The Chinese University of Hong Kong, Shenzhen

CSC4005

Distributed and Parallel Computing

Assignment3 N-body Simulation

Author:
Shi Wenlan

Student Number:
119010265

November 15, 2022

Contents

1 Introduction

In this assignment, I programmed 6 versions of programs to simulate an astronomical N-body system in two-dimensions (inlcuding bonus):

1. A sequential version
2. An OpenMP version
3. A Pthread versoin
4. A CUDA version
5. A MPI version
6. A MPI + OpenMP version (bonus)

Introduction of 2D N-body Simulation: The goal of the program is to find positions and movements of bodies in space that are subject to gravitational forces from other bodies using Newtonian law of physics. The bodies are initially at rest. Their initial positions and masses are to be selected randomly.

The gravitational force between two bodies of masses m_a and m_b can be calculated by $F = \frac{Gm_am_b}{r^2}$ where G is gravitational constant and $r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$ is the distance between bodies. The forces are resolved in two directions as following:

$$F_x = \frac{Gm_am_b(x_b - x_a)}{r^3}$$

$$F_y = \frac{Gm_am_b(y_b - y_a)}{r^3}$$

where bodies of mass m_a and m_b have coordinates (x_a, y_a) and (x_b, y_b).

Subject to forces, a body will accelerate according to Newton's second law:

$$a = \frac{F}{m}$$

where m is the mass if the body, F is the force it experiences, and a is the resultant acceleration.

Let the time interval be Δt. Then, for a particular body of mass m, the new velocity is

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where v^{t+1} is the velocity of body at time $t+1$ and v^t is the velocity of body at time t.

If a body is moving at a velocity v over the time interval Δt, its new position is $x^{t+1} = x^t + v\Delta t$ where x^t is its position at time t.

In addition, the program considers the collisions between bodies and that bounces on borders. All such interactions are calculated as perfect elastic collisions. If a body of mass m_1 and velocity v_1 collides with another body of mass m_2 and velocity v_2, then the new velocity v_1' of the body of mass m_1 is:

$$v_1' = \frac{(m_1 - m_2)v_1 + 2m_2v_2}{m_1 + m_2}$$

And if a body of x-axial velocity v_x collides with left or right border, its new x-axial velocity will be $-v_x$. Similarly, if a body of y-axial velocity v_y collides with top or bottom border, its new y-axial velocity will be $-v_y$

Once bodies move to new position, the forces change and the computation has to be repeated.

My Preset of 2D N-body Simulation: The physics variables are declared in src/headers/physics˙cuda.h (for CUDA version) and src/headers/physics˙cuda.h (for other versions).

```
1  int bound_x = 4000;
2  int bound_y = 4000;
3  int max_mass = 40000000;
4  double error = 4.0f;
5  double dt = 0.0001f;
6  double gravity_const = 1000.0f;
7  double radius1 = 1000.0f;
8  double radius2 = 20.0f;
```

Figure 1: Physics Variables

bound_x is the upper bound of X axis. x is between [0,*bound_x*].

bound_y is the upper bound of Y axis. y is between [0,*bound_y*].

max_mass is the maximum mass of a particle.

gravity_const is the gravity constant G.

dt is the time span between two iterations.

error is a small number used to avoid $DivisionByZero$ error.

$radius2$ is the detection radius of collision between bodies.

$radius1$ is the range radius where error should be calculated into accelerations.

In order to see the iteration effect faster (bodies show a visible movement trend with fewer iterations), I increased $gravity\_const$ to 1000.

This will cause the body to miss the collision detection range due to excessive speed when colliding, so in order to alleviate this situation, $radius2$ is set to be larger than the body's radius and increased to 20.

When the block collides, it will have excessive acceleration because the distance is very close. In order to limit this acceleration, the calculation of the acceleration will add an $error$ to distance r when the r is less than $radius1$.

But because this program is a discrete simulation rather than a continuous one, the bodies might collide faster and faster, and even move from below the lower bound to above the upper bound in one iteration. The above parameter settings can alleviate but not guarantee to eliminate this influence. In order to ensure the visual effect (the movement trajectory of the block is visible), after calculating the speed in each iteration of the program, the absolute value of the sub-velocities in the x-axial and y-axial would be limited at most 20000000.

2 Method

2.1 Flow charts

The computation part of N-body Simulation can be divided into 2 parts as following:

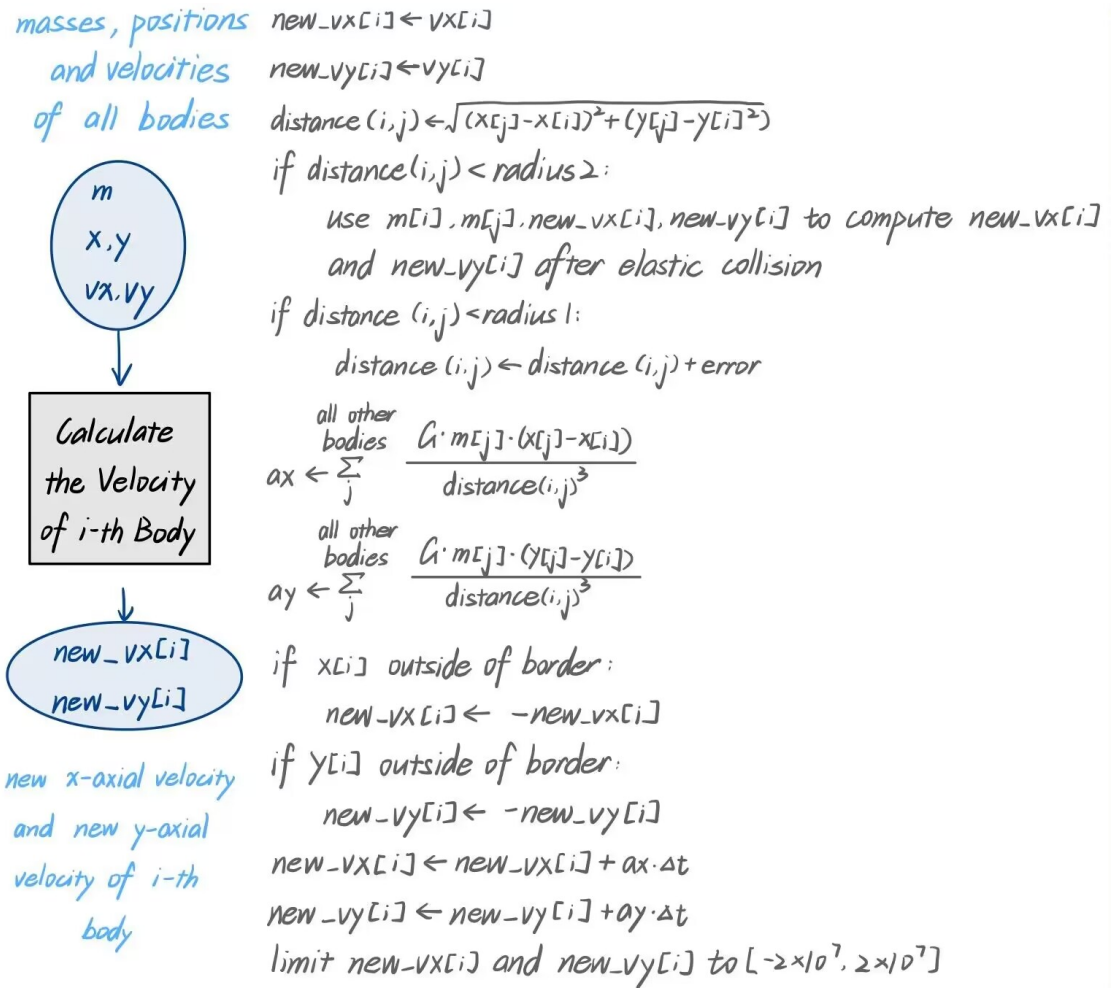

Figure 2: Computation Part 1

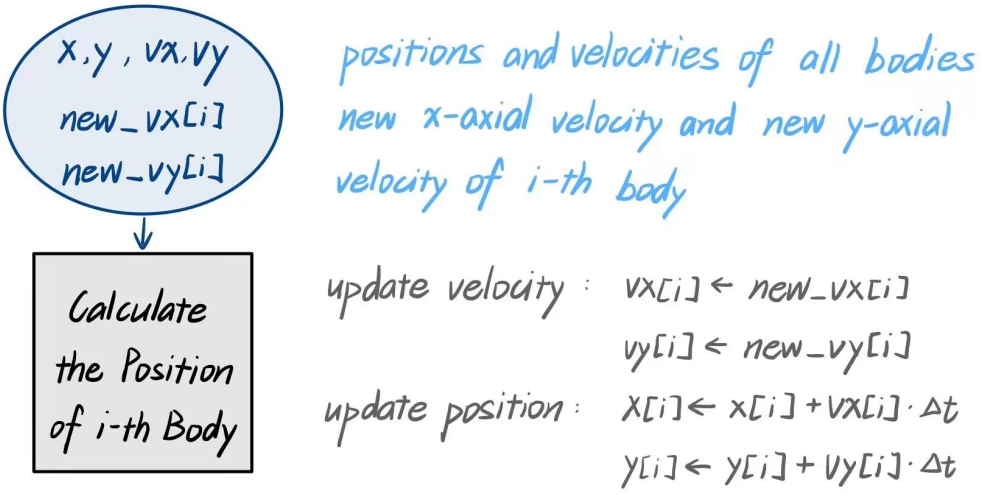

Figure 3: Computation Part 2

In order to avoid data dependency, two arrays, *new_vx* and *new_vy*, are introduced to temporarily store the new velocities. Although the sequential code doesn't need this temporary storage logically, in order to ensure the fairness of the experiment, this part of code remains.

In addition, in order to avoid data dependence, the velocity of each body after collision is calculated independently, which will cause certain errors due to the inconsistent order of calculating collisions. For example: bodies 1, 2, and 3 collide at the same time and are calculated independently in three different processes/threads. Ball 1 collides with 2 first and then 3 during calculation. Ball 2 collides with 1 first and then 3 during calculation. Ball 3 collides with 1 first and then 2 during calculation. The error occurs because when ball 1 and ball 2 collide with ball 3, they are not in the initial motion state (they have collide with each other), which is inconsistent with the situation when ball 3 is calculated.

The general idea flow charts of my 6 versions of program are as following:

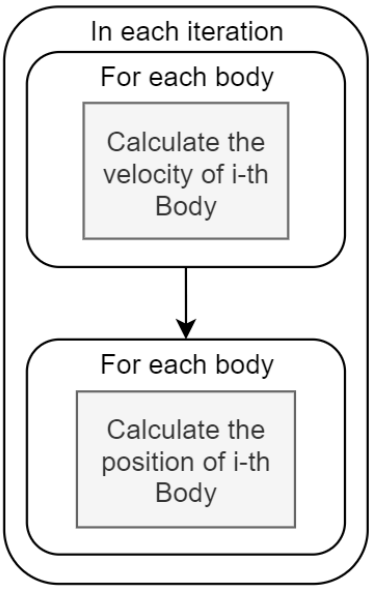

Figure 4: General Flow Chart of Sequential Version

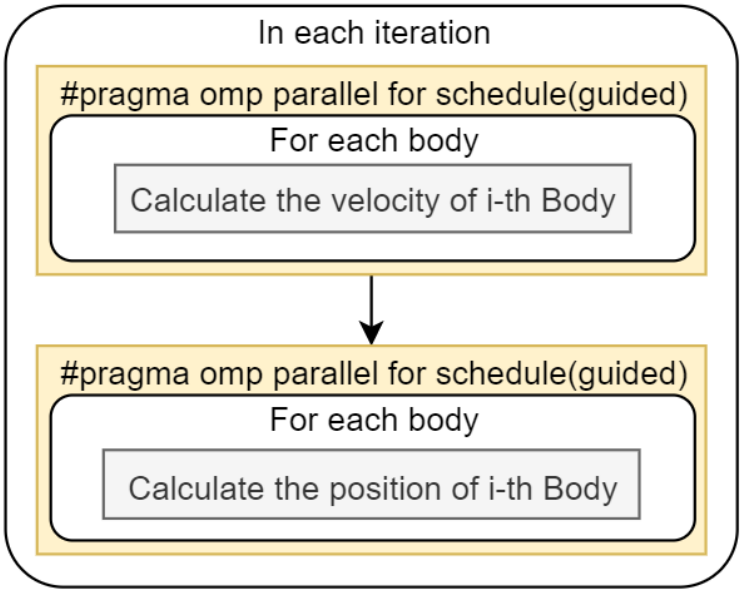

Figure 5: General Flow Chart of OpenMP Version

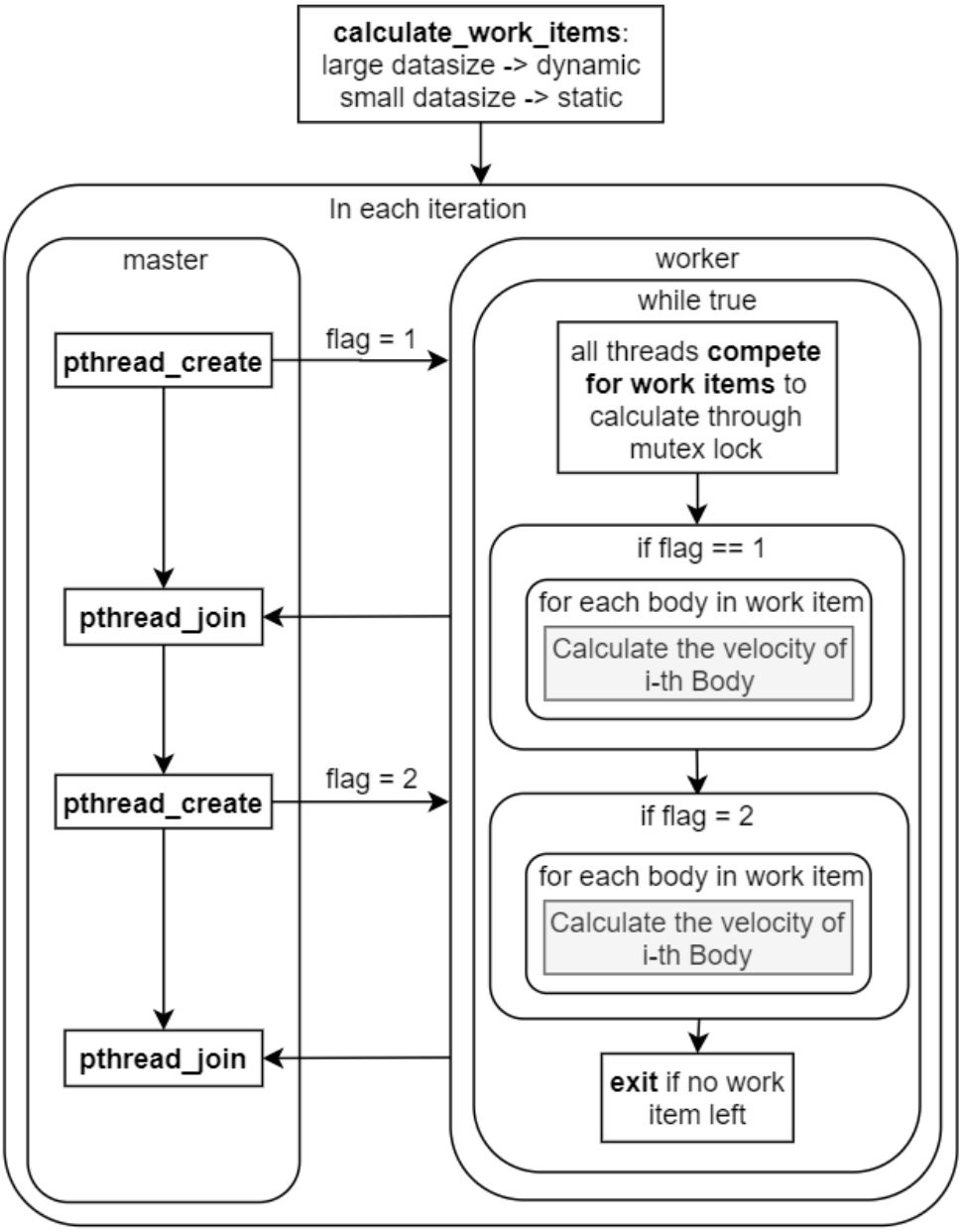

Figure 6: General Flow Chart of Pthread Version

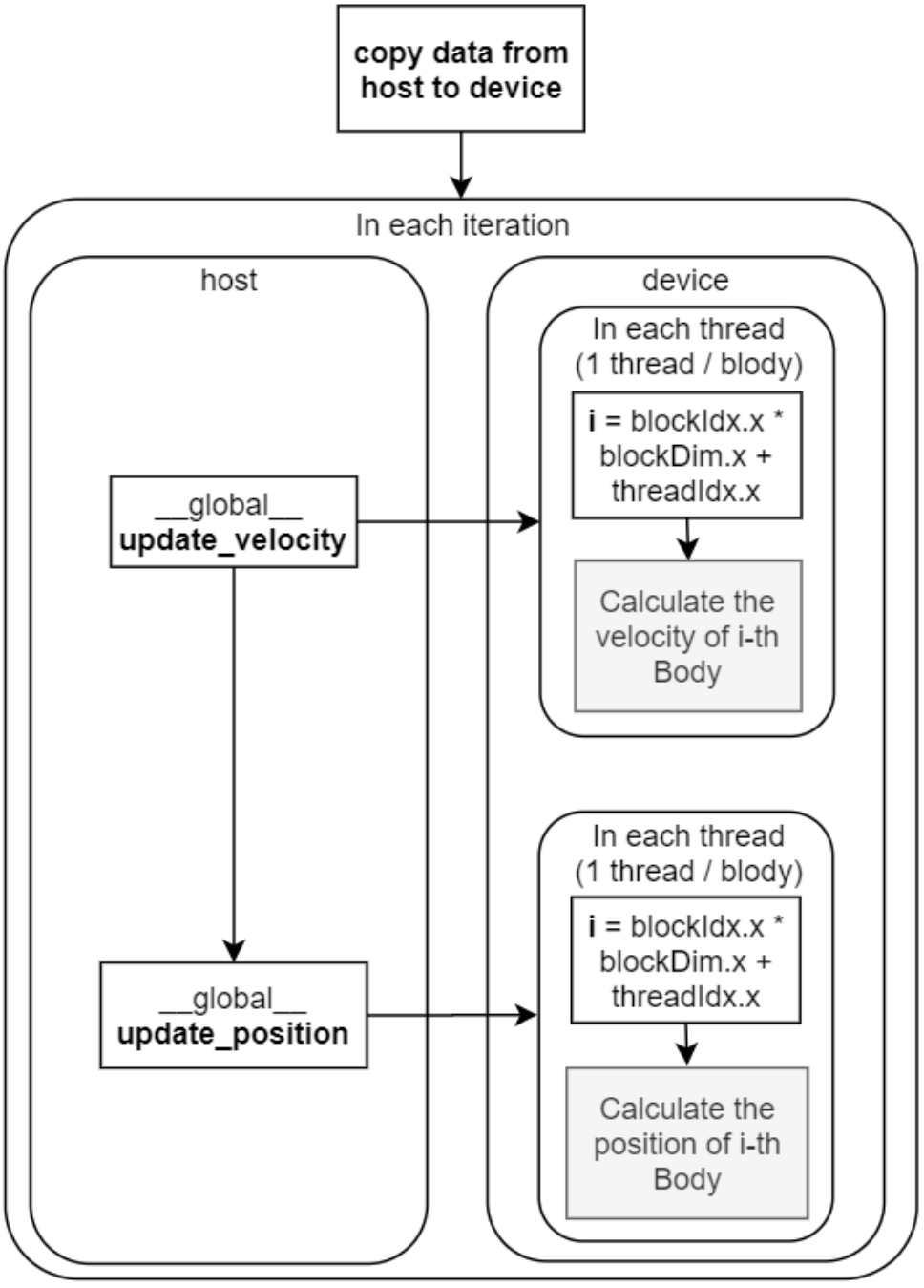

Figure 7: General Flow Chart of CUDA Version

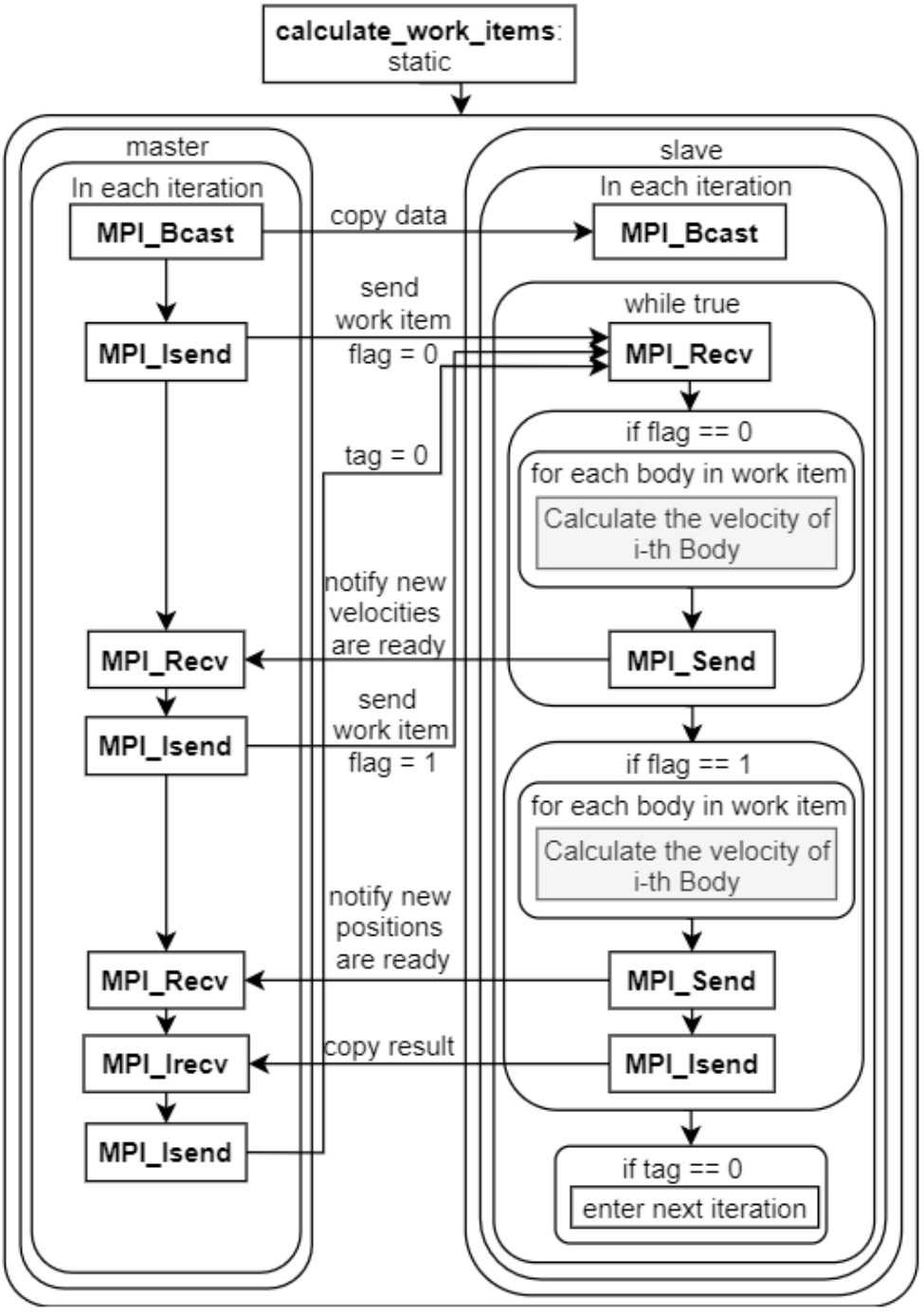

Figure 8: General Flow Chart of MPI Version

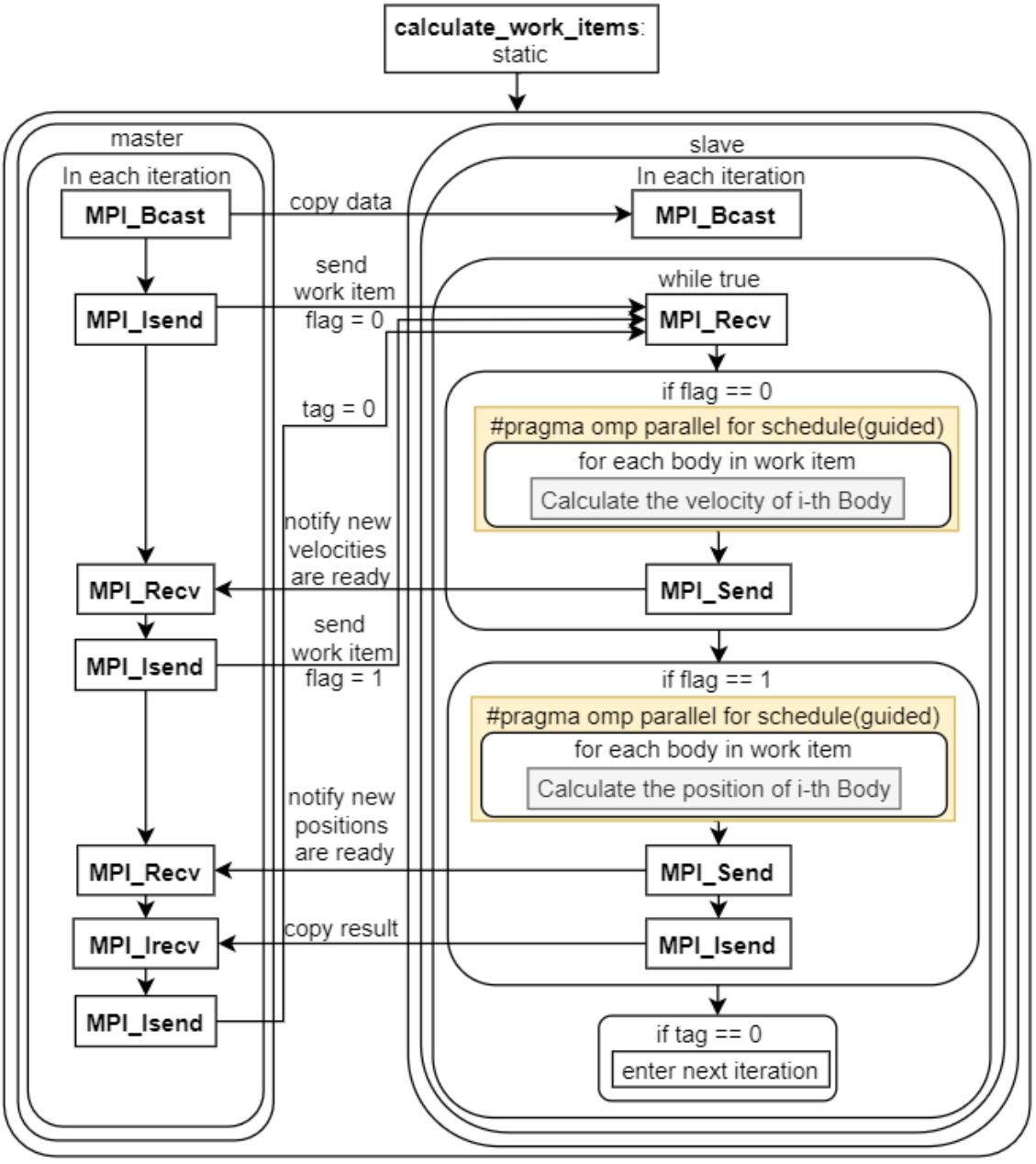

Figure 9: General Flow Chart of MPI+OpenMP Version

2.2 Extra explainations

The following are some extra explainations:

Why not use Barnes-Hut Algorithm: Because of the need of collision detection. The tree constructed by Barnes-Hut Algorithm can only accelerate the calculation of gravity, but cannot accelerate collision detection. And the calculation of gravity can be calculated by the way during collision detection. It seems that the advantage of the Barnes-Hut Algorithm has very little advantage under this topic setting.

How to avoid data race: The data race will happen if two or more threads/processes are fetch the same data, while at least one of them are trying to write the data. My solution is to let all the threads/processes read on shared original vx and vy, and write on their own specific block of new_vx and new_vy. No thread/process will read the block of new_vx and new_vy that another thread/process is responsible for.

On the other hand, this is also the reason why the updapation of vx and vy is in the calculation part 2: vx and vy should not be written before all new_vx and new_vy are calculated.

Why to use schedule(guided) in OpenMP version: OpenMP has three types of scheduling: static, dynamic, and guided. The static version would lead to a waste of resources when some threads end much faster than other. The dynamic version allocates one iteration to each thread at one time by default, which would cause some overhead on allocation. But if I set a minimum number of iterations for a single allocation, it will end up leaving some idle threads when the number of available threads is too large with respect to the number of iterations. However, the guided version uses a heuristic scheduling algorithm that starts allocating larger chunks and then gradually gets smaller, which perform the best.

