CS3210 Assignment 1 Report A0244219H

Overviews and Key ideas of implementation

1. Spatial Partitioning where the grid sizes were made to be ceil(2.1 * radius).

This is because the overlay condition was that the center of the particles were less than or equal to 2 * radius, if I were to use 2 * radius, there were edges cases where a cell could overlay with a particle two cells away, hence ceil(2.1* radius) were used.

2. Checking of grids to the simulation wall is only done for grids at the border.

This is doable due to the size of our grid sizes, making sure that besides the grids at the border, no other particles in any grids could interact with the wall, so we can safely ignore them.

3. Checking and Resolving simultaneously

Instead of splitting the checking for collisions and resolving collisions into two separate sections, they are done simultaneously.

So each grid which is parallelized by threads, constantly checks for collisions and resolves them, and they restart this process as long as one other grid finds a collision.

OpenMP constructs Used

Reduction keyword with parallel for

```
do
  #pragma omp parallel for schedule(static) reduction(|| : collisions_detected)
  for each grid in grids:
      if (grid at border):
        check and resolve with wall
        if collisions found:
          collisions_detected = true

      for each neighbor grid:
          check and resolve particles between grid and neighbor grid
          if collisions found:
            collisions_detected = true
  while(collisions_detected)
```

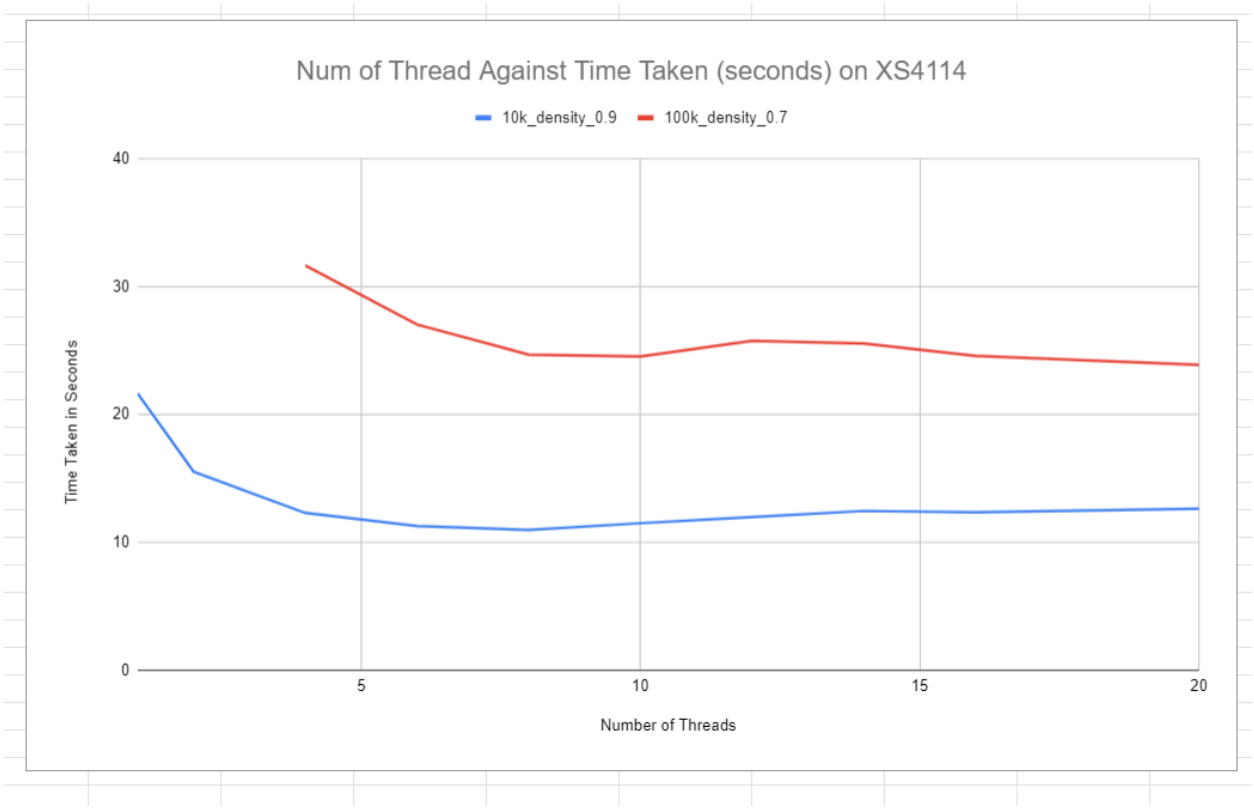
In order for checking and resolving to be done simultaneously, the check and resolve functions both return true if any collisions are detected.

