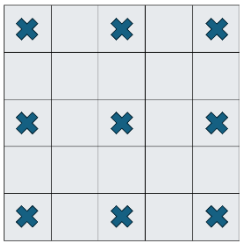
Brief Description

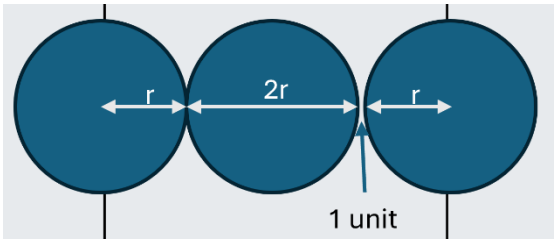
An overview of your chosen algorithm, data structures, and parallelization strategy, why these were chosen.

My solution is to separate the code into finding all the overlaps using broadphase collision, storing them in vectors, and then resolving them repeatedly in parallel in a "checkered" fashion (resolve the "x" in parallel in the image below).

By splitting the grid into smaller boxes of length 4 * r + 1, we are guaranteed that if a particle



"Checkered" fashion.



Particles Unable to Collide

collides with an element on the left box, it cannot collide with an element on the right box at the same time (vice versa for the top and bottom box).

As such, we are free to resolve the left & right boxes in parallel without synchronisation. This pattern continues by shifting this checkered pattern around until we resolve all collisions.

Only vectors were used to store all required data, as they are contiguously stored in memory, allowing for effective exploitation of cache lines. This makes it fast to access the data sequentially in loops (faster than unordered_set even if there are duplicated calculations).

What OpenMP constructs did you use, and why did you use those specific constructs.

For parallelisation, on top of parallel for, collapse(), reduction(), and schedule() were used to modify parallel for to make the parallelisation more efficient. collapse() was used with nested for loops to further divide the loop iterations into threads, instead of dividing only the outer for loop. reduction() was used to combine the results of the threads at the end, so that no synchronisation between threads was needed to store this result. schedule(guided, k) and schedule(static) was used to modify how the work is scheduled between threads to make it more efficient.

How work is divided among threads.

schedule(guided, k) is such that at the start, the size of the chunk (no. of loop iterations) allocated to each thread is large, and decreases as the work continues, until the chunks reach size k. Since the amount of work per iteration could vary widely for these loops, this allows for load balancing to reduce the idle time of threads, while lowering the overhead of the task pool scheduling as compared to schedule(dynamic) (there are less chunks allocated overall with guided). For loops that have a fixed amount of work, schedule(static) was used to evenly distribute them (used only for reserving vector space and clearing vectors).

For finding overlaps, each small grid box in broadphase collision is assigned to 1 thread. For resolving, each checkered "x" is assigned to 1 thread. Both are scheduled with guided.

How you handled synchronization in your program.

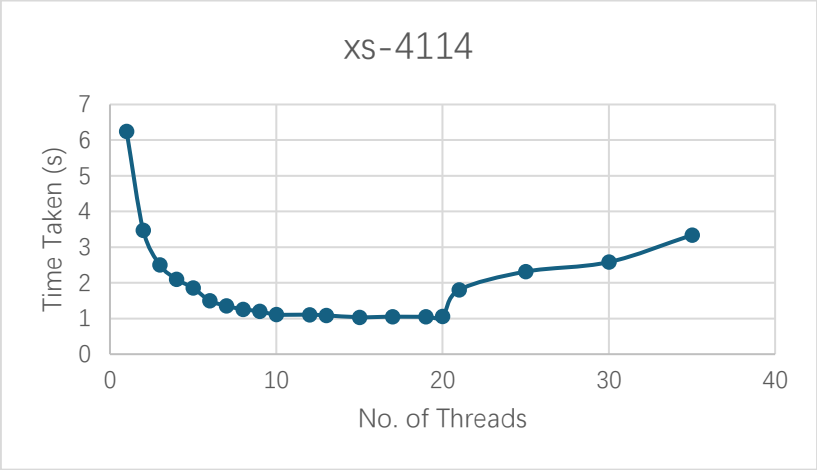
There are no explicit synchronisation constructs used in my code, as it was made such that each thread will not modify the same element/memory space.

