CS3210 Parallel Computing
Assignment 2 Report

Discrete Particle Simulation with MPI

Keven Loo Yuquan & Lee Yong Jie, Richard

Contents

1 MPI Program Design

The discrete particle simulation is implemented fully in MPI with the master-slave pattern. The overall architecture of the simulator is summarised in Figure 1 below.
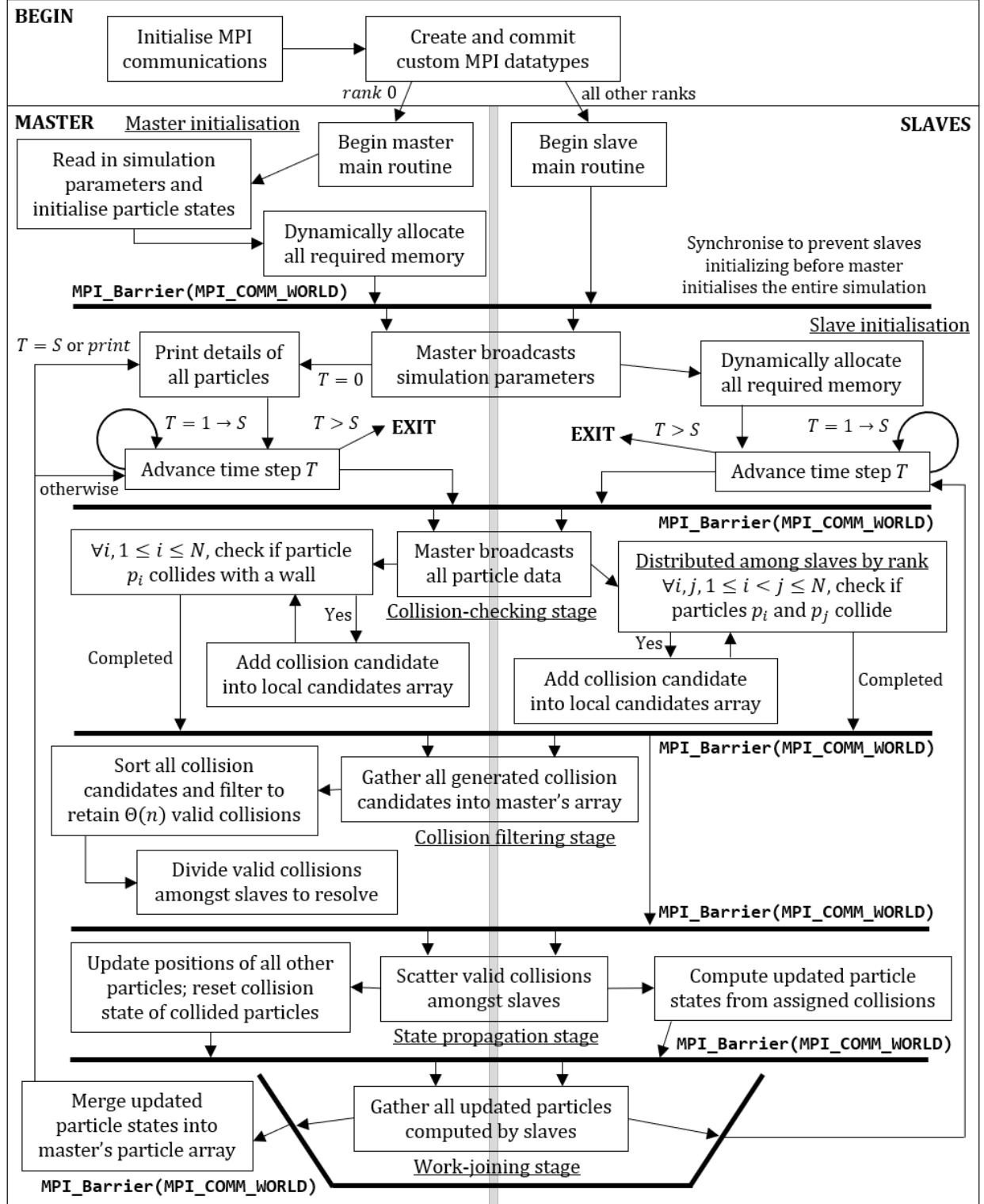

Figure 1: Overall simulator design for MPI implementation

2 Implementation Assumptions and Details

2.1 Assumptions

There are no changes to the assumptions made in the discrete particle simulation from the CUDA implementation. They are reproduced here for convenience.

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

   - Implications

     (a) If a particle collides with a wall after any collision, it is placed at the wall at the end of that time step, and will immediately collide with the wall at the beginning of the next time step

     (b) 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

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

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

   - Implication

     (a) 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 <u>grid arrangement</u> in a square with area L^2)

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

   - Implication

     (a) 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

Changes in general implementation details from the previous report are in blue.

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

   (a) We use the pseudo-random number generator function `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: $ID, x, y, v_x, v_y, w_c, p_c$.

3. Our simulation has an additional parameter `SLOW_FACTOR` in `simulator.cu` 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 multiplied by `SLOW_FACTOR` to compensate, i.e. each original step now corresponds to `SLOW_FACTOR` "micro-steps".

4. Collisions are of two types: particle-wall collisions or particle-particle collisions. We describe a particle-wall collision as $(P, \texttt{WALL})$ and a particle-particle collision as (P, Q). `WALL` is defined with `#define WALL -1`.

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

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 (priority by time)
   ■ If C_1 and C_2 occur at the same time
      □ $C_1 < C_2$ if ID of P in $C_1 <$ ID of P in C_2 (priority by ID)
      □ If C_1 and C_2 both involve the same particle P
         ■ $C_1 < C_2$ if C_1 is a wall collision (priority by type)

■ If C_1 and C_2 are both particle-particle collisions

□ $C_1 < C_2$ if ID of Q in $C_1 <$ ID of Q in C_2 (priority by ID)

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

   (a) A particle is treated to have collided with a wall if it would come within a distance of $\epsilon = \texttt{1E-8}$ to the wall within that time step.

7. All possible particle-particle 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 position 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 time step $0 \leq \Delta t \leq 1$. From Pythagoras' theorem, the distance between them is

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

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

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

   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 $A(\Delta t)^2 + B\Delta t + C = 0$,

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

5

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

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

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,

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

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

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

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.

10. To ensure the collision candidates array in the master process can accommodate $\Theta(N^2)$ potential collisions (in the limit of large N when particle-particle collisions dominate), we dynamically allocate an array with sufficient space to store $N^2/2$ `collision_t` structs.

11. To allow our `particle_t` and `collision_t` structs to be transmitted by MPI, we created the matching MPI datatypes `MP_PARTICLE` and `MP_COLLISION`.

    • Note that the `MPI_` prefix is reserved for the OpenMPI library, hence the use of the `MP_` prefix instead.

12. The structs themselves have also been modified for the MPI implementation.

    • Due to the distributed-memory nature of MPI, the `particle_t` pointers in `collision_t` are no longer useful - they have been replaced with two `int`s, specifying the particle(s) involved in the collision by their ID

    • The original `particle_t` struct resulted in alignment issues during transmission with MPI, due to **structure padding** in C. To mitigate this, a dummy `int` member `dummy` was added after the `id` member to align the `double` members along 4-byte word boundaries.

3 Parallelisation Strategy

3.1 OpenMP and CUDA Implementation

There are no changes to the parallelisation strategy of our OpenMP and CUDA implementations from Assignment 1. Please refer to the reports for Parts 1 and 2 of Assignment 1 for details.

3.2 Task Decomposition

Similar to the CUDA implementation, the computation of each step comprises three distinct stages, two of which contain multiple tasks:

- Collision-checking: (1) particle-wall and (2) particle-particle

- Filtering: prioritising valid collisions

- State propagation: (1) resolving collisions and (2) updating positions

These three stages must be completed serially for each step, which requires inter-stage synchronisation points. However, within each stage, it is possible to compute the tasks in parallel, by dividing the work amongst MPI processes.

3.3 Programming Pattern

Since the tasks in the simulation have significant dependencies and thus require heavy coordination, we decided to implement our MPI program with the **master-slave** pattern for P processes.

The process with rank 0 is designated the **master process** and will be responsible for:

- Reading in the simulation parameters and initial particle states, initialising particles with random positions and velocities using a **uniform random distribution** if not provided

- Performing all printing operations of the simulation state to `stdout`

- Maintaining the updated simulation state at each time step S

- Assigning work to slave processes, and retrieving all work after computation has completed

All other processes with ranks $1, 2, ..., P - 1$ are designated **slave processes**, and will only perform computation.

3.4 Master-Slave Work Division

Since the master process is heavily involved in coordination, we assign it small tasks. This allows the master to perform some work whilst the slaves complete the bulk of the computation. The computation routines for the master are:

1. `MASTER_checkWallCollisions` - checks all N particle-wall collisions

2. `MASTER_updateParticles` - updates state of uncollided particles; resets collision status of collided particles

In contrast, we assign large tasks to the slave processes, who will divide up the computation amongst themselves (the data distribution is detailed in section 3.5). The computation routines for the slaves are:

1. `SLAVE_checkCollisions` - checks all $\Theta(N^2)$ particle-particle collision pairs

    - **Implicit assignment** of particle-particle collision pairs to each slave - each slave determines the computations it performs based on its rank

2. `SLAVE_resolveCollisions` - resolves assigned valid collisions by computing the updated state of the involved particles

    - **Explicit assignment** of valid collisions to each slave - master assigns each slave a unique set of collisions to resolve using `MPI_Scatterv`

For the explicit assignment of valid collisions to resolve, the master first performs an `MPI_Scatter` to inform each slave of the number of collisions it will be assigned. The master then scatters a varying number of valid collisions to each of the slaves with the variable-message-size variant of `MPI_Scatter`, `MPI_Scatterv`.

Each of the slaves will also generate a varying number of elements (collision candidates or updated particles) from its tasks. For the master to collect all these elements, it first performs an `MPI_Gather` to retrieve the number of items each slave will send. The master then gathers a varying number of elements from each of the slaves with the variable-message-size variant of `MPI_Gather`, `MPI_Gatherv`.