Reduction keyword is used with the shared variable collisions_detected shared across all threads with “||” keyword. This ensures that, at the end of the while loop, the results of all the threads local collisions_detected have an OR operation performed on them, and if even one of them returns true, collisions_detected returns true, and the whole loop is done again. This is important for the synchronization of the program, as without it, threads might wrongly conclude that no collisions are detected and terminate early. This is because, if a grid checks all its neighbors and has no collisions, a grid two cells away might resolve some collisions that cause its neighbor to now have a collision with this current grid due to some particles now facing each other.

Omp critical and static

#pragma omp critical are used when resolve wall collision or resolve particle collisions are used to protect shared resources. Omp parallel for loop static used for updating of positions. Static were found to be faster, as the task sizes are similar (will expand upon in last section of report) This also holds true for the main grid loop. I suspect that as the density increases, most cells have similar particles, hence static scheduling is faster.
The work is distributed equally among the threads, due to the static keyword, and this is generally the case among the grids, as when I use static over dynamic it is faster.
This is to be expected because it is used on two points, the moving of particle location and processing of main loop grids in phase 2, both operations are generally equal in work across all iterations of the for loop.

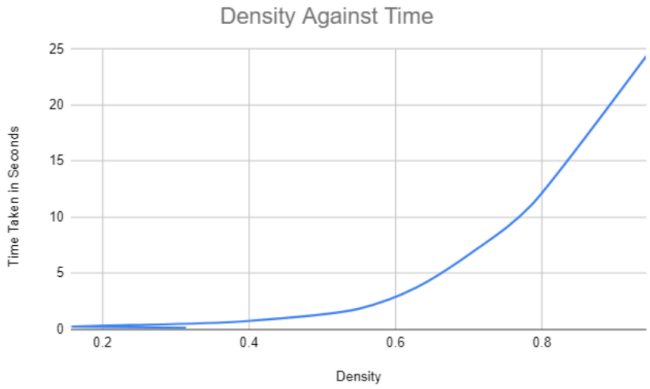
Program Performance as threads increases



Testing on the XS-4114 that has 20 logical cores

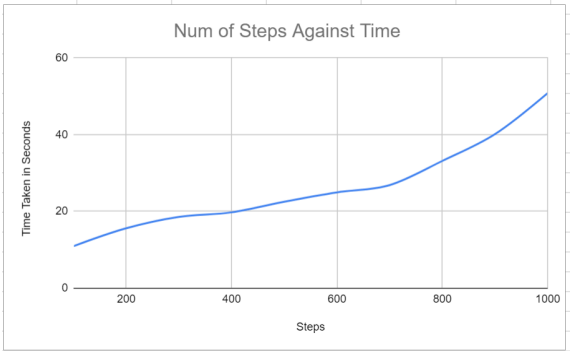
The performance of the program seems to peak around 10 to 12 threads and goes down after 15 threads, even though using a larger particles test case, the higher threads processing power still outweighs the increase in synchronization.
There is a lot of synchronization needed in my code, as when both resolving collisions between particles and walls involve critical sections, hence the higher the thread, the more waiting each of the threads does, the more synchronization overhead causes time delay.
Also, as thread increases, cache contention becomes a problem.
For 10k_density_0.9, at 20 threads, cache_references were 260,125,40, but at 10 threads, it was 226,444,650.

