

THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

Distributed and Parallel Computing

---

## **Report for Assignment 3**

### **N-body Simulation**

---

*Author:*  
Li Jingyu 李璟瑜

*Student Number:*  
118010141

Nov 17, 2021

## Contents

1. Introduction	3
2. Design	6
3. Execution	22
4. Result & Analysis	26
5. Summary	40

## 1 Introduction

Assignment 3 requires us to design a program for N-body simulation and render the points on screen with a type of GUI. In a two-dimension astronomical space, several bodies move under the universal gravitation. The physical formula is expressed as:

$$F = G \frac{m_1 \times m_2}{r^2}$$

With the force, the bodies obtain some acceleration and gain velocity changes and then positional changes. The bodies are initially at rest, whose positions and masses are randomly generated. There is a bound for the body's movement which prevents the bodies from moving outward. Besides the gravitational force, the collision between balls and balls or balls and walls should be taken into account.



Our task in the assignment is to keep computing the movement of the bodies in a certain elapse and rendering all the planets in the screen using ImGui (a C++ supported graphical user interface) and explore the relationship between **computation speed** and different parameters.

The sequential version code is given while we should attempt to implement the parallel versions using **MPI** and **Pthread** and **openMP** and **CUDA** approaches. The combination of **MPI & openMP** version serves as the bonus.

MPI (Message Passing Interface) is the standard of message passing function library that includes many interfaces. It is a parallel technology based on the communication between processes. A sample MPI interface is shown as follows.

MPI\_Send is one of the most popular functions in MPI.

```
MPI_Send(  
    void* data,  
    int count,  
    MPI_Datatype datatype,  
    int destination,  
    int tag,  
    MPI_Comm communicator);  
  
int pthread_create(  
    pthread_t *restrict thread,  
    const pthread_attr_t *restrict attr,  
    void *(*start_routine)(void *),  
    void *restrict arg);
```

Pthread refers to POSIX threads, which is an execution model that exists independently from a language and a parallel execution model. It is primitive and practical in C programming. Here shows the basic grammar of creating a thread and synchronizing the thread.

```
int pthread_join(pthread_t thread, void** return_value);
```

OpenMP is an application programming interface that supports multi-platform shared-memory multiprocessing programming. Only certain `#pragma` is required to add before the codes to realize parallel programming. No extra initialization is needed. It is probably one of the most convenient approaches among the four.

CUDA refers to Compute Unified Device Architecture. It is a parallel computing platform and application interface that utilizes the power of GPU. Only NVIDIA GPU cards are supported. The CUDA approach includes many complex concepts including signs (device, host, global), grid and block, memory management, and so on. Here shows a kernel function to be executed by the GPU.

```
kernal_name<<<grid_size, block_size>>>(args);
```

Although the parallel computing approaches may be different, the core part of this assignment is the same, which is to partition the body calculation reasonably and allocate to different processes/threads. Proper synchronization and data aggregation strategies also require careful consideration.

## 2 Design

In this part the logic of the codes and some details will be elaborated. All of the implementations are based on the ImGui template on blackboard.

- Sequential part

I will first briefly introduce the sequential part and some general configuration in the programs which will be executed by all the different versions.

In the header file, a body is defined as a class having the properties of index (the number of the body), mass, radius, position (with both x and y coordinate), velocity (in x and y directions) as well as the acceleration (also in both directions). A data pool is defined as the aggregation of many bodies' instances, including several vectors (arrays) to record the position, velocity, acceleration and mass of the individual body.

In computation part, first assign the acceleration as zero since it is only related to force and not influenced by the previous values. We first consider the **effects between balls**. We have a *for loop* handling **each pair of bodies**. For each pair, first calculate their distance, if the distance is smaller than the radius (assume all the bodies have the same size), it means two bodies collide. Certain adjustment to their position and velocity needs to be conducted. If they do not collide, calculate the gravitational force and update their accelerations. Then, we perform operations on **each body**, first check if they collide with the wall. If so, reverse their velocity in that direction and update their new position, making sure it is inside the bound. Eliminate the acceleration. After checking the wall collision, update the velocity using the acceleration we

calculated before, and update the new position, assuming in a very short elapse, the bodies move with constant speed. Check the wall collision again. In this way we finish an elapse.

In main function, several meta data with physical meaning are defined and the graphical engine is initialized. In each iteration we render the position of bodies after calculating their property values. By repeatedly computation and updating on positions, it generates a simulation on N-body movement.

- MPI version

In MPI programming, the programmer should attach importance to the message passing between different processes.

In the main function, initialize the MPI setting first. Set the meta-data like gravity and bodies which may be modified in real-time GUI. Initialize the body pool. These steps are expected to do be done in each process.

In the master process, set up the graphical context for plotting. The rendering function will run repeatedly in order to obtain more average and accurate speed. Set some dragger for adjusting the parameters dynamically in the run time. (However, the procedure is usually slow, especially in the server. So it would better fix the parameters in advance). To better allocate the bodies to different processes, I **manually add some bodies** in order to make **the total body number divisible** by the

**number of processes.** However, these additional bodies will not participate in the computational and the graphical rendering procedure.

The following part will be treated as the **computation cost** (time). We only care about computation time in this project, since drawing costs hugely in the GUI rendering context that is often done with a single process/thread.

In the calculation, the related parameters (meta-data) going to be passed to the functions are broadcast to all the processes in `MPI_COMM_WORLD`, since they are subjected to change by the dragger. Also, broadcast the property values to all the processes for calculation, since computation of gravitational force for certain balls requires the data of all the other balls.

```
// Broadcast meta data
MPI_Bcast(&gravity, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&space, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&radius, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&bodies, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
// Broadcast body data
MPI_Bcast(pool.x.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(pool.y.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Enter the member function of body pool. In the computation part, we calculate the **start and end indexes** of the bodies this process will deal with.

```
int task = size() / proc_num;
size_t start = rank * task;
size_t end = (rank + 1) * task;
```

Then in the computation of **ball pair**, we change our indexes in the **first for loop** as **start and end**, but make the **second for loop** include **all the body numbers**, since a process cannot obtain the **calculated data** in other processes, which are **invisible** during the computation. The changes to bodies beyond start and end cannot



synchronize to other processes in the real time. So we should consider forces from all other balls in a single process, making only the **body data numbered from start to end effective**. In the updating of the body state itself, we just replace our indexes directly.

```
for (size_t i = start; i < end; ++i) {
    for (size_t j = 0; j < size_t(ori_size); ++j) { // Only consider original size's bodies effect
        if (j >= start && j < end) { // inside group computation
            if (j <= i) continue; // avoid repeated computation in-group
        }
        check_and_update(get_body(i), get_body(j), radius, gravity);
    }
}

for (size_t i = start; i < end; ++i) {
    get_body(i).update_for_tick(elapse, position_range, radius);
}
```

There is a detail here. When calculating the effects between both balls inside start and end indexes, we need to eliminate **repeat** calculation, only considering the pair with smaller number ahead and larger number behind.

After we evenly distributed the bodies to each process, we use “*MPI\_Gather*” to gather the all results in different processes. **Only indexes between start and end take effects!** Besides, we only require velocity and position here since acceleration will be reset to zero.