When finding overlaps, grid[] was only read from and not modified, so there is no data race. overlaps[] has a different index for each particle, and since each particle is handled by only one thread when finding overlaps, it is safe to write to overlaps[].

Explanation for resolving is in "checkered" fashion as above, requiring no synchronisation.

How and why your program's performance scales with the number of threads. Vary the number of threads and present the data and trends clearly.

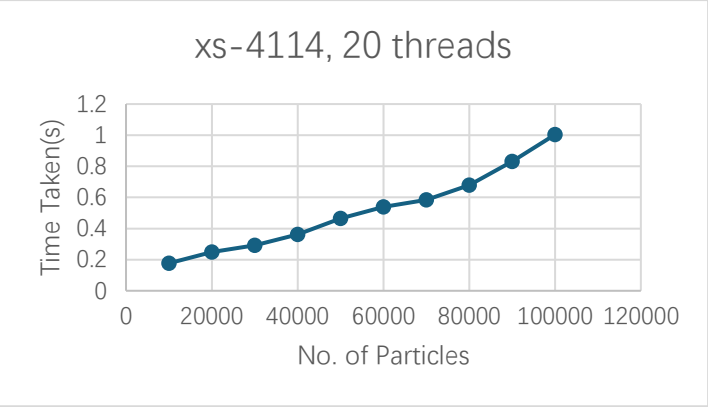
For xs-4114, as the number of threads increases, my program's performance increases as more work is done simultaneously in both finding overlaps and resolving. The time taken decreases until around 10 threads where it plateaus and remains constant until 20 threads as seen in the graph below. This is likely because there is insufficient work to take advantage of the high number of threads, hence the time taken remained constant until reaching the number of hardware threads available. After 20 threads, the time taken starts to increase because the number of allocated threads exceeds the number of physical threads, causing more overhead likely due to more unnecessary context switches, scheduling, and cache contention.



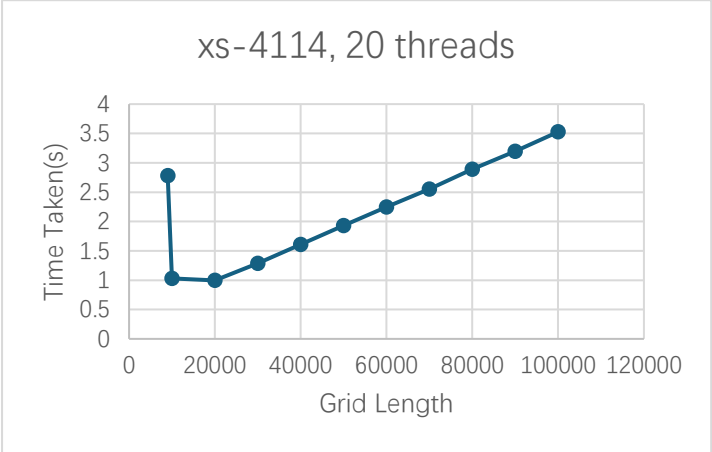
Description, Visualisation, and Data

Yee Jun Hyeok Bryan A02252218L

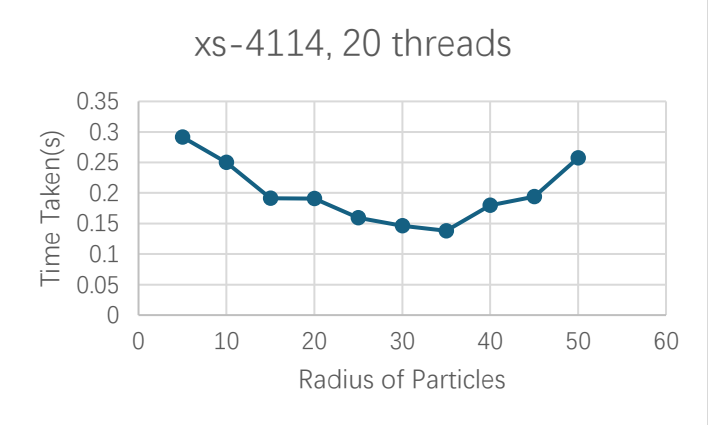
How and why your program's performance changes based on parameters in the input file.



