CS3210 Parallel Computing
Assignment 2 Report

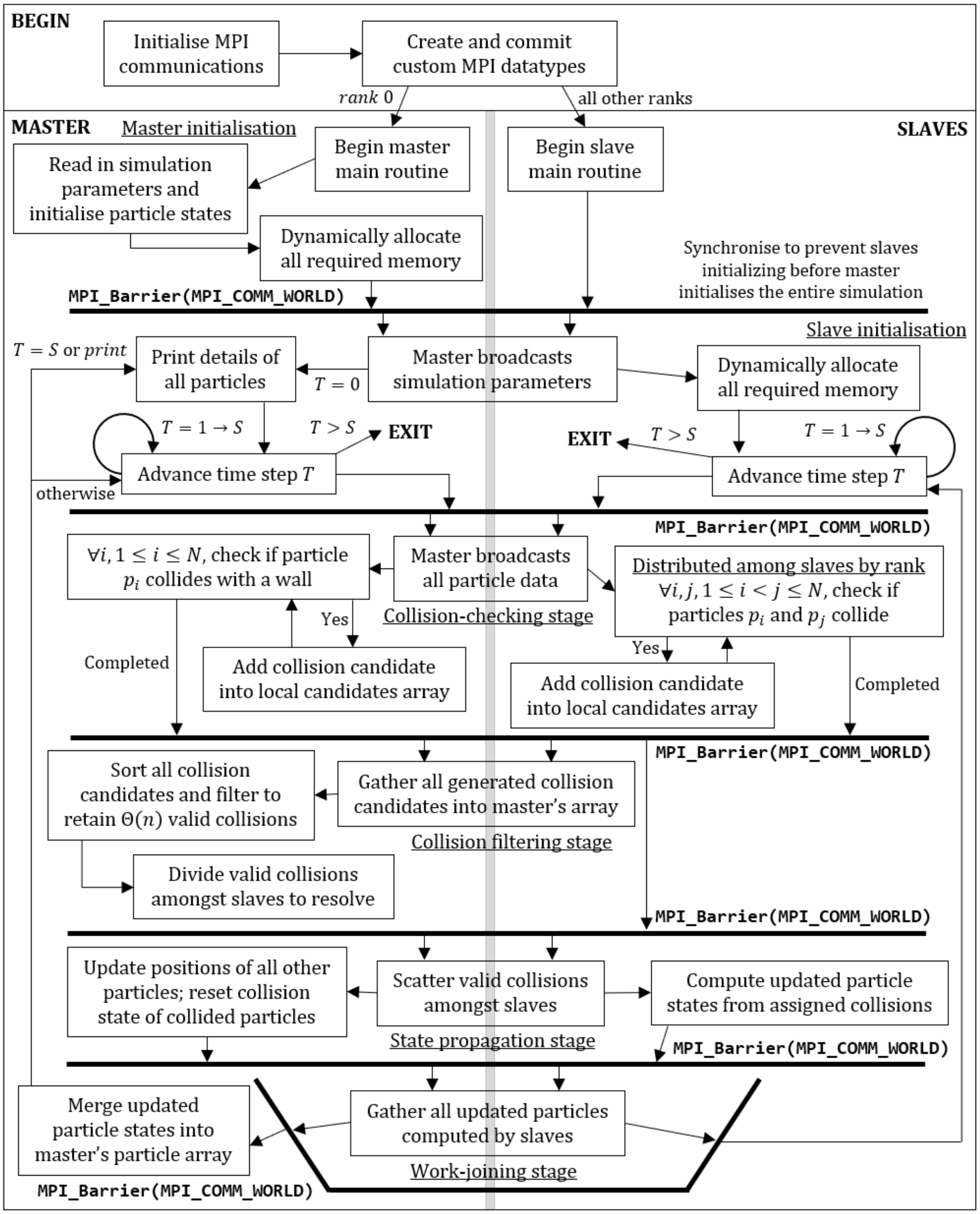
Discrete Particle Simulation with MPI

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

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.



BEGIN
Initialise MPI communications
Create and commit custom MPI datatypes
rank 0
all other ranks
MASTER
Master initialisation
SLAVES
Begin master main routine
Begin slave main routine
Read in simulation parameters and initialise particle states
Dynamically allocate all required memory
Synchronise to prevent slaves initializing before master initialises the entire simulation
MPI_Barrier(MPI_COMM_WORLD)
Slave initialisation
T = S or print
Print details of all particles
Master broadcasts simulation parameters
T = 0
Dynamically allocate all required memory
T = 1 → S
T > S
EXIT
Advance time step T
otherwise
∀i, 1 ≤ i ≤ N, check if particle p_i collides with a wall
Master broadcasts all particle data
Distributed among slaves by rank ∀i, j, 1 ≤ i < j ≤ N, check if particles p_i and p_j collide
Yes
Collision-checking stage
Completed
Add collision candidate into local candidates array
Sort all collision candidates and filter to retain Θ(n) valid collisions
Gather all generated collision candidates into master's array
Collision filtering stage
Divide valid collisions amongst slaves to resolve
Update positions of all other particles; reset collision state of collided particles
Scatter valid collisions amongst slaves
Compute updated particle states from assigned collisions
State propagation stage
Merge updated particle states into master's particle array
Gather all updated particles computed by slaves
Work-joining stage

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 grid arrangement 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: $\text{ID}, x, y, v_x, v_y, w_c, p_c$.
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, `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 $\epsilon =$ `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 $A(\Delta t)^2 + B\Delta t + C = 0$,

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

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.