```
// Receive results from other process
int task = bodies / proc_num;
MPI_Gather(&pool.x[rank * task], task, MPI_DOUBLE, pool.x.data(), task, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
MPI_Gather(&pool.y[rank * task], task, MPI_DOUBLE, pool.y.data(), task, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
```

The calculation part then finishes.

In slave process, the procedure is similar, receive broadcast of meta-data and body data from master process, perform calculation and return back the data part they are responsible for.

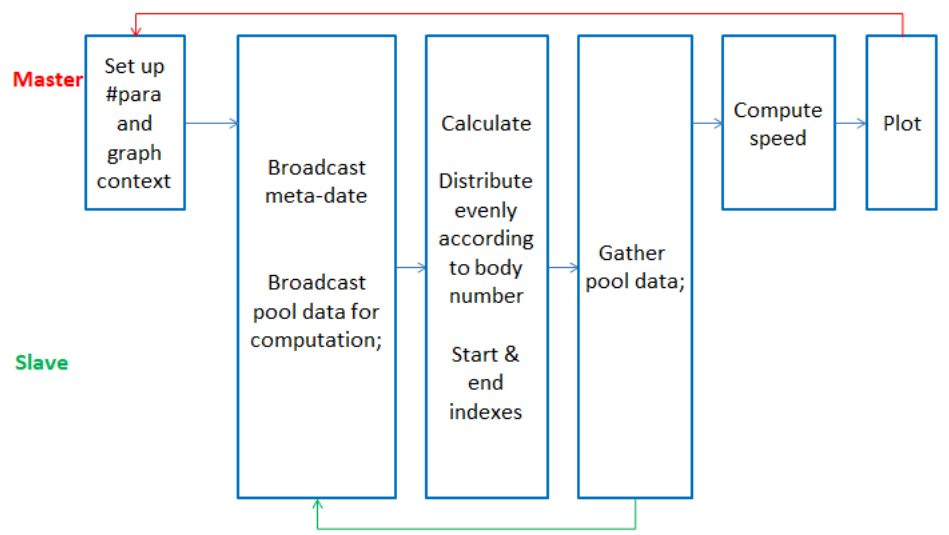
In the main process, the computation speed will be calculated, whose value equals to the processed iteration (# of elapses) divided by the duration (in seconds). The graph is then plotted.

```
duration += duration_cast<nanoseconds>(end - begin).count();
if (count == ITERATION) {
    std::cout << ITERATION << " elapse with " << duration << " nanoseconds\n";
    double speed = double(ITERATION) / double(duration) * 1e9;
    std::cout << "speed: " << speed << " iterations per second" << std::endl;
```

The master process will continue to assign the parameter value and run the computation function, calculate time cost and plot the graph. The slave process will repeat the computation part.

```
// Computation
pool.update_for_tick(elapse, gravity, space, radius);
```

A structural flow figure is displayed.



- Pthread version

In Pthread programming, the programmer should lay more emphasis on thread synchronization and avoid dead lock.

Compared with MPI version, Pthread is similar but easier to implement. All the threads are able to read and write the global class pool and its data directly since they share the same memory space. The parameters for calculation are initialized as global variables to be visible in all threads. To represent different thread, I use an atomic integer as the thread number (rank in MPI version), which can avoid deadlock, ensure thread safety and provide better performance than *mutex*. A mutex and a barrier are initialized for synchronization and **data race prevention**.

```
std::atomic_int current_thread{0};  
pthread_barrier_t barrier;  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

In the main function, the program receives thread number from command line. Similarly, the main thread will set up graphical context. In the calculation part, create threads and execute a **wrapper function** of the calculation function, since the member function of a class has an implicit argument *\*this*. Join threads afterwards.

```
/* Start calculation */  
auto begin = high_resolution_clock::now();  
  
current_thread = 0;  
for (int i = 0; i < thread_num; i++) {  
    pthread_create(&comp_threads[i], nullptr, pool_run, nullptr);  
}  
  
for (int i = 0; i < thread_num; i++) {  
    pthread_join(comp_threads[i], nullptr);  
}  
auto end = high_resolution_clock::now();  
/* Finish calculation */
```

```
void *pool_run(void *) {  
    pool.update_for_tick(elapse, gravity, space, radius);  
    return nullptr;  
}
```

Inside the computation function, one of the tricky parts is to **avoid data race**. A data race happens when two threads try accessing the same position in the memory. For example, when one thread reads a value of position for calculation, another thread writes to the position value immediately, before the first thread can obtain the result. The common scenarios are **Write After Read, Write after Write**. To avoid this, **I separate the reading and writing, distributing the data into two vectors**. During the computation, the threads can only read the data from the read vectors (previous state) and write to another vector (new state). After all computation in this step finishes, transfer the write vectors to the read vectors (new→previous). We avoid dirty data using this strategy. (Adding mutex to avoid data race costs too much!!!)

```
// Local variable to write  
std::vector<double> new_x(x);  
std::vector<double> new_y(y);  
std::vector<double> new_vx(vx);  
std::vector<double> new_vy(vy);  
std::vector<double> new_ax(ax);  
std::vector<double> new_ay(ay);
```

The data partition logic is similar with that in MPI, using start and end indexes. Modification to the functions arguments is required since we now write the values to other vectors.

```
size_t start = remain + rank * task;  
size_t end = remain + (rank + 1) * task;
```

```

for (size_t i = start; i < end; ++i) {
    for (size_t j = 0; j < size(); ++j) {
        if (j >= start && j < end) {
            if (j <= i) continue;
        }
        check_and_update(get_body(i), get_body(j), radius, gravity, new_x, new_y, new_vx, new_vy, new_ax, new_ay);
    }
}

```

A barrier is needed to synchronize all the data before updating the bodies themselves (after computing all the forces). If we do not use a barrier here, after updating a single body, its value may be further changed by forces and collision between bodies.

```

// Synchronize the updation here
pthread_barrier_wait(&barrier);

```

```

for (size_t i = start; i < end; ++i) {
    get_body(i).update_for_tick(elapse, position_range, radius, new_x, new_y, new_vx, new_vy, new_ax, new_ay)
}

```

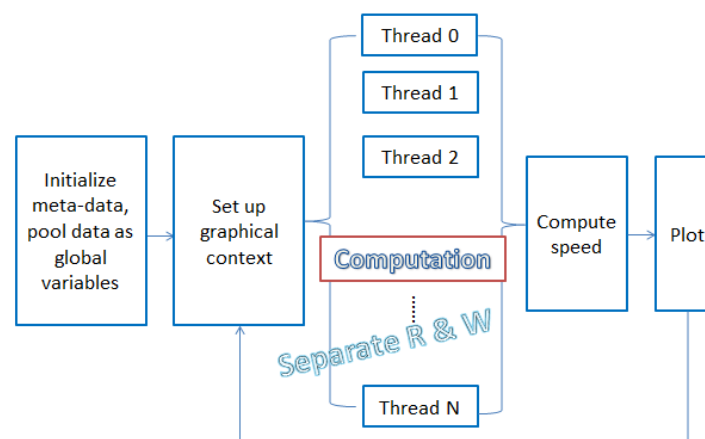
At the end of the computation, we transfer the written data to the global data (where we read). Here a mutex is required to prevent data race. Only one thread can execute the code in a certain time period.

```

pthread_mutex_lock(&mutex);
for (size_t i = start; i < end; ++i) {
    x[i] = new_x[i]; y[i] = new_y[i];
    vx[i] = new_vx[i]; vy[i] = new_vy[i];
}
pthread_mutex_unlock(&mutex);

```

The program flow shows as follows.



