Description of Implementation

Algorithm:

My algorithm has four main stages.

The first step is updating positions. As each particle only requires its own information, I parallelised this using Openmp parallel for.

The next step is assigning particles to their cells in grid. I chose each cell to be a square with length just larger than 2*radius of particles. This helps to reduce redundant checks when calculating overlaps. I stored my grid as an array of vectors where the vector holds the particles in the grid. Since, each particle could belong anywhere in the grid, access to grid had to be protected. So I used openmp for loop with openmp locks to ensure each element of grid is only modified by one thread at a time.

The next step is detecting overlapping particles. I did this because it helps to reduce the amount of searches when finding potential collisions between particles. The overlaps are stored as a list of pairs. The overlaps are detected by locating at only the particles in the adjacent cells. This was also parallelized using openmp for loops.The direction of the overlaps in the grid is also stored to help with parallelisation in the next step.

The last step is repeatedly resolving collisions among the overlapping pair till there is no colliding pair. This step also uses openmp for loop. The parallelisation of this step is explained in more detail in later sections.

Work sharing:

I have not used any explicit work sharing constructs like openmp sections in my algorithm. Since, the only parallel construct i used is the openmp for loop with static scheduling, my work sharinng depends implicitly on my inputs like overlap list and grid.

Synchronisation:

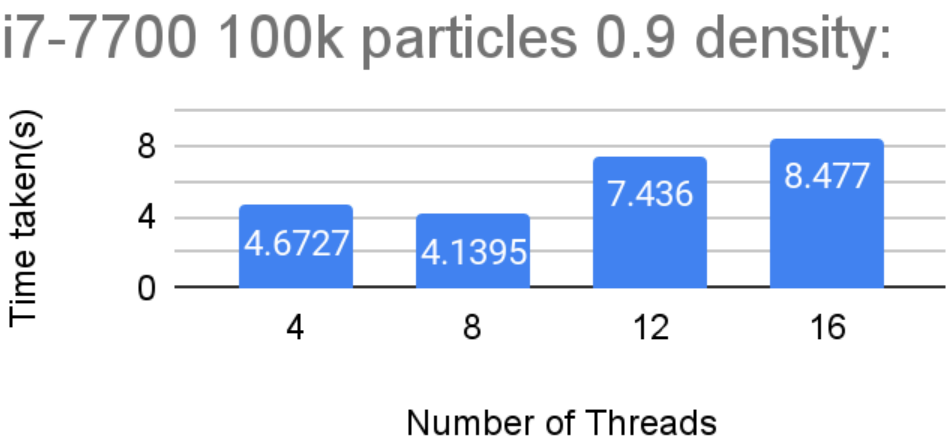
The only step that requires synchronisation is collision resolving step as it is a sequential process. The other steps results are independent of order of operations.

To synchronise collision resolving, I fixed the order in which the collisions are considered which then allows for some parallelism. For example, if the grid is made of cells ABC. where each cell has some particles.

DEF
GHI

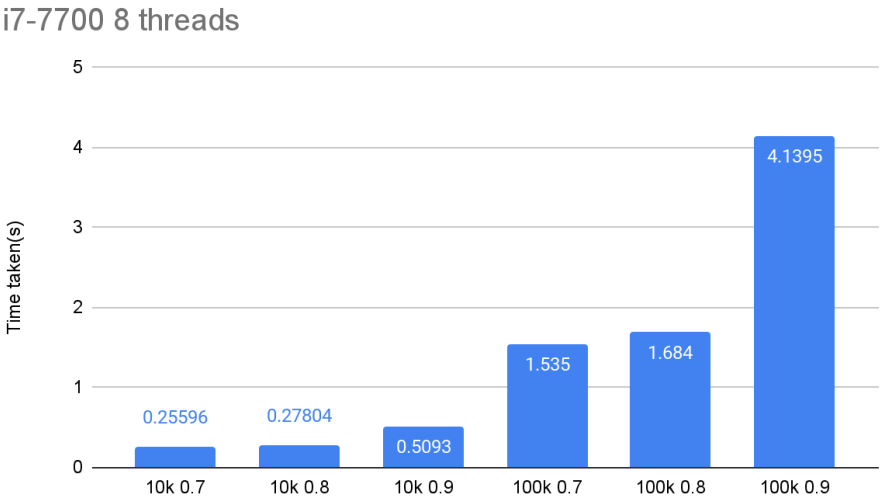
I first considered collisions from particles in A->B, D-> E, G->H. This can be parallelised, with A,D,G running on different threads. Then, I considered collisions from particles in B->C, E->F, H->I. Here B,E, H can run independently on different threads. This two step covers all collisions happening in the positive x direction. This also ensures there is no race condition as only one thread modifies a given particle's velocity. This is repeated for the other 7 directions and one more for collisions happening within a cell.