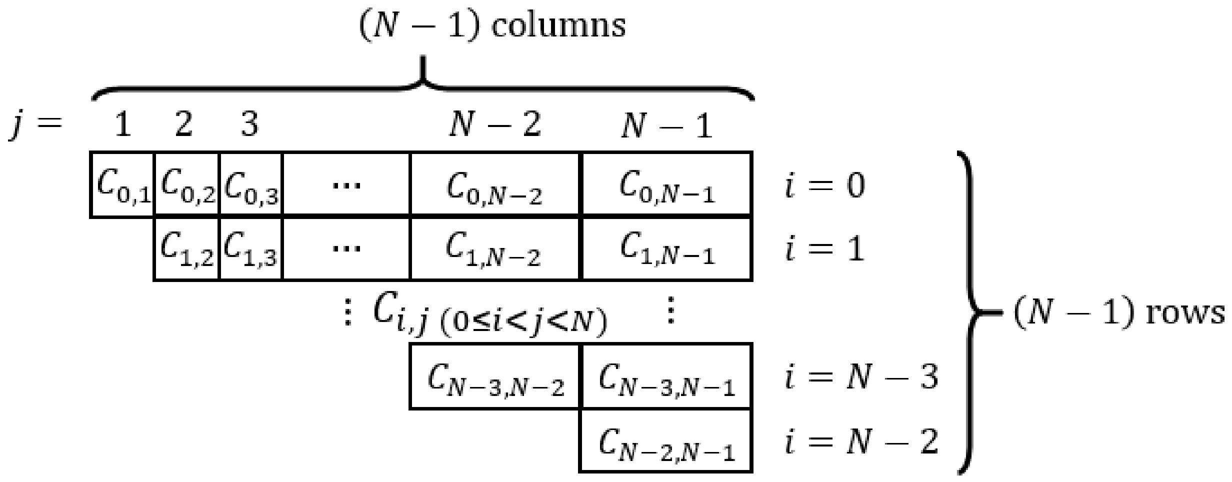

(N − 1) columns
j = 1 2 3 N − 2 N − 1
C0,1 C0,2 C0,3 ··· C0,N−2 C0,N−1 i = 0
C1,2 C1,3 ··· C1,N−2 C1,N−1 i = 1
Ci,j (0≤i<j<N)
CN−3,N−2 CN−3,N−1 i = N − 3
CN−2,N−1 i = N − 2
(N − 1) rows

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.

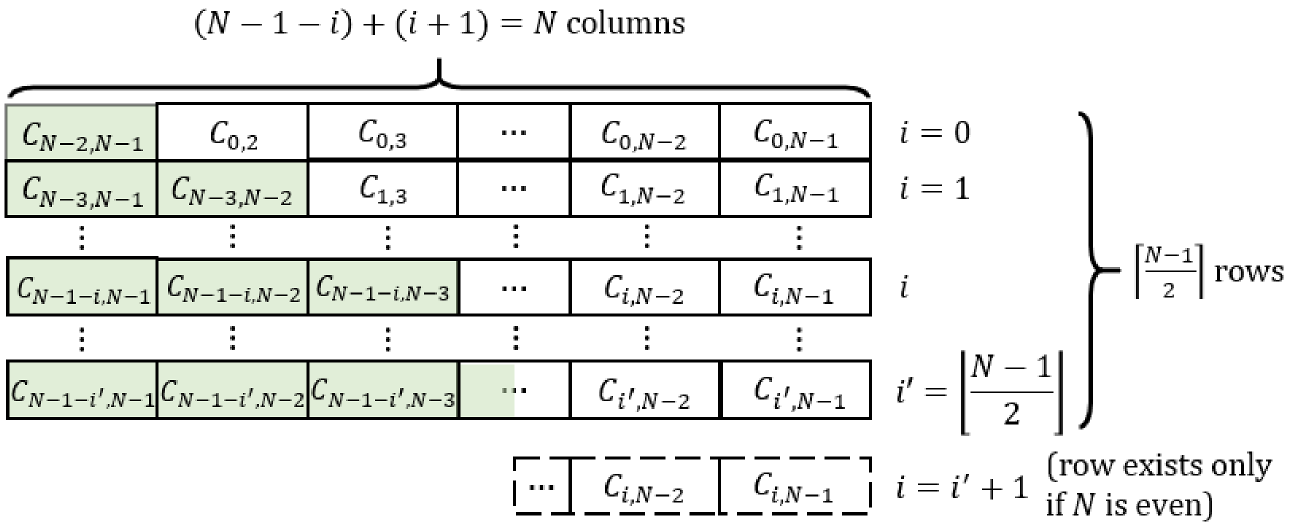
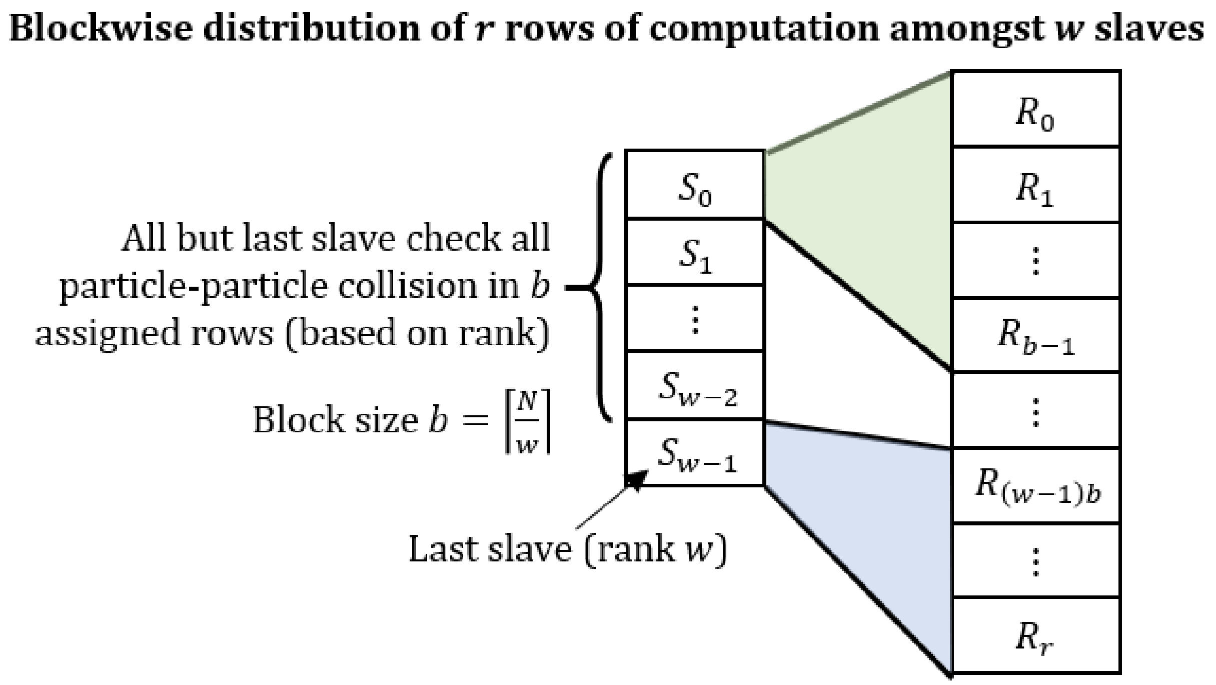


Figure 3: Folded matrix of particle-particle computations

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.