- OpenMP version

OpenMP version is easy to implement, but the **data race** problem is still troublesome! The main logic of openMP program is nearly the same as the sequential function except the addition of speed calculation and thread number obtained from the command line. Programmer only needs to add the `#pragma` just before the *for* loop to parallel. No explicit initialization of processes or threads is even required.

```
/* Start calculation */
auto begin = high_resolution_clock::now();

pool.update_for_tick(elapse, gravity, space, radius);

auto end = high_resolution_clock::now();
/* Finish calculation */
```

In the computation part, to avoid data race, I still make use of the Read & Write Separation technique. I initialize two groups of vectors to write, since when entering the parallel part, the vectors are stated private (or there will be data race) and invisible to each other. To gather the data after one step, we need to use a temporary vector to store the middle state (gather private variables). Another reason to use two vectors is we assign the values in each for loop immediately (each *i*), since the parallel range only includes the *for* loop, or we will lose the data we compute. If we store the data to only one vectors, data race will happen (we do not assign the values at the end but in each for loop).

```
size_t task = size() / thread_num;
size_t start = size() + 1;

size_t i, j;
```

```
// Local variable to write
std::vector<double> new_x(x);
std::vector<double> new_y(y);
std::vector<double> new_vx(vx);
std::vector<double> new_vy(vy);
std::vector<double> new_ax(ax);
std::vector<double> new_ay(ay);

// Temporary vectors to avoid data race
std::vector<double> temp_x(size());
std::vector<double> temp_y(size());
std::vector<double> temp_vx(size());
std::vector<double> temp_vy(size());
std::vector<double> temp_ax(size());
std::vector<double> temp_ay(size());
```

We use the `#pragma` for twice, one for the computation between balls and one for updating the single ball. Carefully deal with the index partition and assign the values between different vectors to avoid data race.

```
#pragma omp parallel for private(j) firstprivate(start, new_x, new_y, new_vx, new_vy, new_ax, new_ay)
for (i = 0; i < size(); ++i) {
    if (start == size() + 1) {
        start = i;
    }
    for (j = 0; j < size(); ++j) {
        if (j > start && j < start + task) {
            if (j <= i) continue;
        }
        check_and_update(get_body(i), get_body(j), radius, gravity, new_x, new_y, new_vx, new_vy, new_ax, new_ay);
    }
}
```

Here several private variables in threads need to be determined. Or threads will affect each other.

```
temp_x[i] = new_x[i]; temp_y[i] = new_y[i];
temp_vx[i] = new_vx[i]; temp_vy[i] = new_vy[i];
temp_ax[i] = new_ax[i]; temp_ay[i] = new_ay[i];
```

```
new_x.assign(temp_x.begin(), temp_x.end()); new_y.assign(temp_y.begin(), temp_y.end());
new_vx.assign(temp_vx.begin(), temp_vx.end()); new_vy.assign(temp_vy.begin(), temp_vy.end());
new_ax.assign(temp_ax.begin(), temp_ax.end()); new_ay.assign(temp_ay.begin(), temp_ay.end());
```

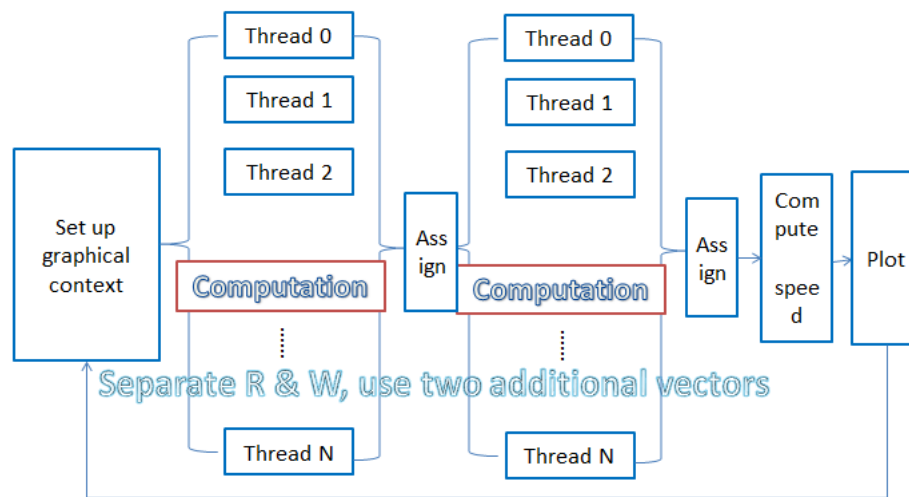
```
#pragma omp parallel for firstprivate(new_x, new_y, new_vx, new_vy, new_ax, new_ay)
for (i = 0; i < size(); ++i) {
    get_body(i).update_for_tick(elapse, position_range, radius, new_x, new_y, new_vx, new_vy, new_ax, new_ay);

    temp_x[i] = new_x[i]; temp_y[i] = new_y[i];
    temp_vx[i] = new_vx[i]; temp_vy[i] = new_vy[i];
}
```

```
x.assign(temp_x.begin(), temp_x.end()); y.assign(temp_y.begin(), temp_y.end());
vx.assign(temp_vx.begin(), temp_vx.end()); vy.assign(temp_vy.begin(), temp_vy.end());
```

The above show some assignment between the vectors and the #pragma sentence.

The below shows the program flow of openMP version.



### ● CUDA version

In the CUDA version, I include all the codes of body.hpp into main.cu for convenience. During the programming, the global data both visible to CPU and GPU also provide great convenience for the implementation. The body data are initialized as the \_\_device\_\_ \_\_managed\_\_ global data, which can be accessed by both CPU and GPU directly. **BODIES** are defined in advance for quick initialization.

```
#define BODIES 200
```

```
__device__ __managed__ double x[BODIES];
__device__ __managed__ double y[BODIES];
__device__ __managed__ double vx[BODIES];
__device__ __managed__ double vy[BODIES];
__device__ __managed__ double ax[BODIES];
__device__ __managed__ double ay[BODIES];
__device__ __managed__ double m[BODIES];
```



The original member data in body pool are all separated as a `__device__` function only executed on the GPU and defined in the main.cu as a normal function.

```
__device__ void handle_wall_collision(double position_range, double radius, int index)
```

```
__device__ void update_for_tick(double elapse, double position_range, double radius, int index)
```

```
__device__ void check_and_update(int i, int j, double radius, double gravity)
```

The major function to be executed by the GPU called from CPU (the host) is signed with `__global__`.

```
__global__ void update_for_tick(double elapse, double gravity, double position_range, double radius, int thread_num)
```

In the computation function, the thread number is obtained by certain dim calculation. Then using the thread number to partition the task, using the start and end indexes, still. The computation is similar with that in pthread version. Each loop will operate the global memory data directly. Since each thread is responsible for different indexes, no data race will happen.

```
int rank = blockIdx.x * blockDim.x + threadIdx.x;
int task = BODIES / thread_num;
int start = rank * task;
int end = start + task;

for (int i = start; i < end; ++i) {
    for (int j = i + 1; j < BODIES; ++j) {
        if (j > start && j < start + task) {
            if (j <= i) continue;
        }
        check_and_update(i, j, radius, gravity);
    }
}
```

After the computation between bodies, we need a synchronization function to maintain the data consistency. Or there will be data race.

```
// Synchronize all the threads in this point
__syncthreads();
```

Then compute the single body using the start and end indexes.

```
for (int i = start; i < end; ++i) {
    update_for_tick(elapse, position_range, radius, i);
}
```

With the help of `__device__` `__managed__` global arrays, the computation part is smooth to implement. It is really a present for CUDA programmer.

