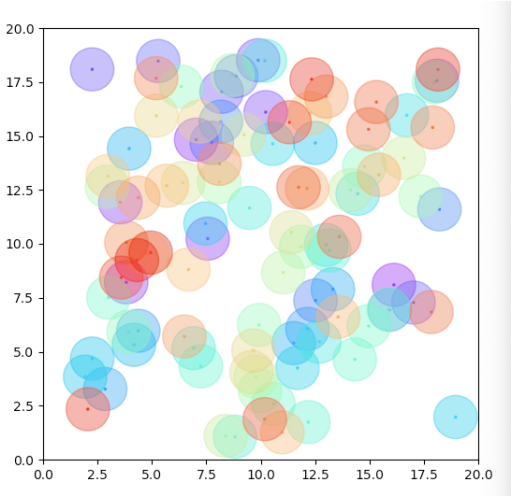
Parallel Computing
CS3210
Assignment Two

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.



Within our report, after implementing the sequential simulation algorithm, we use OpenMP, Cuda and OpenMPI to parallelize the implementation. Different parallel methods are described and compared.

After comparing different strategies, we find the best ones for both OpenMP, Cuda, OpenMPI, and then use them to test different inputs individually. Different square size, particle number, radius and thread number as inputs are compared.

Besides, horizontal comparison between the best algorithms of OpenMp, Cuda and OpenMPI are done regarding different inputs. Some conclusions are reached from our observations.

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, unless they at the exact same place (in this case there will be no normal direction), calculation will be done from the overlapping location.
- If a particle has already had one collision that moves beyond the boundary of the square, this particle will stop at the corresponding boundary to be the final position of this time step.
- When a particle is detected to have collisions with both the wall and other particles at the same time, we will first consider the particle-wall collision.

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 162:

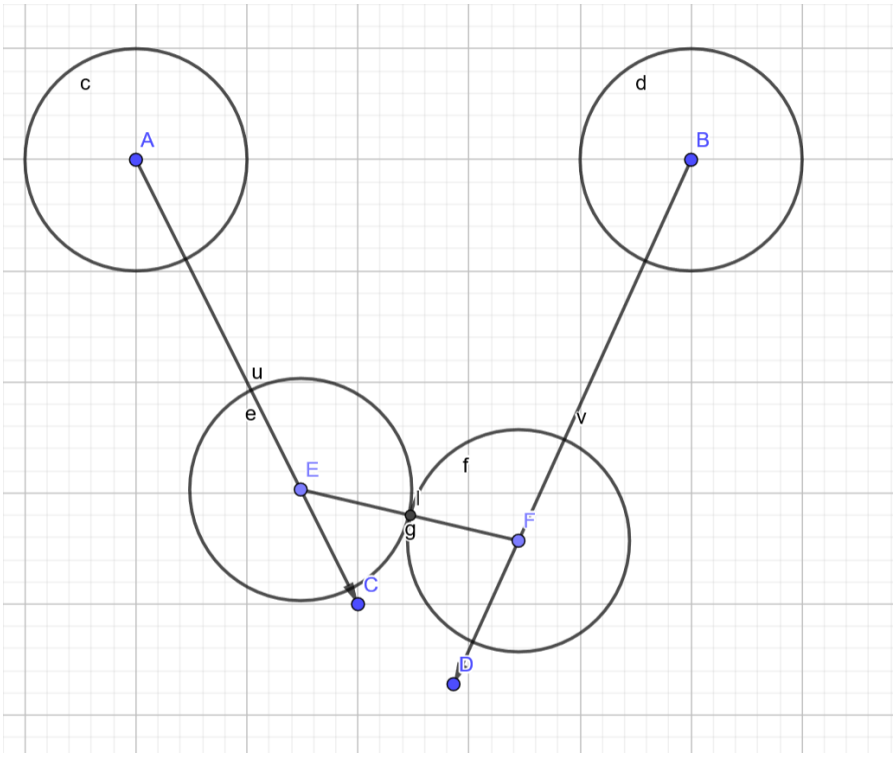
Firstly, we deal with wall collisions. Simply check the position that the particle would reach by the end of one time step and we will find the answer.

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:

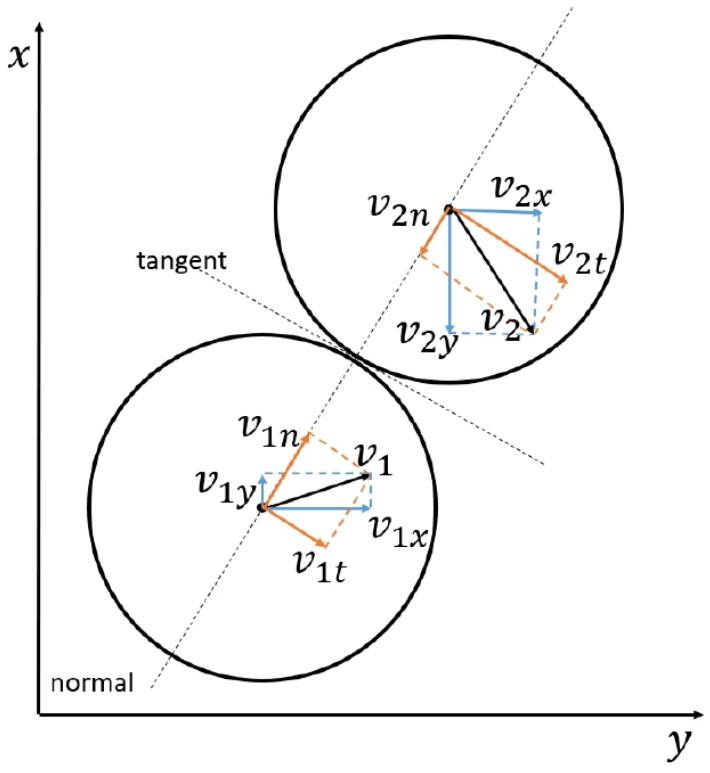
At this part, we have already identified collision with the exact time and two particles involved. We need to update the new velocites after collision.

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 162 since the time lambda must be positive.
- It's difficult to deal with particles overlapping for randomization. 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.
- In order to let wall collisions have a higher priority than particle-particle collisions, we set the particle number of corner (both x and y walls), x-wall and y-wall collisions to be -1, -2 and -3 so that we don't need to change the compare function.