These variable-message-size variants of the collective communication operations provide more flexibility but require two additional arrays:

1. `counts` - integer array with the ith element specifying the number of elements to send to the process with rank i

2. `displs` - integer array with the ith element specifying the displacement

- (For `MPI_Scatterv`) relative to the start of the send buffer of the root, at which to take the outgoing data for the process with rank i
- (For `MPI_Gatherv`) relative to the start of the receive buffer of the root, at which to place the incoming data from the process with rank i

For `MPI_Gatherv`, the first array `count` is constructed from the initial collective communication operation, whereas it is built by the master for `MPI_Scatterv`. The following are the helper routines executed by the master for these:

1. `MASTER_buildDisplacement` - builds the displacement array `displs` from the counts array `counts` obtained from an earlier call to `MPI_Gather`

2. `MASTER_divideCollisions` - divides the number of valid collisions amongst all the slaves, building both `counts` and `displs` in the process

As part of the coordination role performed by the master, it also executes the following additional routines:

1. `MASTER_filterCollisions` - after collecting all collision candidates into the master's array, sorts and filters to retain the valid collisions

- `MASTER_buildDisplacement` returns the total number of elements sent by all processes in `MPI_Gatherv`, which is used to update the master's `numCollisions`
- Filtering is only performed by the master due to its inherently serial nature (since selecting an earlier collision may invalidate later collisions, as a particle can only collide once in a given time step)

2. `MASTER_mergeResolvedParticles` - after collecting all particles updated from resolving collisions, copies the updates into the master's particle array

3. `MASTER_printAll` - prints the simulation state to `stdout`

3.5 Task and Data Distribution

Since the slaves will only check particle-particle collision pairs for p_i and p_j where $0 \leq i < j < n$ (where i, j are the indices of the particles in the array `ps`), we can visualise all the computations to be performed for each time step as the section above the upper diagonal of a $N \times N$ matrix, as shown in Figure 2. This is similar to the CUDA implementation.
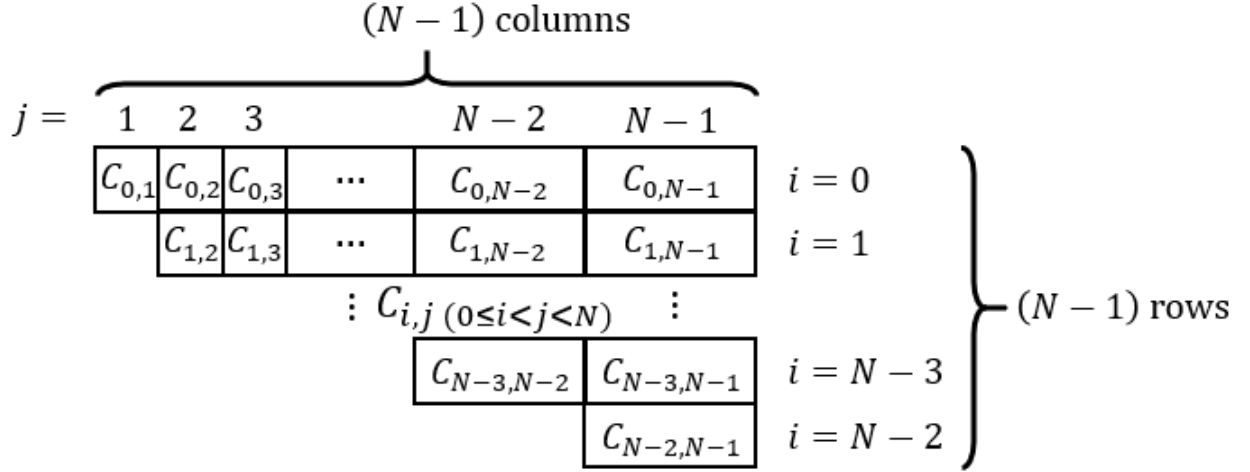
Figure 2: Matrix of particle-particle computations

We considered assigning evenly-sized blocks of rows of computation to each of the slaves. However, this leads to an uneven distribution of work, since the number of potential collisions to check in the ith row is $(N - 1 - i)$. Since the master cannot proceed to filter collisions until all slaves have completed, our MPI program will be severely bottlenecked by the (slowest) slave assigned the block with the longest set of rows.

We then considered letting the master dynamically assign one row of computation to each slave, collecting the result when the row is complete, before assigning another, until all rows are fully computed.

However, this will increase the granularity of the tasks and the amount of communications required, increasing the overhead substantially. Furthermore, it prevents the master from performing any useful work alongside the slaves, which necessitates distributing the particle-wall collision checks amongst the slaves as well.

Therefore, similar to our CUDA implementation, since the upper half of the matrix resembles a triangle, we decided to "fold" the lower half of the matrix onto the upper half, by reflecting it in both axes and stitching it together to form a rectangle, as shown in Figure 3.
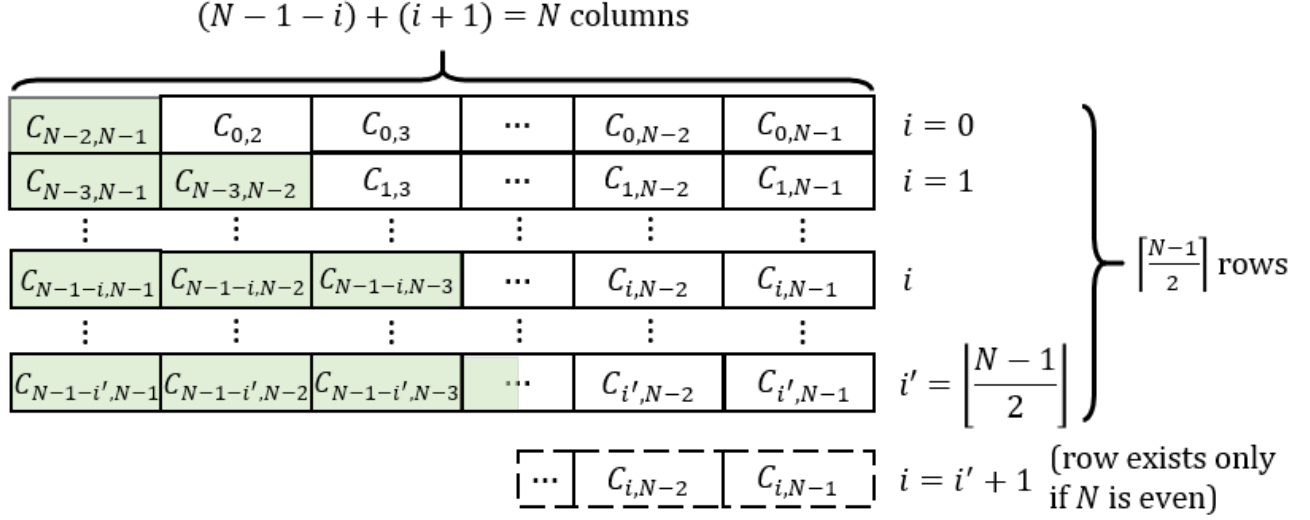
$(N-1-i)+(i+1) = N$ columns

$C_{N-2,N-1}$	$C_{0,2}$	$C_{0,3}$	$\cdots$	$C_{0,N-2}$	$C_{0,N-1}$	$i=0$
$C_{N-3,N-1}$	$C_{N-3,N-2}$	$C_{1,3}$	$\cdots$	$C_{1,N-2}$	$C_{1,N-1}$	$i=1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$C_{N-1-i,N-1}$	$C_{N-1-i,N-2}$	$C_{N-1-i,N-3}$	$\cdots$	$C_{i,N-2}$	$C_{i,N-1}$	i
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$C_{N-1-i',N-1}$	$C_{N-1-i',N-2}$	$C_{N-1-i',N-3}$	$\cdots$	$C_{i',N-2}$	$C_{i',N-1}$	$i' = \left\lfloor \dfrac{N-1}{2} \right\rfloor$

$\left\lceil \dfrac{N-1}{2} \right\rceil$ rows

| $\cdots$ | $C_{i,N-2}$ | $C_{i,N-1}$ |

$i = i'+1$ (row exists only if N is even)

Figure 3: Folded matrix of particle-particle computations

This produces a rectangular matrix with $ceil((N-1)/2)$ rows and exactly N columns, with the shaded green cells denoting the reflected portion. Note that if $(N-1)$ is odd, i.e. N is even, that the matrix will not fold perfectly and there will be a leftover (last) row with only $N/2$ computations.

Then, we performed **blockwise assignment** of the rows of computation to each of the slave processes by their rank, as shown in Figure 4.

Blockwise distribution of r rows of computation amongst w slaves

All but last slave check all particle-particle collision in b assigned rows (based on rank)

Block size $b = \left\lceil \dfrac{N}{w} \right\rceil$

| S_0 |
| S_1 |
| $\vdots$ |
| S_{w-2} |
| S_{w-1} |

Last slave (rank w)

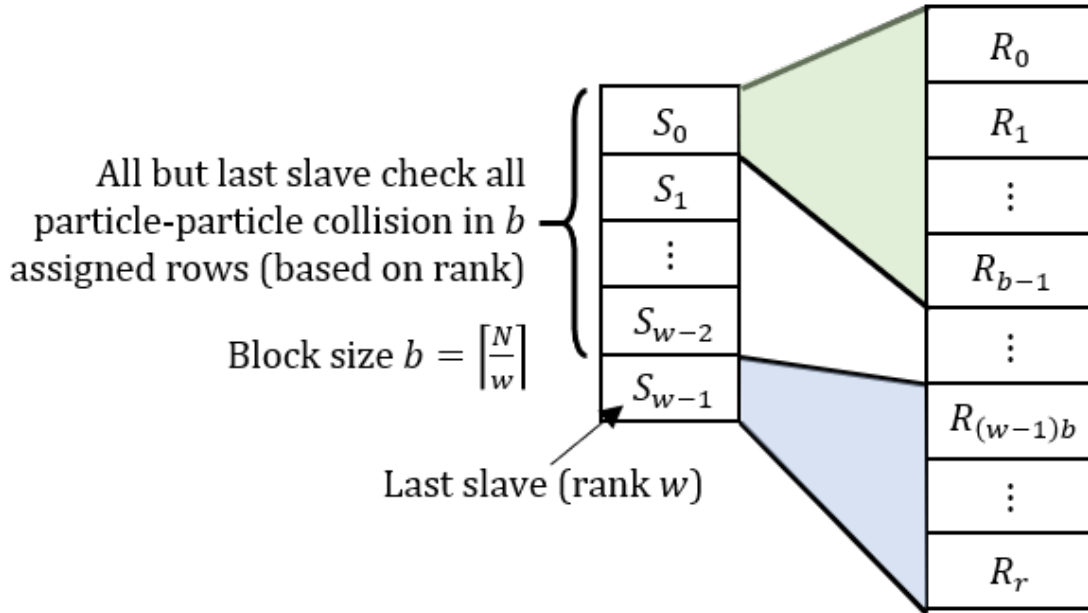| R_0 |
| R_1 |
| $\vdots$ |
| R_{b-1} |
| $\vdots$ |
| $R_{(w-1)b}$ |
| $\vdots$ |
| R_r |

