香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

CSC4005 Distributed and Parallel Computing

Report 3: A Comprehensive Implementation and Analysis of Many-Bodies Simulation by MPI, thread, OpenMP and CUDA

Submitted by:

Changwen LI

118010134

Submission date:

11th of November 2021

香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

Table of Contents

A Comprehensive Implementation and Analysis of Many-Bodies Simulation by MPI, thread, OpenMP and CUDA

1. Introduction:

This assignment requires students to implement many-bodies simulation by MPI (Message Passing Interface), thread, OpenMP and CUDA (Compute Unified Device Architecture). The report consists of 4 parts. In the first part, main concepts are introduced. The second part introduces the program architecture and the procedures to execute the program. In the third part, the analysis part, we test the programs with different number of processes, threads, CUDA cores and different length of data. We also compare the performance of implementation of MPI, thread, OpenMP, CUDA and sequential method.

1.1. Introduction to many-bodies simulation

Astronomers may need to calculate the movements of many celestial bodies. The movements can be inferenced by the law of universal gravitation put forward by Sir. Isaac Newton: $F=(G\times M_1\times M_2)/R^2$. In our approximated simulation, we only consider 2-dimentional movements. The influence of acceleration, velocity between every two bodies is completely independent. Thus, we can exploit the parallelism by calculating the gained acceleration, velocity between 2 stars concurrently or simultaneously.

1.2. Introduction to MPI

MPI (Message Passing Interface) is a protocol for passing messages between processes which is widely used in information transferring in parallel and distributed computing. In this project, the block point-to-point communication is used. The blocking sending, receiving, broadcasting and other information passing schemes provided by MPI are used.

1.3. Introduction to thread

Thread is the basic smallest sequence of programmed instructions that can be managed by a scheduler, which is typically a part of the operating system. Threads are executed concurrently with sharing resources such as memory and global variables. Compared to process, thread saves more resources in general. To program with threads, library <thread>, the standard library of C++ 12 is used in this project. There are 2 versions of thread codes in this project. One of them is the static version, where all threads are scheduled by program to do the jobs assigned by the programs before compiling and executing. Another version is the dynamic one, where threads' workload is not assigned by programs beforehand. They are assigned with jobs automatically once they are free. More details are presented in the discussion part.

1.4. Introduction to OpenMP

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. Programmers can use it without changing the sequential codes significantly. What is more, if the parallel version of codes goes wrong, it still can work as a sequential one automatically. Thus, the OpenMP provides great convenience and compatibility.