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** in our **first submission** are different parallelization methods attached with this pdf. Details will be explained.

As it's quite misunderstanding which one is our best try, we only upload **colliison_openmp.c**, which is same as **collision_p2d.c** this time. Original codes for all our different tries are within our first uploading.

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

Line 139 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 186 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*).

3.4 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

3.5 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.

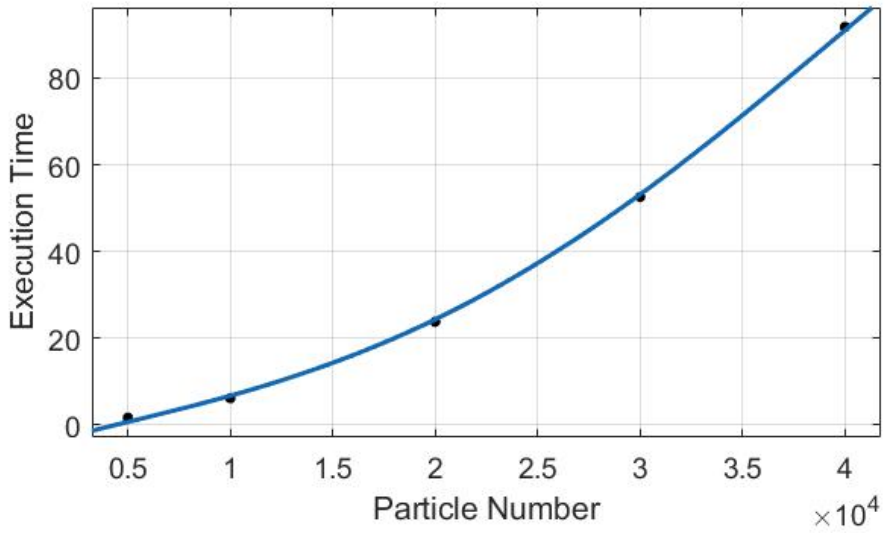
We found that parallelism of the inner j loop performs better than both parallelism of the outer i loop and parallelism of both loops when the schedule method is static.

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.

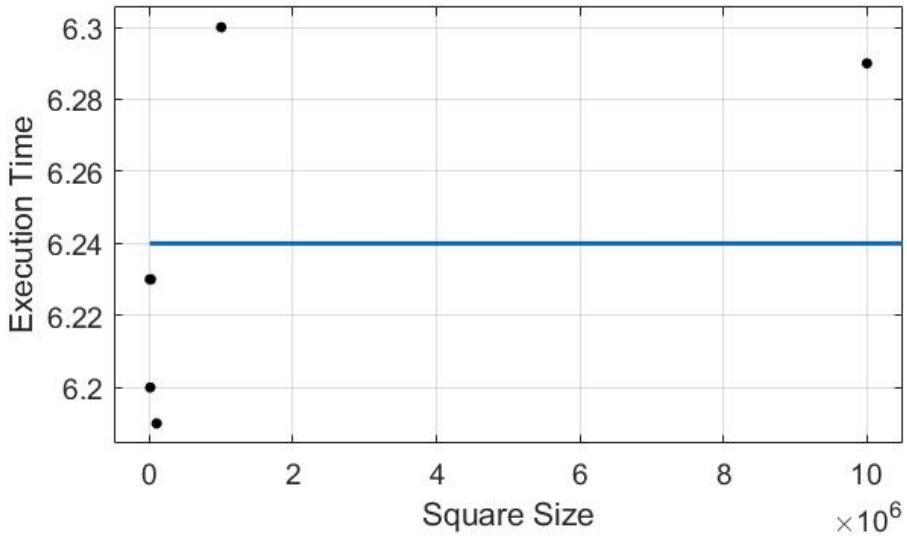
In Table 3, we try inputs with different numbers of particles. With the increase of particle number, the execution time increases exponentially. When the particle number reaches 5,000 segmentation fault happens because the computer can't handle it.

Figure 3: Particle Number



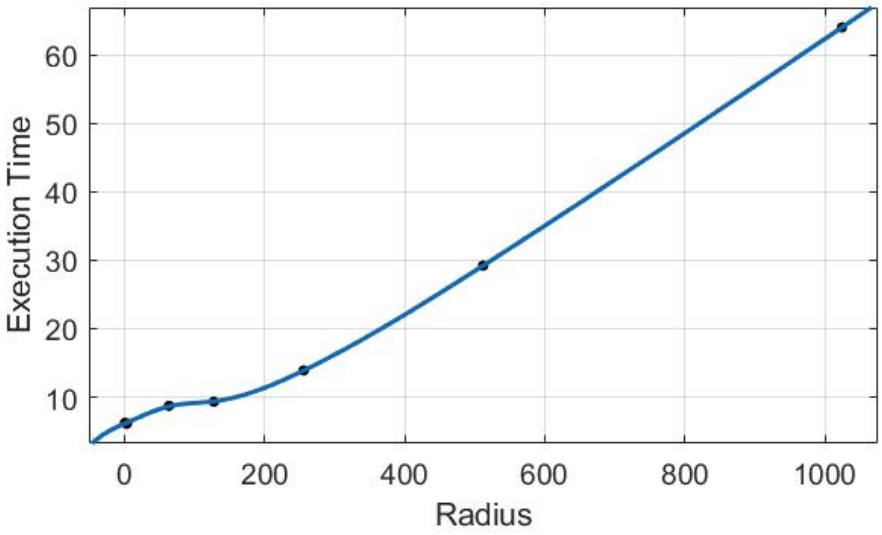
In Table 4, we try inputs with different square sizes. Difference in square size does not matter much and execution time barely changes.

