma omp for`**, as the pointer to each particle is written to different indices of the array **`particleArray`**. We did not parallelise this loop however as the overhead of parallelisation is much greater than the benefit.

File, Line	**random.c: 37-55**

Program Fragment	`for (int i = 0; i < n; i++) {`
	`  // Repeatedly place particles without overlapping`
	`}`

Decision	This loop cannot be safely parallelised, since each particle placed randomly in the square needs to check if it overlaps with any of the particles that were previously placed.

File, Line	**random.c: 73-80**

Program Fragment	`for (int i = 0; i < n; i++) {`
	`  // Generate random velocities`
	`}`

Decision	This loop can be safely parallelised with **`#pragma omp for`**, since each iteration of the loop updates different indices of the random velocity array. We did not parallelise this loop due to overhead.

File, Line	**io.c: 47-59**

Program Fragment	`for (int i = 0; i < n; i++) {`
	`    // Print details of every particle for a given step`
	`}`

Decision	This loop cannot be parallelised. Print statements are not atomic, and attempting to parallelise this only messes up the ordering of the printed output to amusing effect.

File, Line	**simulator.c: 55-59**

Program Fragment	`for (int i = 0; i < n; i++) {`
	`    // Initialise collision state of each particle to false`
	`}`

Decision	This loop can be safely parallelised with **`#pragma omp for`**, since each iteration of the loop updates different indices of the collision state array. We did not parallelise this loop however due to overhead.

File, Line	**simulator.c: 62-131**

Program Fragment	`for (int step = 1; step <= s; step++) {`
	`    // Simulate step T`
	`}`

Decision	This loop is inherently unparallelisable. Steps of a simulation are sequential in nature since the $(n+1)$-th step is dependent on the state of the n-th step.

File, Line	**simulator.c: 69-106**

Program Fragment	`#pragma omp parallel`
	`{`
	`  #pragma omp for schedule(dynamic, 1)`
	`  for (int p = 0; p < n; p++) {`
	`  // Check if this particle p will colide with a wall`
	`  // For every particle q of greater ID, check if p, q`
	`  // will collide`
	`  }`
	`}`

Decision	We opted to parallelise this loop, since checking all $\Theta(N^2)$ potential collisions is expensive. However, each iteration of the outer loop (for a particle P) performs a different number of checks on other particles Q. Thus, we opted to use dynamic scheduling with chunk size 1 – which uses an internal work queue to assign threads 1 outer loop iteration each time they are finished with the previous iteration. Despite the extra overhead from the queue, this minimises total execution time relative to other scheduling policies since it distributes work equitably amongst the threads. All threads are implicitly synchronised at the end of the **`#pragma omp for`** construct, before the simulation proceeds.

File, Line	**simulator.c: 81-87, 96-102**

Program Fragment	`#pragma omp critical`
	`{`
	`  // Add a new collision candidate to an array`
	`  // Increment number of collision candidates for that step`
	`}`

Decision	In the parallelised for loop for generating collision candidates, race conditions may occur when multiple threads attempt to add a new collision candidate to the **`cs`** array and increment **`numCollisions`**. Therefore, we protected both critical sections (for particle-wall and particle-particle collisions) within a **`#pragma omp critical`** construct to prevent concurrent modification.

File, Line	**filter.c: 12-34**

Program Fragment	`for(int curIndex – 0; curIndex < *numCollisions; curIndex++) {`
	`  // Filters array of collision candidates down to`
	`  // only valid collisions`
	`}`

Decision	This loop cannot be parallelised. For a given step, collision candidates need to be processed by their total ordering, since accepting a collision as valid may invalidate some of the later collision candidates (as particles are involved in at most one collision per step).

File, Line	**kinematics.c: 8-20**

Program Fragment	`collision_t* curCollision`
	`#pragma omp parallel private(curCollision)`
	`{`
	`  // Compute chunk size`
	`  #pragma omp for schedule(static, chunk_size)`
	`  for (int i = 0; i < *numCollisions; i++) {`
	`  // Resolve each valid collision after sorting and`
	`  // filtering of collision candidates`
	`  // Update state of particles involved`
	`  }`
	`}`

Decision	A particle can only be involved in at most one valid collision. Thus, parallelising this loop can be done safely since at most one thread will be updating the state of any given particle. Since resolving collisions is computationally expensive, we parallelised this loop, using static scheduling with equal chunk sizes to map loop iterations to threads.

File, Line	**kinematics.c: 27-40**

Program Fragment	`for (int i = 0; i < n; i++) {`
	`  // For a given time step, update position of all particles`
	`  // not involved in a collision`
	`  // Also resets state of collided particles`
	`}`

Decision	This loop can be safely parallelised with **`#pragma omp for`**, as each particle's state is only updated by one thread when parallelised. We did not parallelise this loop however as the overhead of parallelisation is much greater than the benefit.

4 Test Conditions and Testcases

4.1 Test Setup

Benchmarking of our sequential, parallel and early-pruning implementations were done on the same lab machines with aid of a Python script to generate input files and a Bash script to execute and log output for each testcase. These files can be found in the **`<impl>BatchRuns`** sub-folder of the respective implementation.

The Intel i7-7700K node tested was soctf-pdc-001 and the Intel Xeon Silver 4114 node tested was soctf-pdc-010.

Each testcase was run five times and the fastest execution time was retained as the datapoint for that testcase.

To replicate our results for a given implementation, compile all files in that folder with **`gcc *.h`** followed by **`gcc -fopenmp *.c lm`**. Then, execute the corresponding Bash script in the associated **`<impl>BatchRuns`** sub-folder.

4.2 Random Testcases

For benchmarking, testcases ran in *perf* mode and the initial states of particles were not provided. Variables of the simulation was adjusted for each testcase.

The simulation parameters of the default testcase is

- $N = 1000, L = 20000, r = 1, S = 1000$

The testcases that were executed for each implementation are as follows.

