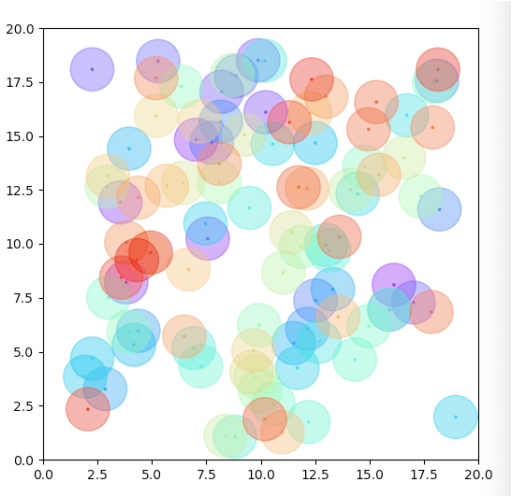
Parallel Computing
CS3210
Assignment One Part One

Xu Yongqing, A0209170M
Liang Ruofan, A0210064Y

1 Introduction

This report deals with the problem of Particles Movement Simulation. Inside a two dimentional square, many particles with velocities collide with each other, creating complex movement patterns. Given the initial status of these particles, we need to calculate their movements afterwards.



In this part of our report, after implementing the sequential simulation algorithm, we use OpenMP to parallelize the implementation. Different parallel methods are described and compared.

After comparing different strategies, we find the best one and then use it to test different inputs. Different square size, particle number, radius and thread number as inputs are compared.

At last, we put forward some thoughts of further improvements and our implementation of them.

2 Sequential Program Design

The full sequential code **collision.c** is attached with this report and won't be posted here as it's too long. Some key thoughts will be further explained.

2.1 Assumptions

During implementation, we made some additional assumptions:

- During initialization, if particles are generated randomly, we ignore the case where they overlap with each other. This is because when data scale becomes huge, it's super expensive to detect the overlapping ones.
- If particles are found overlapped by start of a time step, they will simply go through each other and will not be taken as collisions. This is because if two particles are at the same position at start of a step, there is even no normal direction.
- If a particle already having one collision moves beyond the boundary of the square, this particle will be put to the corresponding boundary as the final position of this time step.
- When a particle is detected to have collision with both the wall and another particles at the same time, we will first consider particle-particle collisions.

2.2 Data Structure Explanation

These are the meaning of some important data structure:

Table 1: Data Structure

Struct Particle	Includes position, velocity etc. of a particle
Struct Collision	Two particles involved and time of collision
colli_p and colli_w	Times of collision with other particles and walls
x_n and y_n	Estimated future position of a particle
cnt	Total estimated collisions that may not happen
real_colli	Number of collisions that truly happen
colli_mat	Bool matrix for particles during one step, true means already collided
colli_time	Store all possible collisions
colli_queue	Store index in colli_time of real collisions
function bound_pos	Function to keep particles wait by walls until the next time step

2.3 Algorithm Explanation

(a.) Collision detection algorithm starting at line 156:

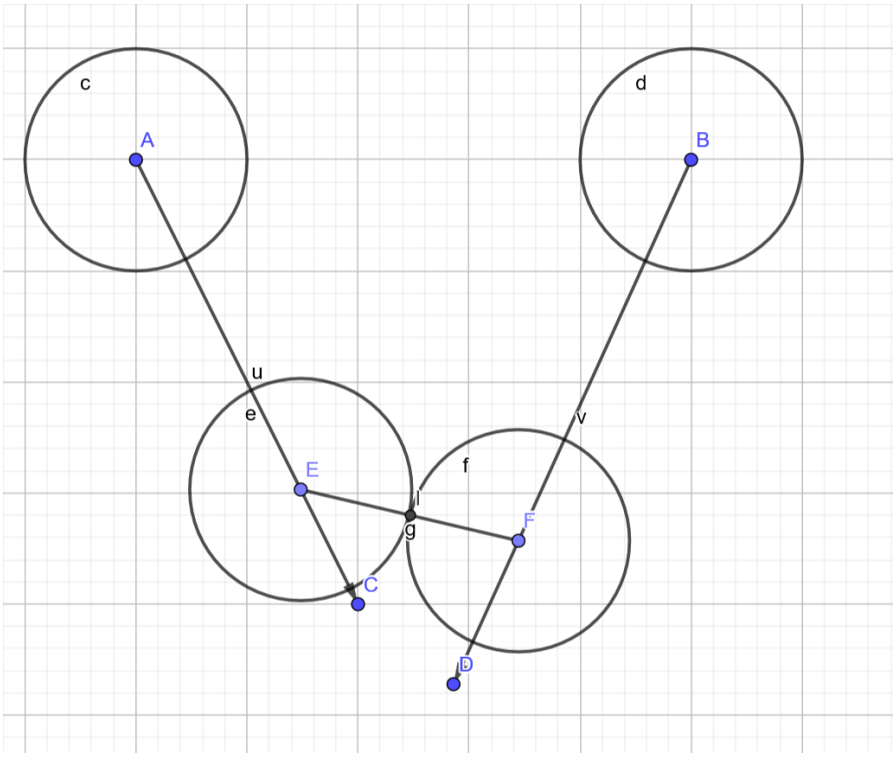
Firstly, we deal with wall collisions. Simply check the position that the particle would reach by the end of one time step and we will find the answer.

Then, for the particle collision detection part, we make early detection and final detection.

For early detection, we note (Δ_x, Δ_y) as the position vector between two particles and (Dx, Dy) as the relative speed vector between them. By making dot product of these two vectors, we can know the angle between speed and position. If the result is equal or more than zero, the acute angle makes them impossible to collide.

For the final detection, we use the formula derived from below:

Figure 1: Collision Analysis



Suppose $\vec{AE} = \lambda\vec{AC}$, $\vec{BF} = \lambda\vec{BD}$, and $||\vec{EF}|| = 2r$

The coordinates are $A(x_1, y_1), C(x_1', y_1'), B(x_2, y_2), D(x_2', y_2')$

Since $\vec{EF} = -\vec{AE} + \vec{AB} + \vec{BF}$, using coordinates we will get

$$\vec{EF} = (\lambda(x_2' - x_1' + x_1 - x_2) + x_2 - x_1, \lambda(y_2' - y_1' + y_1 - y_2) + y_2 - y_1) \tag{1}$$

given $||\vec{EF}||^2 = 4r^2$, set $\Delta_x = x_2 - x_1$, $\Delta_x' = x_2' - x_1'$

Set $D_x = \Delta_x' - \Delta_x = VB_x - VA_x$

Similarly, $\Delta_y = y_2 - y_1$, $\Delta_y' = y_2' - y_1'$ and $D_y = \Delta_y' - \Delta_y = VB_y - VA_y$

Then from equation 1, we will get

$$\lambda^2(D_x^2 + D_y^2) + 2\lambda(\Delta_x D_x + \Delta_y D_y) + \Delta_x^2 + \Delta_y^2 - 4r^2 = 0$$

$$Delta = 4r^2(D_x^2 + D_y^2) - (\Delta_x D_y - \Delta_y D_x)^2$$

Since we already excluded the condition that the direction of movement is opposite to the direction of the other particle during early detection, we get λ as the first time that collision happens:

$$\lambda = \frac{-(\Delta_x D_x + \Delta_y D_y) - \sqrt{4r^2(D_x^2 + D_y^2) - (\Delta_x D_y - \Delta_y D_x)^2}}{D_x^2 + D_y^2}$$

(b.) Velocity update algorithm starting at line 270:

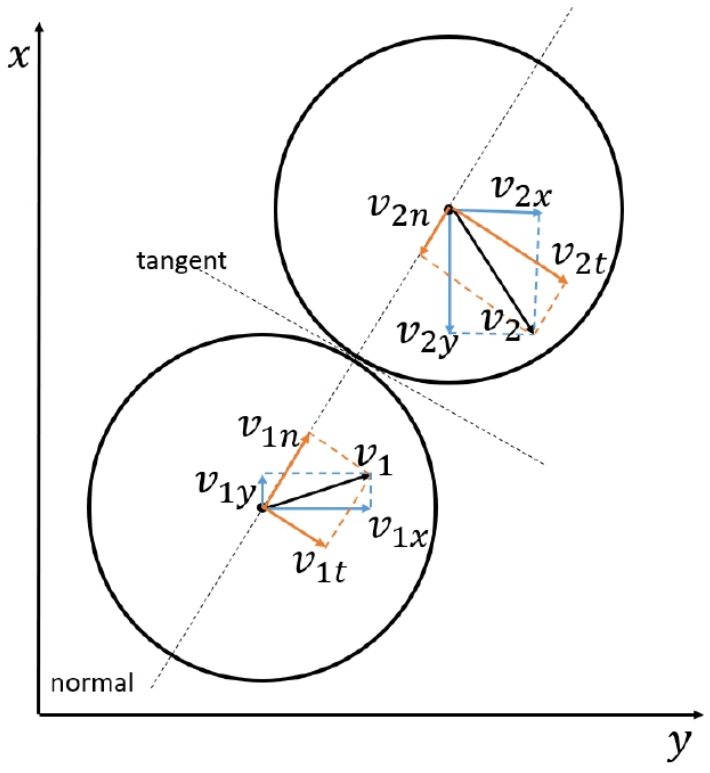
At this part, we have already identified collision with the exact time and two particles involved. We need to update the new velocites after collision.

Principles used in our code is based on the **Physics Engine** part desribed in the assignment.

For wall collisions, only the normal velocity is reversed and the tangent one stays the same.

For particle collisions, we update their velocites by first decomposing them to normal and tangent directions. During decomposition, we first decompose them onto x and y axis and then use dot product.

Figure 2: Picture of Decomposition Principle from Assignment



2.4 Special Considerations

During code writing, we have encountered a couple of problems. Toil and moil, we have them solved. These are the details that we think are non-trivial:

- The direction of velocities can be positive or negative. This has to be taken into consideration when we write the wall collision detection part around line 156 since the time lambda must be positive.
- It's difficult to deal with particles overlapping. During initialization, if the input scale is small, we can create a bool matrix to avoid overlapping. But when the square size is huge, it's impossible to do this. Also, when it's a dense graph, it would be super expensive to check overlapping for every time we create a new particle.

3 OpenMP Parallel Program Design

3.1 Parallel Strategy Description

After realizing the sequential simulation, we firstly use OpenMP to parallelize the implementation.

We tried various ways to parallelize our code. C codes **collision_p1.c, collision_p2.c, collision_p3.c, collision_p1d.c, collision_p2d.c** are different parallelization methods attached with this pdf. Details will be explained.

Take **collision_p1.c** for instance. Some trivial places are parallelized as follows:

Line 138 for-loop:

```
#pragma omp parallel for
        for(i=0; i<N; i++)
        {
            particles[i].x_n = particles[i].x + particles[i].vx;
            particles[i].y_n = particles[i].y + particles[i].vy;
        }
```

Line 185 using synchronization:

```
                int count;
#pragma omp critical
                {
                    count=cnt++;
                }
                colli_time[count].pa = i;
```

Line 346 parallel for-loop:

```
#pragma omp parallel for private(P_a)
        for(i=0;i<N;i++)
        {
            P_a = particles+i;
            P_a->x = P_a->x_n;
            P_a->y = P_a->y_n;
        }
```

Line 276 for-loop:

```
#pragma omp parallel for shared(real_colli,colli_time,
   colli_queue,particles) private(colli,P_a,P_b,Dx,Dy,Delta,dx1,
   dy1,dx2,dy2,DDpDD)
        for(i=0;i<real_colli;i++)
        {
            colli = colli_time + colli_queue[i];
            ..................................
```

These are the parts that doesn't allow much difference in parallelism. But the nested for-loops starting at Line 150 can be parallelized in different ways.