Figure 4: Square Size



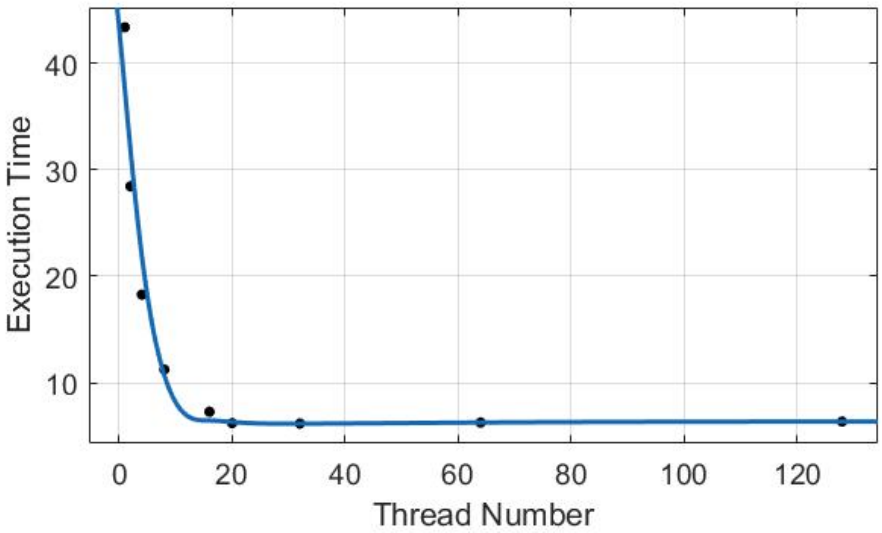
In Table 5, we try inputs with different particle radius. The execution time increases linearly with radius.

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



3.6 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.

3.7 Further Improvements

Grid Parallel Algorithm

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

The grid algorithm divides the square into 4×4 small blocks, 2×2 medium blocks and 1 large blocks respectively, just as the Fig 7 shows. Every time step, each particle has its own

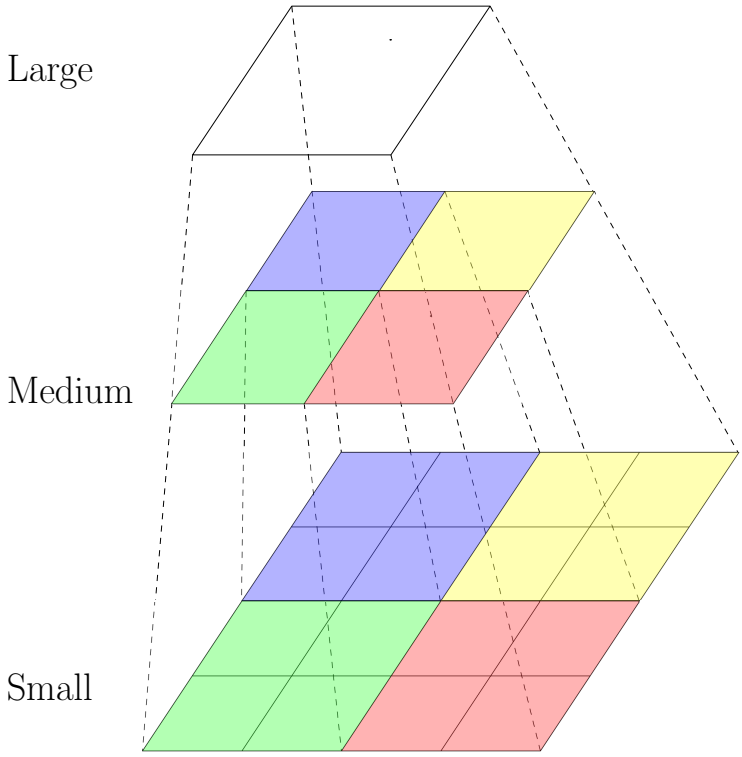


Figure 7: Multi-Level Grids

moving trajectory, which can be bounded by a box. We can then map these bounding boxes into different blocks of the grid. Of course, there will be bounding boxes cannot completely fit into any of 16 blocks. Then these bounding boxes will be tested to fit into a coarser level blocks (Medium level). If bounding boxes still cannot fit into medium level blocks, it will be set into the top level block.

After putting all trajectories into multi-level grids, collision detection will start from the bottom to the top. Particles located in small blocks need collision detection with particles

within that small blocks and particles located in overlapping medium and large blocks. And a similar detecting process will be done for particle located in medium and large blocks.

This algorithm can reduce the total times of collision detection and make the grid-level parallelism possible. The sequential version of this algorithm is in **collision_grid.c**.

We also try to parallelize this gird algorithm with dynamic schedule, the parallelized code is in **collision_grid_p.c**.

Here is the performance comparison on running **inputs.txt**:

Table 7: Comparision between Grid and Normal Algorithm

10000,20000,1	sequential	grid_sequential	p1_dynamic	grid_p_dynamic
first-time	44.92	31.36	6.24	6.92
second-time	42.4	30.84	6.25	6.9
third-time	41.94	30.89	6.25	6.95
shortest	41.94	30.84	6.24	6.9

It can be seen that grid algorithm performs better than the normal algorithm when running sequentially. And it only performs slightly worse than **collision_p1d.c** when being parallelized.

4 CUDA Parallel Program Design

4.1 Parallel Strategy Description

We also tried different strategies to parallelize our cuda code. These different approaches are demonstrated in **collision_cuda_basic.cu** and **collision_cuda_balance.cu**.

4.1.1 Basic Parallelism in CUDA

In **collision_cuda_basic.cu**, we used a simple method to parallel the code.

Basically, we use two kernel functions, *find_collisions()* and *update_particle()*. Firstly, for each step, we use kernel function *find_collisions()* to find all possible collisions, including particle-wall collisions and particle-particle collisions using similar approaches in the sequential code. For each threads in CUDA, it will be assigned a set of particles in a cyclic way, i.e. $\{i, i + T_{all}, i + 2T_{all}, i + 3T_{all}, \cdots\}$, where T_{all} is the number of total threads.

Then, use host function *find_real_collisions()* to sort out the real collisions based on different priorities. This function must wait until the first kernel function ends.