1. **Sequential implementation**

    - Varying N only: $N = 250,\ 375,\ 500,\ 750,\ 1500,\ 2000$
    - Varying L only: $L = 5000,\ 7500,\ 15000,\ 20000$
    - Varying r only: $r = 1, 2, 3, 4, 6, 8, 16$
    - Varying S only: $S = 250, 375, 500, 750, 1500, 2000$

2. **Parallel implementation**

    - T = number of OpenMP threads
    - Varying both N and T together
        - $N = 250,\ 500,\ 1000,\ 2000$
        - $T = 1, 2, 4, 6, 7, 8, 9, 10, 12, 16, 19, 20, 21, 24, 28, 32, 40, 64$

4.3 Special Testcases

In addition to the random testcases, we designed a few additional special testcases to assert correctness of our simulator, alongside an extra Python script (**`generateAnimation.py`**) to visualise the simulation output. We provided the .GIF visualisations for each of these testcases.

Each of these testcases are named accordingly, and should be run with `SLOW_FACTOR` set to a power of two (we recommend 16) for accuracy. Running these testcases with other values of `SLOW_FACTOR` may lead to different results due to floating-point errors or insufficient granularity of simulation.

These testcases are as follows.

- `cradle.in` – a horizontal row of particles that behaves like Newton's cradle, endlessly bouncing between two walls
- `cross.in` – two particles bouncing between opposite diagonals of the square, without ever colliding
- `diagonalLine.in` – a diagonal row of particles bouncing from left to right, with glancing collisions
- `square.in` – four particles colliding at the same time at the centre; all four particles are deflected exactly at right angles from their initial trajectory
- `triangle.in` – three particles colliding at the same time, with resolution to prioritise the collision between the pair of particles with lowest ID

5 Execution Time Results

All plots were generated with the help of R. The raw data from the **`perf stat`** command is available in the submission.

5.1 Sequential Implementation

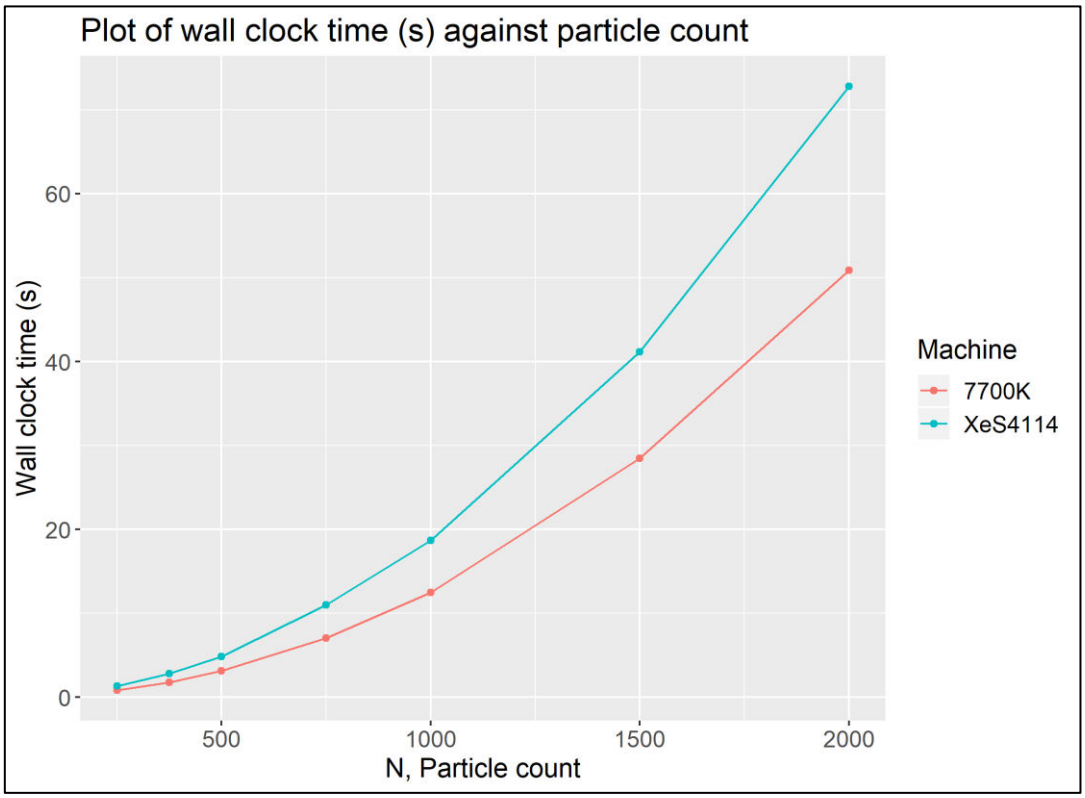
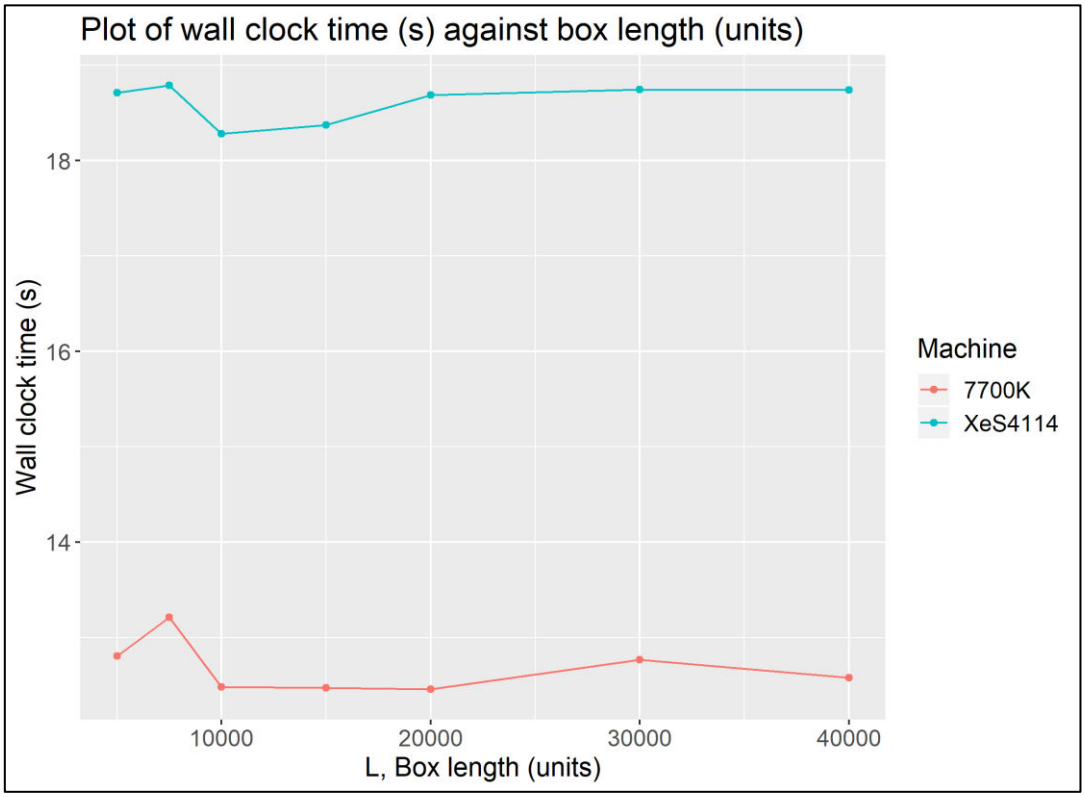