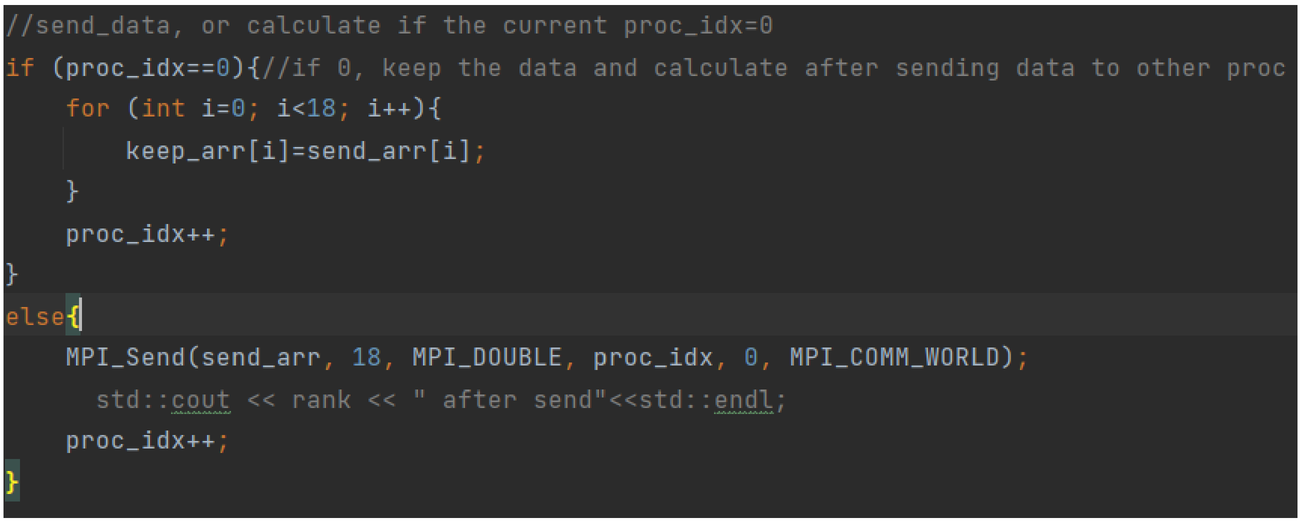
1.5. Introduction to CUDA

CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing unit (GPU) for general purpose processing – an approach called general-purpose computing on GPUs (GPGPU). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

2. Methods

2.1.Program implementation

The program is implemented in 5 ways (sequential, MPI, thread, OpenMP, CUDA).
For the MPI implementation, many data are sent to other processes by process 0 in an array. We first put the data of mass, acceleration, velocity, movement of 2 balls into this array. The configuration data is also written in this array so that they can be sent by MPI. The code of this part is presented in figure 2.1.1. After another process receives the configuration, the calculation starts. When the data has finished calculation, they are sent to process 0. The other process will then send back the result (delta values for acceleration, velocity and movement) to the first process. Then the process 0 will start to write all the data in the body pool. A