Thirdly, call kernel function *update_particle()* to update the velocities and locations of particles in collisions.

```
//collision_cuda_basic.cu Around Line 420
/* Call the kernel */
find_collisions<<<num_blocks, num_threads>>>(total_threads);
/* Barrier */
cudaDeviceSynchronize();
// find real collisions
qsort(colli_time, count, sizeof(Collision), compare);
find_real_collisions();
/* Call the kernel */
proc_colli<<<num_blocks, num_threads>>>(total_threads);
// update_particle<<<num_blocks, num_threads>>>(total_threads);
/* Barrier */
cudaDeviceSynchronize();
```

Within this basic parallelism, we mostly use **unified memory**(UM) to let important variables accessible by both host and device. UM can significantly help us CUDA beginners to manage such heterogeneous memory model.

```
//collision_cuda_basic.cu Around Line 406
cudaMallocManaged((void**)&particles, sizeof(particle_t) *
   host_n);
cudaMallocManaged((void**)&colli_mat,sizeof(int) * host_n);
cudaMallocManaged((void**)&colli_queue,sizeof(int) * host_n);
cudaMallocManaged((void**)&colli_time, sizeof(Collision) *
   host_n*(host_n+1)/2);
```

4.1.2 Optimization for CUDA version

In **collision_cuda_stream.cu**, we further improve the load balance of our basic code. There is a problems in our basic code, that is, each thread may have slightly different workload to do. E.g. To reduce the computation redundancy, a particle with small index i need to compare with particle $i+1, i+2, \cdots, n-1$.

To further utilize many-core feature of GPU, we devise a new parallelization approach to make every thread have similar workload in **collision_cuda_balance.cu**. In checking collision stage, there are two types of collisions, particle-wall(p-w) collision and particle-particle(p-p) collision. We have n times p-w collisions to check, and $\frac{n\times(n-1)}{2}$ times p-p collisions to check. So we divide the `find_collisions()` function in `collision_cuda_basic.cu` into `check_wall_colli()` and `check_pp_colli()`. For the `check_pp_colli()`, we can amortize

$\frac{n\times(n-1)}{2}$ times p-p collisions into T_{all} threads by ordering all collision pairs in an array (we use 2 integer array `pa_idx` and `pb_idx` to represent such array of pairs). Using a cyclic way to distribute data to different threads, we can ensure that each thread has almost the same workload. What's more, note that `check_wall_colli()` and `check_pp_colli()` are 2 independent function, we can achieve task parallelism here by using concurrent multi-stream execution feature provided by CUDA. We also divide the work for two streams when we try to update velocites and locations. One stream will update those particles not involved in any collision. The other will deal with those had collisions. We can do this because these two kinds of updating are independent and are easy to distinguish through our data structure `colli_mat`.

```
//collision_cuda_new.cu Around Line 468
/* Call the kernel */
check_wall_colli<<<num_blocks, num_threads, 0, stream1>>>(
   total_threads);
check_pp_colli<<<num_blocks, num_threads, 0, stream2>>>(
   total_threads);
/* Barrier */
cudaDeviceSynchronize();
```

```
//collision_cuda_new.cu Around Line 483
update_particle<<<num_blocks, num_threads, 0, stream1>>>(
   total_threads);
proc_collision<<<num_blocks, num_threads, 0, stream2>>>(
   total_threads);
```

4.2 Performance Comparison

In this part, we compare the performance of different parallel approaches of CUDA.

We use the same benchmark as in OpenMP 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 8: Comparison of different block and thread numbers between basic and balance version

10000,20000,1	first time		second time		third time		shortest time	
block,thread=(2160,1024)	3.325	0.96	3.19	0.848	3.185	0.882	3.185	0.848
block,thread=(1080,1024)	3.183	0.904	3.152	0.845	3.157	0.837	3.152	0.837
block,thread=(720,1024)	3.208	0.973	3.186	0.84	3.218	0.849	3.186	0.84
block,thread=(360,1024)	3.203	0.883	3.156	0.839	3.182	0.87	3.156	0.839
block,thread=(180,1024)	3.232	0.941	3.163	0.888	3.192	0.888	3.163	0.888
block,thread=(720,2048)	0.223	0.592	0.216	0.54	0.212	0.562	0.212	0.54
block,thread=(720,512)	3.235	0.955	3.175	0.846	3.183	0.855	3.175	0.846
block,thread=(720,256)	1.913	0.899	1.829	0.866	1.864	0.85	1.829	0.85
block,thread=(720,64)	0.978	0.983	0.92	0.903	0.888	0.872	0.888	0.872
block,thread=(720,32)	0.914	0.953	0.804	0.895	0.807	0.873	0.804	0.873
block,thread=(72,32)	1.211	1.616	1.167	1.56	1.162	1.552	1.162	1.552

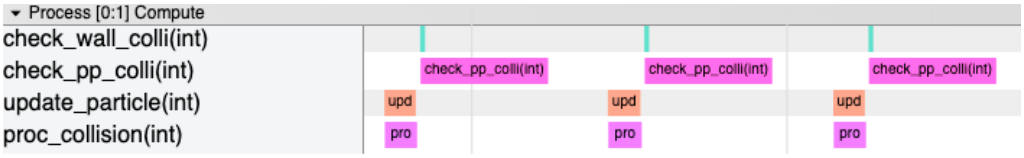


Figure 8: Balance Execution Visualization

Analysis. From Table 8 we can see that the balance version generally outperform the basic version in many cases. When #block and #thread go up, balance version can still get improved, while the basic version will suffer performance loss with more blocks and more threads.

To explain such results, we notice that the basic version has unbalanced workload for different thread when detecting collision. However, GPU executes threads in a warp method, there are 1 or 2 warps in each SM running concurrently. If we increase #threads, then more unbalanced jobs will be assigned to one SM in GPU hardware, resulting to more execution time in this SM, and further resulting total performance loss.

It should be pointed out that when (block,thread)=(720,2058), the execution time is extremely short. But the result is not accurate under this input. The number of wall collisions and particle-particle collisions are **zero** under this case. This is because the number of threads **can not exceed 1024**.

Another interesting point is that, basic version outperforms the balance when we use block,thread = (720,32) settings. Using `nvprof`, we found that basic version has less CPU Page Faults(807), compared with CPU Page Faults(1303) in balance version. To reduce potential

CPU page faults in balance version, we also used Asynchronous Memory Prefetching technique for Unified Memory, but the CPU page faults still larger than that of basic version. This will remain a question for future improvement with further exploit of CUDA.

4.3 Special Considerations

We also encountered with many problems while doing parallelism on cuda. Here are some non-trivial points that we think are worth to mention:

- Problems emerged when zero is expressed in double precision. When comparing the results, we found that sometimes zero is 0.0000000000000001 other than 0 stored. This causes huge problems as our sorting algorithm doesn't work properly in this case. So we define a small value $1e-10$. When the difference between a value and zero is smaller than it, we take it as zero.

- In order to synchronize between threads, we use *atomicAdd()* to synchronize adding on the same variable.