For p1.c, p2.c and p3.c, the difference is which of the nested for-loops is parallelized. For p1.c, we only parallelize the outer i loop, p2.c just parallelizes the inner j loop while p3.c parallelizes them both.

```
//p1.c Line 150
#pragma omp parallel for shared(cnt,colli_time,particles)
   private(j,dx1,dy1,P_a,P_b,lambda_1, lambda_2,lambda,
   wall_colli,Dx,Dy,DDpDD,dDmdD,Delta,dDpdD)
        for(i=0; i<N; i++)
        {..............
```

```
//p2.c Line 206
#pragma omp parallel for shared(cnt,colli_time,particles,P_a)
   private(dx1,dy1,P_b,lambda_1, lambda_2,lambda,dx2,dy2,Dx,Dy,
   DDpDD,dDmdD,Delta,dDpdD)
        for(j=i+1; j<N; j++)
        {..............
```

Note that for p3.c, it's not just the combination of p1.c and p2.c. In default, OpenMP only parallelizes the outer for-loop if you try to parallel nested for-loops. So in order to force it to parallelize both, we use *omp_set_nested(1)* to enable it.

```
//p3.c Line 150 210
        omp_set_nested(1);// enable nested omp
#pragma omp parallel for shared(cnt,colli_time,particles)
   private(j,dx1,dy1,P_a,P_b,lambda_1, lambda_2,lambda,
   wall_colli,dx2,dy2,Dx,Dy,DDpDD,dDmdD,Delta,dDpdD) schedule(
   dynamic)
        for(i=0; i<N; i++)
        {..............
#pragma omp parallel for shared(cnt,colli_time,particles,P_a)
   private(dx1,dy1,P_b,lambda_1, lambda_2,lambda,dx2,dy2,Dx,Dy,
   DDpDD,dDmdD,Delta,dDpdD)
            for(j=i+1; j<N; j++)
            {..............
```

Besides this parallel strategy, we also try to change **schedule method** to further improve performance. In **collision_p1d.c**, we add

$$schedule\ (dynamic)$$

to the outer loop. This will give threads that finish their jobs ealy more work to do. The same strategy is applied to **collision_p2d.c**

3.2 Performance Comparison

To test which parallel strategy is the best, we use the following benchmark with **inputs.txt** (it specifies the initial status of every particle).

N	L	r	S	mode
10000	20000	1	50	perf

The following is the execution time for these strategies:

Table 2: Comparision between Different Parallel Strategies

10000,20000,1	sequential	parallel_1	parallel_2	parallel_3	p1_dynamic	p2_dynamic
first-time	44.92	10.87	8.74	467.82	6.24	94.8
second-time	42.4	10.92	8.54	464.6	6.25	94.46
third-time	41.94	10.65	8.51	466.66	6.25	94.24
shortest	41.94	10.65	8.51	464.6	6.24	94.24

Under the case where schedule are all static, p2 performs best with 8.51s. p1 follows with 10.65s and p3 performs worst with 464.6s.

For p1d.c, we change the schedule in two of the parallelism into dynamic(line 150 and 278). This improves the performance of p1 into 6.24s.

However, if we do the same thing to p2, it actually worsens its performance to 94.24 secs.

We think it's because the outer loop and the inner loop are differently coded. The number of outer loops is the same as the number of particles N, but the number of inner loops is $\frac{n\times(n-1)}{2}$, which is much more. If we apply dynamic schedule to the outer loop, it optimizes the running time because threads finish earlier don't have to stay idle. But for the inner loop, the overhead of doing so would be too much to improve the performance.

Overall, the best parallel strategy is **collision_p1d.c** which parallelizes only the outer loop with dynamic schedule.

Using **collision_p1d.c**, we generate outputs under both correctness mode and speed mode into **outputs_correctness_mode.txt** and **outputs_speed_mode.txt** as attached.

