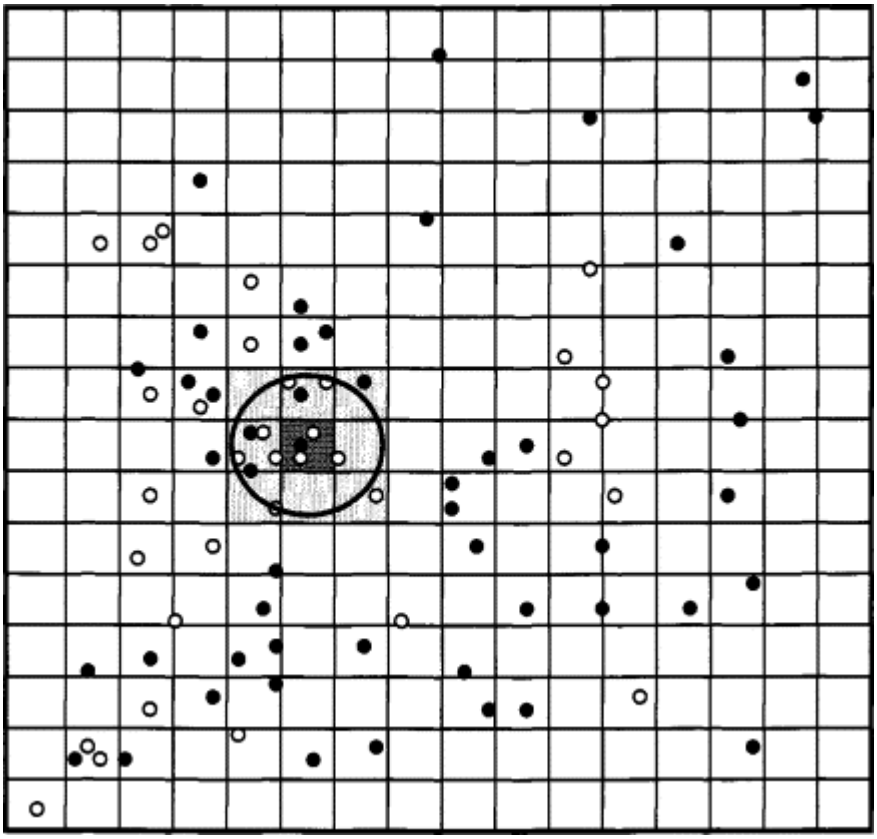
CS3210 - Assignment 1

Student: Nguyen Quang Truong
Student ID: A0293470B

Strategy

The strategy is an attempt to improve broadphase collisions detection by splitting the area of the square into uniform grids (as described in the statement). There are various algorithms for collision detection, but spatial partition appears to be the easiest to implement and optimize with multithreading.

Spatial Partitioning



The space is divided into a grid of "bins" with equal size. Each particle is put

into its corresponding bin. For each bin, the particles inside itself and those inside neighbor bins are more likely to collide. In the above image, the bold-colored bin is checked against its neighbor bins.

With the use of OpenMP directive `parallel for collapse(2)`, a slight speedup can be achieved by merging the 2 for loops together. Race condition may still happen when resolving, thus such process is contained within the `critical` region.

A boolean flag `has_updates` keeps track whether there is any resolved collision. The directive `reduction(|:has_updates)` tells each threaded `task` to keep a their thread-local version of `has_updates`, then update with the global version once finished.