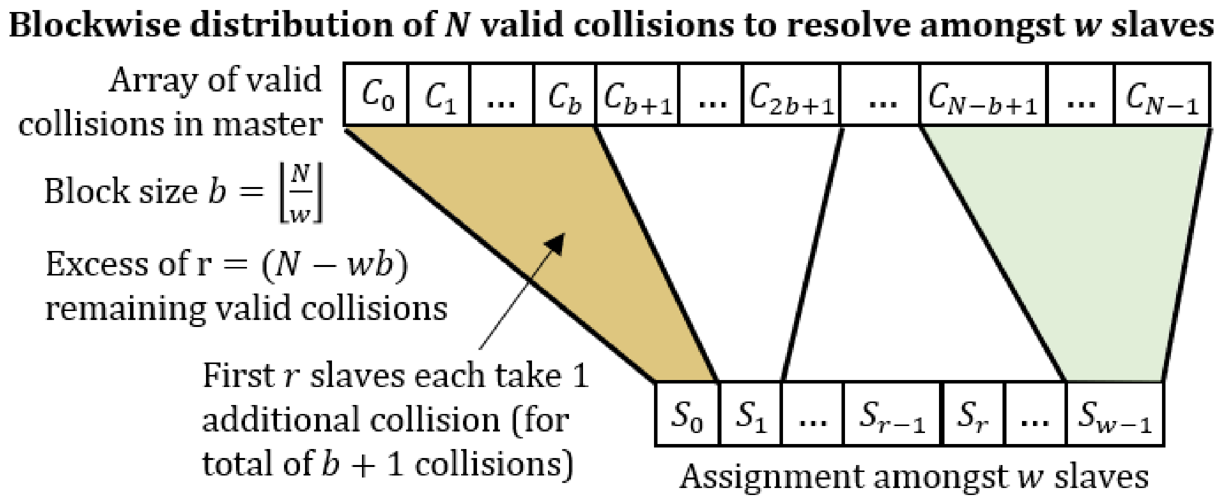
Then, we performed **blockwise assignment** of the rows of computation to each of the slave processes by their rank, as shown in Figure 4.



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 $b = \left\lceil \frac{N}{w} \right\rceil$
Last slave (rank w)
S_0
S_1
S_{w-2}
S_{w-1}
R_0
R_1
R_{b-1}
$R_{(w-1)b}$
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.



Blockwise distribution of N valid collisions to resolve amongst w slaves
Array of valid collisions in master
C_0 C_1 … C_b C_{b+1} … C_{2b+1} … C_{N-b+1} … C_{N-1}
Block size $b = \left\lfloor \frac{N}{w} \right\rfloor$
Excess of $r = (N - wb)$ remaining valid collisions
First r slaves each take 1 additional collision (for total of $b + 1$ collisions)
S_0 S_1 … S_{r-1} S_r … S_{w-1}
Assignment amongst w slaves

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.

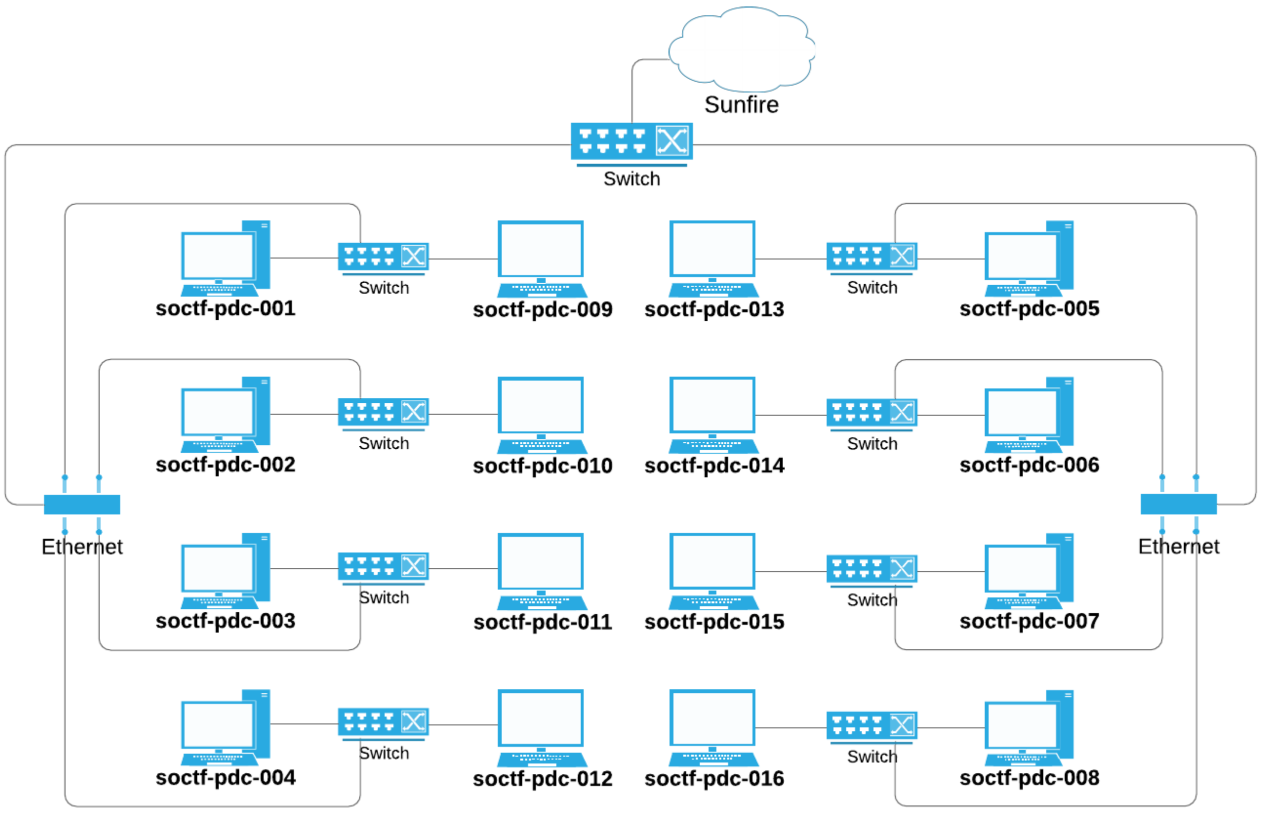


Figure 6: Assumed network layout of computing nodes in COM1-B102

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, test-case 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. [1]