Why to use static work item allocation in MPI version: If the work items are allocated dynamically, the child process needs to transmit new velocities back to the main process after calculating them, and then the main process needs to pass an additional copy of the new velocity data when allocating tasks to calculate the new position. In my experiment, I found that the overhead of dynamically assigning work item data transmission is relatively large, even greater than the calculation time. This is because in each iteration, the main process needs to send all x, y, vx, vy to each child process, and all processes add up to receive a copy of new_vx, new_vy, x, y, vx, vy and send a new_vx, new_vy. And when the main process receives the results, it will only receive notifications from other processes after it has received all the results from a child process. At this time, other processes that complete the work item are blocked.

To alleviate the overhead, I used static work item allocation. Because in this way, each child process would be allocated the same block of data to calculate in two calculation parts. That is to say, the new_vx, new_vy can be child processes' local vairable, and do not need to be transmitted. In addition, I used non-blocking send and receive to send work items and receive results.

How to add OpenMP to MPI: Add "$\#pragma\ omp\ parallel\ for\ schedule(guided)$" to the for loop in computation parts in child processes. Because applying for more than 40 cores to HPC at a time will lead to an endless queue, I hard coded the number of openmp threads that each process can use to 2.

The memory of each MPI process is independent with each other, and is shared by its own OpenMP threads.

3 Result

3.1 How to run my code

If you want to reproduce the experiment in Subsection *Output screenshots*: Open source code folder in the terminal, garantee that there are enough processors available, then type in the following instruction to run the script.

```
1 /bin/bash show.sh
```

Note that if enough resources are not prepared, the program will not report an error, but logically different processes and threads will take turns to use the CPU to simulate multi-threading and multi-process, and the context exchange caused by this will generate a large overhead and make the program run extremely slow. Anyway, the program results are still correct.

If you want to reproduce the experiment in Subsection *Raw data*: Open source code folder in the terminal, garantee that there are enough processors available, then type in the following instruction to run the script.

```
1 /bin/bash slurm/cuda.sh
2 /bin/bash slurm/mpi_openmp.sh
3 /bin/bash slurm/mpi.sh
4 /bin/bash slurm/openmp.sh
5 /bin/bash slurm/pthread.sh
6 /bin/bash slurm/seq.sh
```

Note that the number of openmp threads that each process can use in MPI+OpenMP version has been hard coded to 2. To get the data of singe computational thread-/process, the source should be modified.

If you want to run the code more flexibly: Please refer to README.md for more detailed information.

3.2 Output screenshots

The expected output of the program is: the bodies gather to the middle first because of gravity, and then collide and spread all over the screen.

Here are some screenshots of GUI version compile and run:

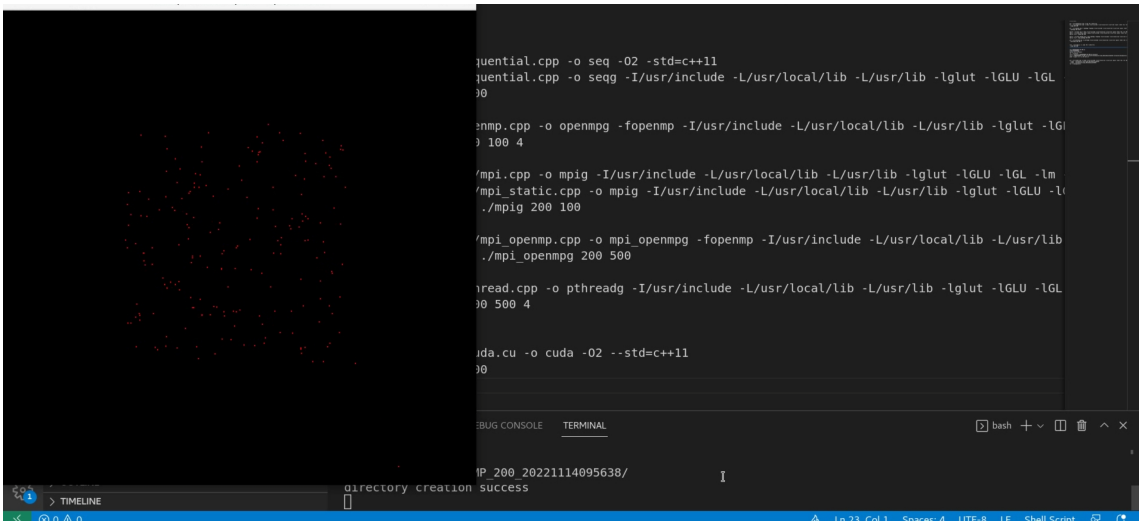

Figure 10: Screenshot 1

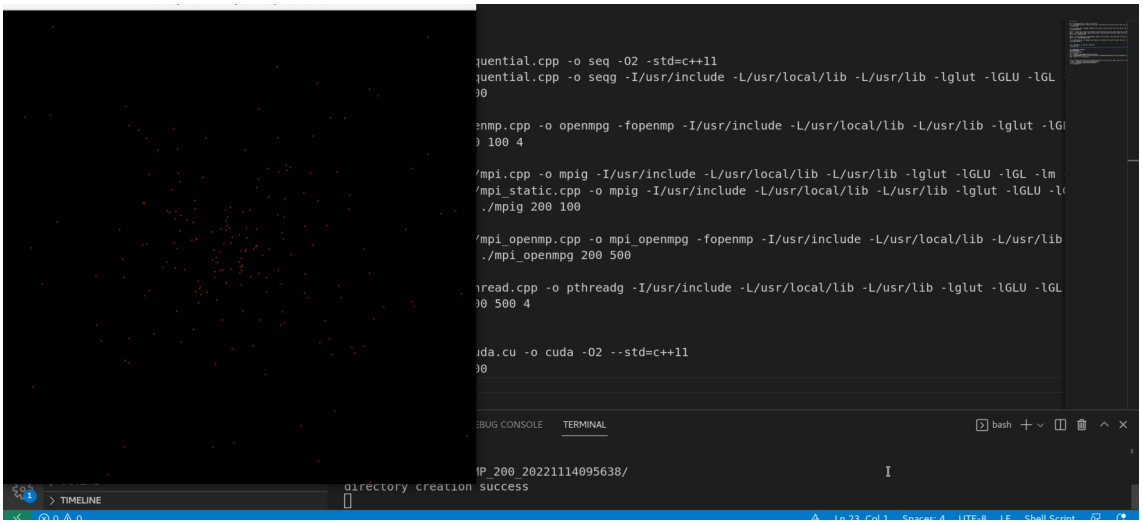

Figure 11: Screenshot 2

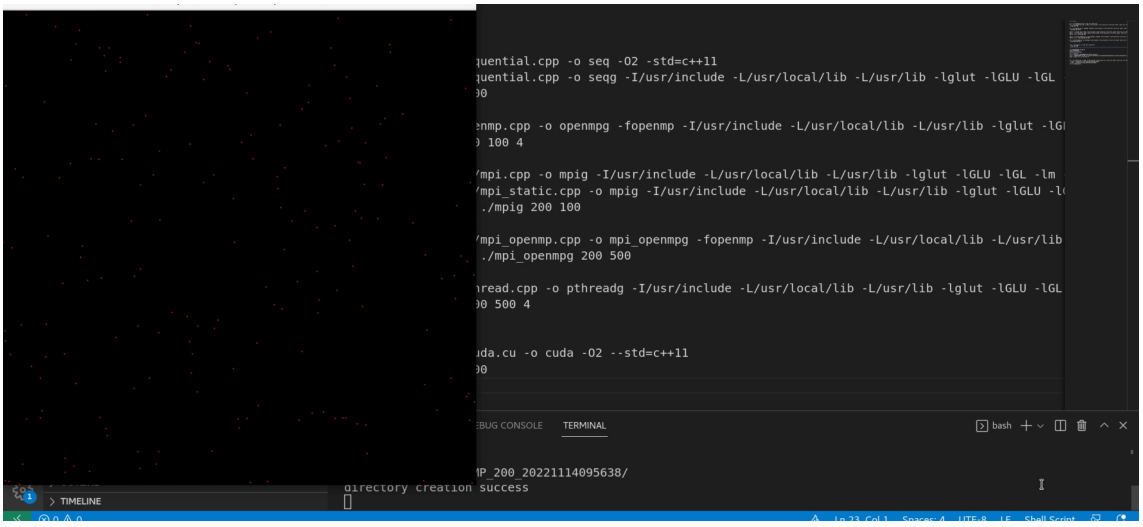

Figure 12: Screenshot 3

See record.mp4 for the complete video.

3.3 Raw data

All programs run for 100 iterations.

Since the initial data is ramdomly generated, there is a certain difference in the amount of calculation for each run of the program.

Because applying for more than 40 cores to HPC at a time will lead to an endless queue, and multi-process programs need an extra main process. The data of MPI related programs used up to 33 processes (i.e. used up to 32 computational processes).

Data size	200	1000	2000					
Sequential	0.913	16.218	62.796					
OpenMP (1 thread)	0.618	15.321	61.084					
Pthread (1 thread)	1.029	16.037	62.81					
MPI (1 slave process)	0.641	15.691	62.253					
CUDA	0.108	0.732	1.35					
execution time								
Number of computational processors/threads	1	2	4	8	16	24	32	40
OpenMP (datasize = 200)	0.618	0.349	0.198	0.162	0.162	0.149	0.142	0.177
Pthread (datasize = 200)	1.029	0.846	0.485	0.317	0.273	0.309	0.365	0.381
MPI (datasize = 200)	0.641	0.348	0.176	0.093	0.069	0.062	0.055	
MPI+OpenMP (datasize = 200)	0.645	0.356	0.187	0.102	0.075	0.067	0.092	
OpenMP (datasize = 1000)	15.321	7.789	3.982	2.199	1.319	1.127	1.004	1.131
Pthread (datasize = 1000)	16.037	9.252	5.7	3.799	2.581	2.394	2.136	2.022
MPI (datasize = 1000)	15.691	7.963	4.087	2.102	1.254	0.962	0.742	
MPI+OpenMP (datasize = 1000)	15.451	8.15	4.082	2.132	1.236	0.931	0.843	
OpenMP (datasize = 2000)	61.084	31.144	15.839	8.437	4.773	3.876	3.483	3.302
Pthread (datasize = 2000)	62.81	33.4	17.408	10.685	7.649	5.347	4.459	4.023
MPI (datasize = 2000)	62.253	31.35	15.764	8.299	5.164	3.666	3.981	
MPI+OpenMP (datasize = 2000)	61.393	31.751	16.027	8.376	4.964	3.746	3.049	

Figure 13: Raw data - execution time on different configurations

3.4 Data analysis

The following figure shows the speedup (sequential execution time (parallel version using 1 processor) / parallel execution time) under different configurations (datasize and number of computational cores/threads).

Number of computational processors/threads	1	2	4	8	16	24	32	40
OpenMP (datasize = 200)	1	1.770774	3.121212	3.814815	3.814815	4.147651	4.352113	3.491525
Pthread (datasize = 200)	1	1.216312	2.121649	3.246057	3.769231	3.330097	2.819178	2.700787
MPI (datasize = 200)	1	1.841954	3.642045	6.892473	9.289855	10.33871	11.65455	
MPI+OpenMP (datasize = 200)	1	1.811798	3.449198	6.323529	8.6	9.626866	7.01087	
OpenMP (datasize = 1000)	1	1.967005	3.847564	6.967258	11.61562	13.5945	15.25996	13.54642
Pthread (datasize = 1000)	1	1.733355	2.813509	4.221374	6.213483	6.69883	7.507959	7.931256
MPI (datasize = 1000)	1	1.970489	3.839246	7.464795	12.51276	16.31081	21.1469	
MPI+OpenMP (datasize = 1000)	1	1.895828	3.785154	7.247186	12.50081	16.59613	18.32859	
OpenMP (datasize = 2000)	1	1.961341	3.856557	7.240014	12.79782	15.75955	17.53775	18.49909
Pthread (datasize = 2000)	1	1.880539	3.608111	5.878334	8.211531	11.74677	14.08612	15.61273
MPI (datasize = 2000)	1	1.985742	3.949061	7.501265	12.05519	16.98118	15.63753	
MPI+OpenMP (datasize = 2000)	1	1.933577	3.830598	7.329632	12.36765	16.38895	20.13545	

Figure 14: Speedup on different configurations

Sequential vs CUDA vs vs openMP Pthread vs MPI vs MPI+OpenMP: The execution time of sequential program is similar to the that of parallel programs using only one computational thread or process. It is within the error range caused by randomness.