Figure 4: Blockwise assignment of rows of computation to slaves

During the state propagation stage, the master assigns valid collisions to slaves with a **blockwise assignment**, as shown in Figure 5.
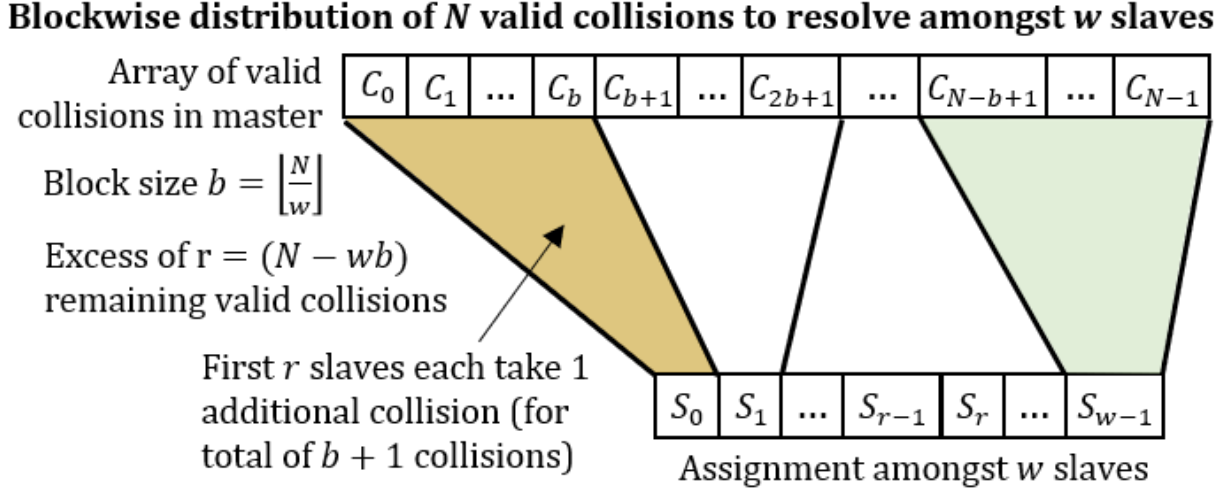

Figure 5: Blockwise assignment of rows of computation to slaves

3.6 Memory Considerations

Since each MPI process has an independent memory space, certain information about the simulation parameters must be replicated across all processes before beginning any computation. These include:

- Simulation parameters `n`, `l`, `r`, `s`

- Derived simulation parameters `minMargin`, `maxMargin`, `max` for the boundaries of the square.

These are broadcast by the master process after initialisating the simulation.

All processes also store the following information required for computation.

- Integer counter for the number of collisions `numCollisions`

  - This is reset to 0 at the beginning of each time step, then incremented locally for each process as it generates collision candidates

  - The master's variable will then be updated with the total count across all processes, and then the total number of valid collisions after filtering

  - As the master assigns valid collisions, `numCollisions` in the slaves are updated to reflect the number of valid collisions assigned to each slave

- Array of N particles `ps`

  – The master's particle array is treated as the **source of truth**, and is broadcast to all processes at the beginning of a time step

  – This was done to avoid complicating the master routine having to send differing data to each slave for each task, since

    (1) the particles required during collision-checking varies non-trivially on the rank of each slave (as the computation matrix is folded), and

    (2) the particles updated by each slave during collision resolution varies in a non-deterministic manner with each time step

  – Whilst this incurs greater communication overhead, it ensures that all slaves see the same the same simulation state; furthermore, `MPI_Bcast` uses an efficient tree broadcast algorithm for good network utilisation

  – It also avoids the need for the slaves to merge in the set of received particles into their particle array, since the `collision_t` struct now references particles by their ID (index in the array)

- Buffer array of N particles `pBuffer`

  – This buffer is used by slaves to store the updated particles computed from resolving collisions

  – This buffer is only used by the master to store all the updated particles gathered from the slaves with `MPI_Gatherv`, to update the master's copy of the particle array with `MASTER_mergeUpdatedParticles`

- Array of collision candidates `cs`

  – This array is used by all processes to store the collision candidates each process generates

  – This array is then used by the master to store all the collision candidates gathered from itself and the slaves with `MPI_Gatherv`, for filtering

  – The array is then re-used by the slaves to store the valid collisions assigned by the master to resolve

  – Its size is $N^2/2$ in the master and $N^2/(2 \times W)$ in each of the $W = P - 1$ slaves

The master process stores the following additional information, as part of its role involves coordinating the entire simulation:

- Array of P integers `numItems`

- Array of P integers `displc`

- Array of N integers describing collision states of all particles, `states`

  - The ith integer describes the collision state of the ith particle
  - `NOT_COLLIDED = 0`; `COLLIDED = 1`
  - This array is only required by the master for filtering valid collisions and is never transmitted
  - The collision states of all particles are reset back to `NOT_COLLIDED` at the end of a time step in `MASTER_updateParticles`

3.7 Synchronisation Requirements

Both the OpenMP and CUDA programming models rely heavily upon shared memory. More specifically,

- In OpenMP, most variables are visible to all threads by default, unless explicitly specified otherwise by the programmer

- In CUDA, all threads have access to the same global and constant memory; additionally, all threads in the same block have access to a common shared memory

In our OpenMP and CUDA implementations, we actively made use of shared memory constructs and thus this necessitates certain synchronisation mechanisms to avoid race conditions. Specifically,

- In OpenMP, all threads added new collision candidates to a common array then incremented a shared counter

  - This was marked a critical section with `#pragma omp critical`

- In CUDA, threads of the `checkWallCollision` and `checkCollision` kernels may be required to add a new collision to an array in global memory and update the `numCollisions` counter (also in global memory)

  - To prevent race conditions, we used the atomic function `atomicAdd` from the CUDA library to increment `numCollisions`
  - Since `atomicAdd` returns the previous value of the counter, this gives each thread a unique index in the array to add its collision candidate to

In contrast, the master and slave processes in our MPI implementation all possess independent memory spaces. Since each process works with its own copy of its variables and data, race conditions are not possible and therefore synchronisation mechanisms to prevent concurrent access are not required.

However, it is essential that all processes are executing in lockstep in the correct stage, as outlined in section 3.2, since dependencies exist between tasks in different stages. Otherwise, the following errors could occur:

- Slaves beginning initialisation before the master broadcasts the simulation parameters, leading to segmentation faults

- Slaves proceeding to the next stage of computation with stale data before the master completes some computation

To achieve this, we explicitly made use of `MPI_Barrier` to synchronise the master and all slaves prior to the beginning of every stage and collective communication operation. Note that this was only possible due to the use of blocking communication operations.

3.8   Network Considerations

For a distributed-memory programming model such as MPI, being sensitive to the network layout of the processing elements is also critical, since message-passing occurs on the network.

We hypothesise that the network layout of the computing nodes in the Parallel-Distributed Computing Lab (COM1-B102) follows closely from their physical arrangement, as shown on the next page in Figure 6.
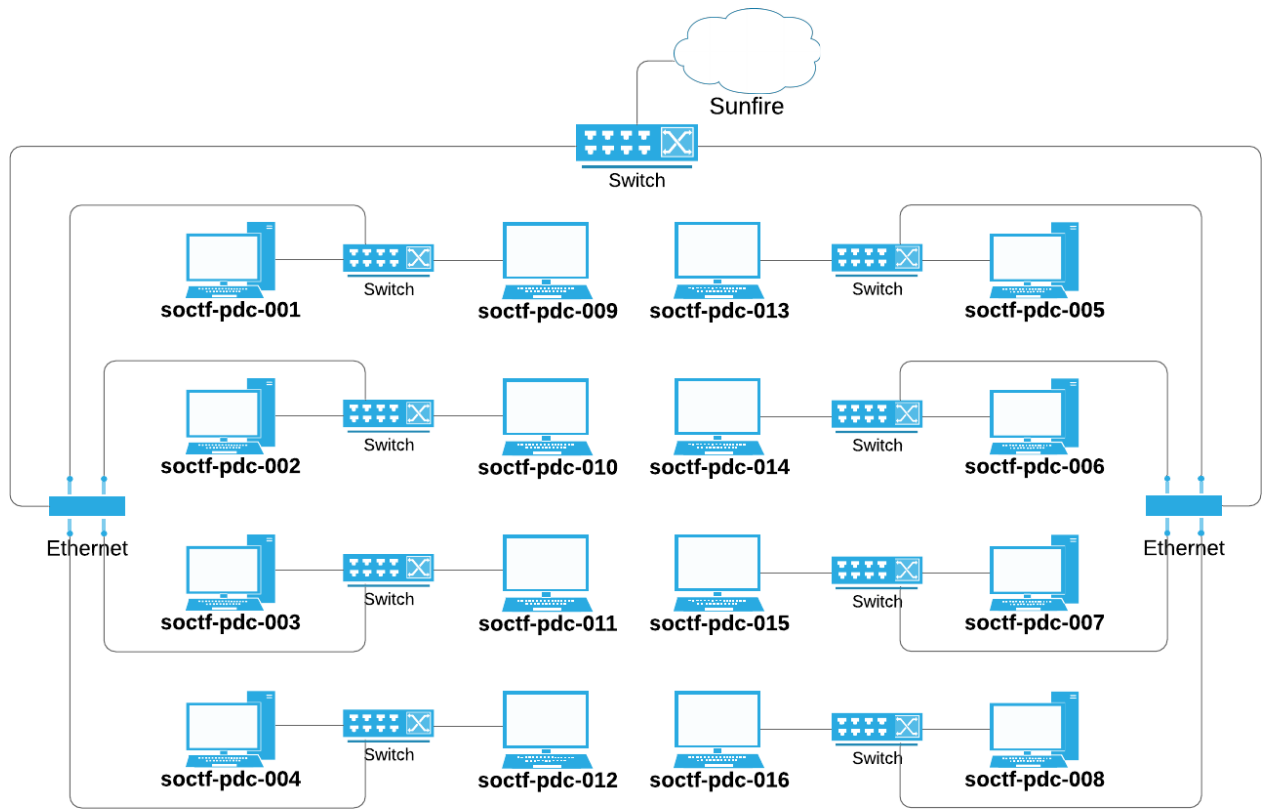
Figure 6: Assumed network layout of computing nodes in COM1-B102

To minimise the effect of network latency on communication operations between MPI processes, we selected nodes in close proximity for each of our testcases.