- When thread number is less than particle or real collision numbers, if we still use the 'one particle one thread' code version, it can't even finish the job. So we give a chunk to each thread in this case.

4.4 Performance with Different Inputs

In this part, we tested different inputs (particle number, square size and particle radius) on **collision_cuda_balance.cu**.

Based on our test of different number of threads and blocks in Table 8, we use $block, thread = (1080, 1024)$ for the following tests.

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

We run our codes on cluster node **xgpe4** with Titan RTX(CUDA10.1). And we use randomized input for testing.

Table 9: Inputs with different numbers of particles

particles, square size, radius	first-time	second-time	third-time	shortest
5000,20000,1	0.736	0.427	0.394	0.394
10000,20000,1	0.848	0.847	0.847	0.847
20000,20000,1	2.614	2.583	2.802	2.583
30000,20000,1	5.492	6.754	5.647	5.492
40000,20000,1	9.35	9.42	9.802	9.35
50000,20000,1	segmentation fault	SE	SE	SE

Table 10: Inputs with different side length of the square

particles, square size, radius	first-time	second-time	third-time	shortest
10000,5000,1	1.155	0.918	0.916	0.916
10000,10000,1	0.906	0.872	0.885	0.872
10000,20000,1	0.849	0.844	0.829	0.829
10000,100000,1	0.821	0.82	0.817	0.817
10000,1000000,1	0.817	0.824	0.824	0.817
10000,10000000,1	0.821	0.826	0.825	0.821

Table 11: Inputs with different particle radius

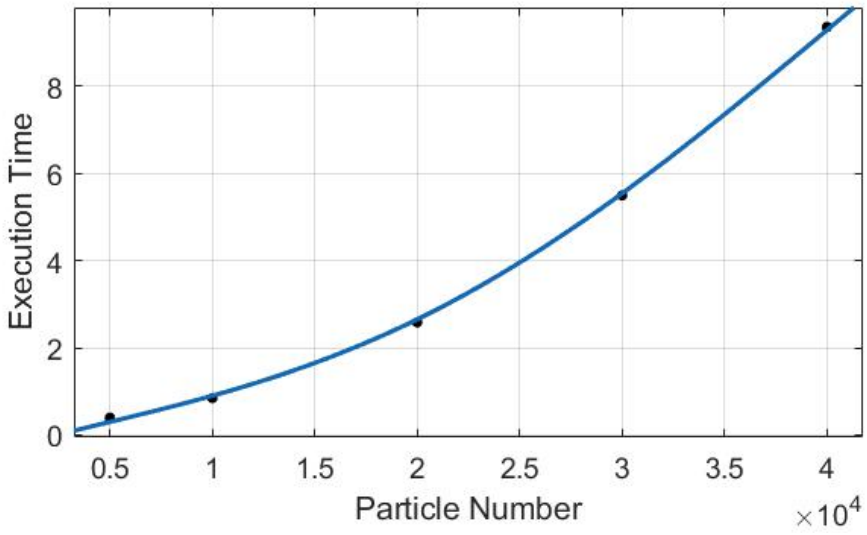
particles, square size, radius	first-time	second-time	third-time	shortest
10000,20000,1	1.157	0.909	0.846	0.846
10000,20000,4	0.915	0.906	0.903	0.903
10000,20000,64	2.447	2.582	2.474	2.447
10000,20000,128,	4.225	4.244	4.205	4.205
10000,20000,256	8.155	8.154	8.055	8.055
10000,20000,512	17.389	17.564	17.091	17.091
10000,20000,1024	39.953	42.265	40.009	39.953

4.5 Observations

We can gain similar conclusions from the tables above as the OpenMp parallelism.

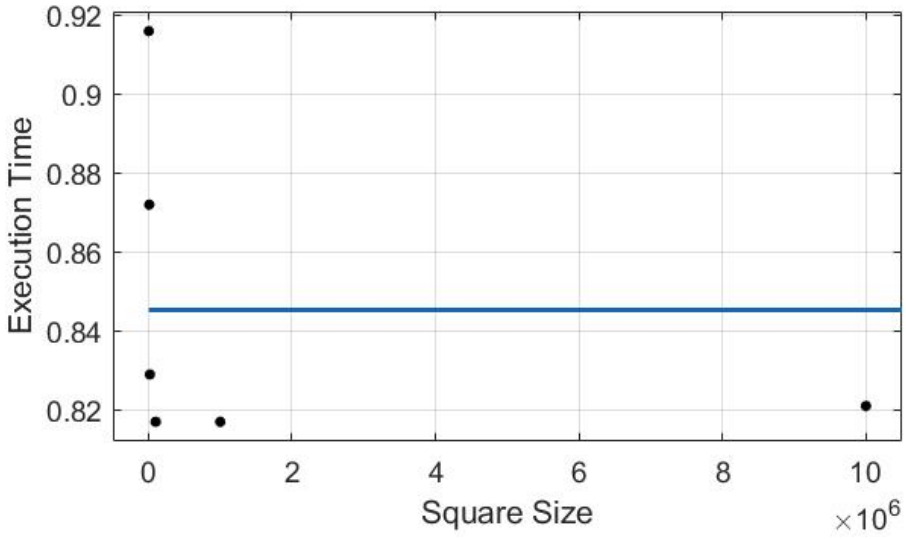
In Table 9, we try inputs with different numbers of particles on cuda code. With the increase of particle number, the execution time increases exponentially. When the particle number reaches 5,000 segmentation fault happens.

Figure 9: Particle Number



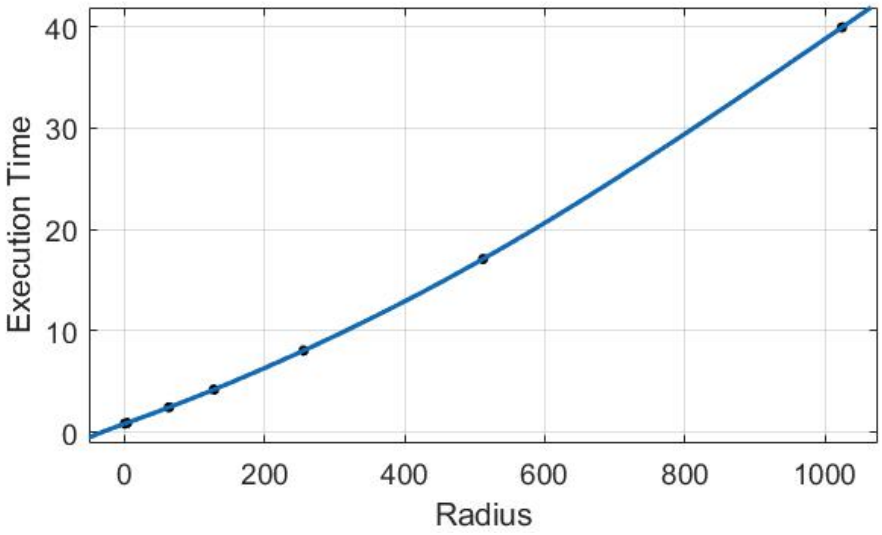
In Table 10, we try inputs with different square sizes on cuda code. Difference in square size also does not matter much and execution time barely changes.