tricky thing to pay attention to is, the MPI_Send may not necessarily provide ideal block transmission if the data size is large. Sometimes the program will continue once the MPI_Send function sends data to the buffer rather than other processes. Thus, here we add another variable as the flag to synchronize the data transmission. When process 0 finishes loading received data on the pool, we send a signal to other processes to continue next iteration of graph computing and drawing. The structural graph is attached in figure 2.1.2.



```
//send_data, or calculate if the current proc_idx=0
if (proc_idx==0){//if 0, keep the data and calculate after sending data to other proc
    for (int i=0; i<18; i++){
        keep_arr[i]=send_arr[i];
    }
    proc_idx++;
}
else{
    MPI_Send(send_arr, 18, MPI_DOUBLE, proc_idx, 0, MPI_COMM_WORLD);
      std::cout << rank << " after send"<<std::endl;
    proc_idx++;
}
```

Figure 2.1.1: Screenshot of configuration and data sending in MPI

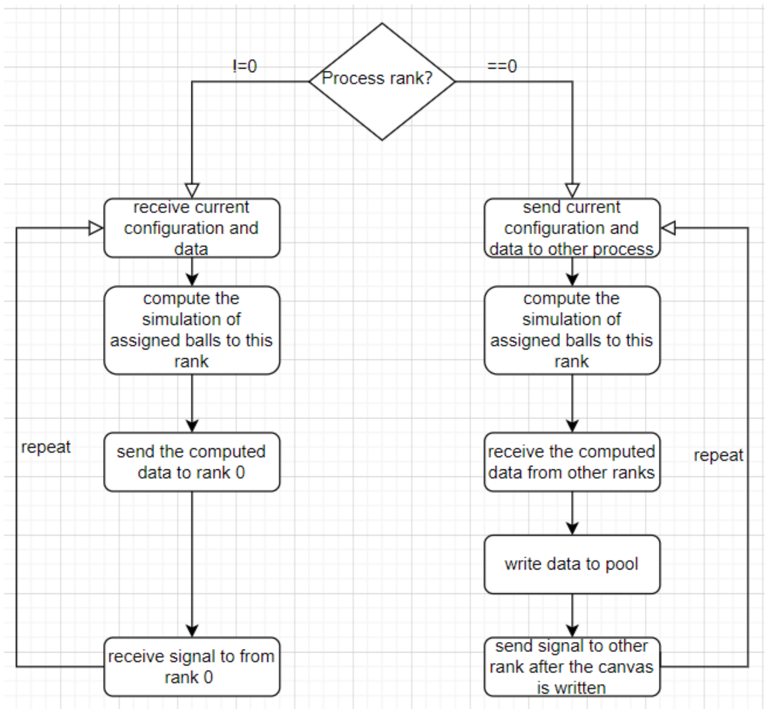
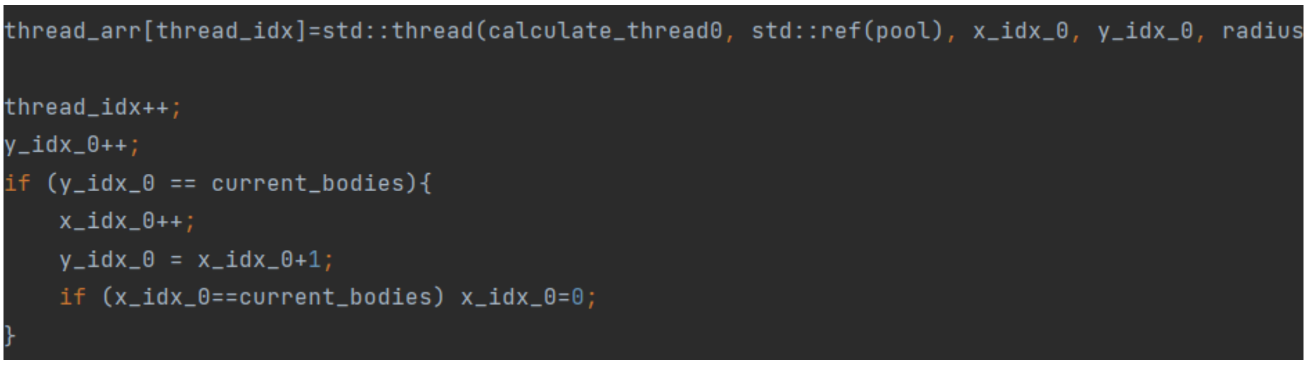