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

Sequential: runtime against N (extrapolated)

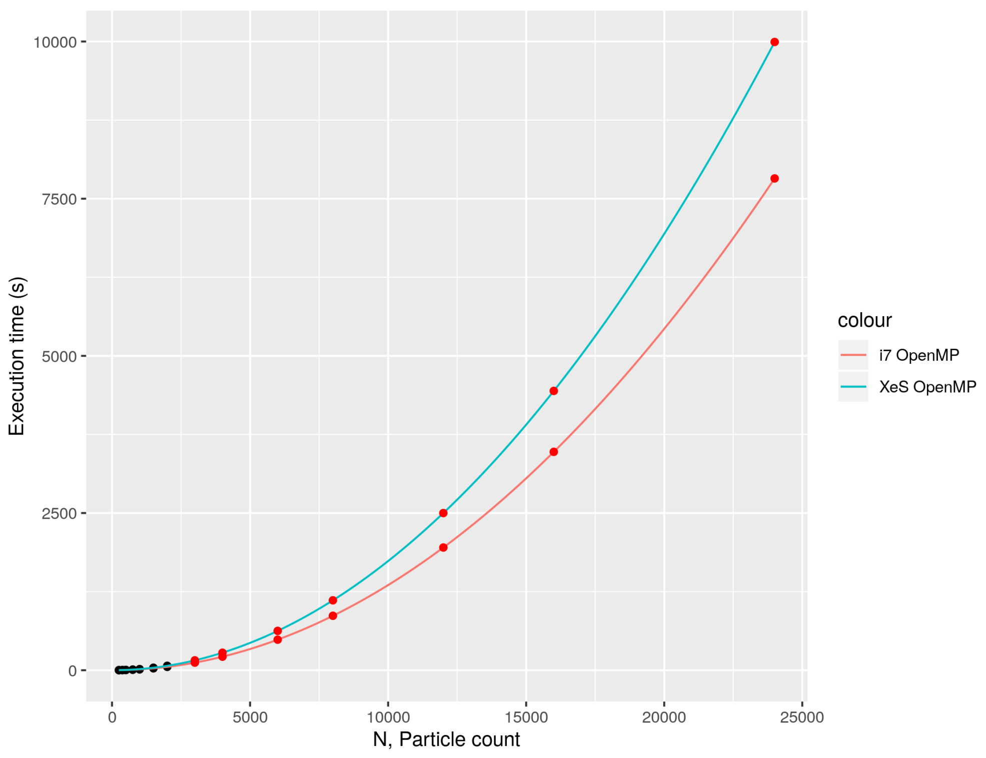


Figure 7: Plot of runtime (s) against N (Sequential implementation)

OpenMP: runtime against N (extrapolated) - both Xeon and i7

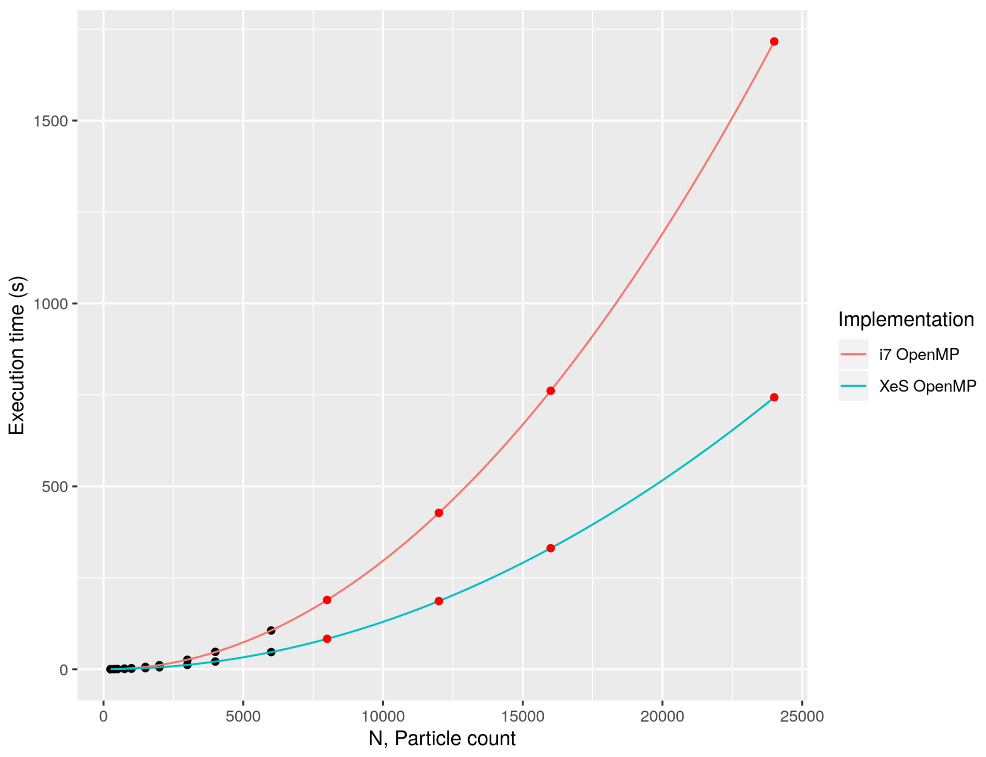


Figure 8: Plot of runtime (s) against N (OpenMP implementation)