In the main function, obtain thread number from command line. Initialize the meta-data, randomize the body data. The core part shows as follows.

```
/* Start calculation */
auto begin = high_resolution_clock::now();
// Cuda initialization
update_for_tick<<<1, thread_num>>>>(elapse, gravity, space, radius, thread_num);

// Wait for computation to finish
cudaDeviceSynchronize();

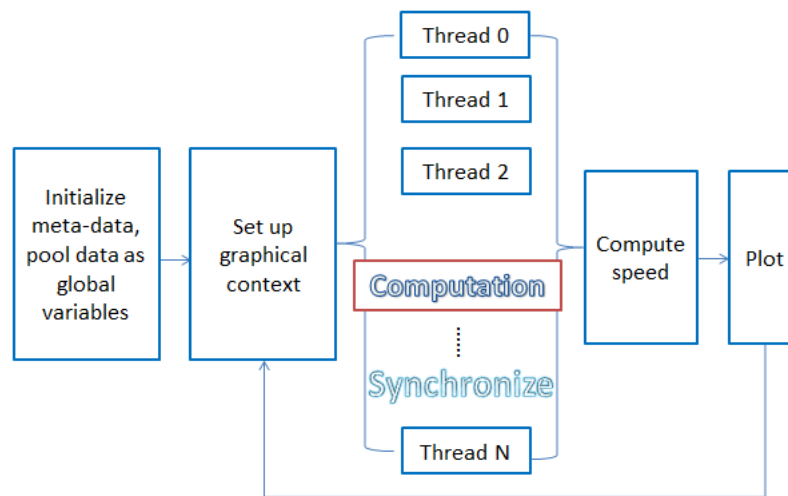
auto end = high_resolution_clock::now();
/* Finish calculation */
```

We use the kernel function `update_for_tick` to distribute tasks to GPU. Remember to synchronize the threads after calling the kernel functions since the control returns to CPU immediately and it should wait for all computation tasks to finish.

Here I only use one grid, making the thread number changeable. The reason for this is to simplify the code logic. If the grid/block number is varied, there are multiple methods to determine the thread dimensional arrangement. Also, the data synchronization between blocks is difficult and time-consuming, which causes unnecessary performance cost. Only one block is sufficient to realize parallel programming.

The CUDA implementation is similar with that in asg3 of CSC3150.

The CUDA program flows:



#### ● Bonus

The bonus part combines both MPI and OpenMP. We distribute the tasks to different processes. In each sub-task, we can also make use of the openMP #pragma to further distribute the workload. It is a multi-process + multi-thread programming.

The main function part is quite similar with that of MPI version. The only difference is that a thread number is entered by the user to implement openMP. The thread number is then broadcast to all processes for sub-division.

```

MPI_Bcast(&thread_num, 1, MPI_INT, 0, MPI_COMM_WORLD);
omp_set_num_threads(thread_num);
  
```

In the body.hpp (computation part), the code is then quite similar with that of openMP version. This time, only one additional vector is required to maintain data consistency since in different process, the data are invisible to each other.

```

// Local variable to write
std::vector<double> new_x(x);
std::vector<double> new_y(y);
  
```

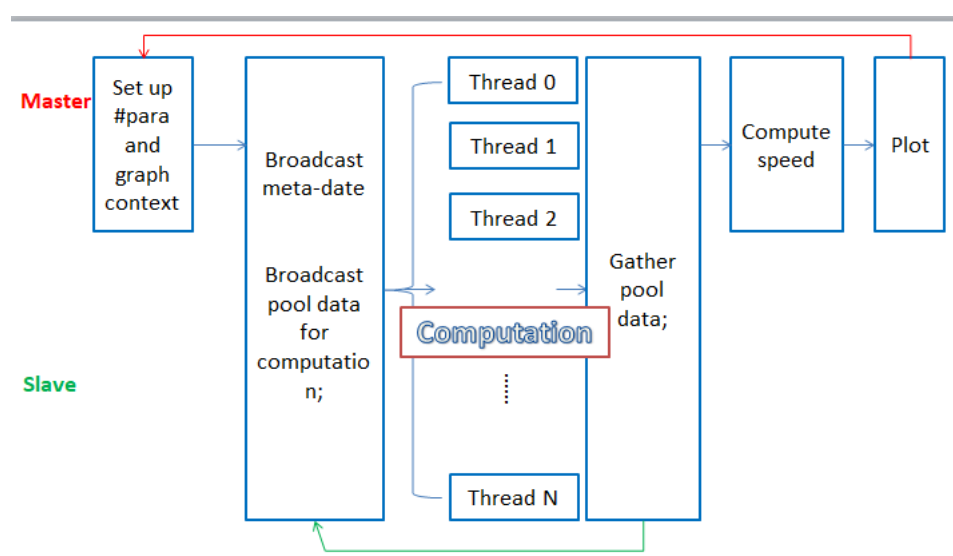
Use start and end indexes to divide the task (MPI). Then in the computation between the bodies, the `#pragma` is used in internal for loop, separately calculated the second body forces or effects and update the additional vector. In the computation on a single body, Use **`#pragma omp parallel`** for to further divide the range between the start and end indexes. No private variables need to be specified since all threads share the same memory, the updating is on the correct positions.

```
for (size_t i = start; i < end; ++i) {
    #pragma omp parallel for
    for (size_t j = 0; j < size_t(ori_size); ++j) { // Only consider original size's bo
        if (j >= start && j < end) { // inside group computation
            if (j <= i) continue; // avoid repeated computation in-group
        }
        check_and_update(get_body(i), get_body(j), radius, gravity, new_x, new_y, new_vx
    }
}
```

```
#pragma omp parallel for
for (size_t i = start; i < end; ++i) {
    get_body(i).update_for_tick(elapse, position_range, radius, new_x, new_y, new_vx, new_
}
```

```
x.assign(new_x.begin(), new_x.end()); y.assign(new_y.begin(), new_y.end());
vx.assign(new_vx.begin(), new_vx.end()); vy.assign(new_vy.begin(), new_vy.end());
```

The program flow:



- Some details

Only the computation part makes sense when calculating the speed, since plotting costs greatly by a single thread.

The program runs repeatedly to average the speed and obtain more accurate results.

The method I use is static allocation, partitioning the bodies by their indexes, which balances the workload to some degrees.

It is meaningless to pass pointer in MPI programming since in different machine the memory address is not the same.

Add `set(CMAKE_CXX_FLAGS "-fsanitize=thread")` in the Cmakefile to check data race.

`Pthread_exit(NULL)` will cost a bug in *fsanitize*.

### 3 Execution

The execution for my code performs on the remote cluster. There are 8 internal nodes each with one **NVIDIA 2080Ti GPU card** with 32 cores together. The maximum CPU time limit is 640 (4 nodes, 128 cores for 5 minutes).

Environment: Linux server equipped with multi-thread library and NVIDIA GPU.