Thread performance:

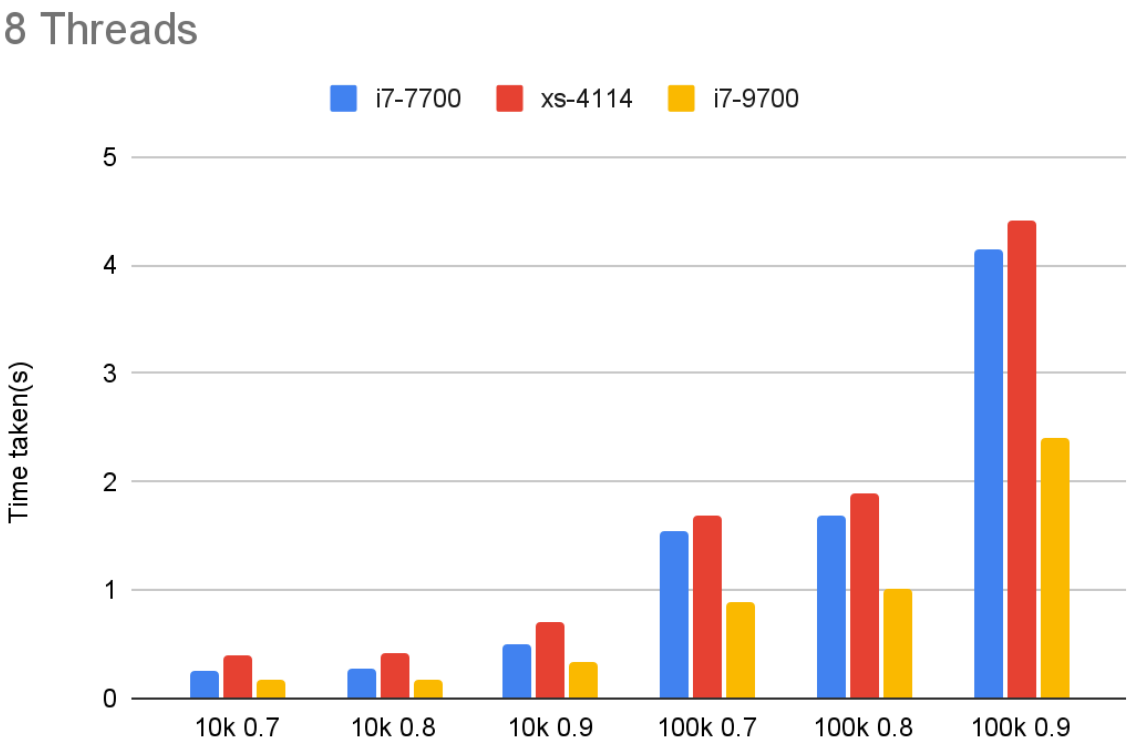


In general, time taken decreases until around 8 threads and then starts increasing. This was also true for other inputs that I tested. Despite exposing all the parallelism, that I could find, it seems that for more than 8 threads, the delay due to overhead of the threads is more than the parallelism offered. So, I decided to limit omp parallel for loops to 8 threads even when more threads were available.

Description and Visualisation



In general, as number of particles and density increases, the time taken increases. This makes sense as there would be more overlaps and collisions between particles. The relationship also does not appear to be linear. When density or number of particle increases, the number of overlaps/collision would increase more than linearly which could large increase in time.