Figure 5.1 – Plot of execution time against particle count, N

Figure 5.2 – Plot of execution time against box length, L



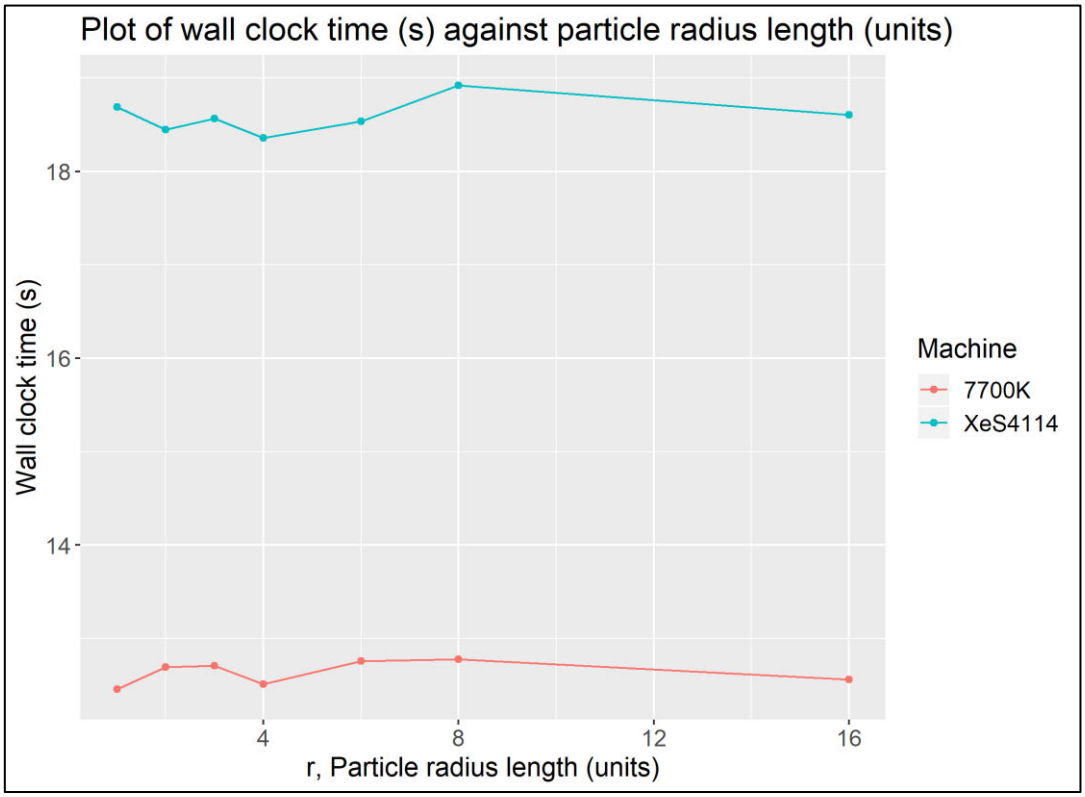
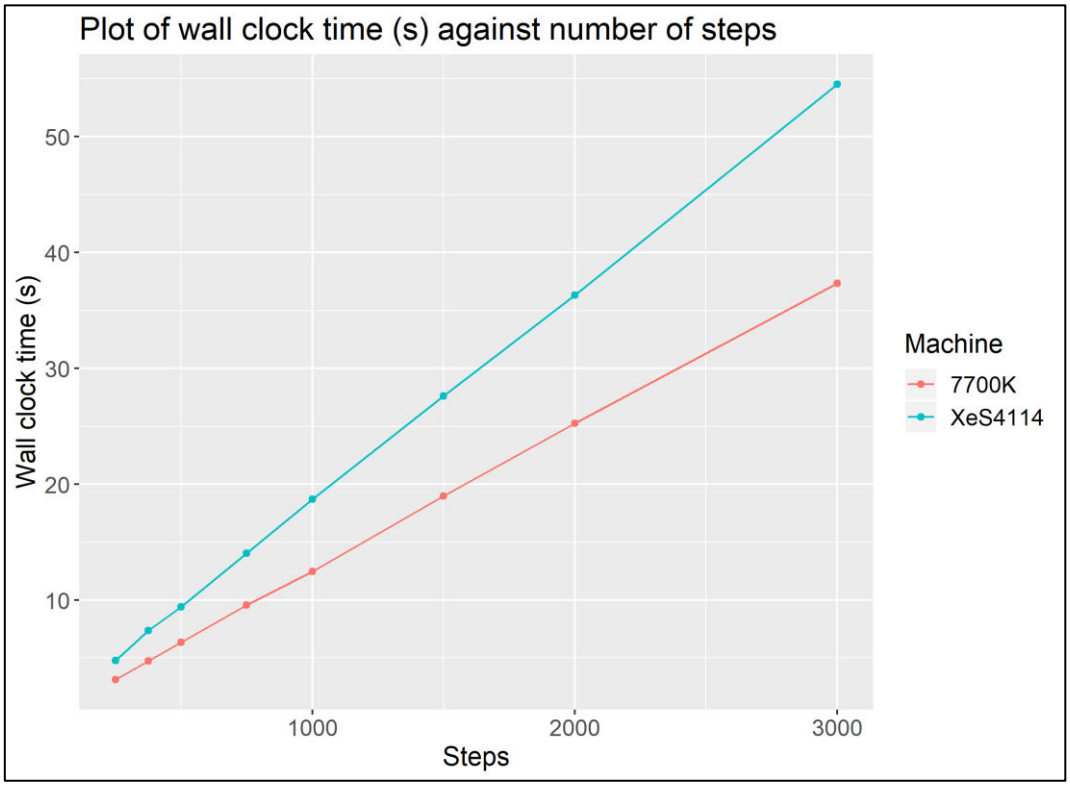