4 Test Conditions and Testcases

4.1 Test Setup

For ease of deployment, we wrote a Bash script to automate the compilation, testcase generation, profiling and transfer of results back to Sunfire with different sets of source code. The scripts can be found in the folder `/testDispatcher`.

Our previous sequential and parallel OpenMP implementations were benchmarked on both an Intel Xeon Silver 4114 node and an Intel i7-7700K node, specifically soctf-pdc-001 and soctf-pdc-009. We ran the OpenMP program on 20 threads for the Xeon and 8 threads on the 7700K, which we found to produce the highest speedup with respect to our sequential implementation. [2]

Benchmarking of our MPI implementation was done in two configurations - with the Intel Xeon Silver 4114 node, specifically soctf-pdc-001, as the host (master) machine and with the Intel i7-7700K node, specifically soctf-pdc-009 as the host (master) machine.

A Python script was used to generate the input files for each of the implementations. 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, run `./run.sh` in the particular implementation's folder. It is recommended to change the target folder for the `scp` command as it will automatically transfer all the results out when it is complete.

4.2 Random Testcases

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

For each node utilised in a testcase, all logical cores of that node will be utilised to run the program, with a one-to-one mapping of each MPI process to a logical core.

- This is 8 MPI processes for each Intel i7-7700K node, and 20 MPI processes for each Intel Xeon Silver 4114 node

The simulation parameters of the default testcase are

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

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

1. Sequential implementation (Xeon 4114 & i7-7700K)

   - Varying N only: $N = 250, 375, 500, 750, 1000, 1500, 2000$

2. Parallel OpenMP implementation - (Xeon 4114 & i7-7700K)

   - $P = 8$ (for i7-7700K) or $P = 20$ (for Xeon 4114)
   - Varying N only: $N = 250, 375, 500, 750, 1000, 1500, 2000, 3000, 4000, 6000$

3. MPI implementation (**only Xeon 4114 nodes**)

   - M = Xeon 4114 nodes utilised
   - P = Total number of MPI processes
   - Varying both N and M together
     - $N = 1k, 2k, 3k, 4k, 6k, 8k, 12k, 16k, 24k$
     - $M = 1, 2, 4, 8 \implies P = 20, 40, 80, 160$

4. MPI implementation (**only i7-7700K nodes**)

   - M = i7-7700K nodes utilised
   - P = Total number of MPI processes
   - Varying both N and M together
     - $N = 1k, 2k, 3k, 4k, 6k, 8k, 12k, 16k, 24k$
     - $M = 1, 2, 4, 8 \implies P = 8, 16, 32, 64$

5   Execution Results

All plots were generated with the help of R. Some of the plots are not reproduced here for brevity.

The processed data is available in the submission as `.csv` files in `/data/cpu`. Raw data files from the **perf stat** command are available in `/data/cpu/<configType>` as text files with no extension.

Due to the incredibly slow speed of the sequential implementation, data points for $N > 2k$ were extrapolated from existing data. This was also done for the large data points for the OpenMP implementation, which became difficult to benchmark above $N > 6k$.

To be precise, since we have observed that the algorithm's execution time is directly proportional to N^2, a quadratic regression (polynomial of degree 2) was fitted onto existing data points for both the sequential and OpenMP implementations. The extrapolated data points at $N = 3k, 4k, 6k, 8k, 12k, 16k, 24k$ was obtained as a rough estimate for computation of *speedup*. The R^2 obtained from the quadratic fits are at least 0.99991, which is sufficiently close to 1 and indicates that the predicted behaviour will likely resemble the actual execution time if the testcases were run.

Hence, all mentions of *speedup* that follow (regardless of OpenMP or MPI implementation) are made with reference to the **extrapolated execution time** of the sequential algorithm.

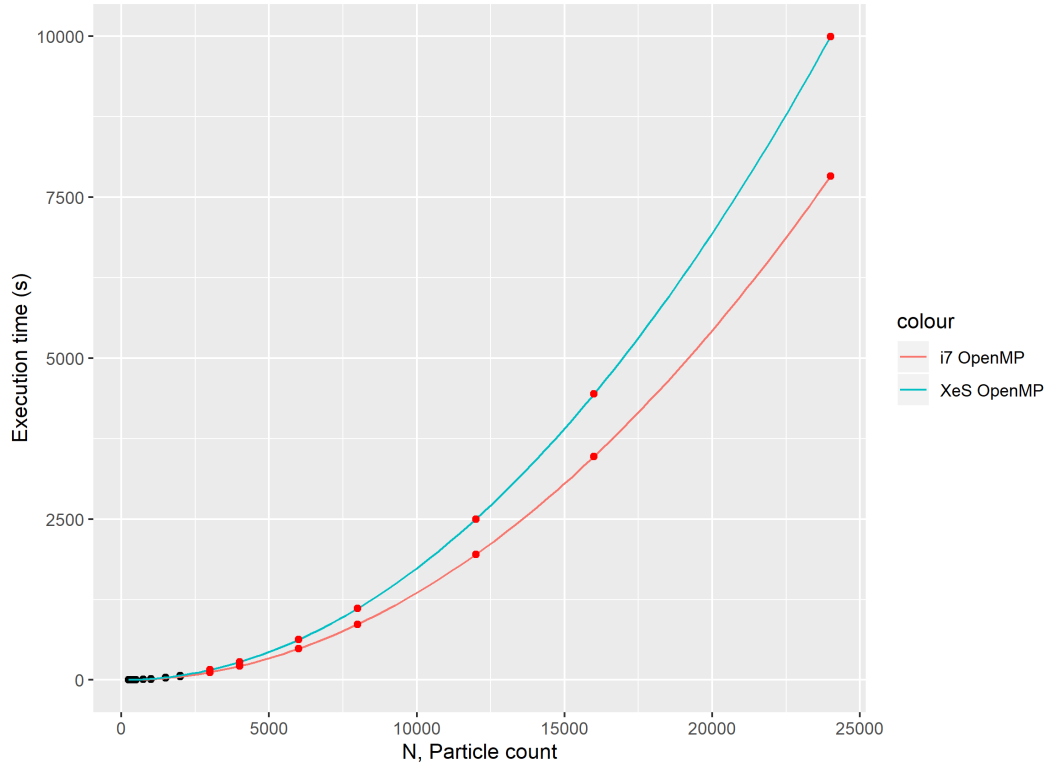## 5.1   Extrapolated Sequential and OpenMP Results

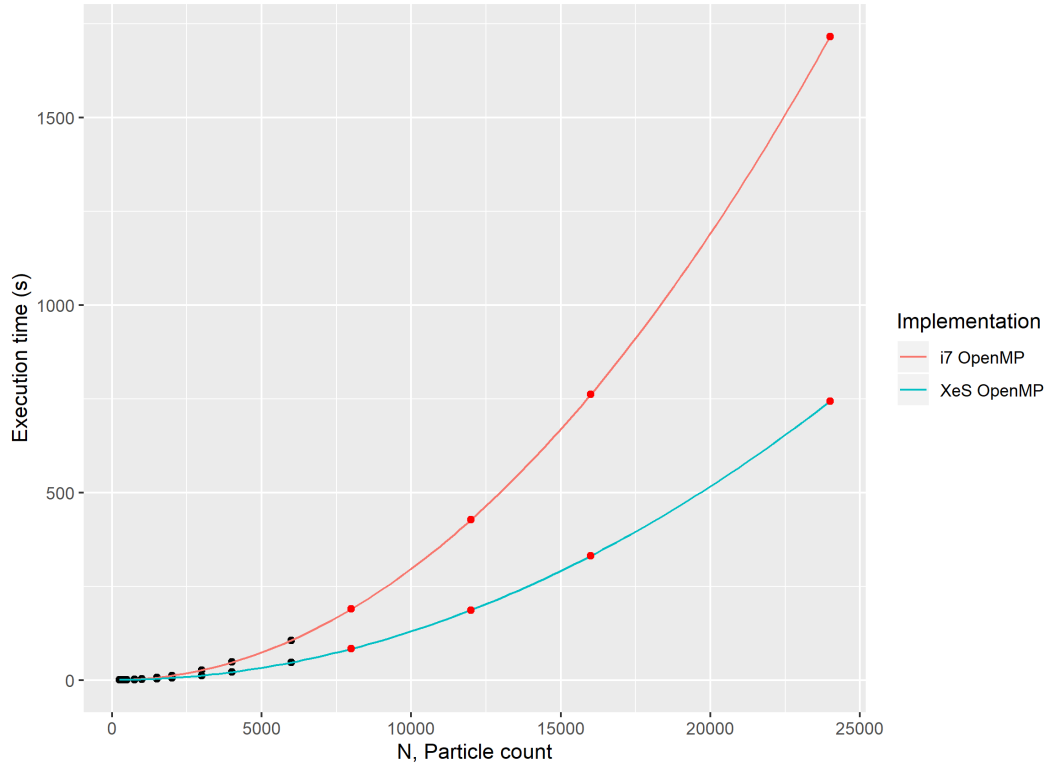Figure 7: Plot of execution time (s) against N (sequential implementation)

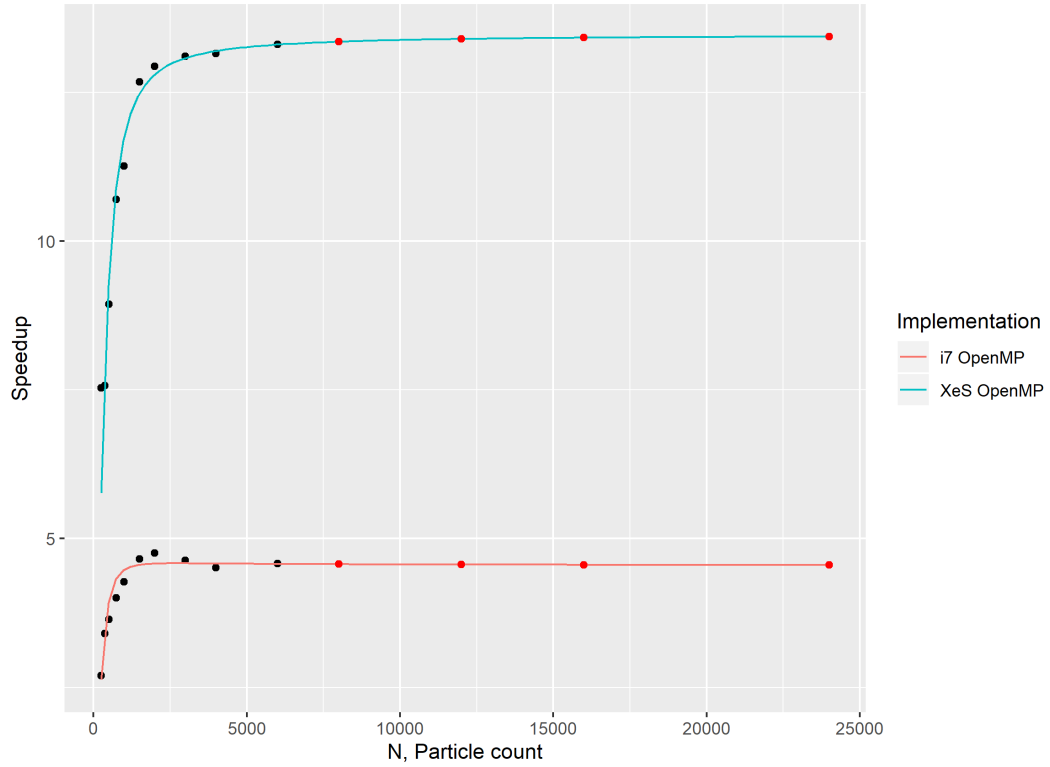Figure 8: Plot of execution time (s) against N (OpenMP implementation)