In terms of the overall trend, even though both OpenMP version and Pthread version are multi-threading programs, the overhead of the latter is larger than the former. This is because the dynamic thread pool implementation method adopted by Pthread version is: multiple threads compete for the read and write rights of the *progress* variable, and after the competition, use *progress* to read the work item from the work item list, and then increment *progress*. This implementation allows threads to automatically compete for the next work item after completing a work item until all work items have been computed. This avoids frequent communication between threads and the main process. But the procedure of competition uses a mutex lock. When a thread is acquiring a work item, other threads that complete the work item will be blocked. However, OpenMP version does not have this blocking problem, and the overhead will be smaller than Pthread version.

The multi-process part (MPI) uniformly uses static work item allocation, and the multi-thread part (Pthread, OpenMP) uniformly uses dynamic work item allocation. When the amount of data is small and the number of threads/processes is large, the acceleration effect of the former is better. Because in this case, the acceleration effect brought by the difference in thread processing speed is small, and the overhead of information interaction caused by dynamic programming is more obvious.

In addition, although the overhead of the newly process creation is larger than that of the new thread creation, because the part of the creation process is outside of the timing range, the part of MPI version that counts overhead is only the part of message passing. This is also one of the reasons why MPI version is faster than multithreaded versions in the experiment. Because each iteration includes the code to write the result into a file and visualize GUI using OpenGL, both of them are time-consuming and will greatly interfere with the excution time measurement. When they are excluded, MPI initialization time is also excluded.

CUDA has a lot of threads and a high degree of parallelism, and its speed is unsurpassed.

The number of threads/processes vs. Execution time: It can be seen that when datasize is large enough, execution time decreases as the number of computational threads/processes being used increases, and this effect is apparent when there are only a few threads/processes being used. It shows the diminishing marginal utility. This is because when the number of threads/processes being used increases, the

data scale each thread/process processes decreases, and the extent of reduction also decreases. The total overhead of message communication between threads/processes and the overhead pf threads creation increases, which further reduces the marginal benefit and even makes it negative.

Datasize vs. Execution time and Speedup It can be seen that when the datasize is large, the reduction in execution time is significant, and the speedup increases steadily and linearly when the number of threads/processes increases. However when the datasize is small, the reduction in parallel versions' execution time is trivial, and the speedup is also unstable and finally decreasing when the number of cores increases. And as the data size becomes larger, the speedup ratio of multi-threading programs is significantly higher than that of multi-process programs.

This is because in multi-threaded versions' situation, the overhead of thread creations and message passing is fixed with the same number of threads, since the data is stored in the shared memory of threads. Therefore, when the datasize is small, the benefit can hardly cover the overhead if the number of thread is large, while when the datasize is large, the net benefit is more apparent.

In contrast, in multi-process version's situation, the overhead of message passing is proportional to data size with the same number of processes, since the data need to be passed to independent memory of each process. Therefore, the net benefit will not be grately influenced by data size.

Number of computational processors/threads	1	2	4	8	16	24	32	40
OpenMP (datasize = 200)	1	0.885387	0.780303	0.476852	0.238426	0.172819	0.136004	0.087288
Pthread (datasize = 200)	1	0.608156	0.530412	0.405757	0.235577	0.138754	0.088099	0.06752
MPI (datasize = 200)	1	0.920977	0.910511	0.861559	0.580616	0.43078	0.364205	
MPI+OpenMP (datasize = 200)	1	0.905899	0.862299	0.790441	0.5375	0.401119	0.21909	
OpenMP (datasize = 1000)	1	0.983502	0.961891	0.870907	0.725976	0.566437	0.476874	0.33866
Pthread (datasize = 1000)	1	0.866677	0.703377	0.527672	0.388343	0.279118	0.234624	0.198281
MPI (datasize = 1000)	1	0.985244	0.959812	0.933099	0.782047	0.679617	0.660841	
MPI+OpenMP (datasize = 1000)	1	0.947914	0.946289	0.905898	0.781301	0.691506	0.572768	
OpenMP (datasize = 2000)	1	0.98067	0.964139	0.905002	0.799864	0.656648	0.548055	0.462477
Pthread (datasize = 2000)	1	0.940269	0.902028	0.734792	0.513221	0.489449	0.440191	0.390318
MPI (datasize = 2000)	1	0.992871	0.987265	0.937658	0.753449	0.707549	0.488673	
MPI+OpenMP (datasize = 2000)	1	0.966788	0.95765	0.916204	0.772978	0.682873	0.629233	

Figure 15: Cost-efficiency on different configurations

The above figure shows the Cost-efficiency (speedup / number of processes/threads being used) under different number of cores/threads. The horizontal axis is the number of cores and the vertical axis is the Cost-efficiency.

The number of cores vs. Cost-efficiency It can be seen that when datasize is large, the Cost-efficiency keeps decreasing as the number of processes/threads being

used increases.

4 Conclusion

Constrained by discrete simulation, fast visual effects, collision detection, and data dependency requirements, the final code does not strictly follow the laws of physics.

My experiment results show that parallel can greatly speedup the computation task when datasize is large, and the number of processes/threads being used are not so large given the existance of diminishing marginal utility.