Performance as parameters varies

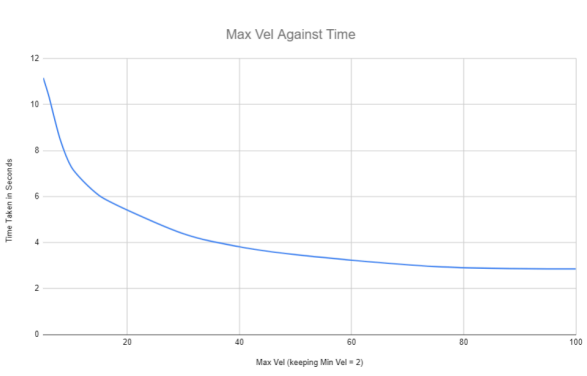


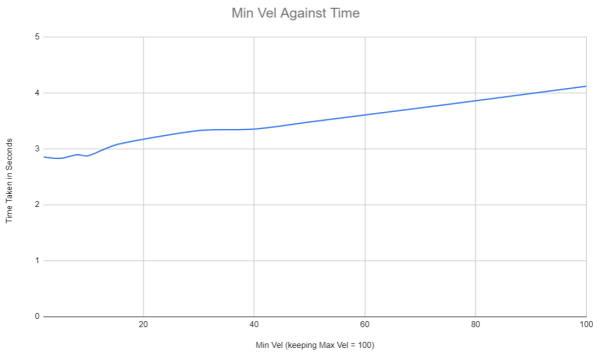
$$\frac{N \times \pi R^2}{L^2}$$ **<- Formula given**

Density parameter seems to affect the time taken the most, as it results in more unresolved collisions due to the particles being closer together with nowhere to go. Hence, we can determine how the number of particles, radius and square_size affects the time taken. Number of particles affects it less so than radius and square_size where a small change to radius or square_size will affect density and hence time taken a lot.



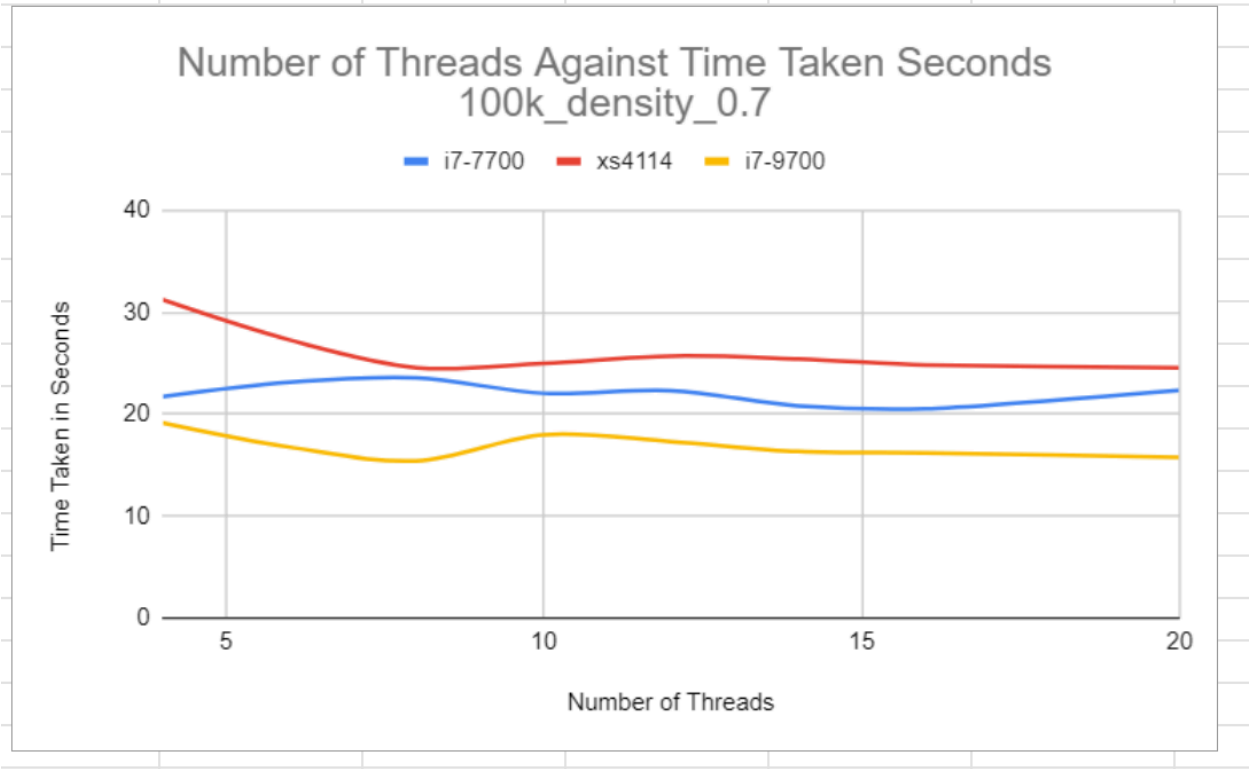
Keeping everything constant, increasing the number of time steps increases the time taken in a linear fashion. This is expected, as we did not allow the energy in the system to decrease, hence the particles will always collide and have their overall velocity/energy constant.