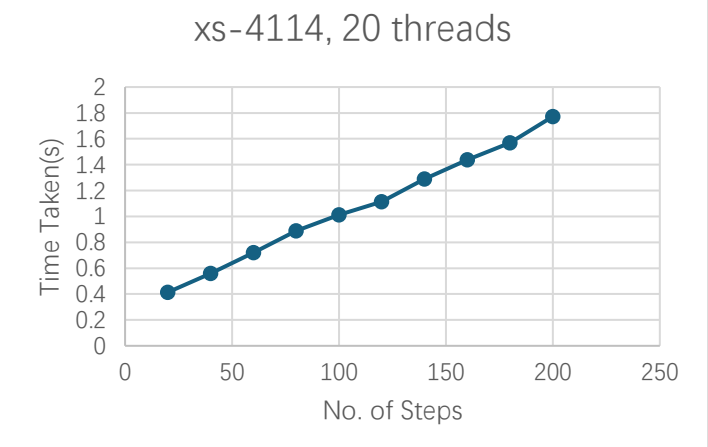
- As the No. of Particles increases, the time taken likewise increases. This is because there are more particles that need to be sorted into the grid, increasing the time needed for finding overlaps. Moreover, the chances of particles overlapping increases, increasing the total amount of resolutions needed.



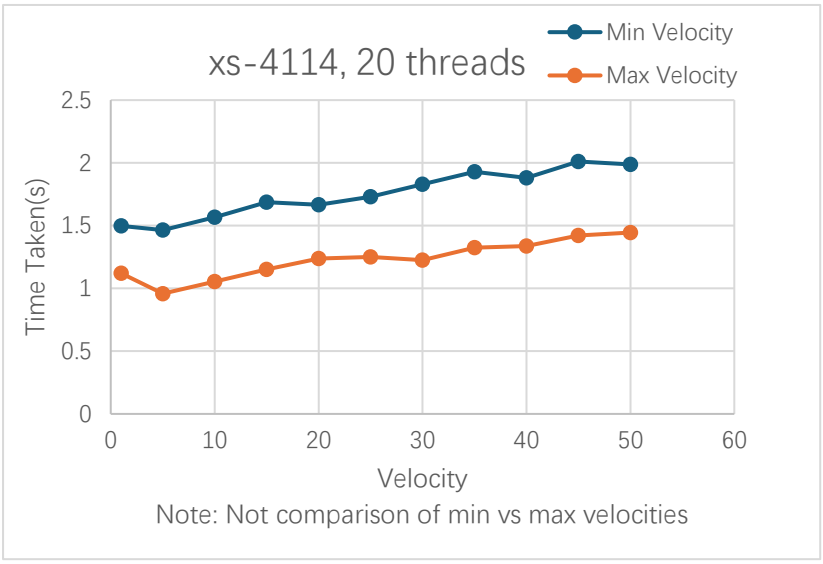
- As the Grid Length increases, the time taken generally increases (almost linearly). This is likely due to the increase in vector sizes needed to store the positions and overlaps of particles, despite the decrease in likelihood of collisions (since density drastically decreases). The spike on the left is likely due to density being near 1, drastically increasing the number of collisions.



- As the Radius of the Particles increases, the time taken generally decreases until a certain point (35 units in this eg). This is likely due to the no. of grid boxes decreasing (since the box width scales with radius), lowering the overhead from the number of vectors created. After that point, the time taken increases since density increases, increasing the likelihood of collisions.