*Way to compile and execute (MPI, Pthread, and OpenMP):*

1. Unzip 118010141.zip
2. `cd csc4005-assignment-3 & csc4005-imgui-xxx (mpi, pthread, openmp)`
3. `mkdir build && cd build`
4. `cmake .. -DCMAKE_BUILD_TYPE=Release`
5. `cmake --build . -j4` (or **make**)

```
cd /path/to/project
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug # please modify this to `Release` if you
want to benchmark your program
cmake --build . -j4
```

6. **MPI:** `mpirun -np 4 ./csc4005_imgui` [Can specify the process number here]

**Pthread/OpenMP:** `./csc4005_imgui 4` [Can specify the thread number here]

7. Adjust the parameters (space, gravity, radius, bodies, elapse, and max mass) to obtain different effects.
8. Check the duration and speed information in the bash.
9. **Ctrl+C** to **forcedly** end the program.

***Way to compile and execute (CUDA):***

*[Thanks to the warm-hearted student in the wechat group!]*

1. `cd csc4005-imgui-cuda`
2. `mkdir build && cd build`
3. `source scl_source enable devtoolset-10`
4. `CC=gcc CXX=g++ cmake ..`
5. `make`
6. (Add *srun* in the server) `./csc4005_imgui 4` [Can specify the thread number here]
7. No adjustment to the parameters for performance consideration.

Only adjust them in the code.

8. Check the duration and speed information in the bash.
9. **Close** the window to stop the program

Some notice:

For **graphical exception**:

Use **ssh -Y** to register.

In /pvfsmnt/118010141 directory:

Copy XAuthority file:

```
cp ~/.Xauthority /pvfsmnt/${whoami}
```

Set XAuthority env:

```
export XAUTHORITY=/pvfsmnt/${whoami}/.Xauthority
```

I run in /pvfsmnt/118010141, the share file system.

For most of the experiment, I use *salloc* to run interactively:

**Salloc -N1 -n32 -t10**

I also use *sbatch* to submit the task for some experimental group. A sample

script is shown as follows:



```
#!/bin/bash
#SBATCH -o cuda64.out
#SBATCH --account=csc4005
#SBATCH --partition=debug
#SBATCH --qos=normal
#SBATCH --nodes=2
#SBATCH --ntasks=64
#SBATCH --time=03:00

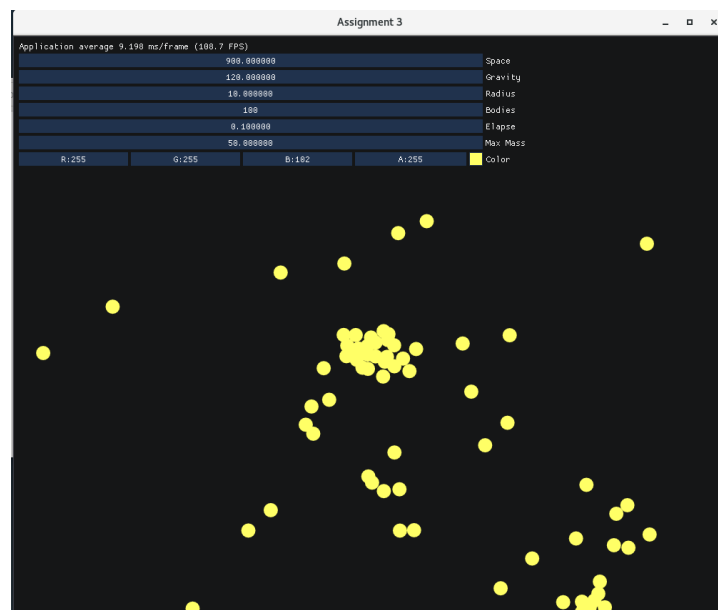
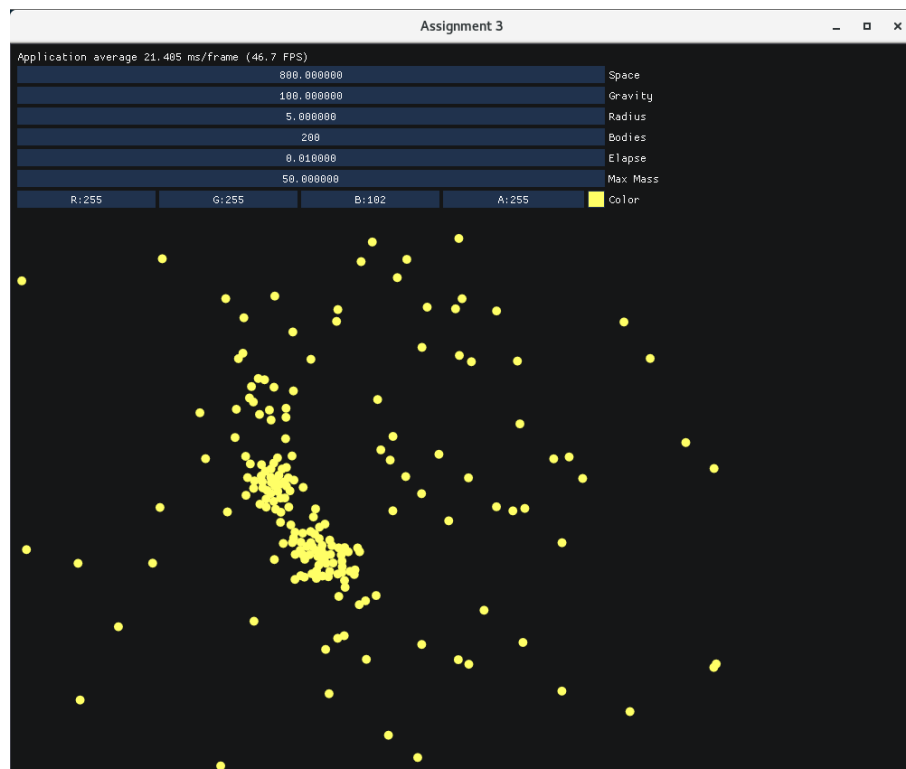
srun /pvfsmnt/118010141/asg3_cuda/csc4005_imgui 64
```

Again, the program should exit after a certain time limit defined by the *sbatch* script or forcedly ended by the user [ctrl + C] since it is running repeatedly.

