Algorithm:

1. Divide the grid into squares. The length of the square is `max((double)params.square_size / sqrt(params.param_particles), 5.0 * (double) params.param_radius)`. This ensures the number of squares is linearly proportional to the number of particles and the length of each square is at least 5 times the radius of 1 particle
2. Bin the particles. The condition for a particle to be in a square is that its center has to be in the square. Each particle is inside 1 square only
3. For each particle, resolve collisions between the particle and wall.
4. For each square, resolve collisions between particles inside the square.
5. For each square, resolve collisions between particles inside the square, and particles inside of the 8 neighboring squares. It is guaranteed that only the 8 neighboring squares are relevant for square length > 2*radius, since sqrt(dx^2+dy^2)>=max(dx,dy)>2*radius for any particle outside

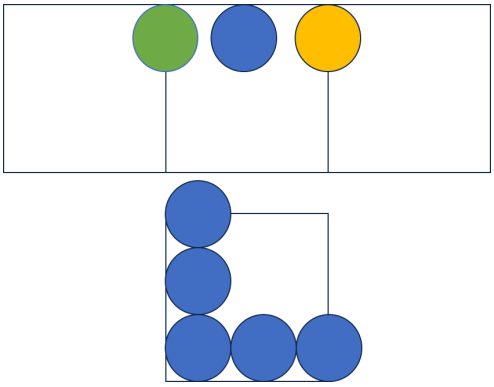
Openmp constructs used: parallel for, collapse

Parallelisation Strategy:
We can parallelize steps 3, 4, 5. For step 3, we can resolve each particle wall collision independently. For step 4, we can parallelise by letting each thread handle resolving collisions for particles in each square.

For step 5, we parallelise this by considering collisions with each of the 8 separate neighbors separately and dividing the work into 4 iterations. In each iteration, for all squares, resolve collisions between particles inside the square, and one of its neighbors. For example, in iteration 4, resolve collisions between square and its top neighbor. In iteration 1, resolve collisions between square and its top right neighbor. This requires iterating through four directions (left/right, top/down, diagonals) and since it is impossible for a particle to intersect particles in diametrically opposing squares, this check can be parallelised for all squares when fixing a direction.

How work is divided among threads:
Step 3: Each thread can take any particle, then check whether this particle is colliding with a wall
Step 4: Each thread can take any square. Then the thread checks particle-particle collisions, for all ordered pairs of particles inside the square.
Step 5: The iterations are run sequentially. But for each iteration, each thread can take any square. Then the thread checks for any collisions between particles inside the square, and inside the neighbor square that is being checked in this iteration.

How we handle synchronization:
We did not use any explicit omp synchronization constructs, such as critical, atomic and lock, as we processed the collisions in a way that prevented race conditions from happening. For step 3 and 4, clearly there cannot be any race conditions. For step 5, race conditions happen when we update collisions (p1,p2) and (p2,p3). But this cannot happen, as the length of square is >= 5 * radius. From the diagram below, it is impossible for p2 to collide with both p1 and p3



p1 is the green ball, p3 is the yellow ball, p2 is the blue ball.

We use the implicit barrier between different directions, as there can be race conditions if particle 1 collides with 2 particles in boxes in different directions.

How and why our program scales with number of threads:
As the number of threads increases, the program can resolve collisions faster, as work is clearly divided among the threads given.

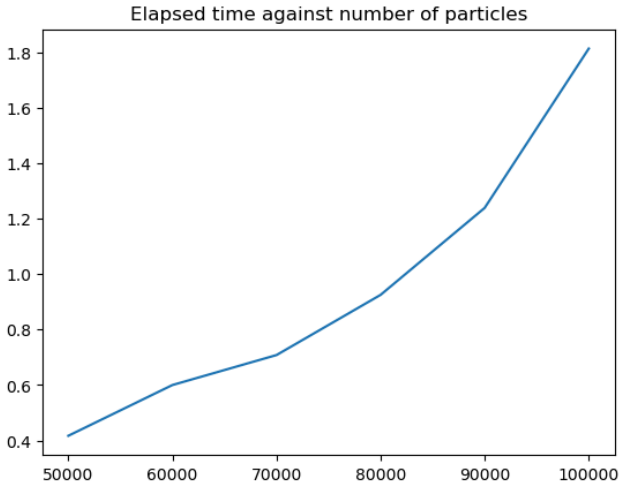
Command: srun –partition xs-4114 time ./sim.perf tests/standard/10k_density_0.9.in <num of threads>