Process rank?
!=0
==0
receive current configuration and data
compute the simulation of assigned balls to this rank
send the computed data to rank 0
receive signal to from rank 0
repeat
send current configuration and data to other process
compute the simulation of assigned balls to this rank
receive the computed data from other ranks
write data to pool
send signal to other rank after the canvas is written
repeat

Figure 2.1.2: Brief architecture of MPI implementation

For the threading programming, we assign job to each threads in the codes. Each thread is assigned to calculate the acceleration, velocity and movements between two balls. In the main program, there is an iteration where the total iteration of it is all the non-duplicated combination of pairs of balls. In each iteration, we assign a thread. An important thing to notice is that we need to make sure the number (index) of the two balls is correct and changes according to the iteration. Here is the screenshot demonstrates how the index of these 2 balls are modified (shown in figure 2.1.3). The brief architecture of dynamic scheduling by thread is shown in figure 2.1.4.



```
thread_arr[thread_idx]=std::thread(calculate_thread0, std::ref(pool), x_idx_0, y_idx_0, radius

thread_idx++;
y_idx_0++;
if (y_idx_0 == current_bodies){
    x_idx_0++;
    y_idx_0 = x_idx_0+1;
    if (x_idx_0==current_bodies) x_idx_0=0;
}
```

Figure 2.1.3: Screenshot of ball indexes modification in thread

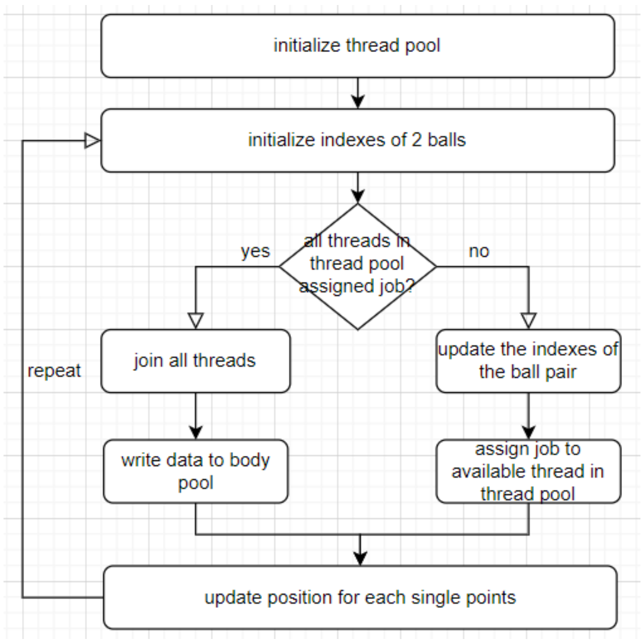
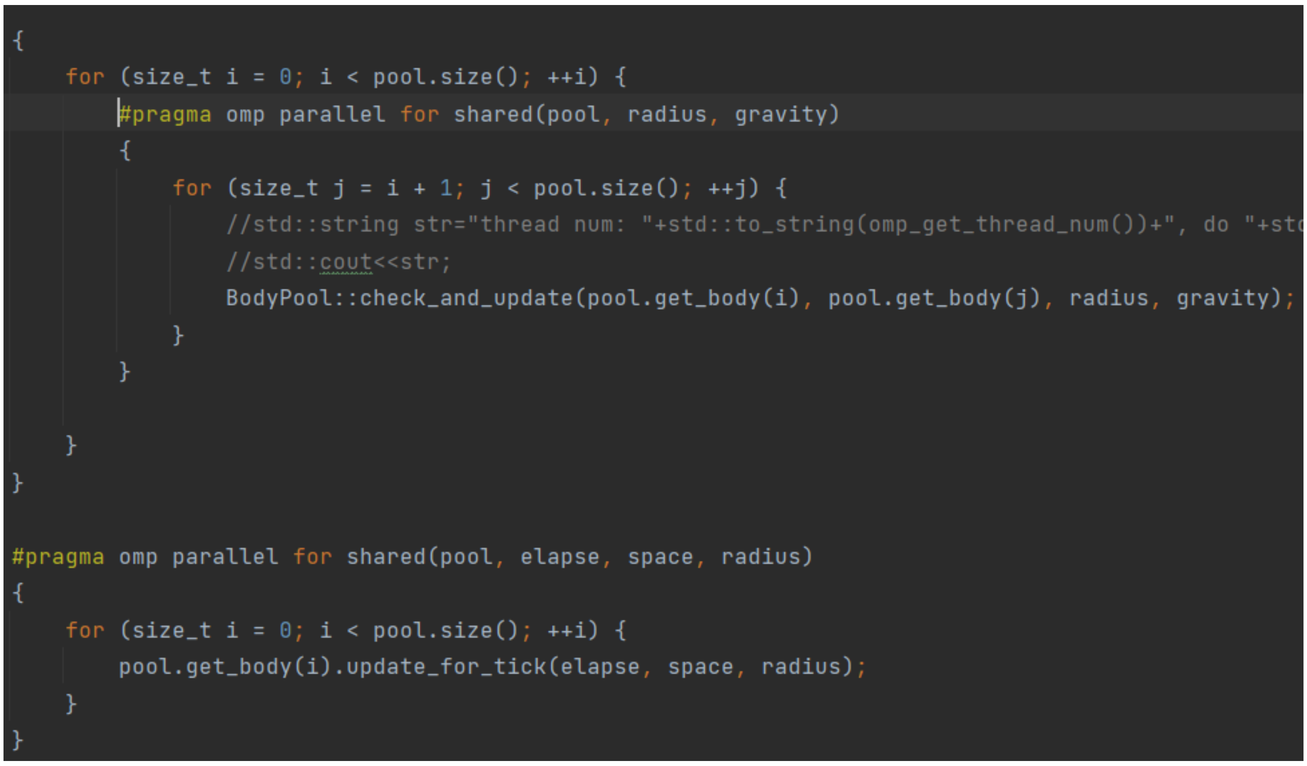

initialize thread pool
initialize indexes of 2 balls
all threads in thread pool assigned job?
yes
no
repeat
join all threads
update the indexes of the ball pair
write data to body pool
assign job to available thread in thread pool
update position for each single points

Figure 2.1.4: Brief architecture of threading implementation

For OpenMP implementation, it is quite easy since OpenMP is highly convenient. We only need to modify a little bit to the original sequential version. We may add some sentences to denote the for loop in the parallel paradigm like the screenshots shown in figure 2.1.5.