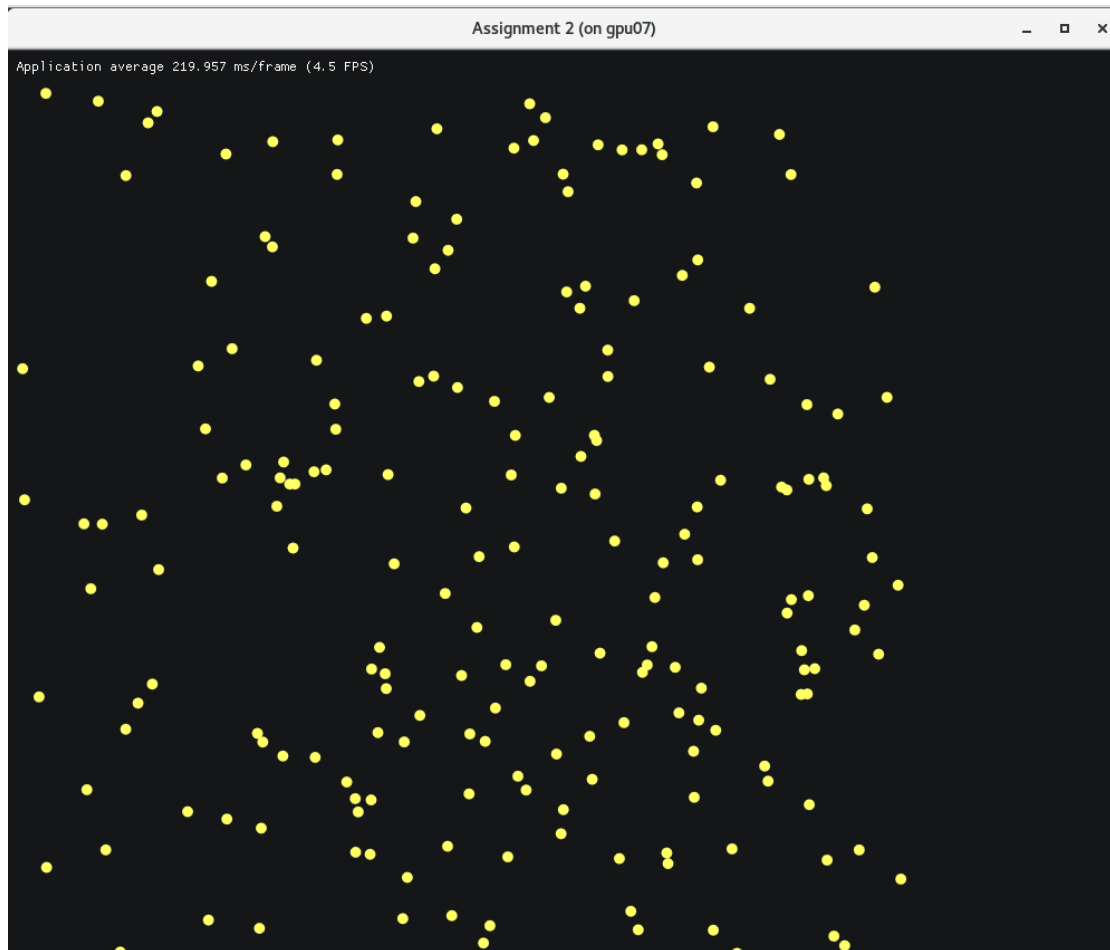
## 4 Result & Analysis

### ● GUI Result

The GUI window is shown as follows. Drag different parameters to make influences. (For example, adjust radius to make the body size larger). The following graphs run on the virtual machine (MPI, Pthread versions).



The following figure (CUDA version) run on the remote server. For performance and code logic consideration, no draggers are provided here. (The network communication speed of GUI is amazingly slow between server and local host)



**Two videos** are attached in the homework folder for animation display. The first video runs under the MPI configuration with 4 processes. As time goes by, some of bodies will **gather to form clusters**, showing the effect of gravity. The second video is about openMP configuration with 6 threads. Adjustment to the parameters is performed. Modification to different parameters will bring different effects, which will be illustrated below.

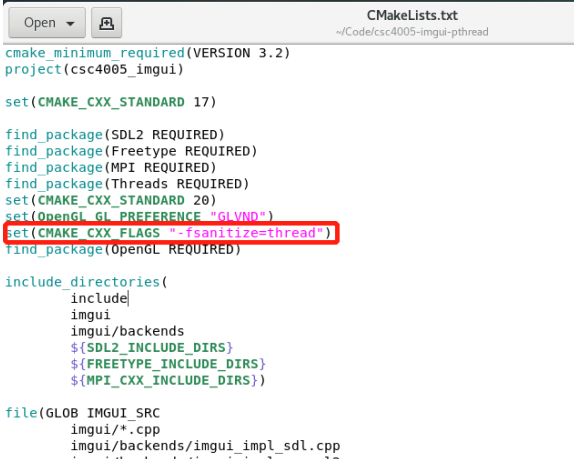
## ● Bash Result

In the bash, the computing time (in nanoseconds) is shown. The speed is calculated and output then (iterations/second). The output is continuous.

```
[csc4005@localhost build]$ mpirun -np 4 ./csc4005_imgui
[WARN] cannot get screen dpi, auto-scaling disabled
100 elapse with 416035343 nanoseconds
speed: 240.364 iterations per second
100 elapse with 487663640 nanoseconds
speed: 205.059 iterations per second
100 elapse with 426077359 nanoseconds
speed: 234.699 iterations per second
100 elapse with 322146212 nanoseconds
...
bash-4.2$ srunk ./csc4005_imgui 8
[WARN] cannot get screen dpi, auto-scaling disabled
100 elapse with 671658480 nanoseconds
speed: 148.885 iterations per second
100 elapse with 673872981 nanoseconds
speed: 148.396 iterations per second
100 elapse with 672239332 nanoseconds
speed: 148.757 iterations per second
100 elapse with 673380782 nanoseconds
speed: 148.504 iterations per second
```

The first screenshot shows the bash result of mpi version and the second shows the CUDA result on the serve. The speed is unstable due to the random distribution (it may cause different situations, either collision or acceleration). As time goes by, in each iteration the calculation workload is also different (due to point distribution and multiple situations of collisions).

Add a sentence to check data race. **No data race during computation appears in my program.**



```
Open [icon] CMakeLists.txt
~/Code/csc4005-imgui-pthread

cmake_minimum_required(VERSION 3.2)
project(csc4005_imgui)

set(CMAKE_CXX_STANDARD 17)

find_package(SDL2 REQUIRED)
find_package(Freetype REQUIRED)
find_package(MPI REQUIRED)
find_package(Threads REQUIRED)
set(CMAKE_CXX_STANDARD 20)
set(OpenGL_GL_PREFERENCE "GLVND")
set(CMAKE_CXX_FLAGS "-fsanitize=thread")
find_package(OpenGL REQUIRED)

include_directories(
    include
    imgui
    imgui/backends
    ${SDL2_INCLUDE_DIRS}
    ${FREETYPE_INCLUDE_DIRS}
    ${MPI_CXX_INCLUDE_DIRS})

file(GLOB IMGUI_SRC
    imgui/*.cpp
    imgui/backends/imgui_impl_sdl.cpp
    imgui/backends/imgui_impl_opengl3.cpp
    ...)
```

There may be some strange warning by the checker at the very beginning of the code (data race in one byte of mutex??), but **it should not be the data race during the computation** (should be size 8 bytes), I guarantee no data race happens **after successfully launch** the computation part and enter iteration. (Use )

- Robustness

The program is able to deal with the case with different process/thread numbers (1-N), different space, gravity, radius, elapse, maximum mass, color, and so on. The program can handle the case where bodies number is smaller than process/thread number.

Before the analysis, let me state the default setting for the meta-data (physical data)

[gravity=100 (decide force level), space=800 (decide bound for the body movement), radius = 5 (body size); elapse = 0.01 (time period for one iteration); **bodies=200 (Number of bodies)**; max\_mass=50 (maximum random mass), iteration=100 (Number of iteration to calculate speed)]

- Relationship between speed & process(thread) number

To explore the relationship between running speed in iterations per second (y-axis) and the process (thread) number (x-axis), the following experiments are presented.