OpenMP: speedup against N (extrapolated to 24k) - both machines

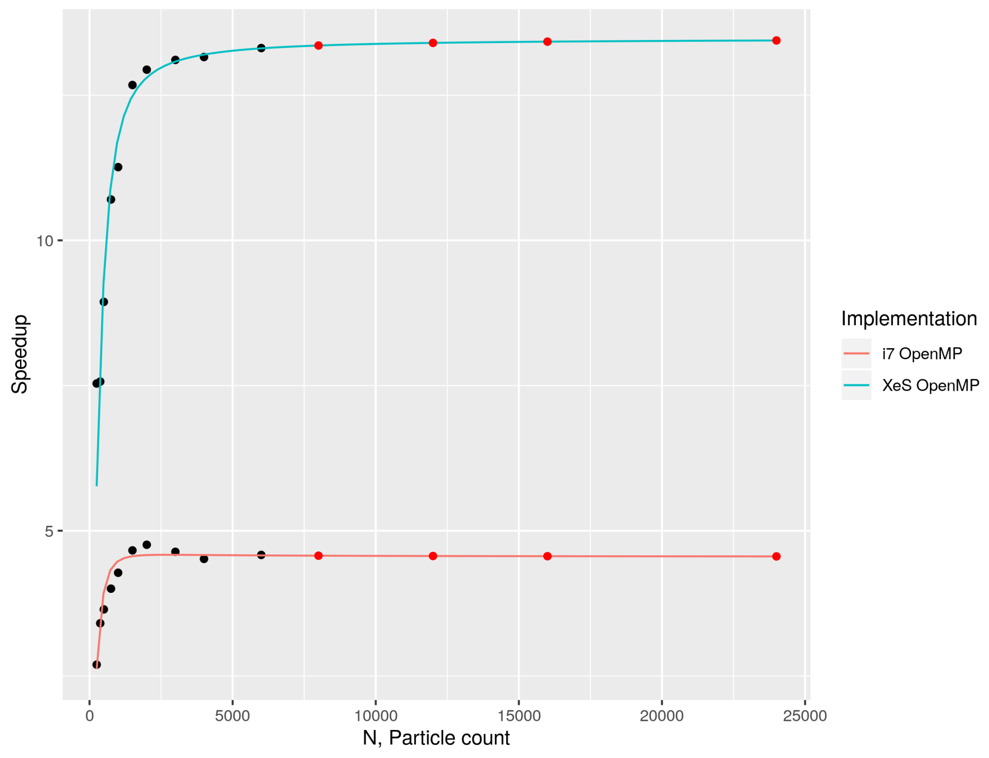


Figure 9: Plot of speedup against N (OpenMP implementation)

5.2 Updated CUDA Results

CUDA: speedup against N - both Titan RTX and Tesla T4, fmad and nofmad (speedup measured against extrapolated sequential)

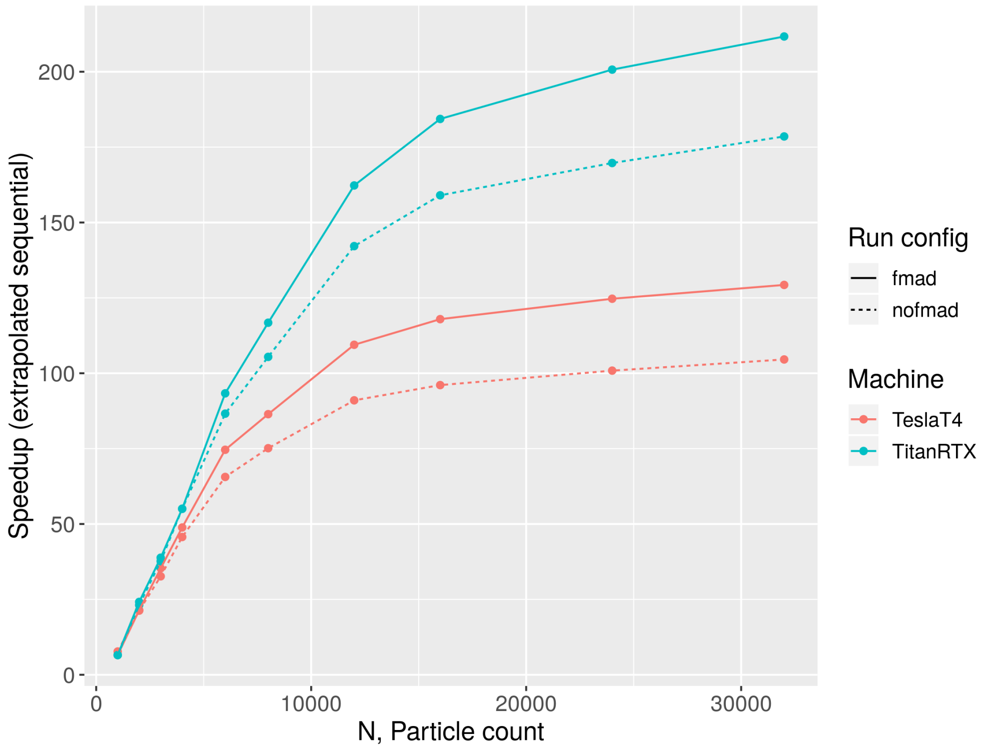


Figure 10: Plot of speedup against N (i7 standard)
)

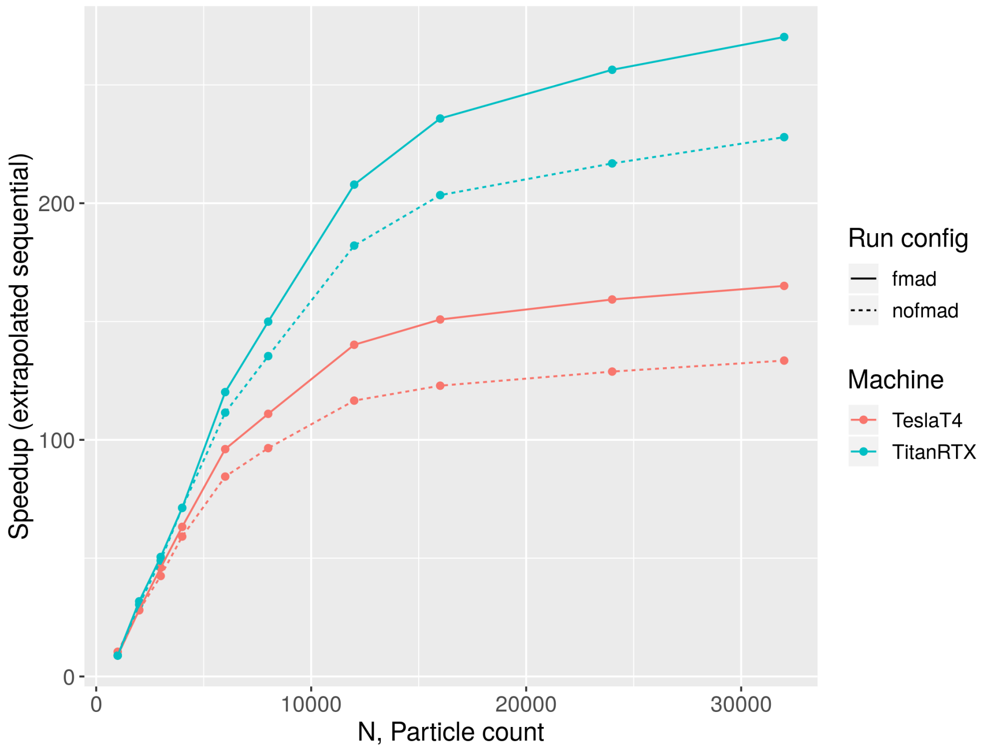


Figure 11: Plot of speedup against N (Xeon standard)
)