Figure 5.3 – Plot of execution time against particle radius, r

Figure 5.4 – Plot of execution time against box length, L



5.2 Parallel Implementation

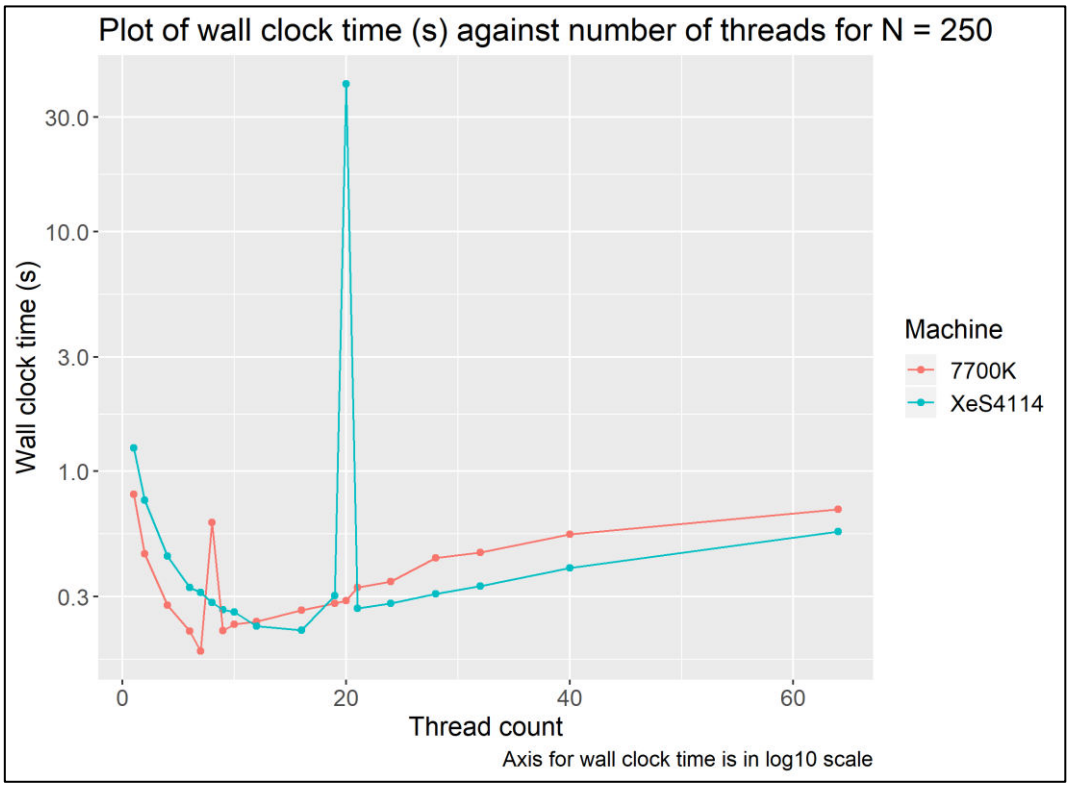
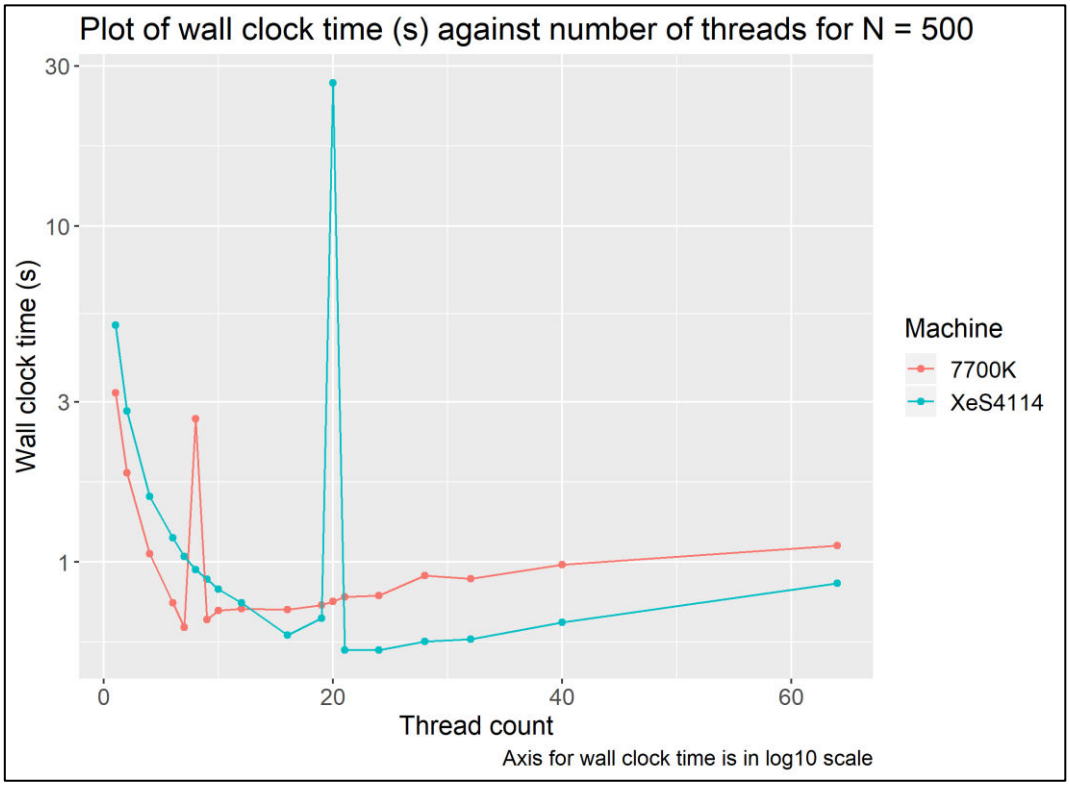