Using the control-variable method, the gravity, space, radius, elapse, bodies (200), max\_mass and iteration should be fixed among different process/thread numbers. In the experiment group, I only set 1 group of body size (200), focusing more on the comparison between different implementation. This is because the server is too crowded this time and it is hard to run too many groups of data smoothly. There are also four versions this time, doubling!! Besides, **I have analyzed the relationship between speed and bodies separately in the below part.**

In each group, process/thread number from **1** (sequential) to **10** (parallel) will be tested (if the number is larger, the graphical window responses pretty slow, so 10 is sufficient to show some insights), whose speed (iterations / duration time) will be recorded and graphed in a line chart to make the comparison and analyze the tendency.

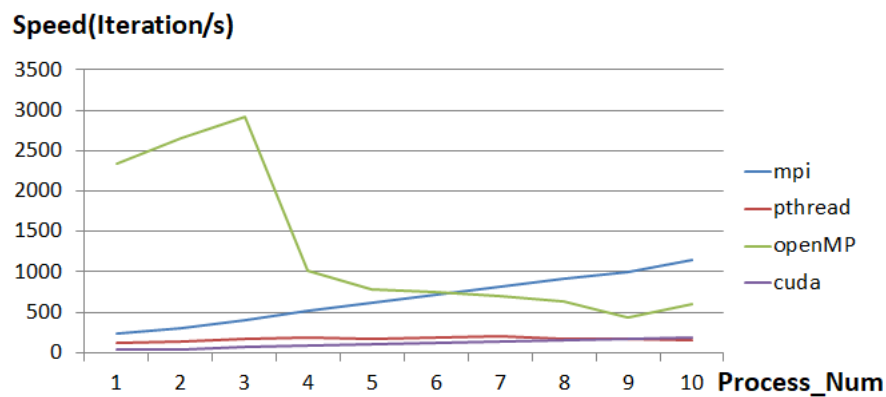
**[More than 33 processes/threads setting will be analyzed in the later part separately]**

The detailed data and the line chart are shown as follows.

The unit for the value is **iterations/s**, which will not be explained again later.

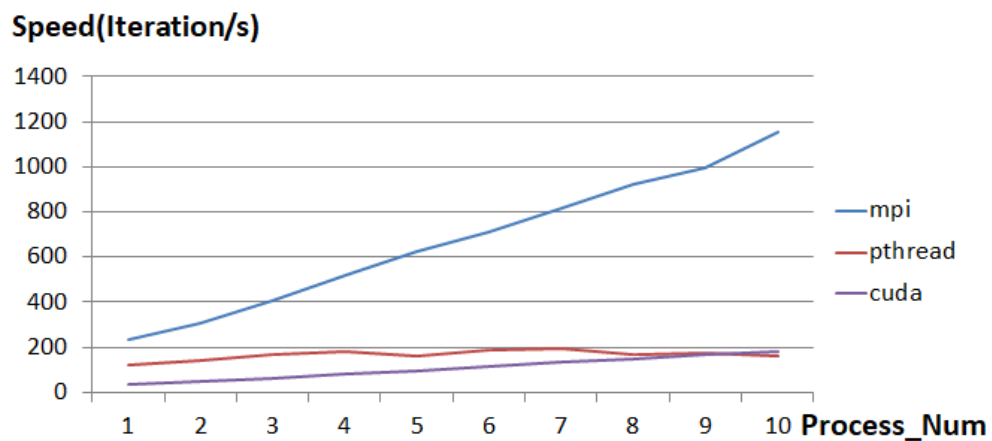
Process/Thread Number	mpi	pthread	openMP	CUDA
1	231.371	119.497	2333.75	35.3688
2	307.325	144.234	2657.7	47.1253
3	409.575	169.297	2917.77	63.9923
4	517.388	182.818	1023.14	80.427
5	622.503	163.463	791.39	97.5372
6	710.608	187.874	751.102	115.48
7	814.2	196.921	697.785	134.083
8	923.412	166.268	638.972	148.835
9	995.698	173.089	437.514	167.136
10	1153.01	161.591	596.357	182.891

Relationship between speed & process number (1-10) MPI, Pthread, openMP, CUDA



Relationship between speed & process number (1-10) MPI, Pthread, CUDA

(openMP's value is too large!)



As can be seen from the raw data and the graph, the program runs the most slowly sequentially in MPI, Pthread and CUDA version, but runs not so slow in the openMP versions. It means that in the first three versions, the parallel approaches are effective, bringing some speedups. However, in openMP, the  $n=1$  means it is exactly the sequential code initially! It is fast at the first 3 processes, but drops greatly from 3 to 4, this may be due to the extra cost of thread initialization and vectors assignment as threads increases (I use two vectors to avoid data race in openMP version). The performance is also related to coder's implementation.

Notice that pthread version also experiences a bottleneck from 4 to 5. When the thread number is larger than 4, no more speedup is gained in both pthread and openMP version! This may be partly because of the thread initialization cost, but the major cause should be the CPU core limit. The threads seem to can reach infinite; however, only one thread can run simultaneously in one core, corresponding to one processor. When the thread number is greater than the processor number, the operating system will run the threads alternately, forming a fake appearance of multiple threads, with the performance bottleneck in fact the number of the cores. I run the Pthread and openMP in my virtual machine whose core number is 4. As a result, these two versions have a bottleneck near 4. Unfortunately, the shape of the line chart for Pthread and openMP version is not that perfect. They also fluctuate a lot (in a certain range).

For the MPI and CUDA versions, the speed increases linearly as process number increases, forming a **perfect straight line**! The slope is fixed, meaning that the



average computation speed is fixed among different processes. The result shows the advantage of my approach for distribution——by body indexes. The balanced workload renders each process similar time to finish the task and return the results back for plotting. There is little interference issue like process synchronization or data transfer events when the process number is relatively small (1-10). In MPI version, the communication does not cost much to change the linear shape. In CUDA version, the modification is directly to the global variable `__device__ __managed__`, even smaller extra cost besides computation. The two results are reasonable.

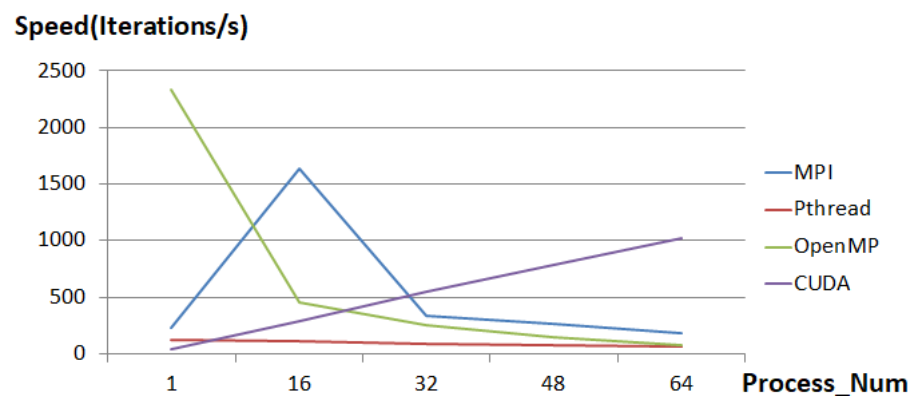
In conclusion, the MPI and CUDA version performance is pretty stable and fits the expectation when the process number is in range 1 to 10. The Pthread and openMP versions meet a bottleneck around thread number 4.

## ## Large process number

I made another experiment, setting the number of processes sequential and the multiple of 16. Because every node has 32 cores, I would like to explore the efficacy when the cores are distributed in different (2) GPU nodes for MPI & CUDA version. To make a comparison, the Pthread and openMP versions data are also shown.