Figure 10: Square Size



In Table 11, we try inputs with different particle radius on cuda code. The execution time increases linearly with radius.

Figure 11: Particle Radius



4.6 Reproducing Results

(a.) To reproduce results in Table 8, run **run_cuda.sh** on cluster node xgpe4. The instructions required are all contained in bash.

(b.) To reproduce results in Table 9-11, run bash **test_l_size.sh**, **test_r_size.sh** and **test_p_size.sh** on cluster node xgpe4.

5 OpenMPI Parallel Program Design

In the previous two sections, we use both OpenMP and Cuda to do parallelism, which use shared memory. In this section, we are going to use OpenMPI and do parallelism using distrubuted-memory system.

5.1 Parallel Strategy Description

In OpenMPI realization, we use master-slave pattern to do parallelism. Besides, we use blocking communication.

Firstly, master generates the way to distribute according to inputs. This is done by function **gen_comm_mat**. We distribute the collision matrix (#particles$\times$ #particles size) into #slaves $\times$ #slaves chunks. Then we divide theses chunks to slaves. Our way of dividing theses chunks is using the diagonal method. The following table is *job_mat* and *send_mat* under the case of dividing 5×5 chunks to 5 slaves.

Table 12: job_mat

0	0	4	2	4
Na	1	1	0	3
Na	Na	2	2	1
Na	Na	Na	3	3
Na	Na	Na	Na	4

Table 13: send_mat

0	1	3	Na	Na	3
1	2	4	Na	Na	3
2	3	0	Na	Na	3
3	4	1	Na	Na	3
4	0	2	Na	Na	3

Between master and slave, master firstly transfer particle information contained in the specific chunks (derived from above) to specific slaves. This diagonal method can **reduce communication overheads** because each slave only needs information of parts of particles .

Then, for each salve, it does collilsion detection and send the result back to master. Master then gather all collision information and sort out the real collisions happened.

After that, master transfer collision information that require updating particles involved to slaves and let slaves update. At last, slaves send the updated particles back to master.

This is what happens in each step.

Implementation assumptions are similar to OpenMP and Cuda.

The following are some important data structure:

Table 14: Data Structure in **collision_mpi.c**

chunk_size	Number of particles contained in each chunk
last_chunk_size	Number of particles contained in the last chunk
function gen_comm_mat	Ways of distributing chunks
send_mat	Store the specific chunks that are involved in detection by a slave
job_mat	Store the specific slave to detect collision within chunk (i,j)
num_chunk_cmp	The number of chunks assigned to a slave
major_chunk	For a slave, this is the chunk at the diagonal
extra_send	particles beyond the major chunk that are updated by a slave

5.2 Performance Comparison

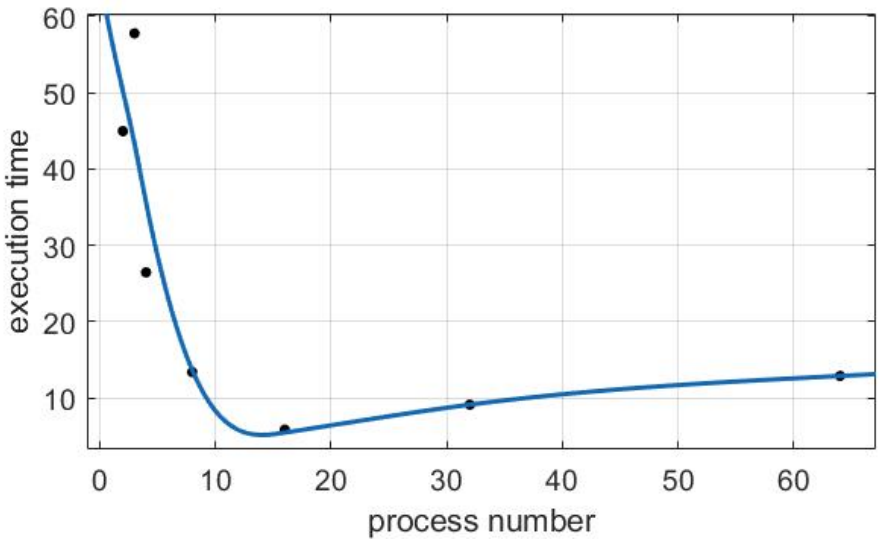
In this part, we compare the performances of our OpenMPI program using different inputs.

5.2.1 Different Number of Processes

As we only change process number in this subsubsection, we use the same input size as in OpenMP and Cuda with **inputs.txt** (it specifies the initial status of every particle).