Figure 5.5 – Plot of execution time against number of threads T for $N = 250$

Figure 5.6 – Plot of execution time against number of threads T for $N = 500$



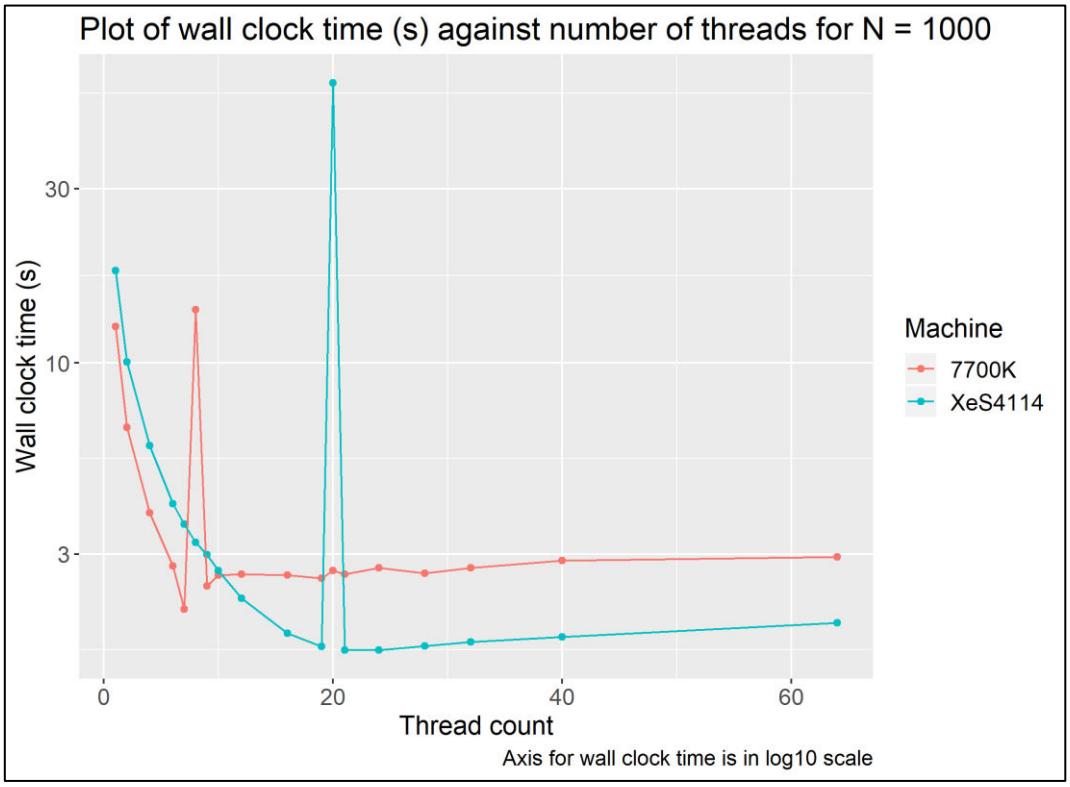
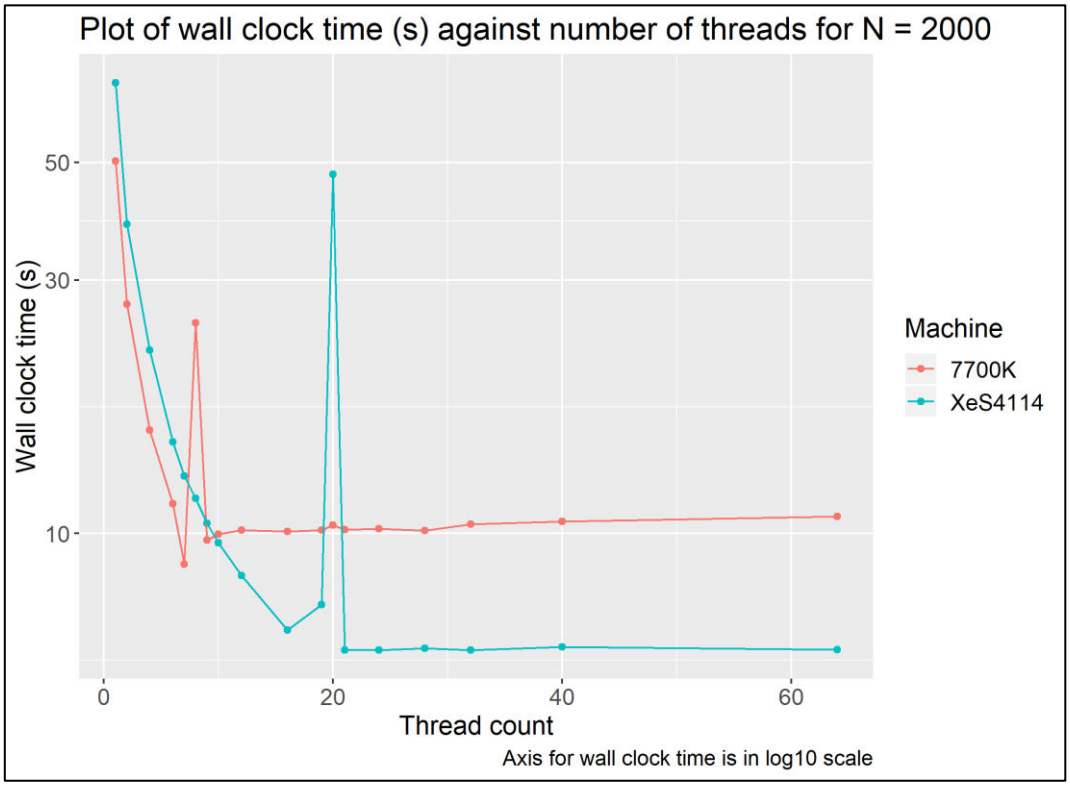