Max Vel seems to affect the time taken a lot more than Min Vel, I suspect as the differences between min and max vel increases, the chances of the particles colliding decreases as their velocities are so different, which explains the trend for min vel increasing causing time taken to increase too. Also as max vel increases, particles are more likely to collide with the wall which is caught earlier in the code, which might lead to faster execution times.

Performance Across Machines



We notice that the initial drop in performance (increase in time) is different for all 3 machines. For XS-4114 and i7-9700 the thread is similar. This was surprising as I thought XS4114's time would just keep decreasing as it has 20 threads to use, but I guess the synchronization overhead slowed it down before the computing power of multiple threads aid in speeding it up again. For i7-7700, I suspect that the number of threads hit the number of physical cores at 4 early on, causing it to slow down due to synchronization overheads as threads increased a lot faster than the rest before eventually speeding up. Even though it does have 2 logical core per physical core, the two logical cores still share resources resulting in it slowing down.

Performance Optimiziation Tried

Most of my optimization were just fine tuning, as I was unable to implement the collision caching

1. Utilizing openmp's reduction construct as synchronization.

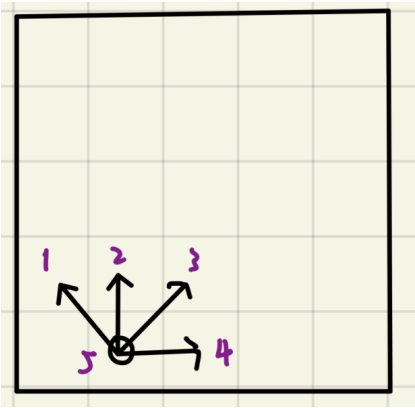
Before doing this, I separated the checking and solving, so in the main loop, will be one set of checking and resolving, then to determine whether the whole iteration of time step would to check and resolve again, I would have an individual function that accesses all grids but only checks whether there are unresolved collisions.