The following is the execution time for different process number (run on one machine *soctf-pdc-007*):

10000,20000,1	first-time	second-time	third-time	minimum
2 processes	44.964	45.232	45.201	44.964
3 processes	58.347	57.775	58.185	57.775
4 processes	26.976	26.605	26.48	26.48
8 processes	14.295	13.681	13.419	13.419
16 processes	5.884	6.113	5.978	5.884
32 processes	9.151	9.281	10.42	9.151
64 processes	14.422	13.105	12.904	12.904



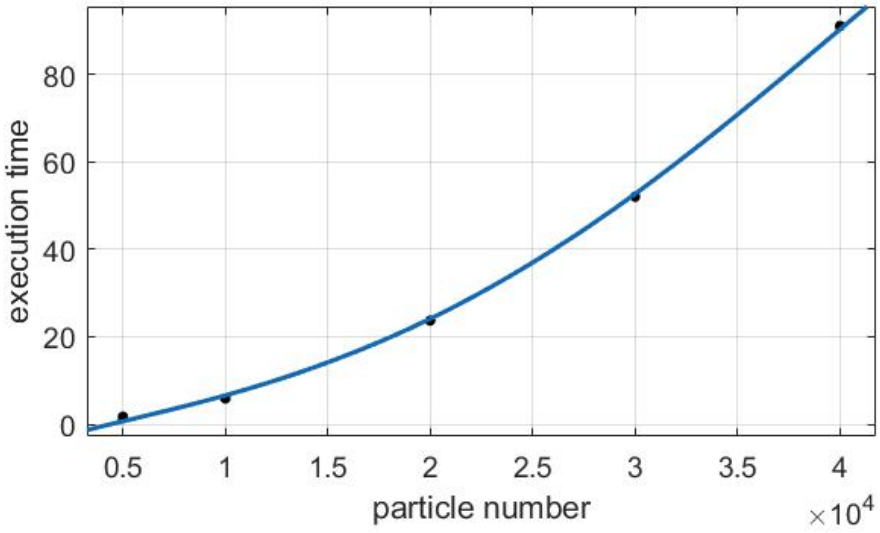
Analysis. As shown in the figure and table, the algorithm is the most efficient when there are 16 processes. When process number exceeds 16, the execution time starts to increase due to communication overhead.

5.2.2 Different Number of Particles

According to the last section, we can see that our program behaves best when there are 16 processes. We are going to use 16 processes to run in this subsubsection.

The following is the execution time for different particle number (run on one machine *soctf-pdc-007*):

particles, square size, radius	first-time	second-time	third-time	shortest
5000,20000,1	1.726	1.697	1.701	1.697
10000,20000,1	5.884	6.113	5.978	5.884
20000,20000,1	24.238	23.749	23.688	23.688
30000,20000,1	52.039	62.235	60.964	52.039
40000,20000,1	91.078	110.075	91.357	91.078
50000,20000,1	segmentation fault	SF	SF	SF



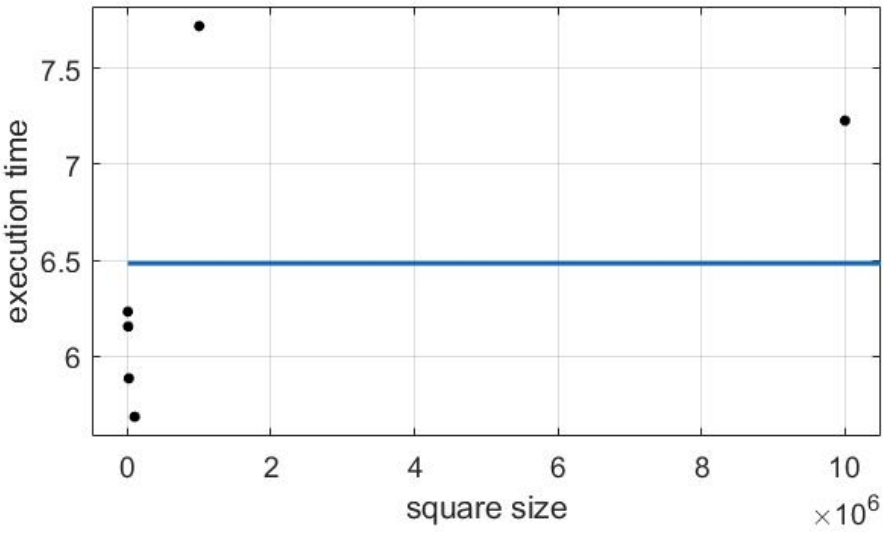
Analysis. As shown in the figure and table, execution time increases exponentially with particle number under OpenMPI, which is similar to OpenMP and Cuda.

5.2.3 Different Square Size

We will still use 16 processes to test.

The following is the execution time for different square sizes (run on one machine *soctf-pdc-007*):

particles, square size, radius	first-time	second-time	third-time	shortest
10000,5000,1	7.371	6.232	6.253	6.232
10000,10000,1	6.189	6.903	6.154	6.154
10000,20000,1	5.884	6.113	5.978	5.884
10000,100000,1	5.684	5.749	5.843	5.684
10000,1000000,1	7.72	7.785	7.768	7.72
10000,10000000,1	7.227	7.464	7.628	7.227



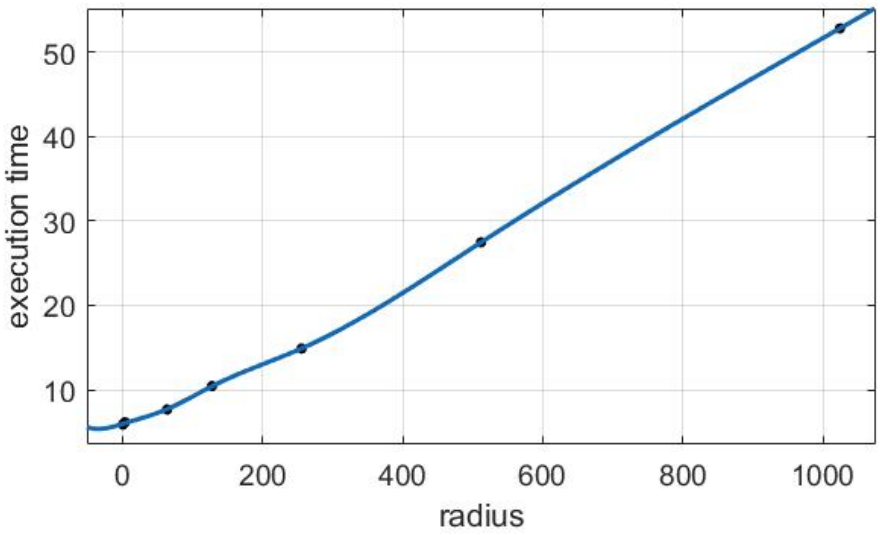
Analysis. As shown in the figure and table, execution time barely changes with square size under OpenMPI, which is similar to OpenMP and Cuda.

5.2.4 Different Radius

We still use 16 processes to do tests.