Figure 5.7 – Plot of execution time against number of threads T for $N = 1000$

Figure 5.8 – Plot of execution time against number of threads T for $N = 2000$



5.3 Speedup between sequential and parallel implementation

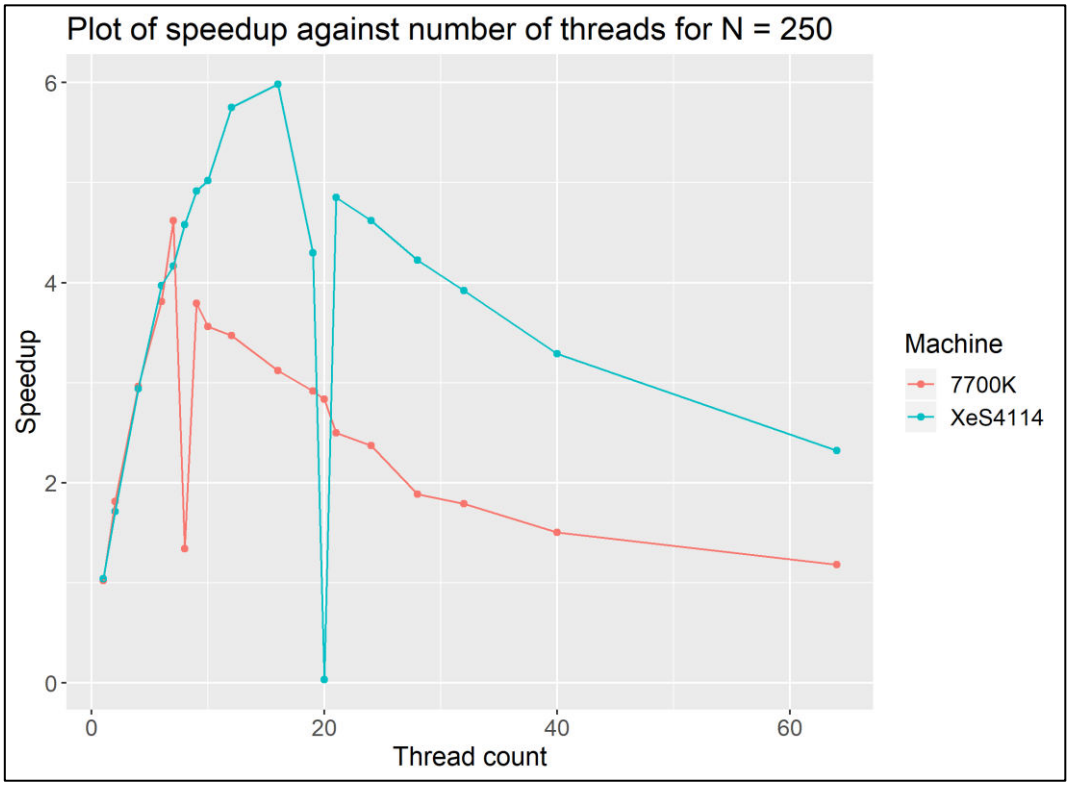
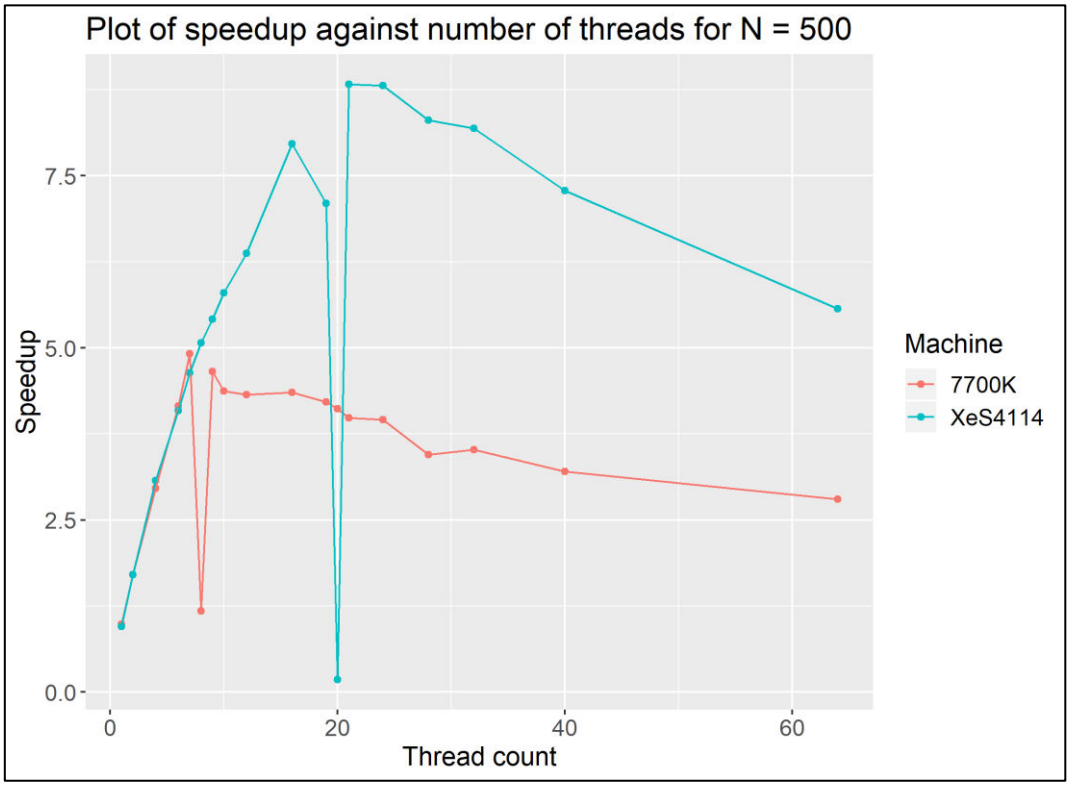