Before: fp_arith_inst_retired.scalar_double = 35,040,523,796, time taken: 25.576

After: fp_arith_inst_retired = 34,096,499,597, time taken: 24.585

We can see that the number of floating points computation has decreased as is_particle_collision is called less

2. Visiting less grid neighbor (no longer used in new method)



Previously, when checking for whether there are collisions, instead of checking all 9 (including itself) adjacent grids. One can just check 5 grids, in order to achieve the same thing

3. Only checking wall collisions for grids that are bordering the simulation space
As the grid cells are ceil(2.1 * radius), this makes it so that only grids bordering the space can interact with the wall as the condition to collide with the wall is <= r. This decreases the number of checks needed.
Without: fp_arith_inst_retired.scalar_double = 34,550,449,457
With: fp_arith_inst_retired.scalar_double = 34,096,499,597
4. Making isBorder function inline.
This was a shock for me as it seems to speed up things by a considerable margin.
Running 10k_density_0.9 with 16 threads in xs-4114 went from 14.5 seconds to 11.9 seconds.
There are less instructions and fewer branches, as inline avoids function calls and simply pastes the whole code at that point of calling; this is expected, as there are no function calls and no setting up of the function stack.
Before -> Instructions: 186,870,249,747 Branches: 32,973,370,191
After -> Instructions: 184,800,456,686 Branches: 32,570,468,540
5. Flattening of grid vector from 3D to 2D
To improve spatial locality, I flatten my vector of grids from a 3D array to an 2D array.
./sim ./tests/large/100k_density_0.7_fixed.in 8
Before: Cache-refernces: 909,182,250
After: Cache-references: 844,675,052
6. Passing vector of Particles instead of vector of particles indices
In order to improve spatial locality, I tried working with vector of particles instead so that it can be directly referenced instead of referencing a vector of indices where the indices can be eg. [0 999 67 14444] which results in more cache misses as the indices are not contiguous in memory. However, this blew up the memory required and ultimately did not work.
7. Switching is_particle_collision to be called after is_particle_moving_closer
There is a small speed up by doing this. I suspect that it is because is_particle_moving_closer returns false more often.
8. Adding quick rejection test for is_particle_overlap, using integer over double type
Failed because grids are already small, added wasted computation and more time delay.

Appendix

Performance as Trend Increases

Ran on XS-4114 using 10k_density_0.9 and 100k_density_0.7 with varying threads

Supporting perf code to show cache contention is that section is ran using

srun --partition xs-4114 perf stat -e cache-references,cache-misses -r 3 ./sim ./tests/standard/10k_density_0.9.in 10
and
srun --partition xs-4114 perf stat -e cache-references,cache-misses -r 3 ./sim ./tests/standard/10k_density_0.9.in 20

Performance across machines ran using

srun --partition i7-7700 perf stat -r 2 ./sim ./tests/large/100k_density_0.7_fixed.in 10

Comparing across 3 machines						
i7-7700			xs-4114		i7-9700	
time	threads		time	threads	time	threads
21.72321538	4		31.22797515	4	19.15036791	4
23.12250812	6		27.28011286	6	16.76253689	6
23.54168315	8		24.55726469	8	15.42055676	8
22.04086795	10		24.97731429	10	17.98237316	10
22.30659	12		25.69374947	12	17.3101423	12
20.81901026	14		25.41587654	14	16.33903759	14
20.523	16		24.83586248	16	16.20021012	16
22.346	20		24.55903998	20	15.76057326	20

Performance as parameters changes

srun --partition xs-4114 perf stat ./sim <file_name> 8

File_name are in tests, labeled folders

1. varyMinMax
2. varyParticles (for density)
3. varySteps

Performance Optimizations tried, guide on how to run them

1. Refer to code sim.cc vs sim static 2D grid
2. NA
3. Refer to code sim.cc and remove the isBorder to replicate
4. Refer to code sim.cc and remove inline in isBorder to replicate
5. Compare sim static 2D grid vs static 1D grid working
   -> note: in the naming i refer to 2D as 3D, and 1D as 2D, as I do not consider the vector of integers as a dimension which I should have
6. Refer to sim gridwithcontinguousmemory, running it with a sufficient large test case will crash due to memory
7. Refer to sim.cc
8. NA