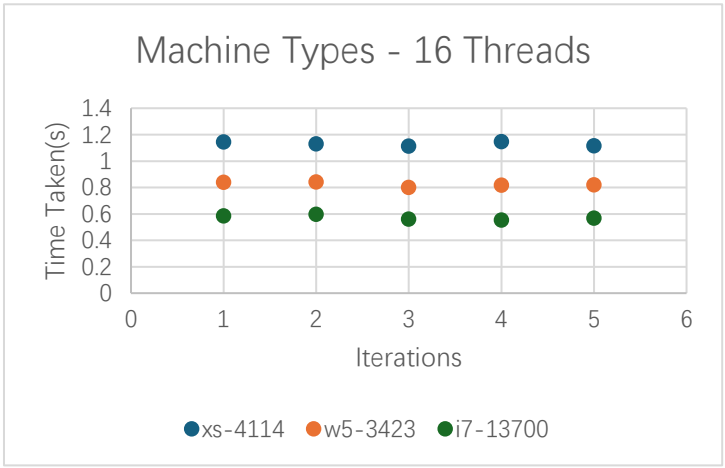


- As the Number of Timesteps increases, the time taken increases. This is because in general, the number of collisions per timestep would stay roughly the same (especially in high density scenarios like the one in the graph – 0.7), thus requiring the same amount of work per timestep. Hence, the time taken would increase linearly with the no. of timesteps.



- As the min / max velocity increases, the time taken generally increases. This could be because as the average velocity of the particles increases, the chance of them exceeding the walls increases, causing more particles to collect at the edges of the grid (since before assigning particles to grids, they are unbounded by the walls and can exceed by however much they want to). This causes the distribution of particles to become uneven, reducing the effectiveness of parallelisation and increasing the time taken.

<u>How and why your program's performance is affected by the type of machine you run it on.</u>



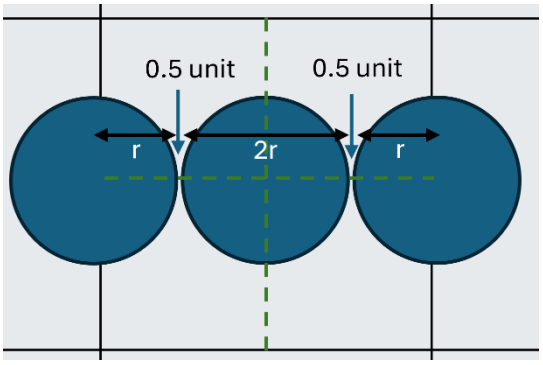
- While all 3 of the chosen nodes (xs-4114, w5-3423, i7-13700) have at least 20 threads, since i7-13700 only has 8 performance cores, we limit the runs to 16 threads (especially since it was seen earlier to plateau at around 10 threads).

There is a clear distinction between each node, with i7 being the fastest, followed by w5, and then xs. This is likely due to the difference in the processor frequencies, with the max turbo of i7 being the fastest at 5.10GHz, followed by w5 at 4.60GHz, and lastly xs at 3GHz. Hence, the higher the frequency of the processor, the lower the time taken.

Describe at least TWO performance optimizations you tried

1) 9 Boxes to 4 Boxes

Taking advantage of the grid box sizes of 4 * r + 1, since a particle cannot simultaneously collide with both the left and right neighbouring boxes (vice versa for top and bottom boxes), instead of checking for collisions between all 9 neighbouring boxes (including its own box), we instead only have to check 4 neighbouring boxes based on the position of the particle within its box.



When the particle is exactly in the centre of the box, it is unable to collide with particles in any neighbouring box (as seen in the picture). When we nudge the particle slightly to the left by 0.5 units, it is now able to collide with a neighbouring box particle (vice versa for top and bottom).

As such, we can divide each box into quadrants (green lines in the picture), and find out which quadrant the particle is in. If it is for example in the top left quadrant, we only need to check the current box, the left box, the top box, and the diagonally left-top box.

This reduces the number of boxes we need to check to less than half, improving the runtime. While we can exclude a particle exactly in the centre from calculations, it is rare to occur and will simply increase complexity of implementation.