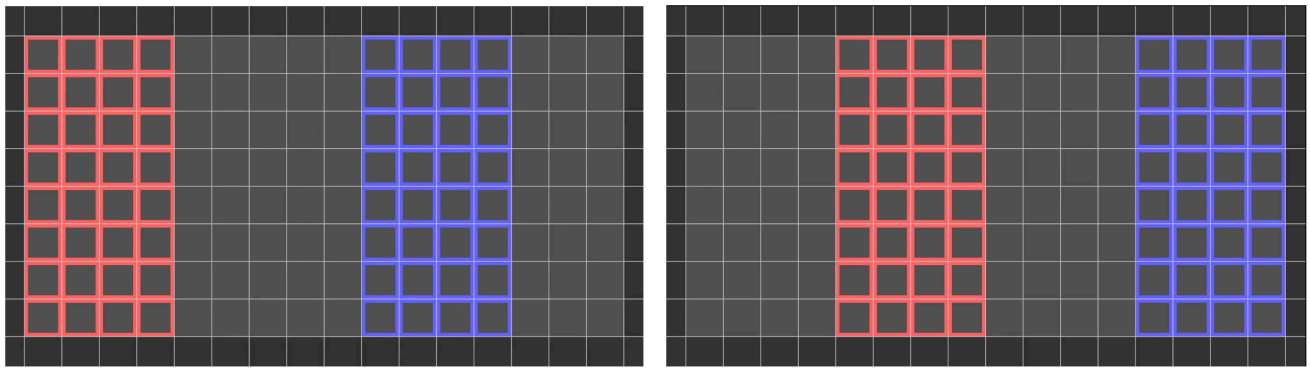
```
#pragma omp parallel for collapse(2) reduction(|:has_updates)
for (const int x : iota(0, bin_num)) {
        for (const int y : iota(0, bin_num)) {
                for (const int nx : iota(x - 1, x + 2)) {
                        if (nx < 0 || bin_num <= nx) continue;
                        for (const int ny : iota(y - 1, y + 2)) {
                                if (ny < 0 || bin_num <= ny)
continue;

                                auto& bin = bins[f(x, y)];
                                auto& other = bins[f(nx, ny)];

                                for (const auto bi : bin) {
                                        auto& [i, loc1, vel1] =
particles[bi];
                                        for (const auto bj : other)
{
                                                auto& [j, loc2,
vel2] = particles[bj];

                                                if (i < j &&
is_particle_collision(loc1, vel1, loc2, vel2, radius)) {
                                                        #pragma omp
critical

resolve_particle_collision(loc1, vel1, loc2, vel2);
```

```
                                                                has_updates
= true;
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
}
```

Parity Spatial Partitioning

"ParallelScan.png" could not be found.

By splitting the space into multiple columns, parallel traversals can be performed. However, if the strides between these scans are small, several bins may be checked at the same time by different threads, causing race condition.



A way to avoid race condition is to check in two passes: check the odd-indexed columns parallelly, and after that, check the even-indexed columns parallelly. This guarantees that for any positive column width, no race condition can occur, and thus there is no `critical` section.

Each check can be considered as an OpenMP `task`, with each pass being an OpenMP `taskgroup`. The number of tasks is proportional to the number of threads.

```
// first pass
#pragma omp parallel shared(has_updates)
#pragma omp single
{
        #pragma omp taskgroup task_reduction(|:has_updates)
        {
                for (const int i : iota(0, num_threads)) {
                        const int start = 2 * i * slice_size;
                        const int end = start + slice_size;

                        #pragma omp task in_reduction(|:has_updates)
                        has_updates |=
resolve_collision_threaded(particles, start, end);
                }

                if (last_cell < square_size) {
                        #pragma omp task in_reduction(|:has_updates)
                        has_updates |=
resolve_collision_threaded(particles, last_cell, bin_num);
                }
        }
}
#pragma omp barrier

// second pass
#pragma omp parallel shared(has_updates)
#pragma omp single
{
        #pragma omp taskgroup task_reduction(|:has_updates)
        {
                for (const int i : iota(0, num_threads)) {
                        const int start = (2 * i + 1) * slice_size;
                        const int end = start + slice_size;

                        #pragma omp task in_reduction(|:has_updates)
                        has_updates |=
resolve_collision_threaded(particles, start, end);
                }
        }
```