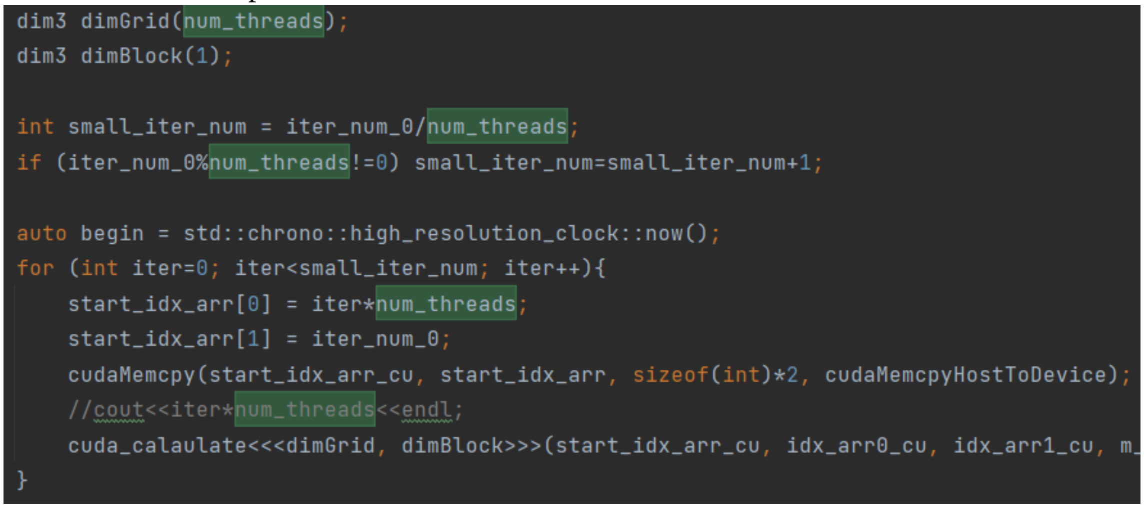
```
{
    for (size_t i = 0; i < pool.size(); ++i) {
        #pragma omp parallel for shared(pool, radius, gravity)
        {
            for (size_t j = i + 1; j < pool.size(); ++j) {
                //std::string str="thread num: "+std::to_string(omp_get_thread_num())+", do "+st
                //std::cout<<str;
                BodyPool::check_and_update(pool.get_body(i), pool.get_body(j), radius, gravity);
            }
        }

    }
}

#pragma omp parallel for shared(pool, elapse, space, radius)
{
    for (size_t i = 0; i < pool.size(); ++i) {
        pool.get_body(i).update_for_tick(elapse, space, radius);
    }
}
```

Figure 2.1.5: Brief architecture of OpenMP implementation

For CUDA implementation, we follow the classical paradigm of data flow in CUDA programming, which is CPU-GPU-CPU. The data we need to pass from the CPU to GPU are the data stored acceleration, velocity, mass and movement. We also need to pass some configuration number to the GPU so that the thread inside GPU knows which number in acceleration, velocity, mass and movement arrays to calculate. After calculation by GPU, the changed value will be written in another array. This array will be later transformed to CPU to update the pool. The screenshot 2.1.6 shows the screenshot of the kernel of CUDA implementation. And figure 2.1.7 demonstrates the basic architecture of CUDA implementation.



```
dim3 dimGrid(num_threads);
dim3 dimBlock(1);

int small_iter_num = iter_num_0/num_threads;
if (iter_num_0%num_threads!=0) small_iter_num=small_iter_num+1;

auto begin = std::chrono::high_resolution_clock::now();
for (int iter=0; iter<small_iter_num; iter++){
    start_idx_arr[0] = iter*num_threads;
    start_idx_arr[1] = iter_num_0;
    cudaMemcpy(start_idx_arr_cu, start_idx_arr, sizeof(int)*2, cudaMemcpyHostToDevice);
    //cout<<iter*num_threads<<endl;
    cuda_calaulate<<<dimGrid, dimBlock>>>(start_idx_arr_cu, idx_arr0_cu, idx_arr1_cu, m_
}
```