The results of this optimisation can be seen in Graph 1 in the Appendix below, showing an improvement of roughly 4.6%. This improvement may not be as huge perhaps due to both variants exploiting cache lines, and most of the time being spent on resolving instead.

2) Self-Contained Resolves

We can immediately resolve particles that overlap and are within the same grid box, without any synchronisation. By having each grid box be handled by exactly 1 thread, this ensures that when resolving particle A with B (that are both within the same grid box), no other thread will be resolving with either particles A or B at the same time. Moreover, since each grid box is handled in parallel, every single one of the grid boxes will have their self-contained collisions be resolved in parallel, enabling many resolves to be done at once.

Moreover, we can do these resolves while finding the overlaps between particles, since they both require a similar distribution of work between threads (1 thread per grid box). This reduces overhead since there will be less duplicate thread creation and distribution, and more exploitation of cache lines since they are accessing the same vector and hence the same region in memory.

The results of this optimisation can be seen in Graph 2, showing almost no difference in timings on average (average of 1.178822 with self-contained resolve, 1.178469 without). It could be that there are too few collisions that occur solely within each grid box to make a difference, or perhaps “checkered” resolve may be too efficient for resolving separately to make an impact.

To test this, by changing the resolving method into a sequential one instead (track only changed), we can see a significant decrease in timing of about 11% in Graph 3. Hence while this optimisation would work well in a sequential main resolve, using it with the parallel “checkered” resolve brings no benefits.

Additional Report – Bonus

Alongside the “checkered” parallel resolve, several other optimisations were made.

- Vectors were used everywhere due to their contiguous storage in memory, allowing for fast iterations.
- Vectors were either pre-allocated or reserved with a reasonable size, minimising the need for vectors to be moved around in memory when running out of space. This leads to a significant performance boost, since moving around vectors in memory is time consuming - O(n) each time, especially for the huge amount of data needed to be stored in this assignment.
- Vectors were mostly cleared of their data & reused rather than making a new vector, since remaking such huge vectors will take longer than clearing (large chunks of free memory needs to be found, and old vector needs to be garbage collected).
- emplace_back() was used for slight increases in speed.
- References were heavily used for function parameters, so that large vectors would not have to be copied, but were instead passed with pointers.
- Finding overlaps with particles within the same grid is done separately from finding in other grid boxes, so that overlaps are only checked for once per pair.
- Stored variables are minimised, and more math calculations are done instead.
- Bitwise operations are used for speed.
- Compiler was changed to Clang++ for more performant compiled code.
- Some of the resolves were done in reverse order of the vector (see below).
- Differing omp parallel for schedules were tested such as guided, dynamic, and chunk_sizes. schedule(guided, 4) was found to be best for finding overlaps, and schedule(guided, 5) is best for resolving. The timings can be found in Graph 4-7 in the Appendix.

Reverse Resolve:

Some of the resolves were done in reverse order of the vector. As these resolves were done within for loops with increasing counters, iterating in reverse decreases the likelihood of back-to-back resolves that are wasted computations. Eg. Particles 1, 4, 5 are iterating in this order, and have these overlaps - 1: [4, 5], 4: [1, 5], 5: [1, 4, 7]. After 1 resolves with 4 & 5, 4 will then resolve with 1 again, even though it has just been resolved (and hence cannot be colliding). By traversing the vector in reverse, after 1 resolves with 5 & 4, 4 will resolve with 5 first. If there is a collision here, the velocity of 4 changes. Then 4 resolves with 1, making it more likely that this resolve is required and not wasted. The grid vector has elements stored in increasing order, and overlaps vector also has elements somewhat stored this way (increasing order but in 4 groups). This allows them to take advantage of reverse order traversal.