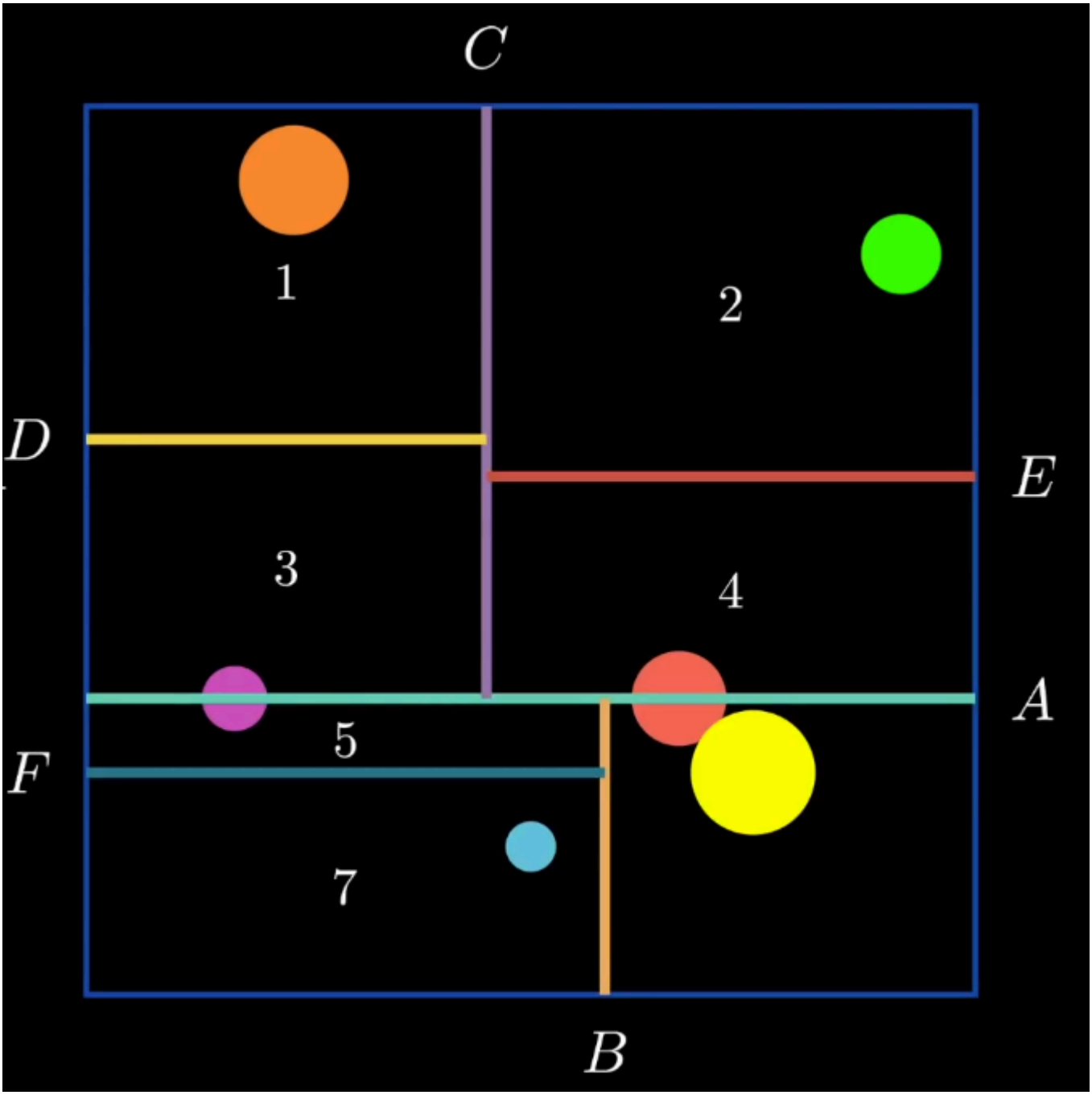
```
}
#pragma omp barrier
```

Rejected Strategy: k-d tree

Spatial Partitioning has one weakness: when the density of the space is low, there are many empty bins, but it traverses through all of them.

The k-d tree solves this problem by dividing the space along the median of the particles on an axis. This "uneven" partition in theory should help speed up the execution, and it is used in single-threaded applications.

However, for this particular problem of particle collisions, k-d tree does not appear to be useful. Since a k-d tree has to be reconstructed every iteration of collision resolution, even building it parallelly does not help.



Other implementation details

One calculation for each pair

Suppose `particle[i]` and `particle[j]` are to be checked against each other. Ideally, only one calculation is needed. Thus, when traversing, the condition `i < j` is applied to ensure only one calculation is carried out.

Multidimensional Array Flattening

`std::vector` allocates data on the heap. `std::vector<std::vector<int>>` allocates an array of pointers, which points to arrays on the heap. These small arrays are not contiguous, and may be inefficient to access.

By flattening it into a one-dimensional array, all elements are on one contiguous buffer.

Performance

The program is run against 3 hardware types: Intel i7-7700, Intel i7-13700, and Intel Xeon Silver 4114. The result is acquired using `perf stat`, each is run for 5 iterations (`-r 5`) to calibrate the result. The time unit is seconds.

The datasets used for displaying the results are `small/random80.in`, `standard/10k_density_0.9.in`, and `large/100k_density_0.7_fixed.in` (all of which are provided with the assignment).

The benchmark for k-d tree will still be included as numbers (to give a reference of why it is rejected). It will not be included in the graph.