Figure 9: Plot of speedup against N (OpenMP implementation)
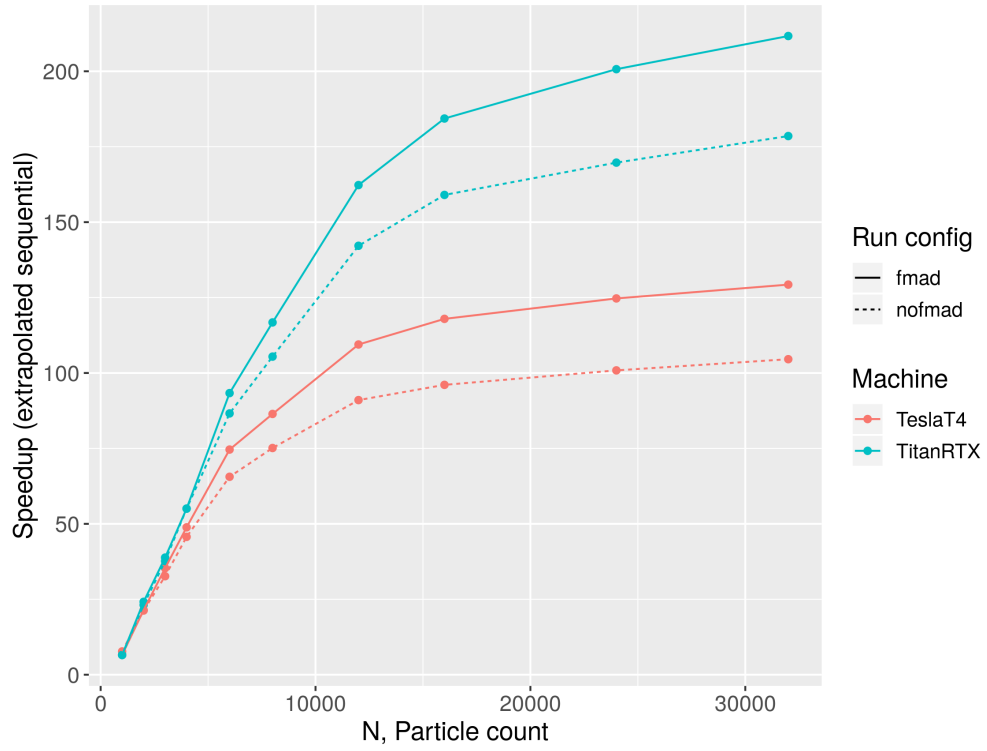
5.2 Additional CUDA Results

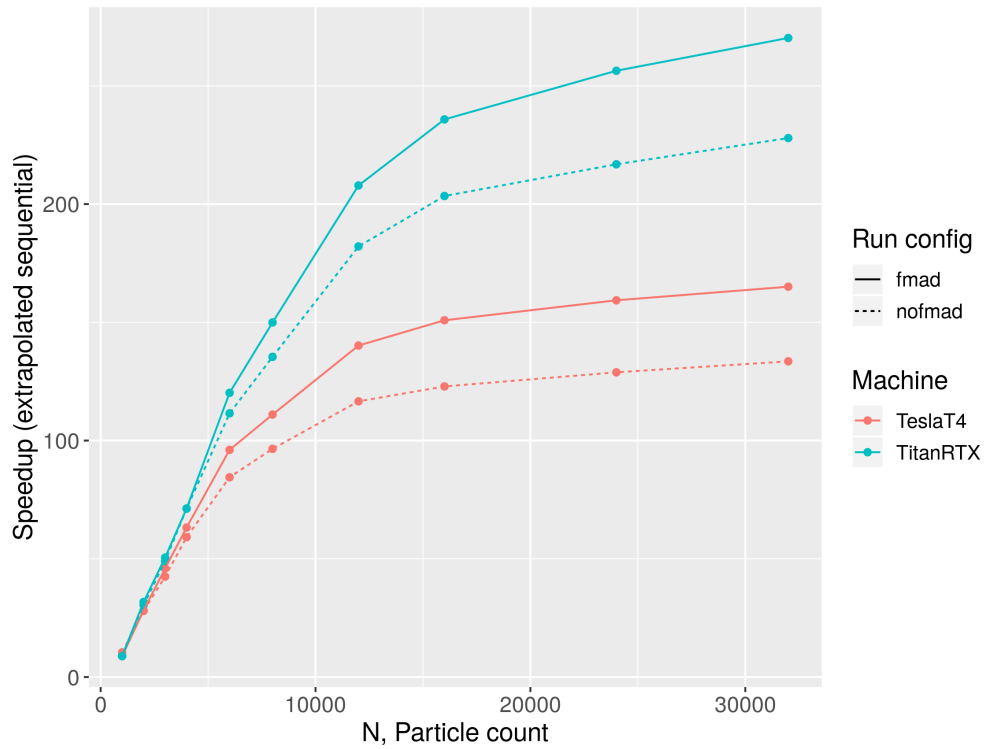Figure 10: Plot of speedup against N (against i7-7700K sequential results)

Figure 11: Plot of speedup against N (against Xeon 4114 sequential results)

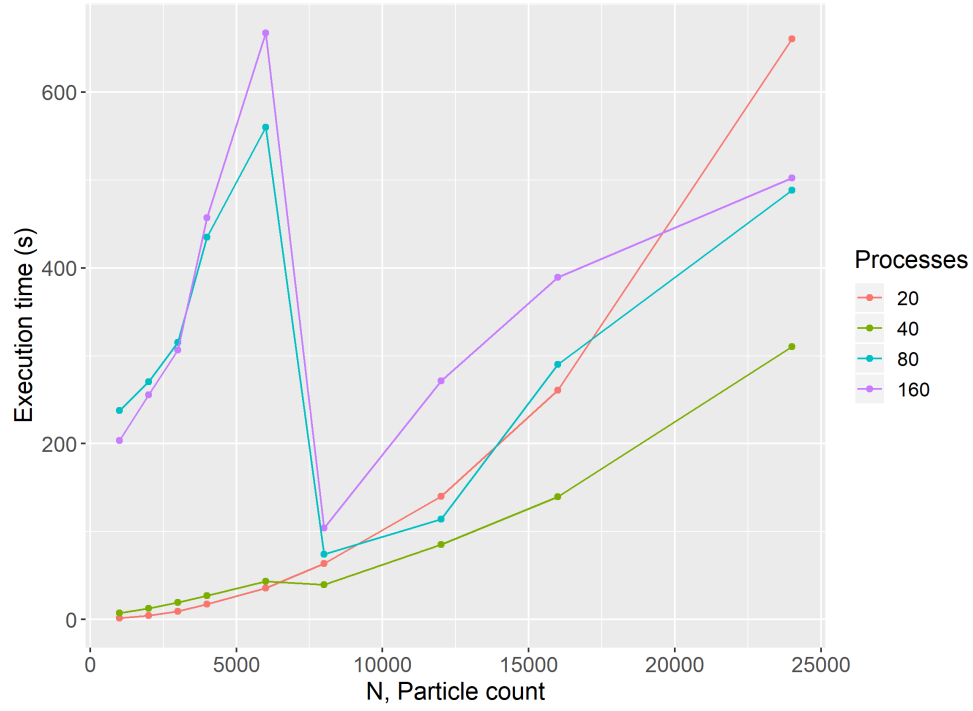## 5.3   MPI Results (Xeon 4114 configuration)

Figure 12: Plot of execution time (s) against N for varying processes (Xeon 4114 configuration)
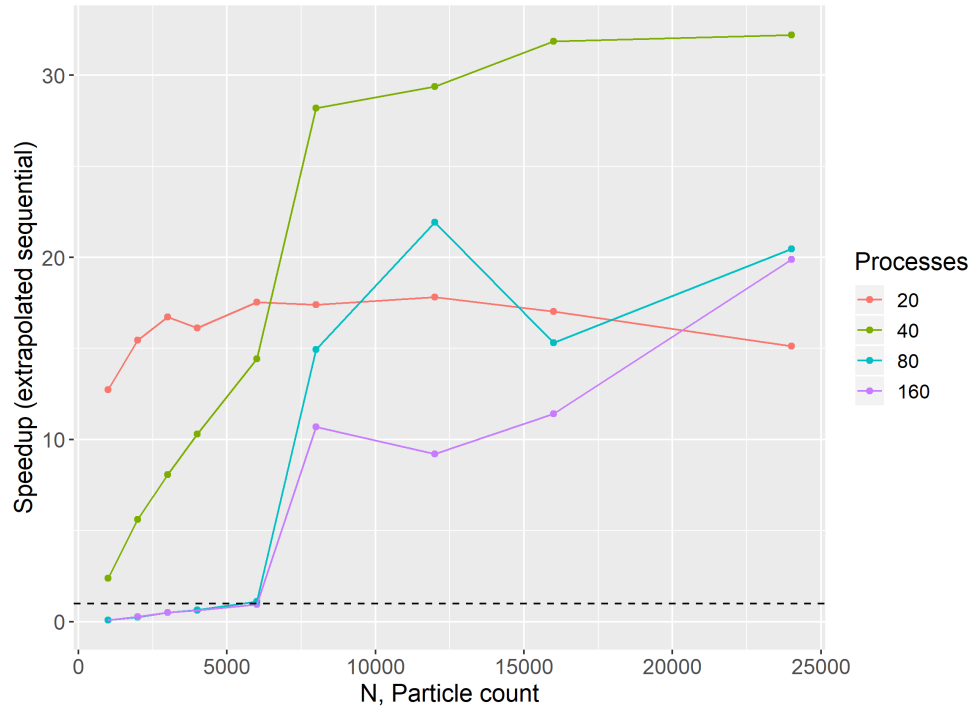