Number of threads	**Time**
1	7.717
2	5.037
4	3.143
8	1.559

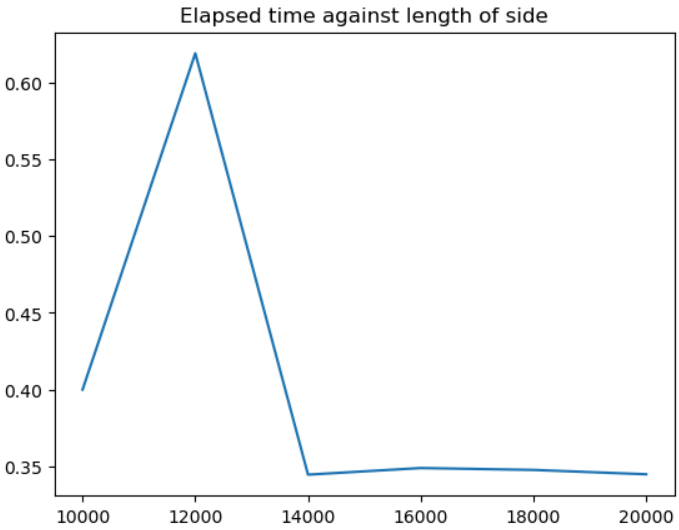
Evaluation of performance of the program
In order to evaluate our program's performance, we tested it against randomly generated input files, varying one parameter at a time. Our default parameters were n=50000, l=10000, r=15, s=100, which has density 0.35. We changed each of these parameters individually to obtain each plot, taking care to avoid increasing density above 1. To test each set of parameters, first we generated an input file satisfying these parameters using gen_testcase.py (with minimum speed 0 and maximum speed 5), then used perf stat to find the average time across 3 runs of our program on this file, all within a sbatch script. The partition used was i7-7700 with a maximum memory of 4GB.

Command: (with appropriate parameter modified) python3 gen_testcase.py 50000 10000 15 100 0 5 > "$temp_file"
srun perf stat -r 3 ./sim.perf "$temp_file" 8

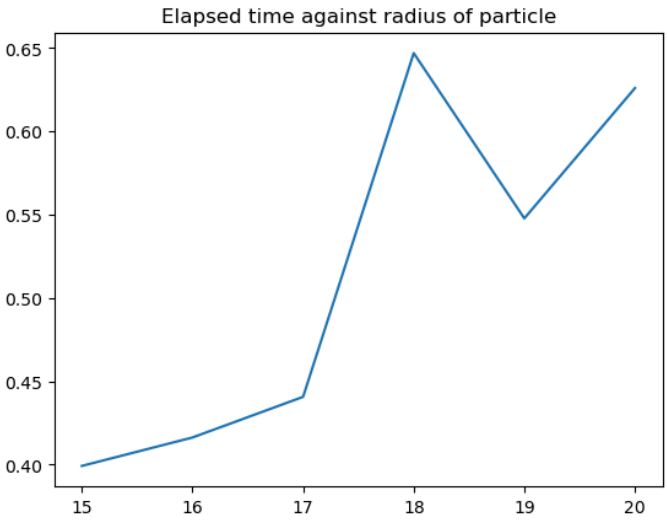
Raw data is available in the Appendix.



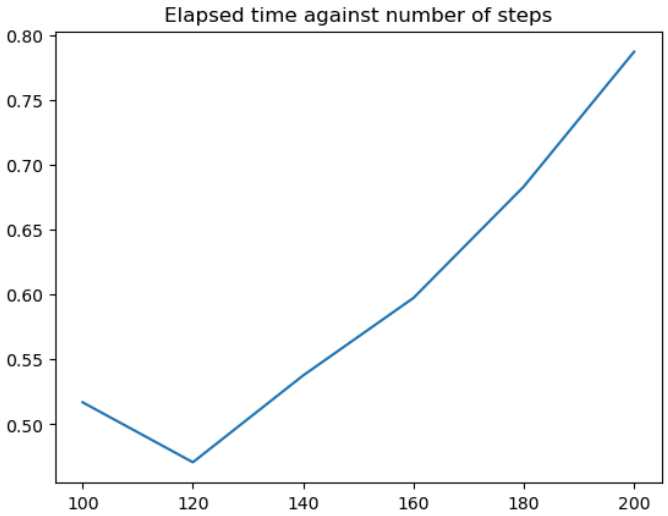
When particles are increased, the elapsed time increases faster than linearly. This is because the expected particles per box increases, and the time taken is roughly quadratic in the number of particles per box (as our algorithm has complexity scaling as the product of the number of particles in the boxes concerned)



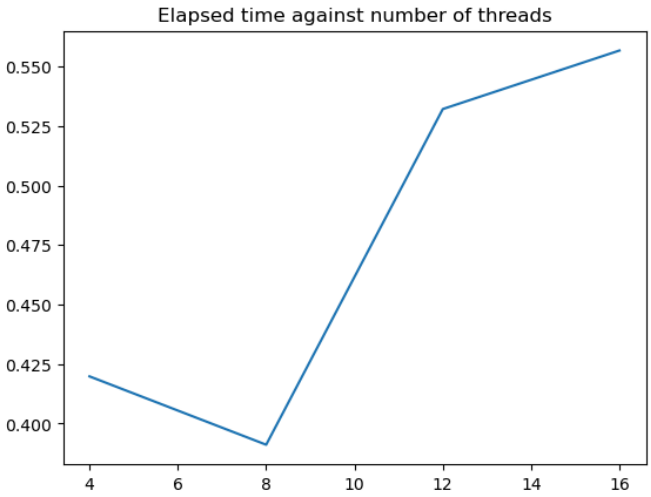
As the side length increases, the chance of particles colliding drops rapidly so the elapsed time decreases to a low value. Most of the time is spent on overhead rather than collisions.