`small/random80.in`

Parallel

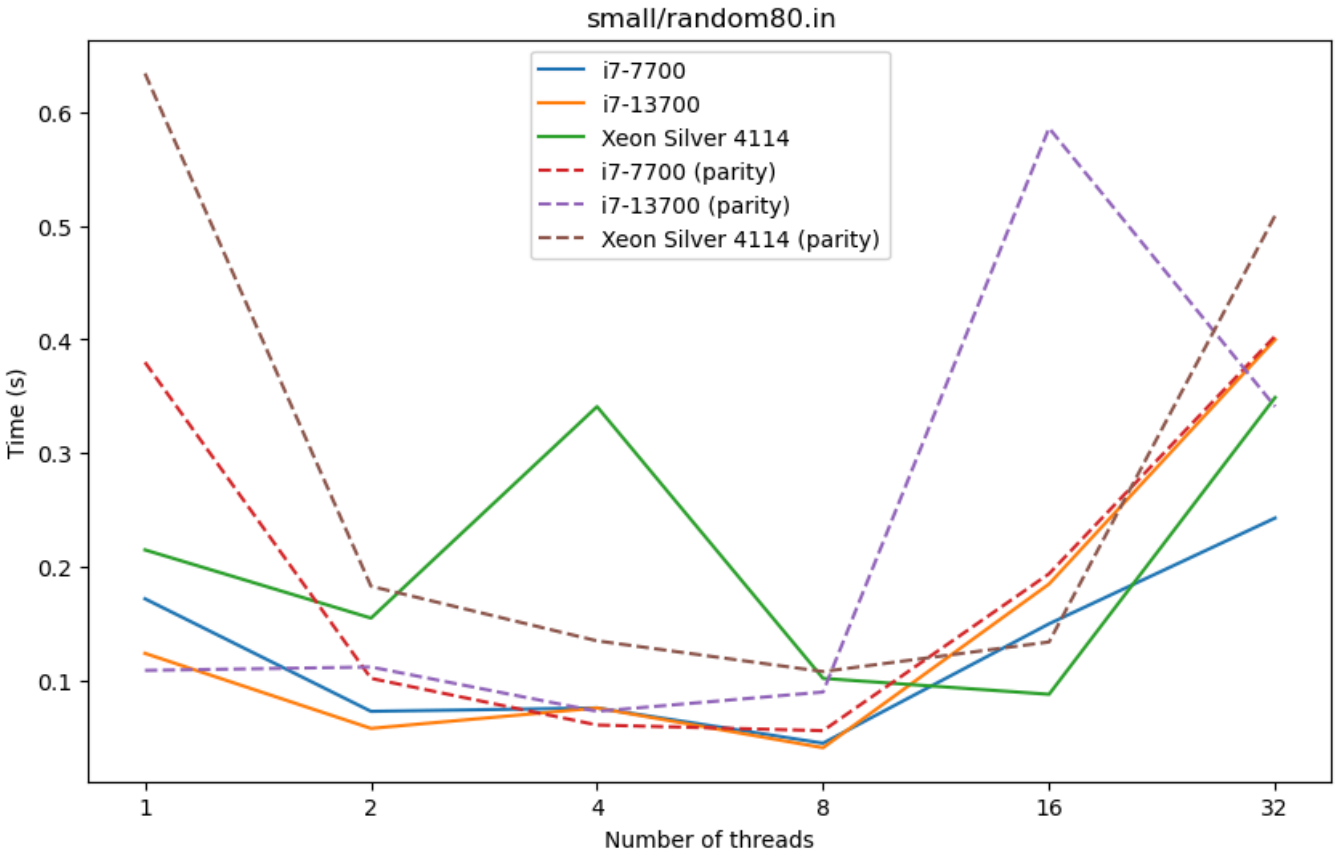
Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	0.172	0.124	0.215
2	0.073	0.058	0.155
4	0.076	0.076	0.341
8	0.045	0.041	0.102
16	0.150	0.185	0.088
32	0.243	0.400	0.349

Parity Parallel

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	0.380	0.109	0.634
2	0.102	0.112	0.183
4	0.061	0.073	0.135
8	0.056	0.090	0.108
16	0.194	0.586	0.134
32	0.403	0.341	0.509

k-d Tree

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	6.635	3.977	9.234
2	3.580	2.411	5.237
4	2.065	1.284	3.065
8	1.826	0.807	2.177
16	1.983	0.675	1.853
32	1.992	1.546	3.452



`standard/10k_density_0.9.in`

Parallel

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	8.564	7.683	12.077
2	4.925	4.380	7.736
4	4.928	3.940	7.041
8	3.202	2.275	4.371
16	3.840	2.051	3.823
32	4.267	5.533	14.034