5.3 MPI Results (Xeon 4114 configuration)

MPI: runtime against N (to 24k) - both homogeneous setups (x2 graphs) MPI: speedup against N (to 24k) - both homogeneous setups; speedup measured against extrapolated sequential (x2 graphs)

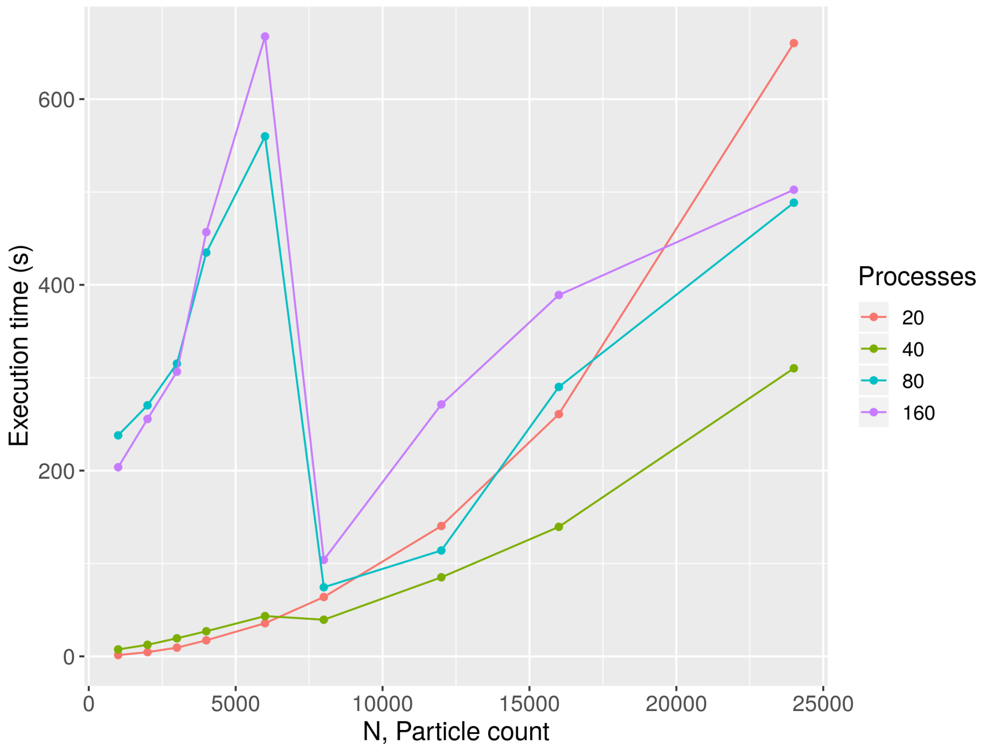


Figure 12: Plot of runtime (s) against N for varying processes (Xeon 4114 configuraion)

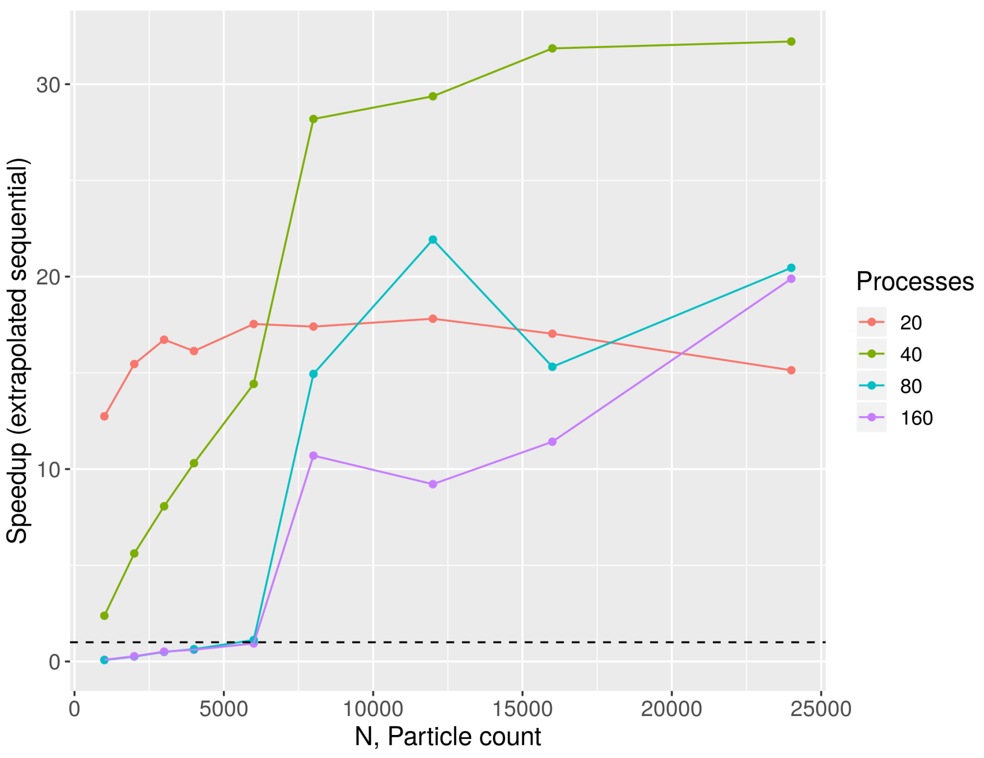


Figure 13: Plot of speedup (extrapolated sequential timings) against N for varying processes (Xeon 4114 configuraion)