As the radius of a particle increases, so does the time taken, as chances of collision increase quadratically.



As our algorithm performs the same actions per step, it takes time linear in the number of steps.



As threads increase, the runtime decreases if the number of threads is less than or equal to 8. The runtime of our program also increases after that., despite the problem being exactly the same. This is due to the cpu only having 8 hardware threads. Creating more than 8 threads results in overhead of creating and managing threads.

How and why our program performance is affected by the machine we run it on.
Command: srun –partition <machine> ./sim.perf tests/larged_fixed/100k_density_0.7.in <number of threads>

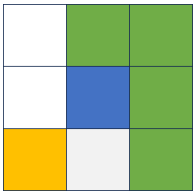
Number of threads	**w5-3423 (12 core, 24 threads, 4.60Ghz)**	**i7-9700 (8 core, 8 threads, 4.70GHz)**	**i7-13700 (8 P cores, 8 E cores, 24 threads, 5.20Ghz for P core, 4.10Ghz for E core)**
1	7.40	5.63	5.10
2	4.53	3.36	3.06
4	2.68	2.11	1.80
8	1.58	1.20	1.07
16	1.20	1.93	0.93
24	0.97	2.15	1.72

When the number of threads is much lesser than the number of hardware threads the cpu has, the i7-13700 has the fastest execution time. This is because the i7-13700 has the fastest frequency among the 3 cpu, at 5.20GHz max frequency. Thus, the i7-13700 is able to execute the program the fastest.

When the i7-9700 is run with 16 and 24 threads, the run time increases, as the cpu only has 8 hardware threads. When run with more than 8 threads, there is no additional benefit, plus there is additional overhead from having to sync more user threads together.

When the i7-13700 is run with 24 threads, the run time increases compared to when run with 16 threads, while the w5-3423 has a decrease in run time when run with 24 threads. This is also when both cpu have 24 hardware threads. This could be due to the i7-13700 having 8 performance cores with 2 threads and a higher clock frequency, and 8 efficiency cores with only 1 thread per core and a lower clock frequency. Hence, when the i7-13700 is run with 24 threads, run time increases as the program has to wait for the slower E cores to finish.

Optimisations:
We realized that we do not have to check, for each square, all of its 8 neighbors. In fact, we only need to check half of its neighbors. For the other half, the neighbor will check for collisions between particles from these 2 squares.

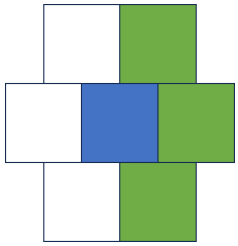


The blue square only needs to check for collisions with the 4 green squares only. For collisions between the blue and yellow square, the yellow square will do the checking. This can halve our execution time. From our data, this optimization decreased our execution time significantly.

Command: srun –partition xs-4114 time ./sim.perf tests/standard/10k_density_0.9.in <num of threads> Take average of 3 runs

Number of threads	Check all 8 neighbors	Check 4 neighbors
1	10.84	7.48
2	6.89	4.66
4	3.77	2.59
8	2.13	1.46

Another optimization we tried is to rearrange the squares in a hexagon manner. This way, we only need to check 3 neighboring squares and not 4. Unfortunately, this reduced execution time slightly. This is because there is additional overhead from having to bin particles into a more complex layout.



The blue square only checks for collisions with the green squares.

Command: srun –partition xs-4114 time ./sim.perf tests/standard/10k_density_0.9.in <num of threads> Take average of 3 runs

Number of threads	Hexagon Arrangement	Normal arrangement
1	5.82	7.35
2	3.77	4.66
3	2.20	2.60
8	1.34	1.48

Appendix
How to reproduce the results is already stated next to each section

Raw data for program performance evaluation

Number of particles	Average Time (3 runs)
50000	0.41695
60000	0.6
70000	0.7081
80000	0.9252
90000	1.239
100000	1.8136

Side Length	Average Time (3 runs)
10000	0.4
12000	0.619
14000	0.3446
16000	0.3489
18000	0.34767
20000	0.34484

Radius	Average Time (3 runs)
15	0.39924
16	0.4163
17	0.4407
18	0.647
19	0.5478
20	0.626

Steps	Average Time (3 runs)
100	0.517

120	0.4708
140	0.5377
160	0.5974
180	0.6832
200	0.787

Threads	Average Time (3 runs)
4	0.4198
8	0.39107
12	0.5322
16	0.5568