Figure 5.9 – Plot of speedup against number of threads T for $N = 250$

Figure 5.10 – Plot of execution time against number of threads T for $N = 500$



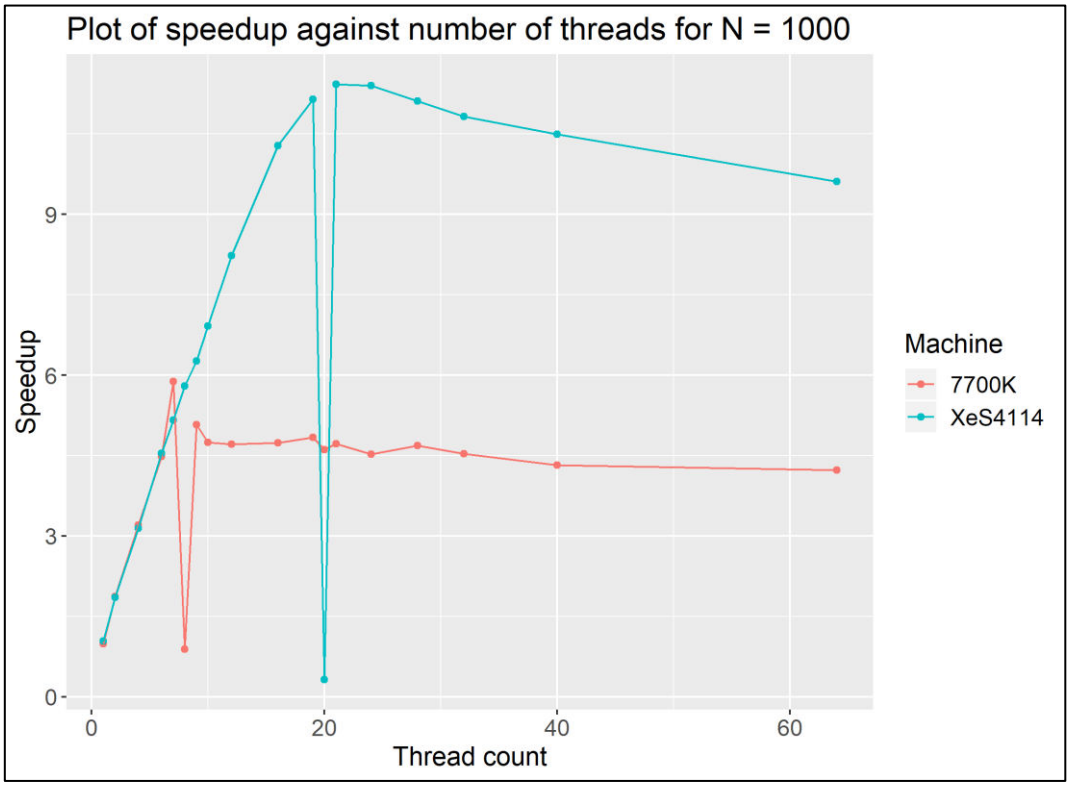
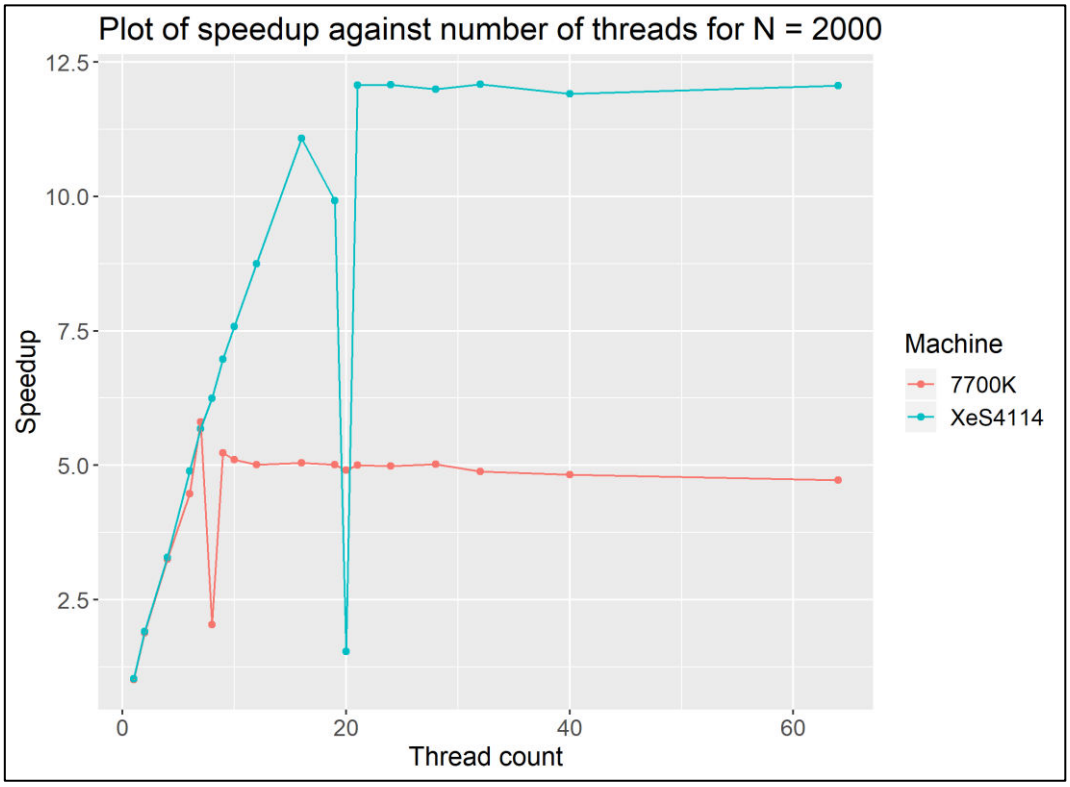