APPENDIX

All of the following code can likewise be found in the files "slurm_job.sh" and "generate_tests.sh"

Varying No. of Threads:

```
srun --partition=xs-4114 --time=00:10:00 perf stat -e task-clock
-r 3 ./sim.perf tests/large/100k_density_0.9.in <threads>
```

where `<threads>` is 1-10, 12, 13, 15, 17, 19, 20, 21, 25, 30, 35

All of the following benchmarks were run with (unless otherwise stated):

```
srun --partition=xs-4114 perf stat -e task-clock -r 3 ./sim.perf
tests/generated/<file>.in 20
```

Varying No. of Particles:

Generation:

```
./gen_testcase.py <particles> 10000 15 100 2 5
```

where `<particles>` is 10000, 20000, …, 100000

Varying Grid Length:

Generation:

```
./gen_testcase.py 100000 <grid_length> 15 100 2 5
```

where `<grid_length>` is 10000, 20000, …, 100000

Varying Radius:

Generation:

```
./gen_testcase.py 10000 10000 <radius> 100 2 5
```

where `<radius>` is 5, 10, 15, …, 50

Varying Time Steps:

Generation:

```
./gen_testcase.py 100000 10000 15 <steps> 2 5
```

where **<steps>** is 20, 40, 60, …, 200

Varying Min. Velocity:

Generation:

```
./gen_testcase.py 100000 10000 15 100 <min_vel> 50
```

where **<min_vel>** is 1, 5, 10, …, 50

Varying Max. Velocity:

Generation:

```
./gen_testcase.py 100000 10000 15 100 1 <max_vel>
```

where **<max_vel>** is 1, 5, 10, …, 50

Varying Machine:

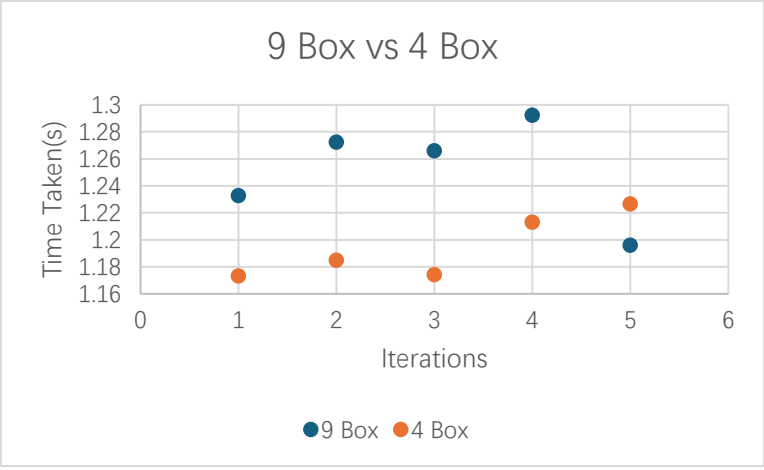
Benchmark:

```
srun --partition=<node> --time=00:10:00 perf stat -e task-clock -r 3 ./sim.perf tests/large/100k_density_0.9.in 16
```

where **<node>** is xs-4114, w5-3423, i7-13700.

The measurements were repeated 5 times per node, (each run consists of -r 3, so 15 per node, added to reduce inconsistencies).

9 Boxes vs 4 Boxes:



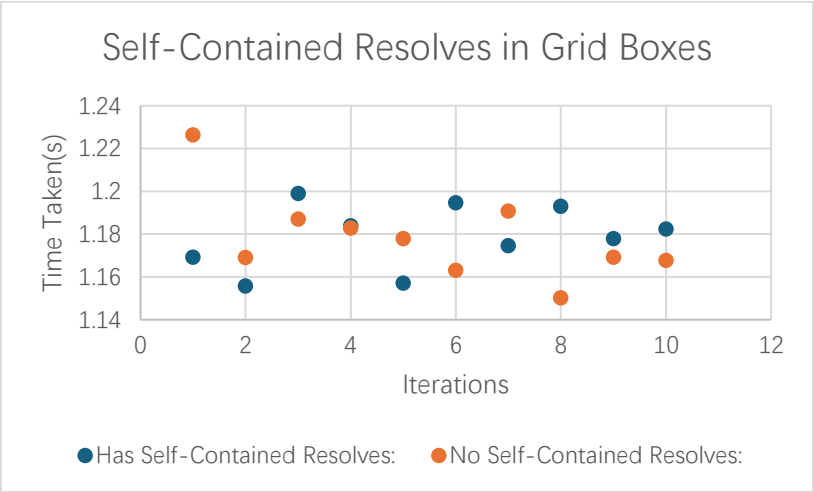
Graph 1

```
srun --partition=xs-4114 --time=00:10:00 perf stat -e task-clock -r 3 ./sim.perf tests/large/100k_density_0.9.in 20
```