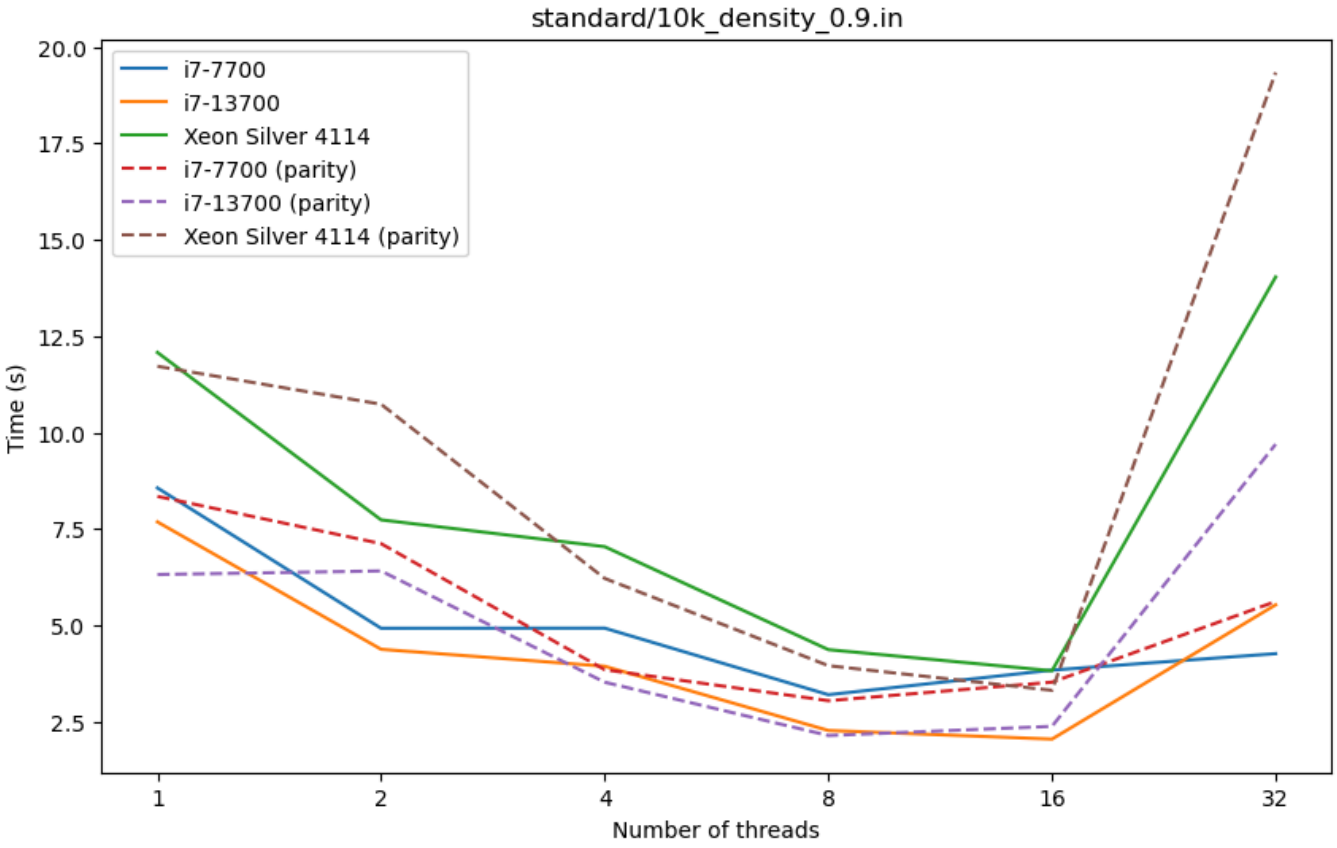
Parity Parallel

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	8.345	6.319	11.718

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
2	7.119	6.411	10.734
4	3.850	3.526	6.218
8	3.047	2.144	3.955
16	3.527	2.382	3.316
32	5.619	9.697	19.336

k-d Tree

Timeout.