Figure 13: Plot of speedup against N for varying processes (Xeon 4114 configuration)

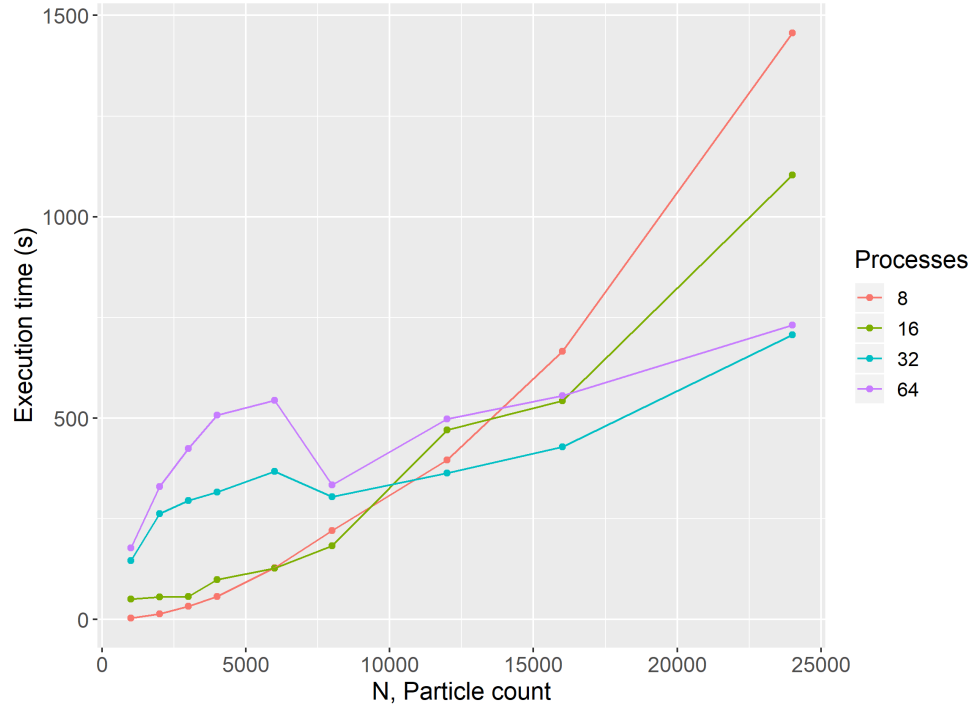## 5.4   MPI Results (i7-7700K configuration)

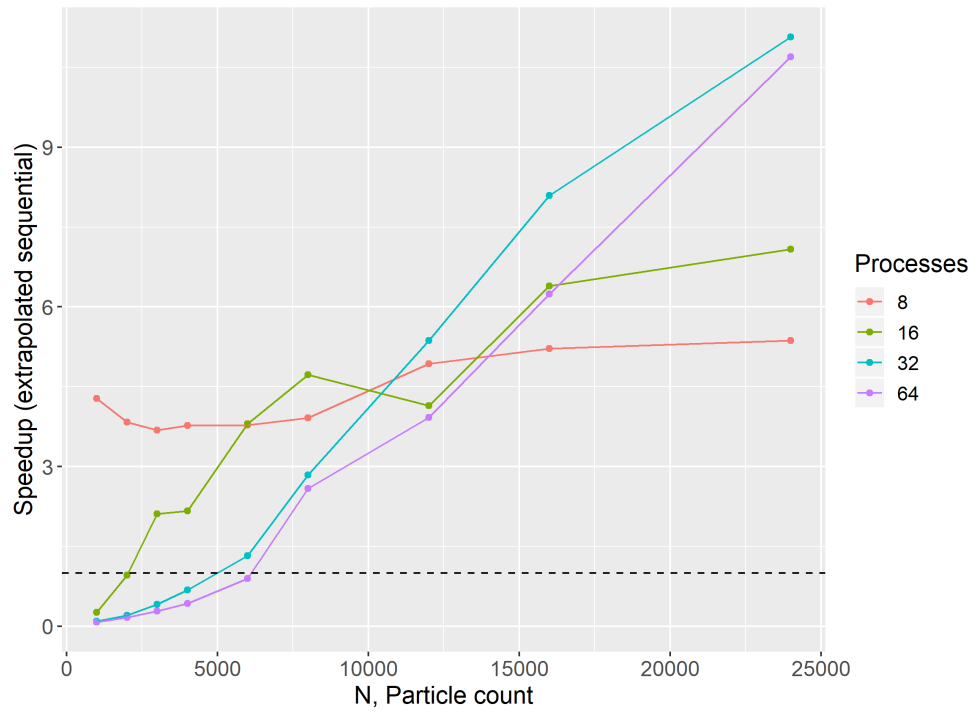Figure 14: Plot of execution time (s) against N for varying processes (i7-7700K configuration)

Figure 15: Plot of speedup against N for varying processes (i7-7700K configuration)

6 Discussion: MPI Implementation

6.1 Expectations

Akin to the OpenMP implementation, we expect our MPI implementation for a given number of processes P to exhibit an increasing speedup with increasing N up to some peak value, demonstrating the benefits of parallelisation.

However, communication between MPI processes and coordination between the master and slaves will introduce significant overhead that is quantifiable in the overall execution time. Therefore, we additionally expect:

(1) The peak speedup for the MPI implementation with P processes to be lower than the equivalent OpenMP implementation with P threads, and

(2) The curve of speedup against N to grow significantly slower towards the peak speedup, as the ratio of computation to communication increases with N

Furthermore, when the number of processes P is large and the problem size N is small, we expect the communication overhead to dominate computation time, resulting in potentially worse performance than the sequential implementation.

6.2 Conclusions

Figures 12 and 14 demonstrate an interesting mix of behaviour with different regimes for different number of nodes M and problem size N.

When the number of nodes M is small ($M \leq 2$), we observe that the execution time grows approximately quadratically with the number of particles N, for both node configurations. This agrees with our expectations of the execution time being dominated by the computation required for particle-particle collision checks.

The peak speedup for small M achieved by both configurations approached the number of processes P. This indicates that for our task and data decomposition pattern, the optimal problem size N that balanced computation time and communication overhead was within our range of testcases.

However, the behaviour of the curve of speedup against N differs significantly between $M = 1$ and $M = 2$, for both configurations. When $M = 1$, the speedup begins close to the peak value at $N = 1000$ and plateaus quickly as N increases, remaining close to the peak speedup throughout. In contrast, for $M = 2$, the speedup begins close to 1 (dotted horizontal line), and then rapidly increases up to $N = 8000$, after which it levels out and approaches the peak speedup asymptotically.

This behaviour demonstrates that the cost of communicating across multiple nodes is significant, even for two nodes. When $M = 1$, all processes are confined to the same node and message-passing does not involve the network, leading to relatively little overhead. As a result, the curve of speedup resembles that of the equivalent OpenMP implementation. However, when $M = 2$, the network is now involved in communication operations between processes on different nodes, introducing additional overhead from packet routing and network congestion. Therefore, the curve of speedup begins close to 1 and rapidly increases with N, as the ratio of computation to communication changes as more work is assigned to each slave with larger N. Notably, there was a small performance regression (speedup < 1) for small N on the Intel i7-7700K configuration with $M = 2$.

The same trends are not observed with $M \geq 4$ nodes. For small N, we observed a huge performance regression (speedup $\ll 1$) relative to the sequential results, with the execution time up to $N = 6k$ being orders-of-magnitude worse than the equivalent execution time with fewer nodes.

We attribute this observation to the combination of a small problem size N with a large number of processes P. Under such conditions, the tasks assigned to each slave was too granular and required little work, skewing the ratio of execution time heavily towards communication operations instead of computation. This introduced a large amount of network contention from the many slaves attempting to communicate, resulting in much greater overhead. The performance was also further worsened by bottlenecking from the master process, which performs a fair amount of computation alongside its coordination role. This demonstrates the need to carefully select the data distribution and the number of processes P to fit a given problem size N.

Interestingly, this performance regression resolves itself dramatically moving from $N = 6k$ to $N = 8k$, with a marked reduction in execution time observed across both configurations for $N \geq 4$. The execution time continues to increase with greater N, but not in a predictable manner, since the curve begins to exhibit signs of plateauing around $N = 16k$. This crossover point at $N = 6k$ between the two regimes also marks the beginning whereby a speedup over the sequential result was observed with the MPI implementation. However, whilst the curve of speedup beyond $N = 6k$ showed a general increasing trend with N, it was also similarly unpredictable and exhibited localised deviations.

For the Xeon 4114 configuration, we suspect the large communication overhead from the many processes on many distinct nodes ($M \geq 4$) will prevent the peak speedup from approaching the 33x peak speedup observed for two nodes, even in the limit of large N.

Implementing the particle simulation with MPI boasts a clear advantage over the alternatives. Whilst the OpenMP and CUDA implementations are constrained by the available system and GPU memory respectively, the MPI implementation allows the computation to be scaled transparently across a variable number of computing nodes, allowing the simulation to be run for a much greater number of particles N in a respectable time.

However, we opine that this is not the key strength of the message-passing programming model, since the performance of a master-slave pattern will be heavily constrained by the network bandwidth. Instead, we believe that message-passing is best suited for problems with some element of spatial structure with no clear master-slave relationship. This allows efficient mapping of spatial elements in the problem to the underlying topology of the network, ensuring related elements are on nodes in close physical proximity, which will both minimise the latency of message-passing and the bandwidth utilisation from communication operations.

7 Discussion: Comparison with OpenMP and CUDA

7.1 Remarks on Previous Implementations

The extrapolated execution times for the sequential implementation in Figure 7 more clearly showcase the difference in single-core performance between the Xeon Silver 4114 and the Intel i7-7700K. Due to its higher clock frequency, each core of the i7-7700K has a computational throughput about 28% greater than each core of the Xeon Silver 4114.

Figures 10 and 11 plot the speedup of the CUDA implementation relative to the extrapolated sequential times for varying N. Both the Titan RTX and Tesla T4 only possess 2 FP64 units per streaming multiprocessor (SM) [1], for a total of 144 and 80 FP64 units respectively.