was used for both the 9 boxes and 4 boxes variants, repeating the measurements 5 times for each variant (each run consists of -r 3, so 15 per variant, added to reduce inconsistencies).

Note: Use lines 198-213, and comment out lines 218-242 to use 9 boxes.

Self-Contained Resolves – Parallel Main Resolve:



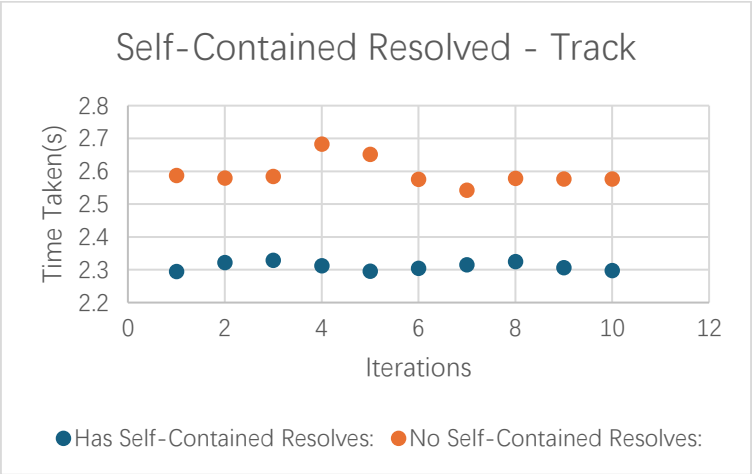
Graph 2

```
srun --partition=xs-4114 --time=00:10:00 perf stat -e task-clock
-r 3 ./sim.perf tests/large/100k_density_0.9.in 20
```

was used for both variants, repeating the measurements 10 times for each variant (each run consists of -r 3, so 30 per variant, added to reduce inconsistencies).

Note: Comment out lines 170 & 178-193 to remove resolving within grid boxes.

Self-Contained Resolves – Sequential Main Resolve:



Graph 3

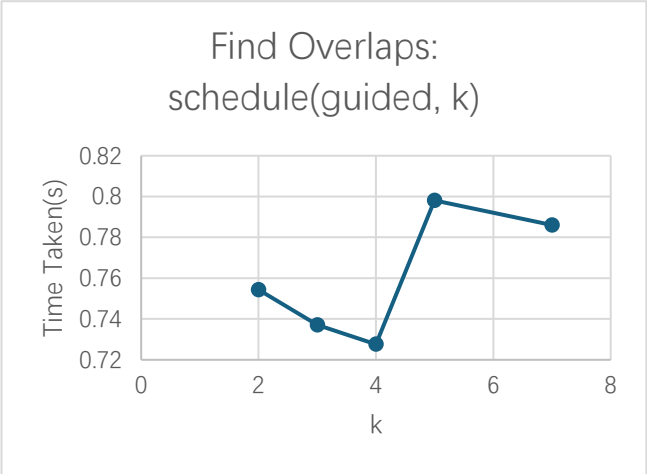
```
srun --partition=xs-4114 --time=00:10:00 perf stat -e task-clock
-r 3 ./sim.perf tests/large/100k_density_0.9.in 20
```

was used for both variants, repeating the measurements 10 times for each variant (each run consists of -r 3, so 30 per variant, added to reduce inconsistencies).