Figure 2.1.6: use of kernel in CUDA implementation

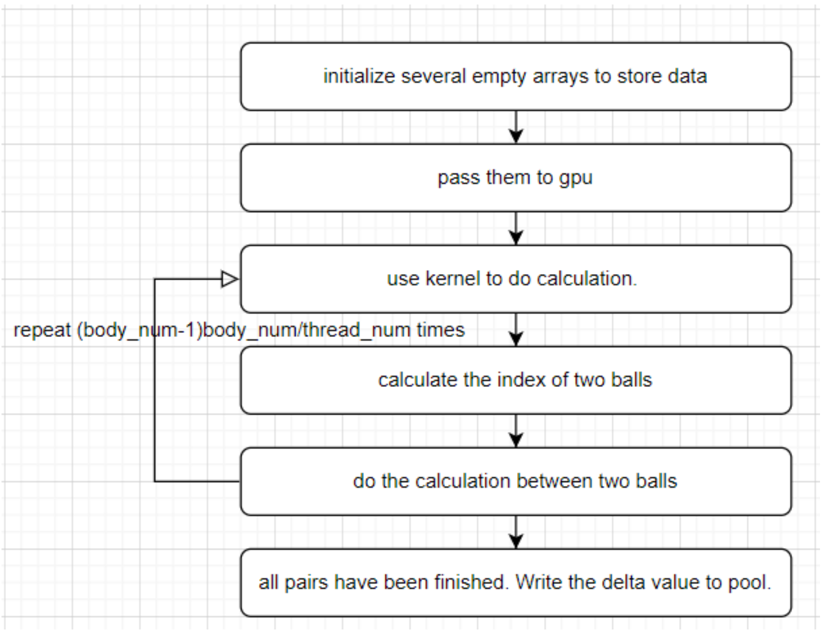

initialize several empty arrays to store data
pass them to gpu
use kernel to do calculation.
repeat (body_num-1)body_num/thread_num times
calculate the index of two balls
do the calculation between two balls
all pairs have been finished. Write the delta value to pool.

Figure 2.1.7: basic architecture of CUDA implementation

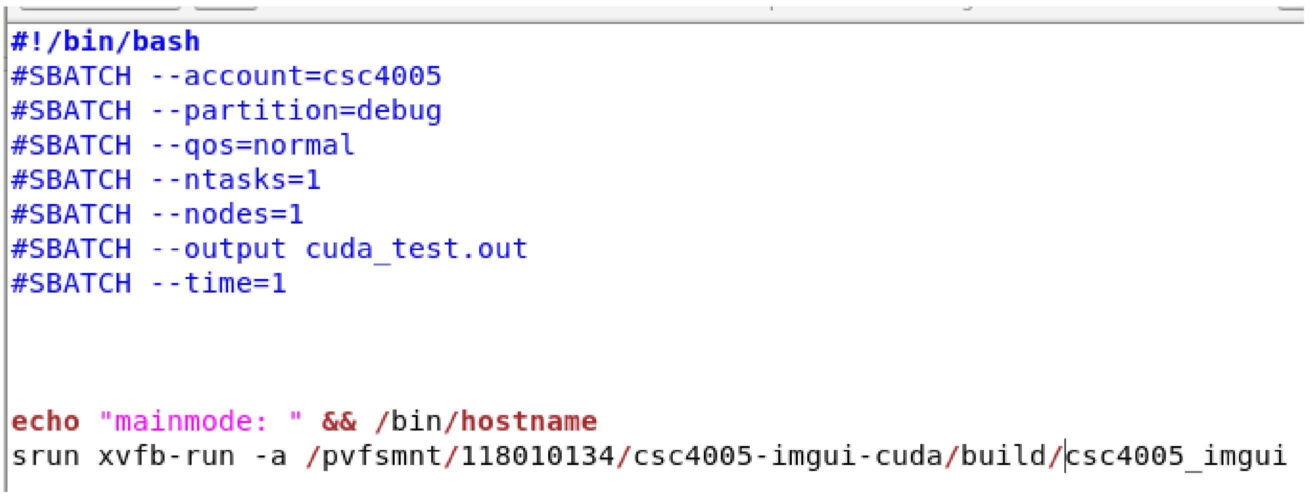
2.2.Procedures to execute the program

The project is based on templates provided by TAs and it is already compiled. There are 5 documents, which provides MPI, thread, OpenMP, CUDA and sequential.