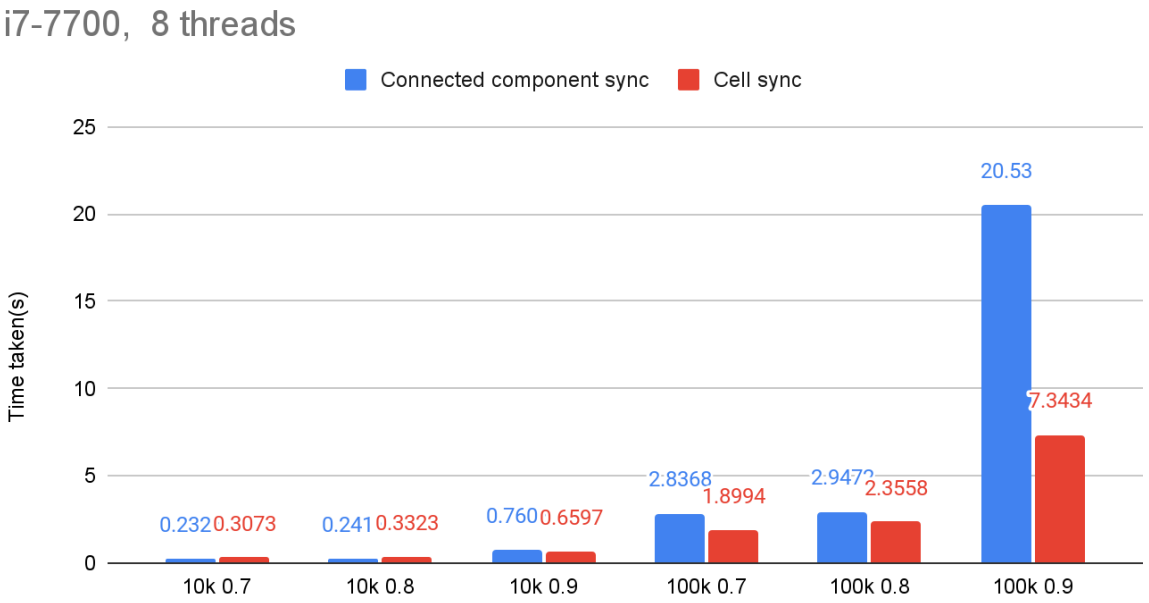


The execution takes roughly similar time on both the xs-4114 and i7-7700 processor. However, the execution on the i7-9700 processor seems to take around half the time as the other processors.

Optimisation: Connected component synchronisation

I tried an alternative synchronisation for the parallel execution of collision resolving step. Instead of parallelising based on the cells in the grid, I tried parallelising it based on components where in each component particles are connected by overlap. So, there is no possibility of collision between two particles from two different components. So, each component can run it's collision resolving step on its own thread safely without data races. Within each component, collision resolution is done sequentially.

To find out the connected components, I first found out all the overlaps as before. After that, I ran a breadth first search to find connected components treating each overlap as an edge. I stored overlaps as an array of vectors. Each element in the array is a different component and each element stored all the overlapping pairs in that component.