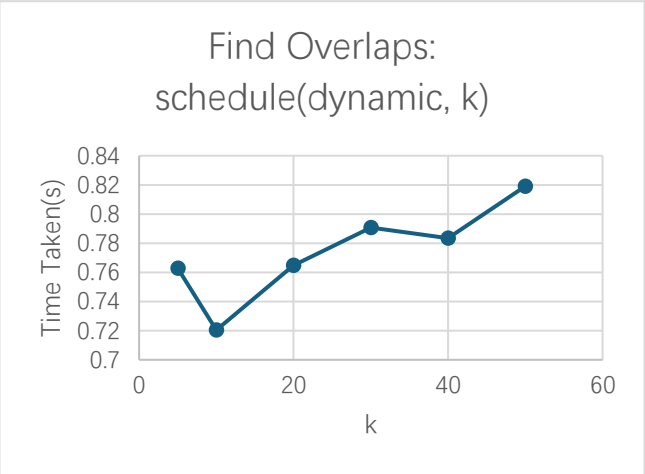
Note: Comment out lines 170 & 178-193 to remove resolving within grid boxes.

Comment out lines 251-269 and uncomment lines 272-284 to use Track resolve

Schedule for Finding Overlaps:



Graph 4

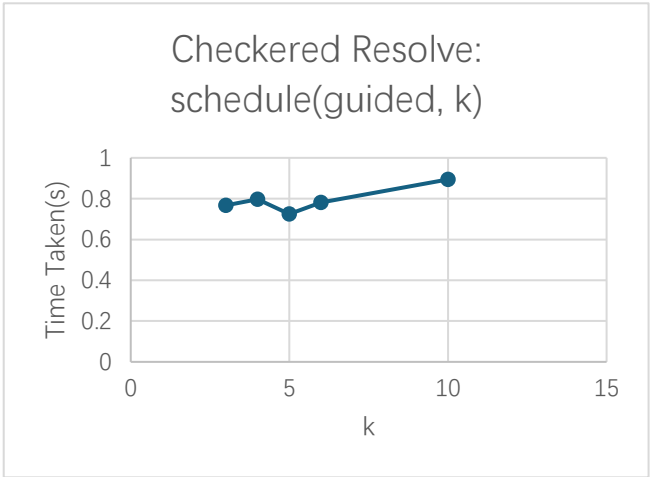


Graph 5

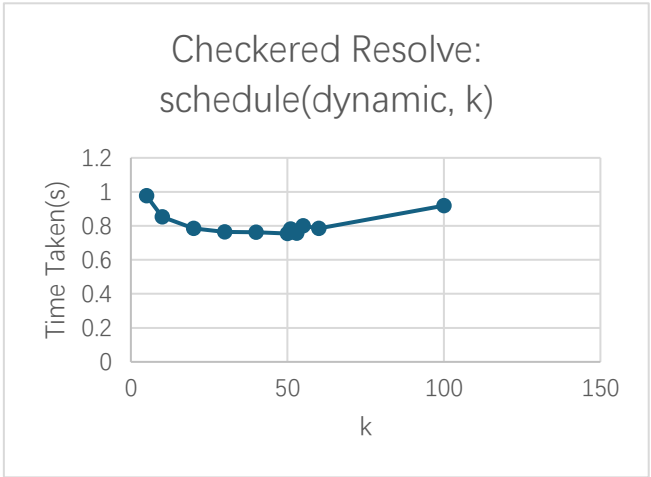
```
srun --partition=i7-13700 --time=00:10:00 perf stat -e task-
clock -r 3 ./sim.perf tests/large/100k_density_0.9.in 8
```

was used for all runs, while fixing resolve to schedule(guided, 5).

Schedule for Resolve:



Graph 6



Graph 7

```
srun --partition=i7-13700 --time=00:10:00 perf stat -e task-
clock -r 3 ./sim.perf tests/large/100k_density_0.9.in 8
```

was used for all runs, while fixing overlaps to schedule(guided).