3.3 Special Considerations

During parallel computing, special attention has to be paid to directives. These are the things that we need to consider:

- We must be careful about private/shared variables. In default, variables except the iteration one are taken as shared. If we failed to declare variables that are used differently in different threads private, mistakes would happen.
- We can not use break during OpenMP for-loop parallelism.
- Using the schedule directive, we can also define the chunksize. We have also tried it, but the result showed little difference with different chunksize. So we simply use *schedule* (*dynamic*).

4 Performance with Different Inputs

Run **collision_p1d.c** with different inputs on soctf-pdc-007 (Intel Xeon). The basic benchmark we use to compare is the one used before to test different strategies:

N	L	r	S	mode
10000	20000	1	50	perf

Table 3: Inputs with different numbers of particles

particles, square size, radius, threads	first-time	second-time	third-time	shortest
5000,20000,1,20	1.7	1.7	1.72	1.7
10000,20000,1,20	6.23	6.25	6.25	6.23
20000,20000,1,20	23.92	23.91	23.82	23.82
30000,20000,1,20	52.66	52.6	52.69	52.6
40000,20000,1,20	92.49	91.76	92.05	91.76
50000,20000,1,20	segmentation fault	SF	SF	SF

Table 4: Inputs with different side length of the square

particles, square size, radius, threads	first-time	second-time	third-time	shortest
10000,5000,1,20	6.23	6.25	6.26	6.23
10000,10000,1,20	6.2	6.21	6.22	6.2
10000,20000,1,20	6.23	6.25	6.25	6.23
10000,100000,1,20	6.25	6.28	6.19	6.19
10000,1000000,1,20	6.33	6.31	6.3	6.3
10000,10000000,1,20	6.34	6.29	6.34	6.29

Table 5: Inputs with different particle radius

particles, square size, radius, threads	first-time	second-time	third-time	shortest
10000,20000,1,20	6.23	6.25	6.25	6.23
10000,20000,4,20	6.34	6.13	6.2	6.13
10000,20000,64,20	8.72	8.81	8.78	8.72
10000,20000,128,20	9.33	9.33	9.5	9.33
10000,20000,256,20	13.97	13.89	15.8	13.89
10000,20000,512,20	30.58	31.65	29.2	29.2
10000,20000,1024,20	64.3	64.01	64.19	64.01

Table 6: Inputs with different numbers of threads

particles, square size, radius, threads	first-time	second-time	third-time	shortest
10000,20000,1,1	43.35	43.51	43.82	43.35
10000,20000,1,2	28.69	28.47	28.44	28.44
10000,20000,1,4	18.28	18.35	18.53	18.28
10000,20000,1,8	11.45	11.26	11.68	11.26
10000,20000,1,16	7.3	7.4	7.36	7.3
10000,20000,1,20	6.23	6.25	6.25	6.23
10000,20000,1,32	6.24	6.24	6.19	6.19
10000,20000,1,64	6.31	6.29	6.32	6.29
10000,20000,1,128	6.38	6.76	6.84	6.38

5 Observations

In our report, we have yet made two comparison. One is at Section 3. We use the basic benchmark to test different parallel strategies. Its result is in Table 2.