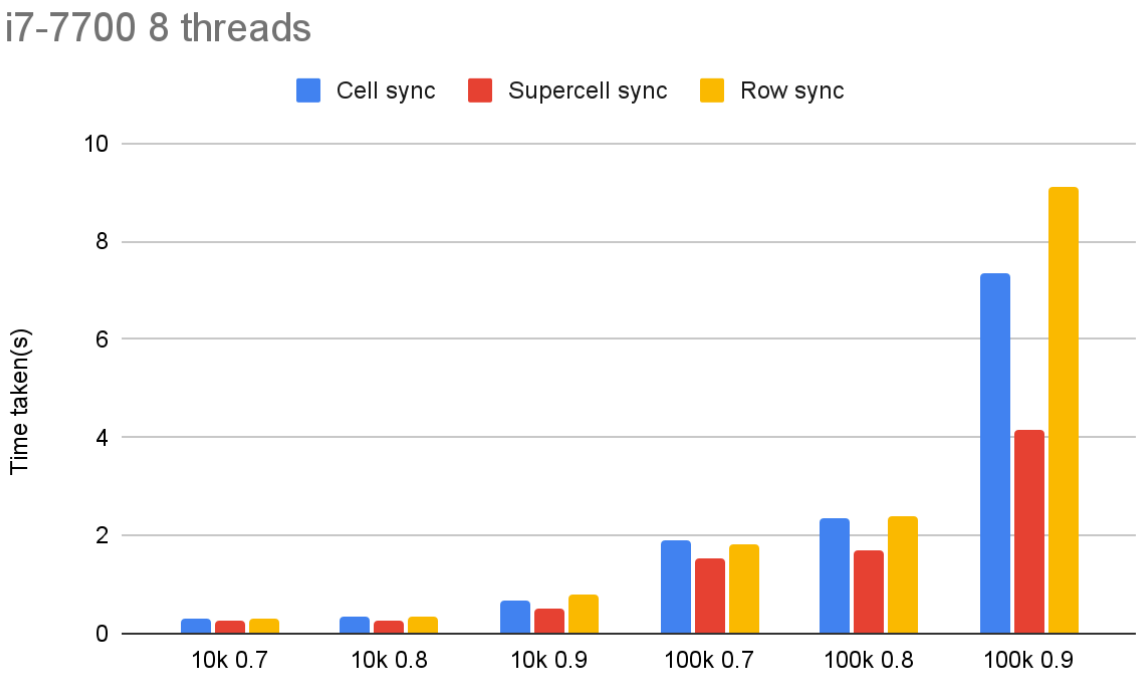
However, it did poorly compared to cell synchronisation when number of particles increased and especially when density increased. As the density increased, the probability of big component sizes increases drastically. When I watched the simulation video for the 100k 0.9 input, all the particles were moving to the right and when they bounced of the wall, it created a huge connected component. When I checked the component sizes through the program, the biggest component size was almost 80% of all the overlaps. Hence, this resulted in almost sequential like collision resolution which explains the poor performance compared to cell synchronisation which doesn't suffer from this problem. So in the end, I didn't use this.

Optimisation: Cell vs Supercell vs Row synchronisation.

When I profiled the solution with cell synchronisation, I obtained these results.

Cell size vs time taken	CELL Size multiple of radius			
.	2r	5r	10r	20r
Time for Detect overlap	0.054	0.048	0.128	0.457
Time for Collision resolution	0.939	0.52	0.372	0.347

So, detect overlap step seems to prefer smaller cell size, while collision resolution step prefers larger cell sizes in the grid. So,I decided to use different cell sizes for each step. When detecting overlaps, for each cell, I would do the same as before and check for overlaps in all adjacent cells with cell size of 2 time radius. However, instead of storing these overlaps in the cell itself, I would store it in a “bigger object” that contains this cell. I tried two different “bigger objects”. First is a supercell which is adjacent cells grouped to form a bigger cell and the second one I tried is a group of rows so all the cells would store overlap information in their row group they belonged to. . Then, I would run the parallelised algorithm on these bigger objects as described in the synchronisation section in description.



The supercell synchronisation is likely faster than cell synchronisation. This might be because in supercell synchronisation the supercells are bigger than the cells in cell synchronisation, so the ratio of thread overhead to work done in collision resolution step is lower. So, it is faster.
However, in row synchronisation, the row groups are bigger than the supercells. Since, in collision resolution, the collision is handled sequentially within each of the bigger object, the row synchronisation is slower.
So, in the end, I used supercell synchronisation.

Appendix:

1) The different synchronisation can be found in different branches in github repository. The main branch contains supercell synchronisation, the ‘rowsync’ , ‘gridsync’ , ‘bfs’ contains the other respective synchronisations. The inputs such as threads, partition is in the charts and figures.
2) Raw data: https://docs.google.com/spreadsheets/d/1b3dJIJFFAB9WmiwxfZ8-ATwNVciFm9oDrEFaWje2eOM/edit?usp=sharing