Despite each CUDA core being simpler and less powerful than a single CPU core, we observed a peak speedup (relative to the i7-7700K) of 210 and 130 respectively on the two GPUs, greater than the number of FP64 units. We opine this demonstrates the high **latency-hiding** capability of GPUs, which are able to exploit their large register file on each SM to switch between occupant threads at a very low cost to minimise stalling.

7.2 Comparison with OpenMP

For a fair comparison between OpenMP and MPI, we compare the speedup for the same number of OpenMP threads and MPI processes with varying N. The results for both parallel programming models are plotted together in Figures 16 (Intel i7-7700K) and 17 (Xeon Silver 4114).
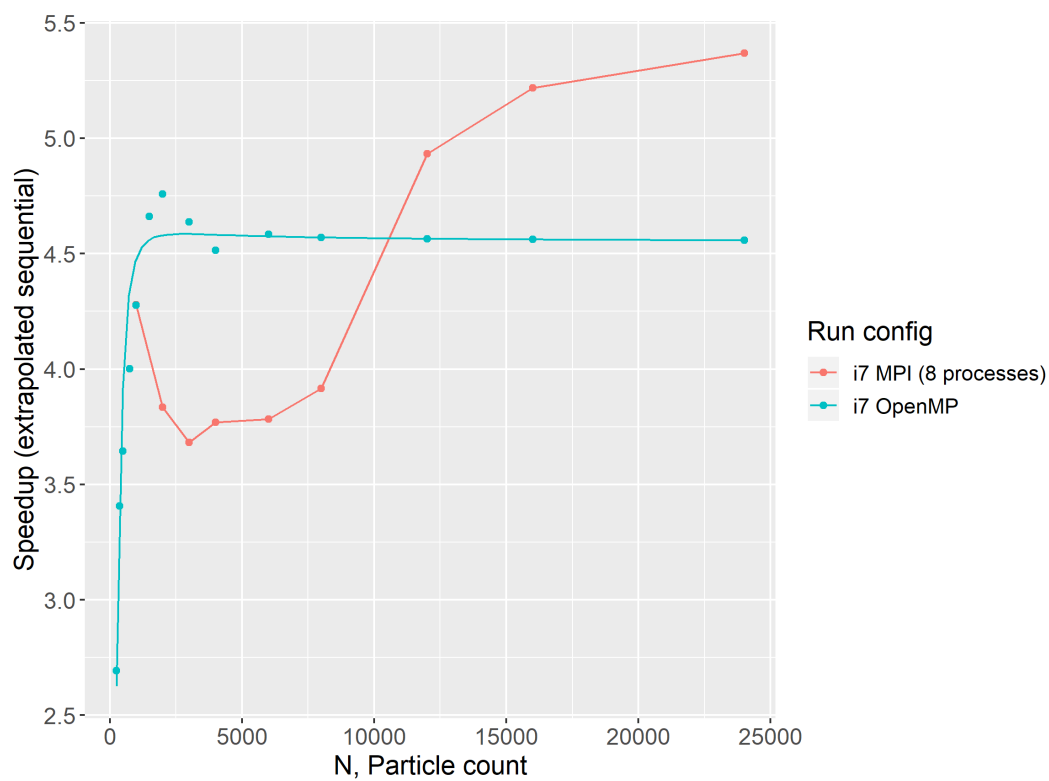
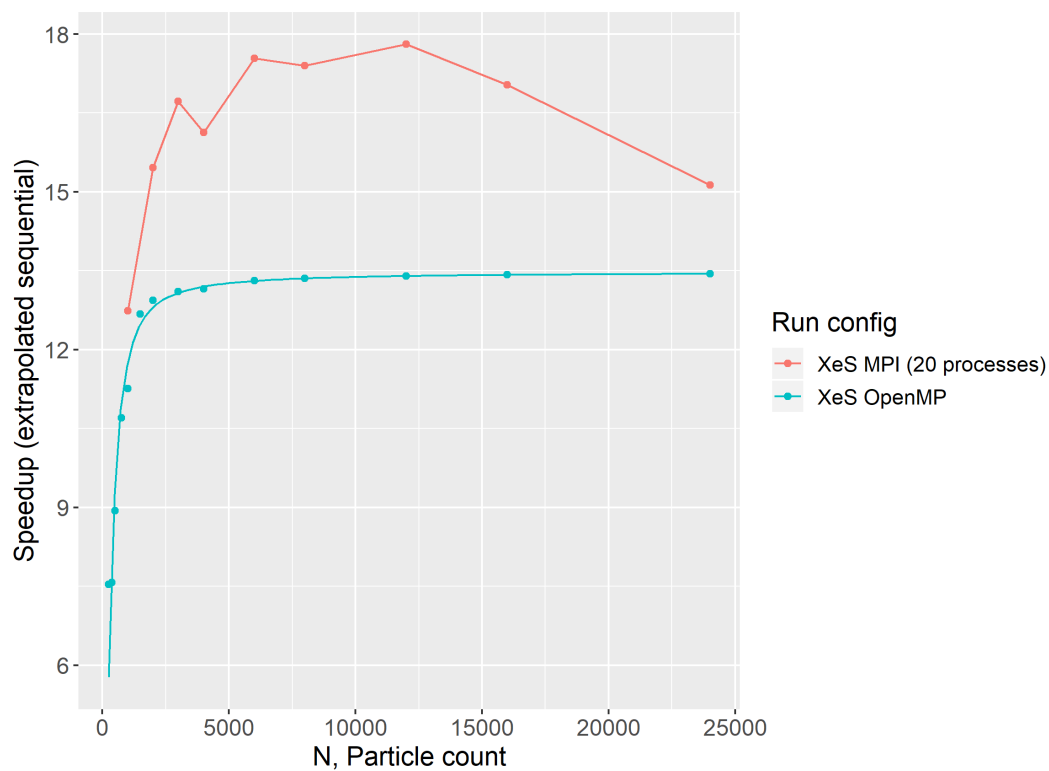Figure 16: Plot of speedup against N for 8 OpenMP threads/MPI processes (Intel i7-7700K)

Figure 17: Plot of speedup against N for 20 OpenMP threads/MPI processes (Xeon Silver 4114)

Several observations can be drawn from these two graphs. For the same number of threads/processes, the peak speedup achieved by MPI exceeds that of OpenMP for both machines. However, the speedup for MPI only approaches this maximum value at moderately large problem sizes N, in contrast to OpenMP where the speedup attains this peak value early and remains remains close thereafter.

We attribute these observations to the differing nature of the programming models:

(1) Initial gradient of speedup curve

- For OpenMP, the speedup rises quickly towards the peak value at small N due to the benefits of parallelisation
- For MPI, the small problem size N leads to tasks that are too finely divided, resulting in the communication overhead dominating the computation time, leading to the curve of speedup growing more slowly towards the peak value

(2) Peak speedup attainable

- For OpenMP, the peak speedup attained is lower due to contention between multiple OpenMP threads for the single critical section to add a collision candidate
- For MPI, there are no shared resources between MPI processes except for the network; however, since all MPI processes are confined to the same node, communication operations do not incur additional overhead from the network and hence the peak speedup achieved is greater

(3) Sensitivity of speedup to problem size N

- For OpenMP, we see the speedup generally remains close to the peak value after attaining it, since the density of particle-particle collisions dominates with large N
- For MPI, we see the speedup varies noticeably with problem size N, demonstrating the sensitivity of the execution time to the task and data distribution pattern

Furthermore, we observe that the speedup of the MPI implementation exhibited a trough between $N = 1k$ to $N = 8k$ - we suspect this occurred due to the greater single-core performance of the i7-7700K processor, which skews the ratio of computation to communication more heavily towards the latter.

7.3 Comparison with CUDA

Since CPU and GPU computing are wholly different computing platforms, it is difficult to draw a fair comparison between the CUDA and MPI implementations.

However, the empirical benchmarks demonstrate that the highly parallel nature of GPUs provide the CUDA implementation a significant advantage in performance, due to the GPU's high latency-hiding capability. Furthermore, GPUs are able to exploit the high degree of data parallelism in the simulation (since the same task is repeated on many different pieces of data), allowing it to achieve **superlinear speedup** despite the heavily divergent nature of the computation.

8 Discussion: CUDA FP64 performance on the Titan V

8.1 Rationale

In the previous assignment [3], we analysed the performance of our CUDA implementation on two Nvidia GPUs, namely the Tesla T4 and the Titan RTX. In particular, we observed that when the implementation was switched to using lower-precision `float`s from `double`s, the performance improvement exhibited did not agree with our expectations from the specifications of the FP32 and FP64 FLOPS performance of the GPUs. Only a modest speedup of $2 - 3$ was observed between the two variants, instead of 32 (the ratio of the FP32 : FP64 throughput).

We decided to investigate this further with the Titan V, a datacentre-class GPU that boasts the full 32 hardware FP64 units per SM. This gives the Titan V the maximum available FP64 instruction throughput, at half that of the FP32 throughput. The testcases ran on the Titan V did not change from the previous assignment.
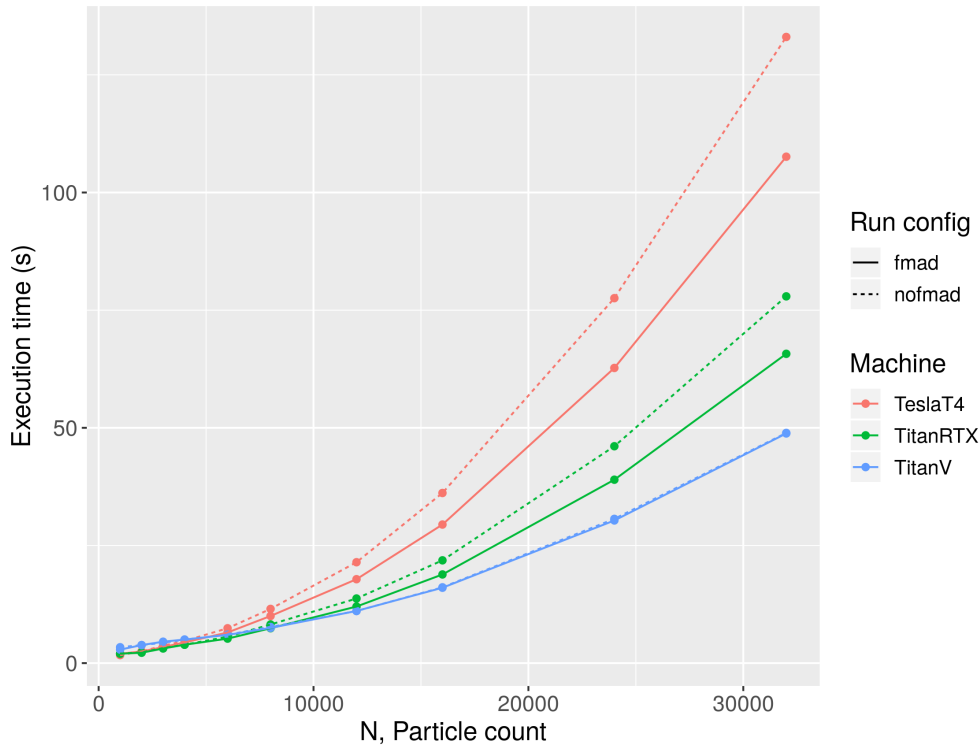
8.2 Titan V Results