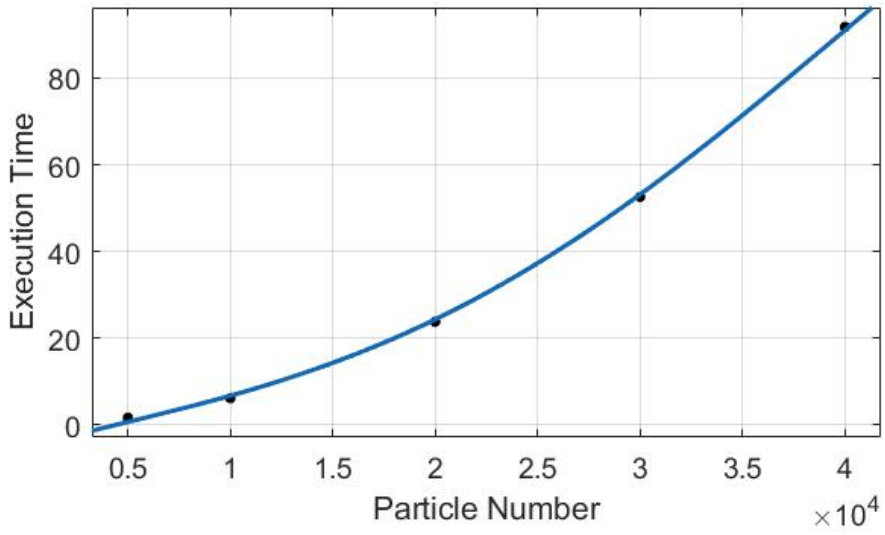
We found that parallelism of the inner j loop performs better than both parallelism of the outer i loop and parallelism of both loops when the schedule method is static.

But when we use dynamic schedule, the parallelism of outer loop performs even better than the inner one. Meanwhile, dynamic schedule not only does not make the parallelism of the inner loop performs better, but actually worse. We guessed the reasons in Section 3 that the reason might be the loop size difference.

The other comparison is the one in Section 4. Analyzing these tables, we can observe some interesting facts.

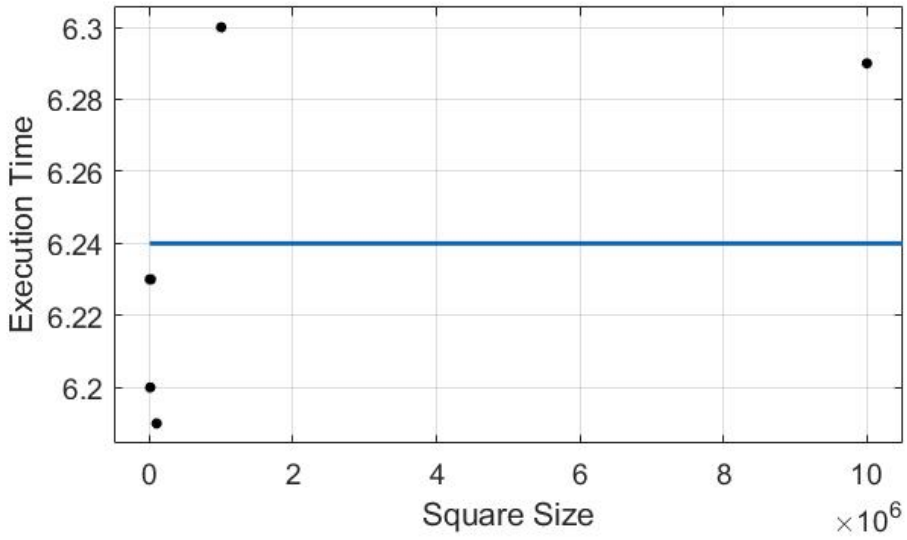
In Table 3, we try inputs with different numbers of particles. With the increase of particle number, the execution time increases exponentially. When the particle number reaches 5,000 segmentation fault happens because the computer can't handle it.