5.4 MPI Results (i7-7700K configuration)

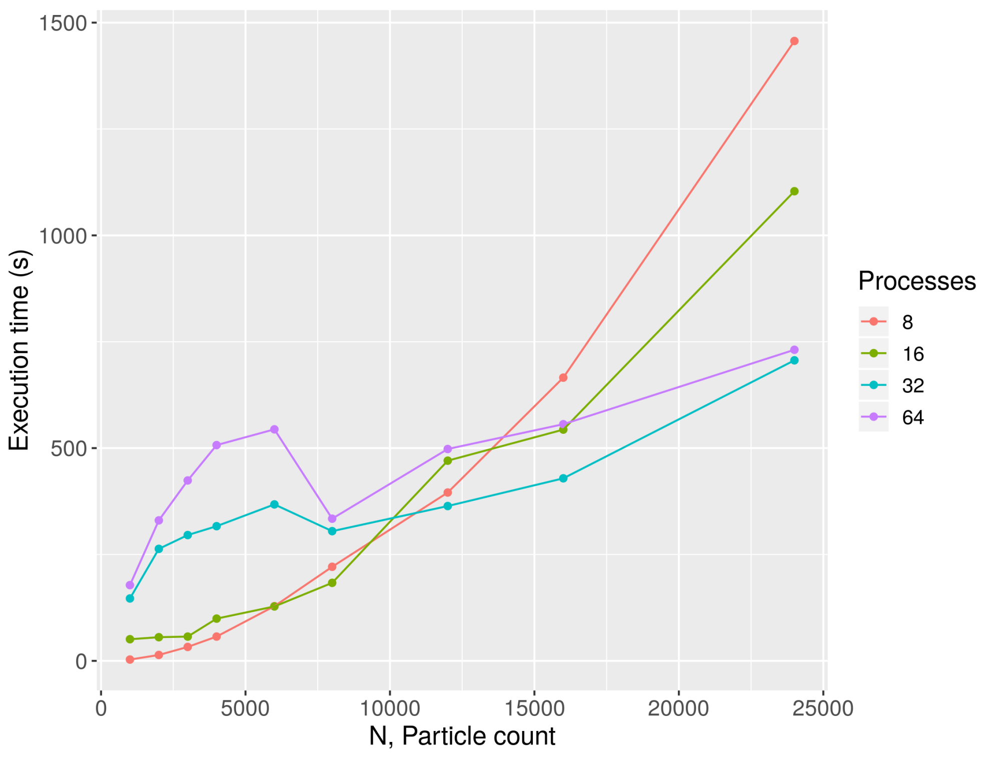


Figure 14: Plot of runtime (s) against N for varying processes (i7-7700K configuraion)

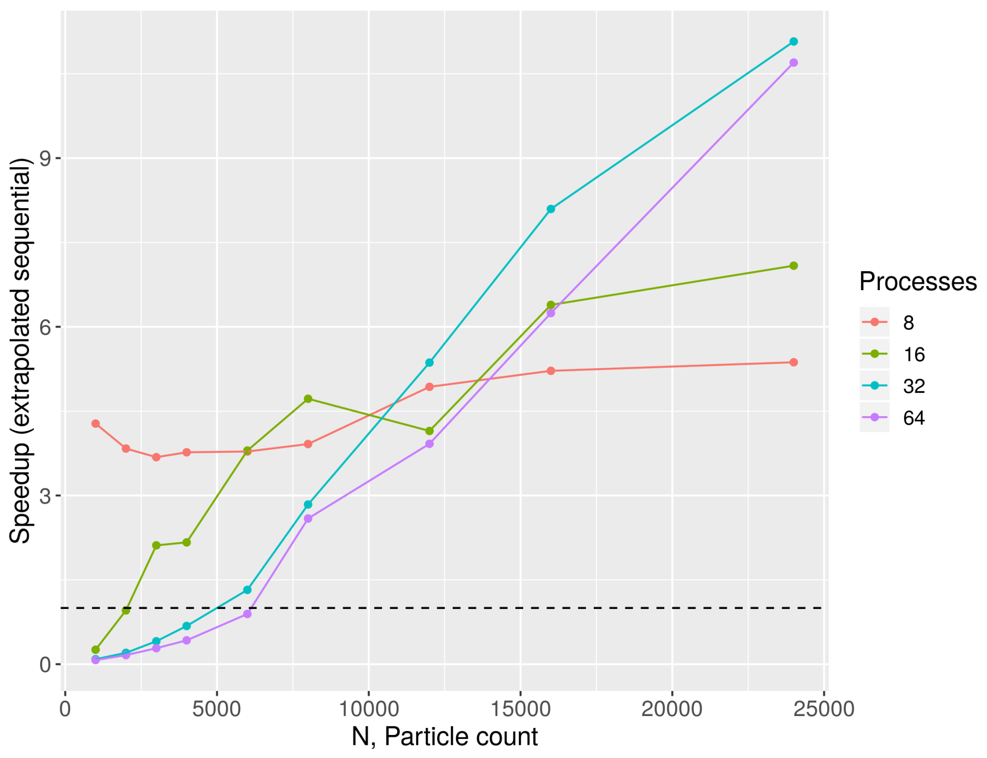


Figure 15: Plot of speedup (extrapolated sequential timings) against N for varying processes (i7-7700K configuraion)

6 Discussion: MPI Implementation

7 Discussion: Comparison with OpenMP and CUDA

8 Discussion: CUDA FP64 performance on the Titan V

8.1 Rationale

In the previous assignment [2], 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 `floats` from `doubles`, 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 streaming multiprocessor (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