The source files are under the /src directory. The executables are under each /build directory. To execute the programs, you may use either local host or the Slurm platform.

If you are using a localhost, the command to run the executables for MPI version is $mpirun -np num_of_process ./csc4005_imgui$. The command to run the executables for thread, CUDA and OpenMP version is $./csc4005_imgui thread_num bodies$. Please notice that the command is slightly different from the original version provided on BB. I modify the program so that the program can take two integers as input in command. If you forget to type an integer input, the program can still be executed with a default number of threads and bodies. The command to run the executables for sequential version is $./csc4005_imgui$. You are recommended to execute the program on local host if you want to see the graphic results because the graphic user interface is extremely slow on Slurm.

I recommend you to execute the programs on Slurm if you only want to obtain the experiment data. If you are using Slurm, then command approach is not the best approach to execute them on Slurm. The recommended one is using batch approach. There is a batch file (run_a2.sh) in each folder. You can directly execute them if you want. Please modify the parameters in the batch file to suit your requirements. The number after ntasks indicates the number of processors, the number after nodes indicates the number of nodes you want. You can also modify the information after output to denote the name of the file storing the printed information. The command starts with "xvfb" is the same as the commands in localhost (except there is no need to state the number of processes). Modify it if it is necessary for you. An example of batch file is shown in figure 2.2.1. After modifying the batch file, use command $sbatch batch_name.sh$ to run the batch files (You may need to wait in queue).



```
#!/bin/bash
#SBATCH --account=csc4005
#SBATCH --partition=debug
#SBATCH --qos=normal
#SBATCH --ntasks=1
#SBATCH --nodes=1
#SBATCH --output cuda_test.out
#SBATCH --time=1



echo "mainmode: " && /bin/hostname
srun xvfb-run -a /pvfsmnt/118010134/csc4005-imgui-cuda/build/csc4005_imgui
```

Figure 2.2.1: An example of batch file for executable

2.3.Methods of experiments and data collection

The project tests the performance of the calculations with different number of processes/threads and different data size. There are mainly three set of experiments conducted in this project.
In the first set of experiment, the performance of different data size with the same number of processes/threads are tested. The fixed number of processes/threads is 5, with data size of 20 balls, 40 balls and 80 balls. In the second set of experiment, different number of threads/processes are used to test on same size of data. 2 threads/processes, 4 threads/processes, 8 threads/processes, 16 threads/processes, 32 threads/processes, 38 threads/processes are used to test on 40 balls. In the third set of experiment, parallel computing with MPI and thread and OpenMP and CUDA and sequential computing are being compared.

3. Analysis

3.1. Speed-up analysis:

Assume that the workload is fixed. We can make this assumption since most of the experiments are conducted with same data size. Then we may use Amdahl's Law to approximate the speed up. According to Amdahl's Law, we have $S(n) = \frac{n}{1+(n-1)f}$ where n represents the number of nodes and f represents the percentage of sequential part. When the parallel part of the codes takes the majority, the speed up should be closed to n.

3.2. Experiment result analysis

3.2.1. Performance of different data size with the same number of processes/threads

In this experiment set, the number of processes is assigned to be a fixed number. The number of assigned processes/threads is 5. Then, test data with 20 balls, 40 balls and 80 balls are tested. The raw data of threads is shown in figure 3.2.1.1. The graph is very similar for other implementation. So here I only show one of them to demonstrate. The speed of execution drops when the data set gets larger. It is clear that the calculation speed drops as the data size increases. The reason is that each the more the balls, the more calculation is needed. Thus, more time is required. Another finding is that the time did not increase linearly but quadratically as the data size (number of balls) increase. The reason is that the computing time is $O(n^2)$ with the number of balls n.