Figure 3: Particle Number



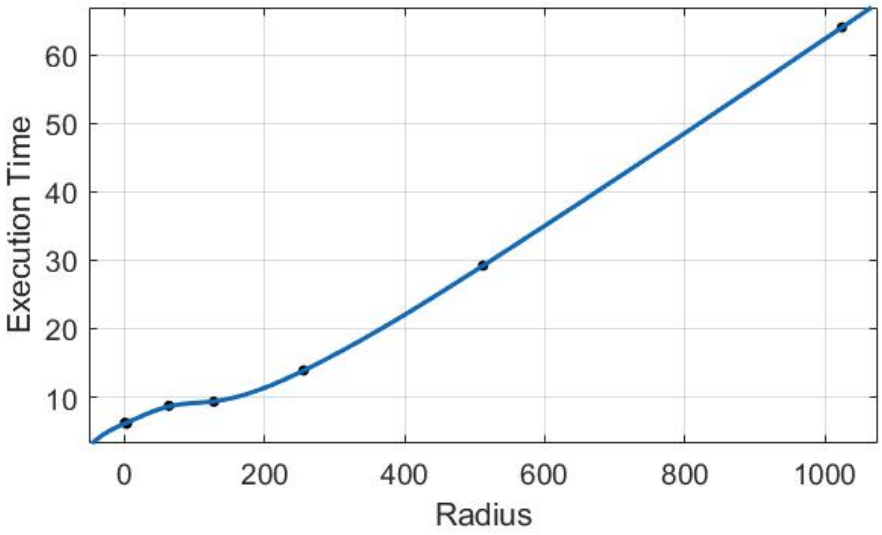
In Table 4, we try inputs with different square sizes. Difference in square size does not matter much and execution time barely changes.

Figure 4: Square Size



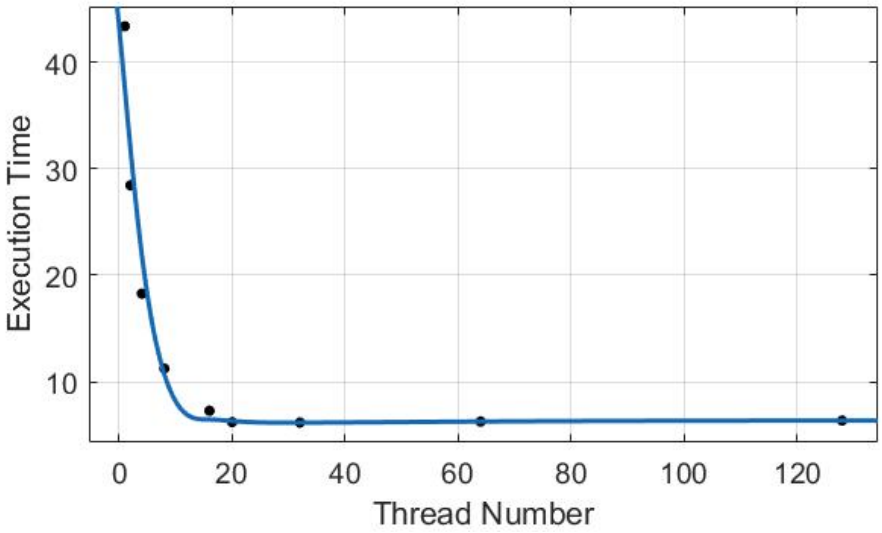
In Table 5, we try inputs with different particle radius. The execution time increases linearly with radius.

Figure 5: Particle Radius



In Table 6, we try inputs with different thread numbers. When thread number is less than 20, execution time reduces sharply with the increase of thread number. Meanwhile, after thread number exceeds 20, execution time stays nearly the same.

Figure 6: Thread Number



6 Reproducing Results

(a.) To reproduce data in Table 2 (comparison between different parallel strategies), use **input.txt** attached. Firstly compile different C codes, for example, use *gcc -fopenmp -o collision collision.c -lm* to compile the sequential code. Then use *perf stat – ./collision* on

Intel Xeon to measure the execution time.

(b.) To measure the influence of number of threads on performance, use **collision_p1d.c**, change Line 12: *define T_NUM 4* to the required thread number and uncomment Line 78: *omp_set_num_threads(T_NUM);*

(c.) For measurement of the influence of particle number, square size and particle radius, simply change **inputs.txt** into the required ones. Then use the methods described in (a.) to get execution time.

7 Further Improvements

7.1 Better Collision Detection Algorithm

In **collision.c**, we use the mathematical formula derived by ourselves in Section 2.3.

However, performance can be further improved using methods described in `https://www.gamasutra.com/view/feature/131424/pool_hall_lessons_fast_accurate_.php?page=2`.