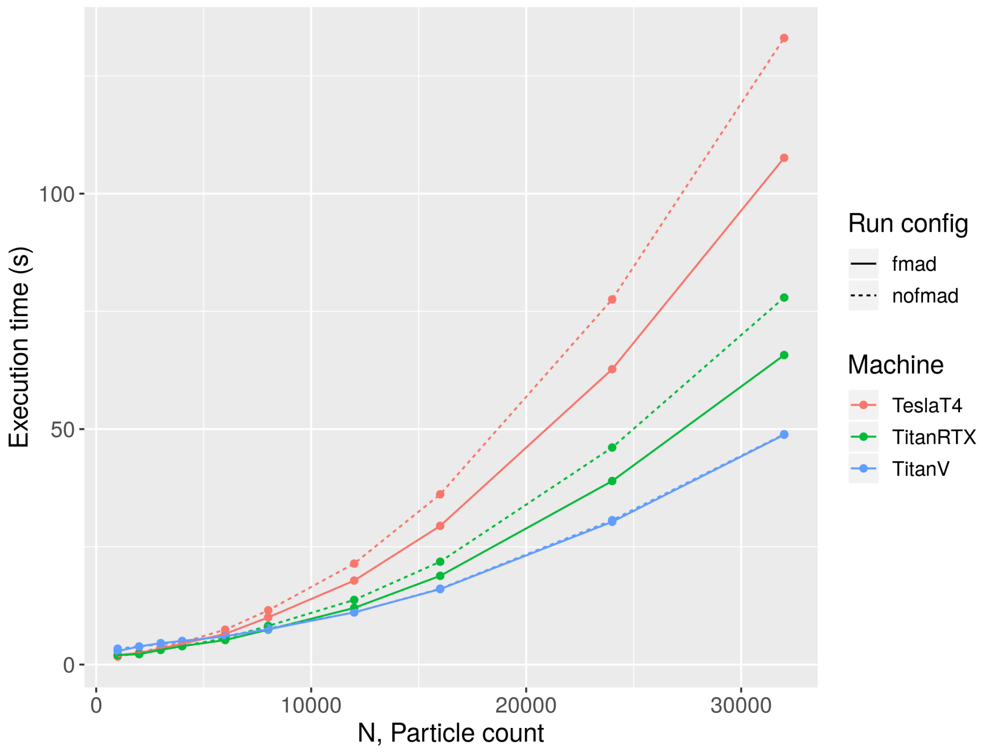


Figure 16: Plot of execution time against particle count, N

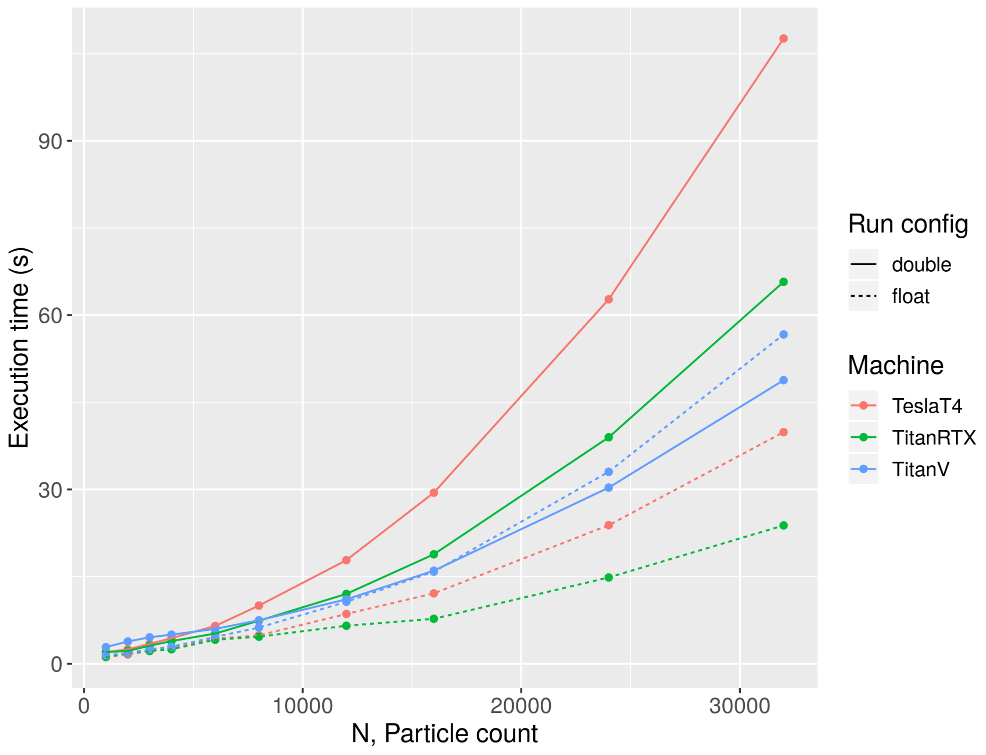


Figure 17: Plot of runtime (s) against N for `float` and `double` programs (`--fmad=true`)

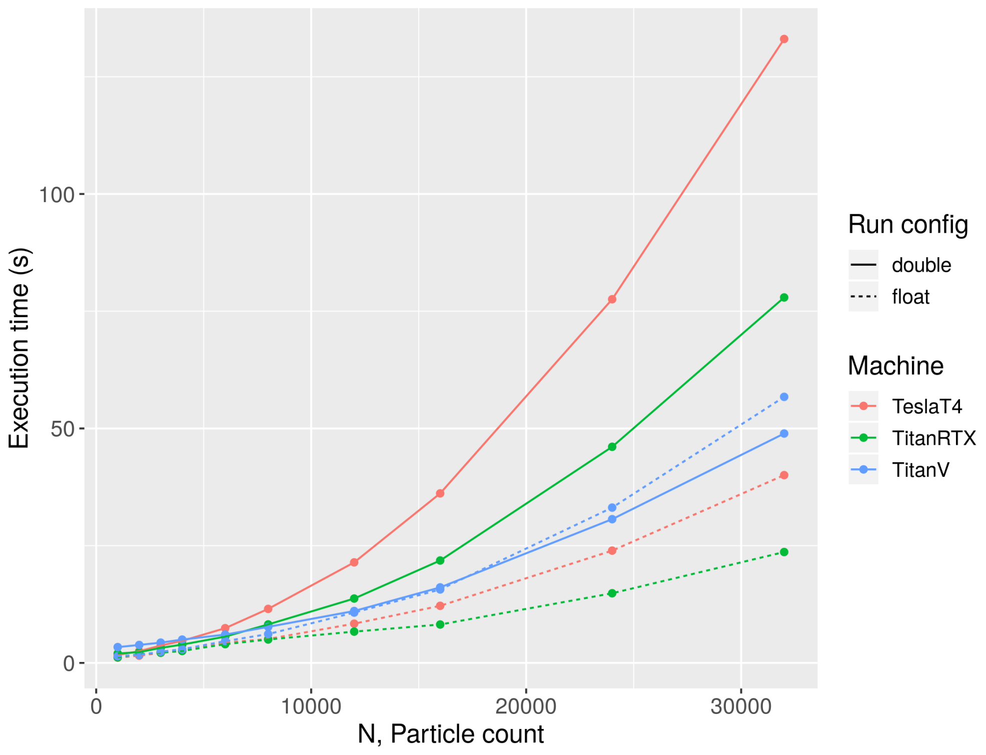


Figure 18: Plot of runtime (s) against N for `float` and `double` programs (`--fmad=false`)

Figure 16 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 17 and 18 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

10 References

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

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