The following is the execution time for different radius (run on one machine *soctf-pdc-007*):

particles, square size, radius	first-time	second-time	third-time	shortest
10000,20000,1	5.884	6.113	5.978	5.884
10000,20000,4	6.241	6.162	6.384	6.162
10000,20000,64	7.666	9.344	8.315	7.666
10000,20000,128	10.775	10.693	10.444	10.444
10000,20000,256	14.891	16.411	15.518	14.891
10000,20000,512	27.462	28.804	28.856	27.462
10000,20000,1024	59.181	56.115	52.795	52.795



Analysis. As shown in the figure and table, execution time increases linearly with radius under OpenMPI, which is similar to OpenMP and Cuda.

5.3 Reproducing Results

To reproduce different execution time varying with process number, run *run_mpi.sh*.
To reproduce different execution time with different particle numbers, run *run_mpi_p.sh*.
To reproduce different execution time with different square size, run *run_mpi_l.sh*.
To reproduce different execution time with different radius, run *run_mpi_r.sh*.
(The above tests are run on machine soctf-pdc-007)

5.4 Future Improvements

Although our code looks well, there are still some points that we can improve on:

1. We use blocking communication throughout our code. Is it possible to use non-blocking communication somewhere to improve efficiency?

2. Also transfer data between slaves to reduce the burden of master.

3. Use mergesort instead of quicksort when sorting collisions to increase efficiency.

6 Comparison between OpenMP, Cuda and OpenMPI

Table 15: Comparison of different particle numbers on OpenMP, Cuda and OpenMPI

particle number	OpenMP	Cuda	OpenMPI
5000,20000,1	1.7	0.394	1.697
10000,20000,1	6.23	0.847	5.884
20000,20000,1	23.82	2.583	23.688
30000,20000,1	52.6	5.492	52.039
40000,20000,1	91.76	9.35	91.078
50000,20000,1	segmentation fault	segmentation fault	segmentation fault

Table 16: Comparison of different square size on OpenMP, Cuda and OpenMPI

square size	OpenMP	Cuda	OpenMPI
10000,5000,1	6.23	0.916	6.232
10000,10000,1	6.2	0.872	6.154
10000,20000,1	6.23	0.829	5.884
10000,100000,1	6.19	0.817	5.684
10000,1000000,1	6.3	0.817	7.72
10000,10000000,1	6.29	0.821	7.227

Table 17: Comparison of different radius on OpenMP, Cuda and OpenMPI

radius	OpenMP	Cuda	OpenMPI
10000,20000,1	6.23	0.846	5.884
10000,20000,4	6.13	0.903	6.162
10000,20000,64	8.72	2.447	7.666
10000,20000,128	9.33	4.205	10.444
10000,20000,256	13.89	8.055	14.891
10000,20000,512	29.2	17.091	27.462
10000,20000,1024	64.01	39.953	52.795

Observation. It can be seen that overall, Cuda is the fasted implementation method. It is obviously faster than other two methods in either of the three comparisons. In detail, Cuda is around **five times faster** than other methods.

Although OpenMP and OpenMPI use different memory system, their running time are very close.

These three methods show similar trends varying particle number, square size and radius.

7 Conclusions

In all two Assignments, we have managed to finish the sequential simulation of particle collision and use OpenMP, CUDA and OpenMPI to parallelize it.

Our assumptions and algorithms are described briefly for our sequential and parallel code for OpenMP, CUDA and OpenMPI. For both OpenMP/CUDA, we firstly try different strategies to parallelize and find the best one through testing.

After that, we made comparison between different parallel strategies for OpenMP/CUDA. And use different inputs, that is, particle number, square size, radius and thread number to test OpenMP, CUDA and OpenMPI codes.

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

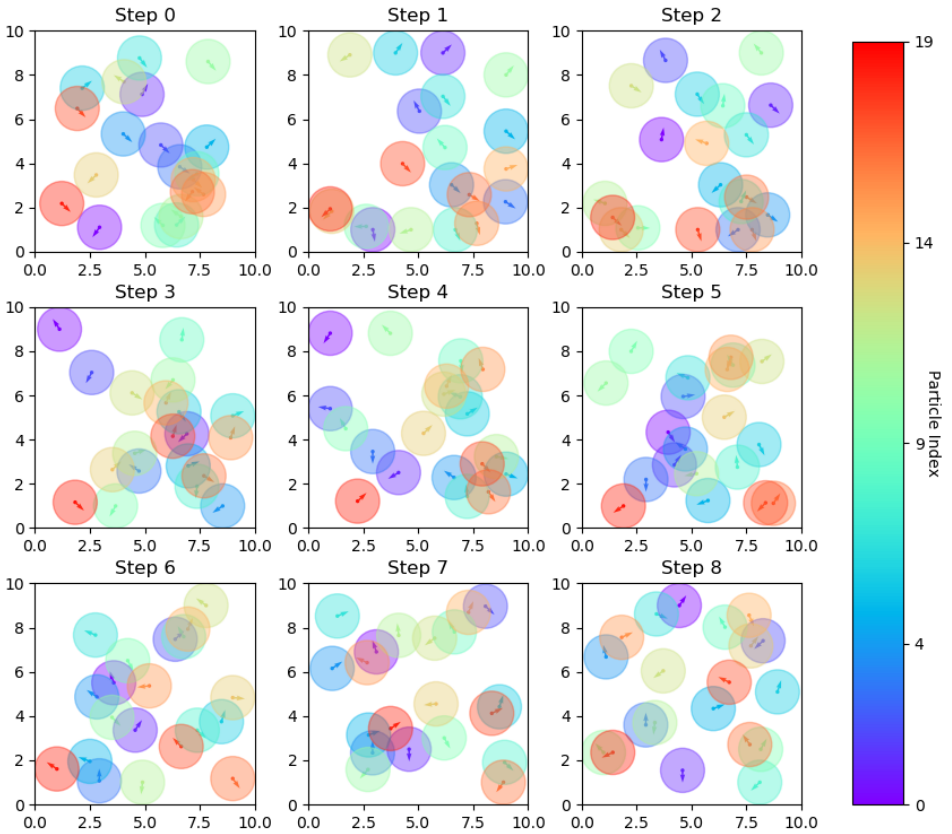


Figure 12: Simulation Step-by-step Visualization

By comparing different type of implementation of collision simulation, we see the great advantage brought by different kinds of parallel methods. We should make use of parallelism for future research and work.

We have finally completed all particle simulation tasks. Thanks a lot to our TAs, Prof. Cristina and classmates for help!