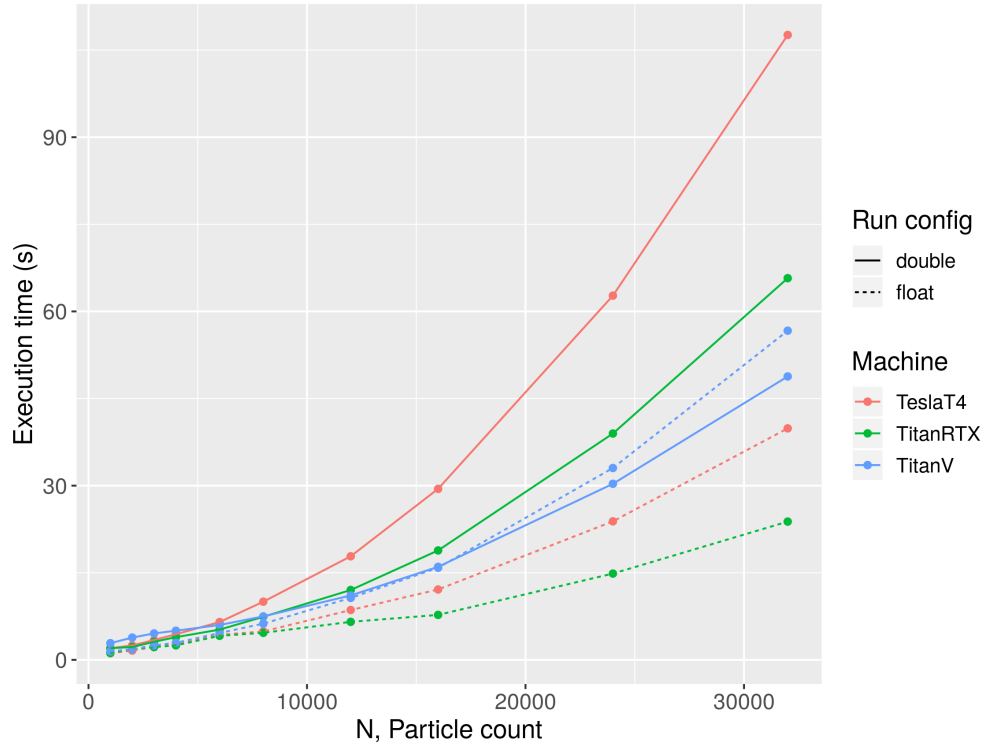Figure 18: Plot of execution time against particle count, N

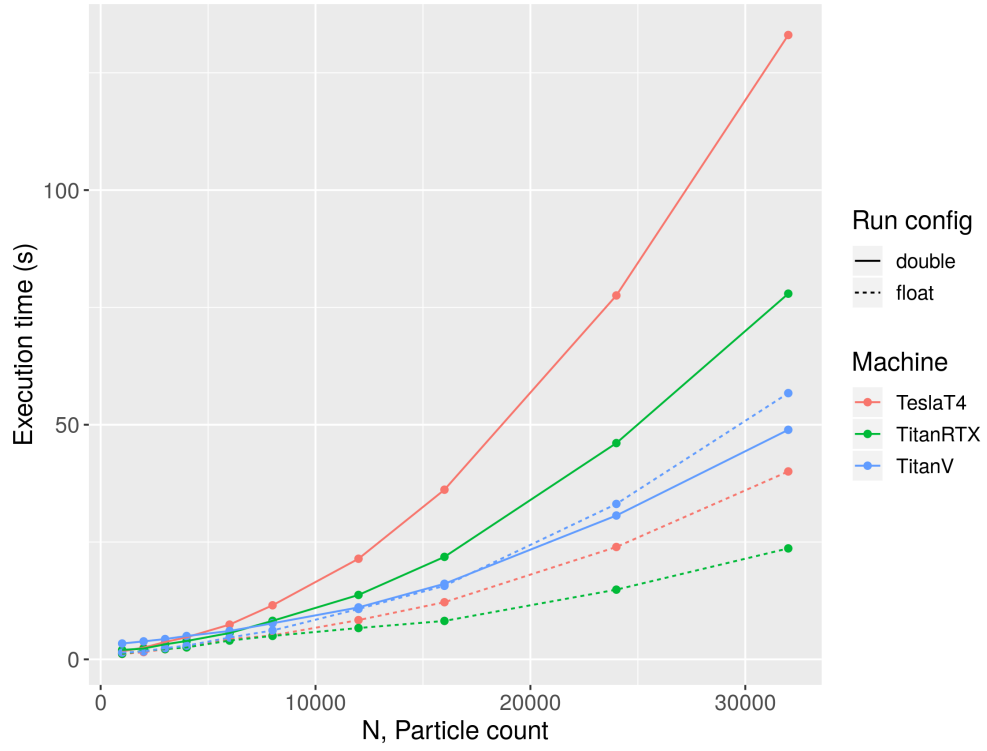Figure 19: Plot of execution time (s) against N for `float` and `double` programs (`--fmad=true`)

Figure 20: Plot of execution time (s) against N for `float` and `double` programs (`--fmad=false`)

Figure 18 shows that the the presence or absence of the FMA optimisation had little to no effect on the execution time of our CUDA implementation on the Titan V, in contrast to the other two GPUs. We suspect that this occurs due to more severe bottlenecking from the Titan V's memory subsystem, due to its slightly lower memory bandwidth (651.3 GB/s vs. 672.0 GB/s for the Titan RTX) making it more difficult to feed the much larger number of FP64 units with data. As a result, both variants run with a similar execution time, and the effect of the FMA optimisation becomes indiscernible.

In contrast, on the other two consumer-grade GPUs, the small number of FP64 units meant that the effect of the FMA optimisation became significantly more pronounced. Since the FMA optimisation reduces the average number of floating-point instructions required for computation, this increases the rate the FP64 units are fed with data, resulting in a small but significant speedup over the non-FMA variant.

Figures 19 and 20 shows two interesting observations. First, the performance of the `float` variant scaled worse on the Titan V with increasing N, relative to the other two GPUs. Second, we see that when $N \geq 24k$, the `double` variant began to perform better than the lower-precision `float` variant on the Titan V, in deviation to the trend exhibited by the other two GPUs.

We attribute both of these observations to a different set of hardware and firmware optimisations on the Titan V GPU, which should favour double-precision FP64 computation over single-precision FP32 computation.

The Titan RTX and Titan V have a similar number of SMs, at 72 and 80 respectively, but the Titan V has the full number of hardware FP64 units. Between the two, we thus expect the execution time of the `double` variant to be significantly faster due to the greater number of hardware resources. However, comparing the execution time directly shows that the Titan V only exhibited a modest (relative) speedup of 1.55, failing to reach the factor of 32 that we had expected. We treat this as clear evidence of limitations imposed by the bandwidth of the memory subsystem and the highly divergent nature of discrete particle simulation.

9 Potential Optimisations

9.1 Hybrid OpenMP-MPI approach

Our conclusions in section 6.2 suggest that the MPI implementation will likely scale very poorly to a large number of MPI processes on physically distinct nodes, due to limitations of the network bandwidth. This is particularly important since such particle simulations are common in scientific research and are typically run for an extremely large number of particles ($N \gtrsim 1M$) on a large cluster of computing nodes ($M \gtrsim 1000$), such as those of a supercomputer.

Since the message-passing model does not possess any problem-specific advantages for discrete particle simulation, we opine that the computation on a node need not be carried out across separate MPI processes.

Instead, we propose that a hybrid implementation combining both OpenMP and MPI is best suited for this problem, as it allows us to take advantage of the strengths of both programming models. The modifications required for this are:

- Our program will now only create 1 MPI process for each node, for a total of P processes across P nodes; one of these nodes will be designated the master

- The master node distributes work between itself and the $P - 1$ slaves; each process then parallelises the computation of its tasks with OpenMP

- OpenMP directives are used to (1) perform parallel execution by creating a team of threads of size equal to the number of logical cores on that node, and (2) to enforce synchronisation between OpenMP threads

The benefits of such a hybrid approach are:

- Retaining MPI's ability to scale the computation across multiple physical computing nodes

- Reducing overall communication overhead, since a smaller number of MPI processes result in fewer communication operations required and thus less network contention

- Retaining the memory footprint advantage from OpenMP, as the use of shared memory avoids the need to replicate portions of the simulation state across many independent MPI processes

9.2 Optimising Communication Operations

Figure 12 demonstrates how network contention from many MPI processes participating in collective communication operations may lead to significant performance regression. To address this, we propose that reworking the message-passing between the master and slaves may improve the performance by reducing the strain on the network in transmitting data.

The changes would include:

(1) The master only transmitting the section of the particle array required by each slave for checking particle-particle collisions, instead of broadcasting the entire array of N particles

(2) The master transmitting only the additional particles required by each slave for collision resolution

(3) Replacing the collective communication operations with their non-blocking variants where safe, allowing processes to perform computation to hide the latency of message-passing

9.3 Optimising Collision-sorting

Presently, the master process is responsible for sorting all the gathered collision candidates in its own `cs` array, which we have shown in our previous report [3] to have complexity of $O(N^2 lg N)$ with quicksort, where N is the number of particles.

Therefore, it is possible to reduce the execution time by optimising this sorting of collisions. A parallel sorting algorithm (e.g. parallel mergesort) could be employed with all P processes to distribute the task of sorting amongst the slaves, reducing the time required for the sorting to be completed by the master process.

10 References

[1] E. Kilgariff, H. Moreton, N. Stam, and B. Bell. NVIDIA Turing Architecture In-Depth. `https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/`. Accessed: 2019-11-16.

[2] K. Loo and R. Lee. CS3210 Assignment 1 - Particle Movement Simulator (Part 1), 2019.

[3] K. Loo and R. Lee. CS3210 Assignment 1 (Part 2), Discrete Particle Simulation with CUDA, 2019.