We try to update our code using this method into **collision_relative.c**. We also make comparison before and after we change our code. Compare with this benchmark:

N	L	r	S	mode
4000	200	1	100	perf

The performance comparision:

Naive	Relative
3.65	3.30

7.2 Grid Parallel Algorithm

We also try the **Grid** Algorithm to further improve performance.

The grid algorithm divides the square into 16 equal grids. Each particle has its own moving trajectory, which can be bounded by a box. Then Map these boxes into the 16 grids. We can repeat this process continuously. Furthermore, we divide particles into two categories: one is those completely inside a grid and the other is those only partly inside a grid.

Using this algorithm, we write **collision_grid.c**.

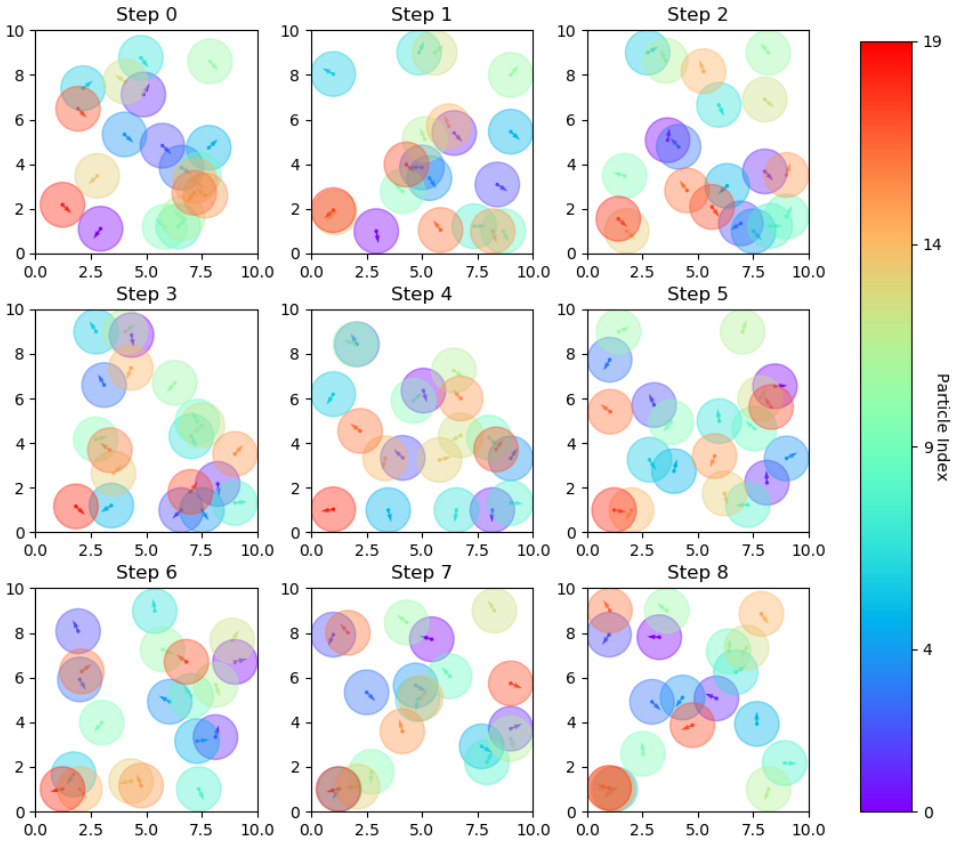
8 Conclusions

During Assignment Part One, we have managed to finish the sequential simulation of particle collision and use OpenMP to parallelize it.

Our assumptions and algorithms are described briefly. Besides, we made comparison between different parallel strategies and different inputs considering particle number, square size, radius and thread number.

The following is the heat map of a nine-step simulation drawn. Different color indicates different particles.

Figure 7: Simulation Heat Map



At the end of our report, we put forward two further improvements to our algorithms and codes realizing them. We hope we can use them in the future assignment.