Figure 5.11 – Plot of speedup against number of threads T for $N = 1000$

Figure 5.12 – Plot of execution time against number of threads T for $N = 2000$



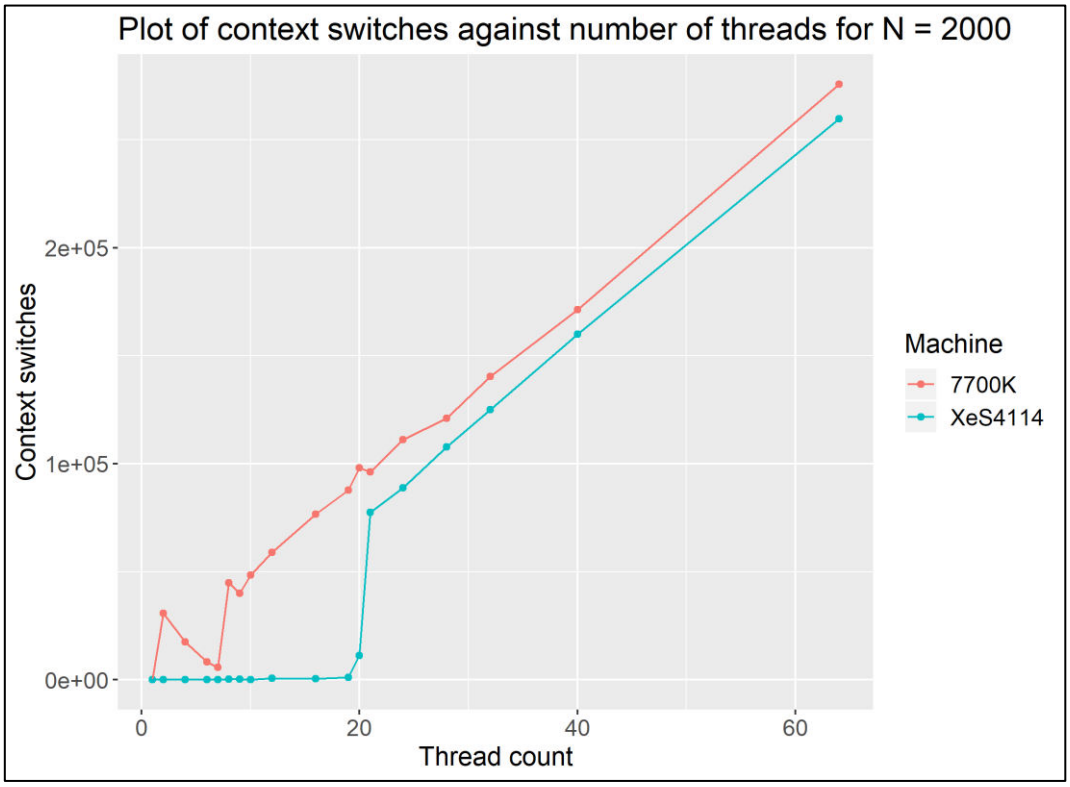


Figure 5.13 – Plot of number of context against number of threads T for $N = 2000$

6 Discussion: Sequential Implementation

6.1 Expectations

For a single step in the simulation, the different components contribute different amounts to the execution time for that step:

- Checking $\Theta(N^2)$ potential collisions, each incurring a cost of T_{check}: $O(N^2)T_{check}$
- Sorting $\Theta(N)$ collisions (since we assume $P(collision) \ll 1$): $O(N \lg N)$
- Resolving $\Theta(N)$ collisions (since each particle is involved in at most one collision), each incurring a cost of $T_{resolve}$: $O(N)T$
- Updating positions of particles not involved in any collisions: $O(N)$

Thus, the execution time per step is $O(N^2)$, since T_{check} is constant $\Rightarrow$ the overall execution time is $O(N^2S)$.

Hence, we expect that the overall execution time is dominated by the work required for the collision checking stage.

6.2 Conclusions

From Figure 6.1, we see that the execution time increases quadratically with the number of particles N, which agrees with our expectations in (6.1).

From Figure 6.2, we observe that the execution time appears to be constant with respect to the size of the box L. Increasing L increases the initial velocity limit of particles and reduces the average particle density – reducing the number of particle-wall collisions but not the number of particle-particle collisions.

From Figure 6.3, we also observe that the radius of particles does not appear to influence the execution time. Even when the radius of particles is $r = 16$, particles still only occupy a negligible fraction of the total area of the square, thus the total number of particle-particle collisions does not rise appreciably relative to the number of particle-wall collisions.

- Changing either L or r does not change the amount of work performed in checking collisions for each step, hence they rightfully do not affect the overall execution time.

From Figure 6.4, the execution time appears to grow linearly with the number of steps S of the simulation. For a random simulation, the average number of collision per time steps should be similar, and all $\Theta(N^2)$ potential collisions are checked per time step. Thus, the computation work done per time step is approximately equal, and hence this agrees with our expectations in (6.1).

Of the above results, only the dependence of execution time on the number of particles N was notable. Hence, for the parallel implementations, we opted to only vary N amongst the simulation parameters.

7 Discussion: Parallel Implementation

7.1 Expectations

We expected our parallel implementation to demonstrate an increasing speedup over the sequential implementation with an increasing number of threads T, up to an optimum number of threads T_{opt}. This is because we employed dynamic scheduling to distribute work equitably amongst the OpenMP threads.

7.2 Conclusions

From Figures 5.5 – 5.8, we observed that the execution time for a fixed N generally decreases with an increasing number of threads T, up to an optimum number of T. Here, the optimal number of threads T_{opt} is shown for the Intel i7-7700K and Xeon Silver 4114, which have 8 and 20 logical cores respectively.

N	Optimal T for i7-7700K	Optimal T for Xeon 4114
250	7	16
500	7	21
1000	7	21
2000	7	21

For both processors and all N tested, the optimal number of threads T_{opt} was not equal to the number of logical cores they possessed. We hypothesise this occurs due to scheduling issues and contention over hardware resources, since Figure 5.13 shows that the number of expensive context-switches begins to rapidly increase when the number of threads T is 1 less than the number of logical cores of that processor.

Notably, when T is equal to the number of logical cores, the overall execution time degrades significantly, to performance that is sometimes worse than the sequential implementation. This is despite us ensuring no other concurrent processes were running on the nodes when the testcases were being executed.

N	Maximum s for i7-7700K	Maximum s for Xeon 4114
250	4.6	6.0
500	5.0	8.8
1000	6.0	11.5
2000	6.0	11.9

From Figures 5.9 – 5.12, we also observe the trend that the maximum speedup s (rounded to 1 d.p.) over the sequential implementation increases with the number of particles N. This is expected since the bulk of the computational work performed by the simulation occurs during the (parallelised) collision checking stage, which increases rapidly with N.

Additionally, we also expected that the asymptotic speedup will not equate the total number of logical cores of a processor, due to three reasons. First, not all of the work in each simulation step is parallelised, as there remains serial portions (e.g. filtering of collision candidates). Next, the work of collision checking may not be distributed equally amongst the threads despite dynamic scheduling, since faster particles have a greater probability to collide with walls or other particles. Third, the critical section also results in some threads blocking whilst waiting to add a new collision candidate to the array.

7.3 Benefits of discrete simulation

Discrete simulation with arbitrary time-units possesses a benefit over execution-time units, in that we can avoid introducing additional floating-point errors associated with simulating with execution-time units. Furthermore, it simplifies the simulation since the state of all particles in the following step can be directly computed from their states in the previous step.