`large/100k_density_0.7_fixed.in`

Parallel

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	31.834	21.238	40.832

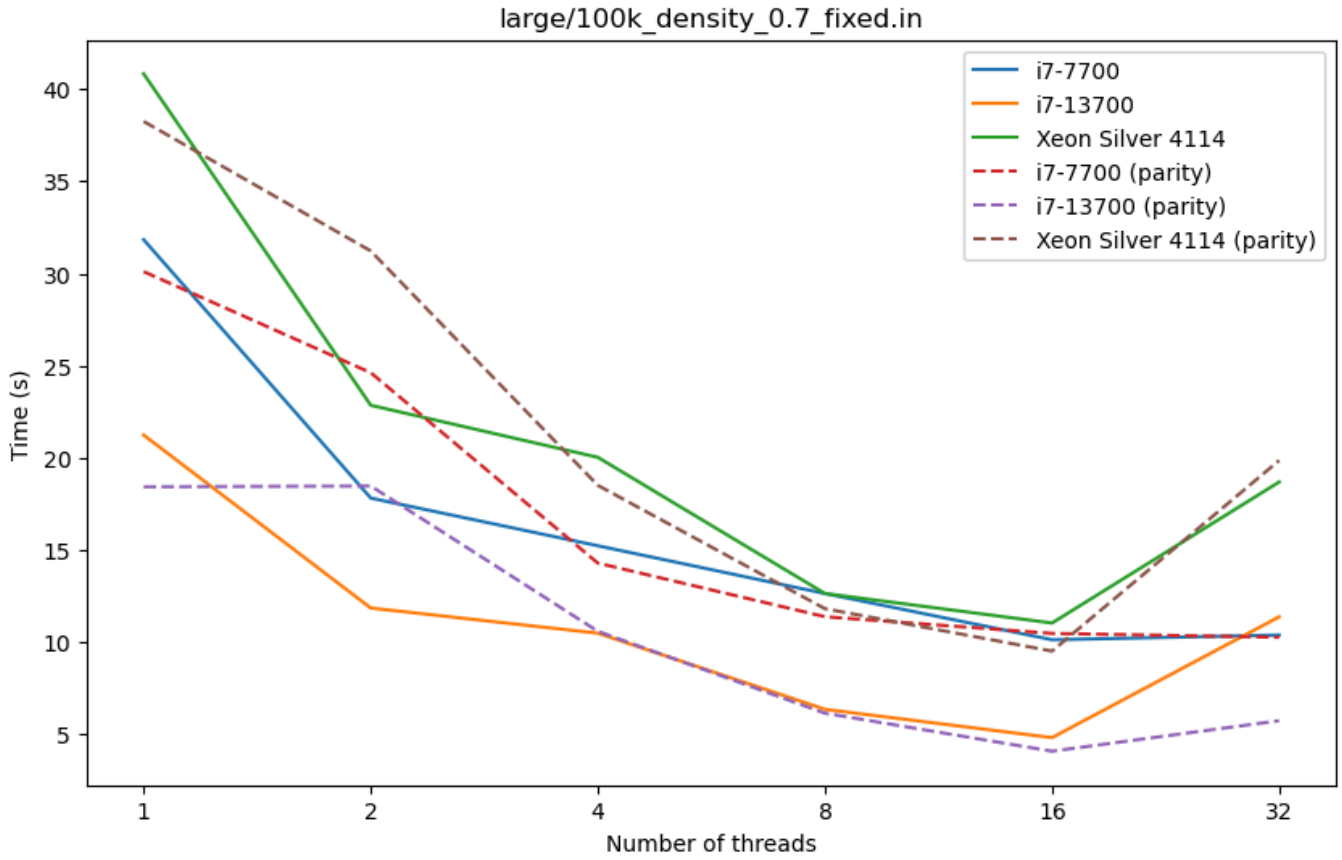
Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
2	17.819	11.856	22.853
4	15.233	10.490	20.029
8	12.642	6.360	12.642
16	10.134	4.826	11.042
32	10.388	11.378	18.692

Parity Parallel

Number of threads	`i7-7700`	`i7-13700`	`xs-4114`
1	30.103	18.429	38.251
2	24.620	18.473	31.221
4	14.292	10.592	18.508
8	11.383	6.151	11.819
16	10.479	4.090	9.520
32	10.268	5.743	19.859

k-d Tree

Timeout.



Conclusion

- The program appears to scale well with the number of threads.
- However, if the number of threads is too high (usually higher than the number of hardware threads), performance may worsen, due to sudden spikes in context switches and CPU migrations.
- For small data, the run time of the programs are inconsistent.

Another interesting property is that Intel Xeon Silver 4114 only appears to beat Intel i7-7700 when the number of threads is from 8 to 16 on several occation. A possible explanation is that despite parallelization in place, the computation still relies heavily on single-core performance. This implies that there is still room for improvement on the implementation.