Process/Thread Number	MPI	Pthread	OpenMP	CUDA
1	231.371	119.497	2333.75	35.3688
16	1641.77	114.158	450.774	289.833
32	329.49	82.3793	255.591	546.056
48	262.72	77.0965	141.882	785.412
64	186.649	60.1371	79.8377	1026.13

Relationship between speed (pixels/s) &amp; process number (1-64)



From the raw data and the graph, for MPI, the running speed increases greatly from 1 process to 16 processes but falls greatly to the same level of sequential at process 32 and worse with larger process number. The speedup is no longer perfectly linear when the core number is relatively huge. Because the cost for process communication and synchronization increases as the number of processes adds up this

time. The main cost lies in the **data gathering and then the parameters broadcast**. Besides, the cost for process communication between different nodes is far greater than that in the same node (32 cores). In the same GPU, the cost is ignorable (proved by my previous experiment 1-10), but in the different GPU, the cost cannot be neglected! As a result, structure of the cluster should also be taken into account when analyzing. There are 32 cores in a node, and every one node addition will reduce the speedup ration (the slope in the graph).

However, for the CUDA version, the line is still a perfect straight line, meaning that CUAD version still not reaches its bottleneck. There are plenty of threads in a single block, up to tens or even hundreds. The communication between threads can be ignored in the same block, thus the additional threads will only bring extra speedup. 64 is not the limit. From the result we can also learn more the internal structure of the CUDA GPU.

For Pthread and openMP versions, conforming to my previous analysis, the performance is stable (reduced for openMP) after certain thread number.

One of the possible reasons (except CPU core bottleneck) to explain the bottleneck and the instability is the cost for thread creation and synchronization. In the main thread, the other threads are created one by one, causing different start time for computation. One thread can only start after the previous thread is successfully launched.

As a result, when the thread number is large, the time interval between the first thread and the last thread should be large.

After computation, the function `Pthread_join()` is used to synchronize all the threads, causing a time delay. Assume the time for creating a thread is fixed, it is normal that when the thread number increases, although the computation time can be shortened, the time difference increases, causing a longer waiting time for thread synchronization and no improvement on the total performance due to the elimination.

```
/* Start calculation */
auto begin = high_resolution_clock::now();
current_thread = 0;
for (int i = 0; i < thread_num; i++) {
    pthread_create(&cal_thread[i], nullptr, calculate, nullptr);
}
for (int i = 0; i < thread_num; i++) {
    pthread_join(cal_thread[i], nullptr);
}

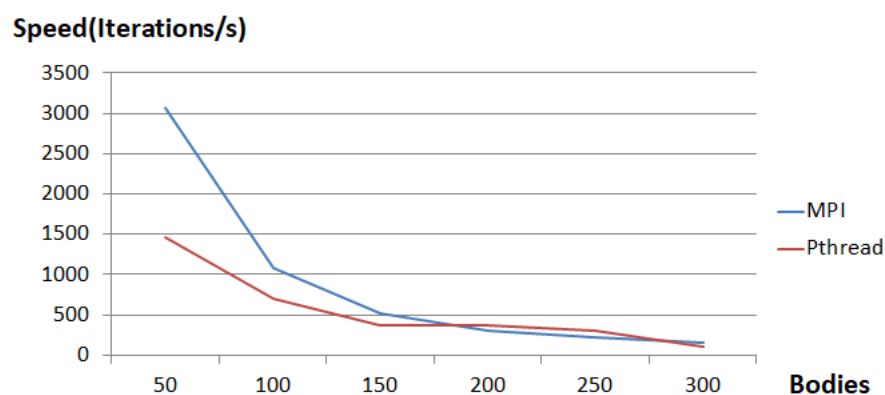
auto end = high_resolution_clock::now();
/* Finish calculation */
```

- Relationship between speed & bodies

The following experiments are conducted when process number is 4. Only MPI version should be taken into account since it is stable. The data is crazy for Pthread and openMP, varying greatly. Also, it is impossible to run CUDA for these less trivial experiments in the crowded server!!!

Similar to the previous experiment, to explore the relationship between the speed and body number, the number of processes (threads) and other parameters should be fixed. Here I choose the parallel scenario when  $N=4$  fix other values. The bodies varies from 50 to 300, with 50 increase each time. Both MPI and Pthread versions are conducted. The results are shown as follows.

Bodies	MPI	Pthread
50	3070.9	1454.77
100	1077.08	707.212
150	522.825	377.171
200	306.324	372.054
250	218.503	303.257
300	160.575	112.276

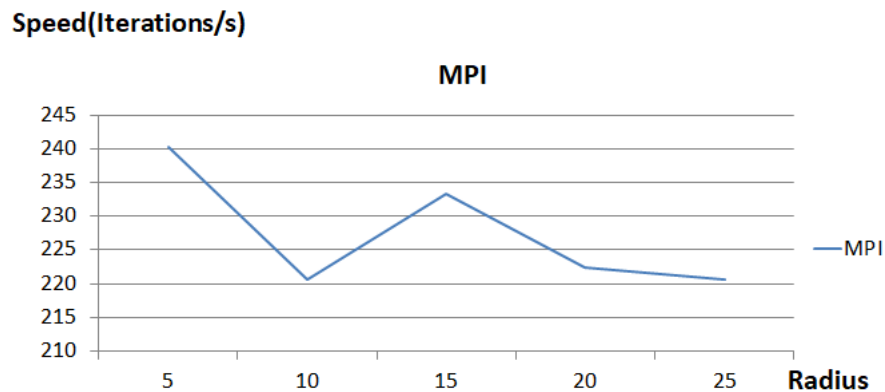


As can be seen from the raw data and the graph, the speed decreases obviously when body number increases. For both MPI and Pthread version, the tendency is true, fitting our previous conclusion that running time increases with the problem size. The larger the body number, the more the tasks are, requiring more computational resources. In each process or thread, it is allocated with more body number to calculate, and the communication and synchronization efforts also increase.

- Relationship between speed & radius

Similar to the previous experiment, to explore the relationship between the speed and radius, I fixed the N=4 and other parameters to default. The radius varies from 5 to 25, increasing 5 each time. The results are shown as follows.

radius	MPI
5	240.224
10	220.559
15	233.417
20	222.459
25	220.697



As can be seen from the raw data and the graph, the absolute value of speed does not vary a lot (though it seems a lot in the graph, but it is just around 20 range). It is reasonable to conclude that radius have little relationship with the speed. Although it may affect the situations of collision (more collisions with larger radius), it seems not to affect the performance greatly.

Other parameters also make trivial influences to the performance, so I won't show more results here. The most important part should be the process/thread number part.

## 5 Summary

Here I will summarize what I have learned when writing assignment 3.

- Understanding of parallel computing. For example, different process shares different memory, and different threads can share the same global variables. The same pointer cannot always make effect. There is visible area to both CPU and GPU in CUDA.
- The problems to be noticed when writing parallel programs (process synchronization, process communication, thread creation and join, cost, **data race**)
- MPI & Pthread & OpenMP & CUDA library usage
- Coding ability improvement
- Analysis ability increases
- The ability to make use of searching engine
- Patience.....**The queue is sometimes too crowded!!**
- Cluster experience. Queuing to obtain the resources!
- By the way, tutorials are quite practical for writing the project and TA & USTF are warm-hearted in answering the questions in the wechat Group, Kudos!

That's all.