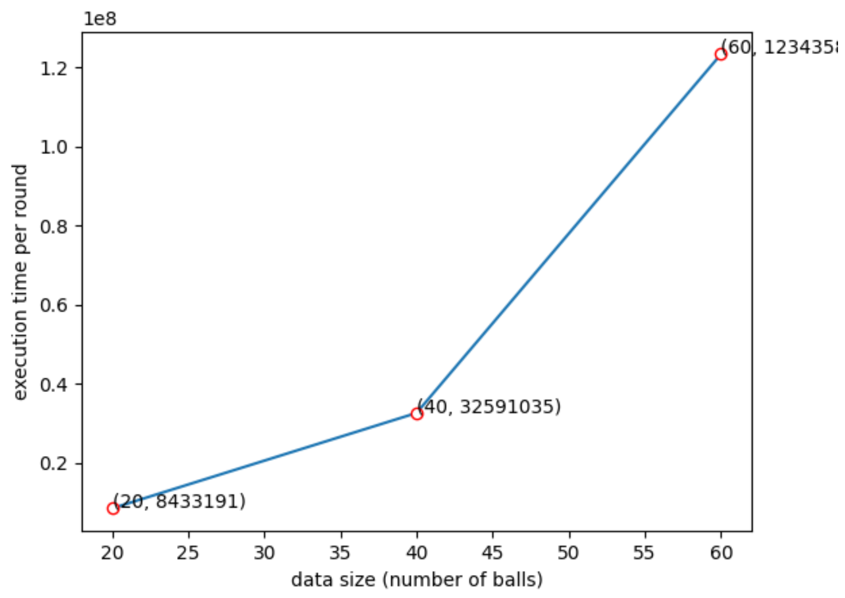


Figure 3.2.1.1: Execution time of different data size with 5 threads by thread

3.2.2. Performance of different number of processes/threads with the same data size

In this experiment set, the data size is assigned to be a fixed number. The data size is 800 pixels. Then, different number of cores are used to test on same size of data. 2 threads/processes, 4 threads/processes, 8 threads/processes, 16 threads/processes, 32 threads/processes and 38 threads/processes are used to test on 100 balls (20 balls for MPI). The raw data is shown in figure 3.2.2.1 for thread and the raw data for MPI is shown in figure 3.2.2.2. The raw data is shown in figure 3.2.2.3 for OpenMP and the raw data for CUDA is shown in figure 3.2.2.4. The experiment results fit the theoretical analysis of Amdahl's Law. The execution speed increases as the number of parallel units increase. An interesting fact is that from all figures, we see no decrease of running time from 32 to 38. Sometimes the running time even increase. This phenomenon account from the architecture of the cluster. Since each cluster only have 32 cores, which means there will be 2 card in use if the process number is 38. Thus, there will be communication between different cards. Hence, significantly more overhead time occurs.

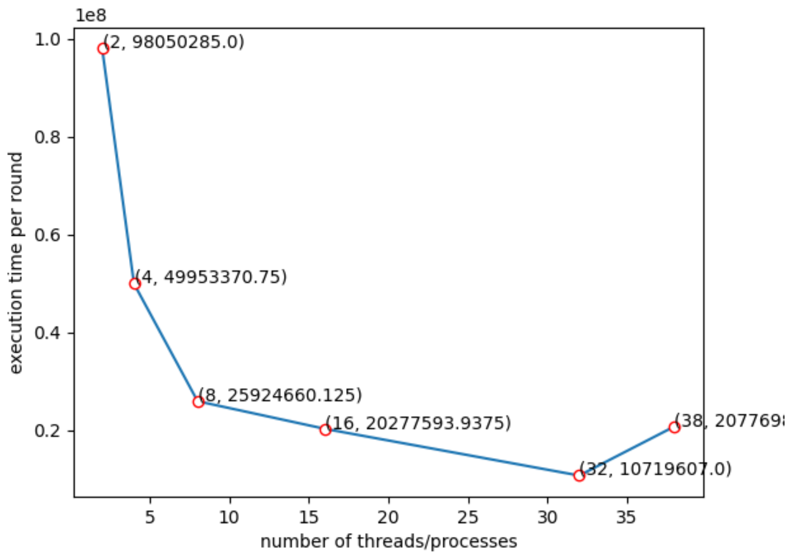


Figure 3.2.2.1: Execution time of different number of threads size with 800 pixels

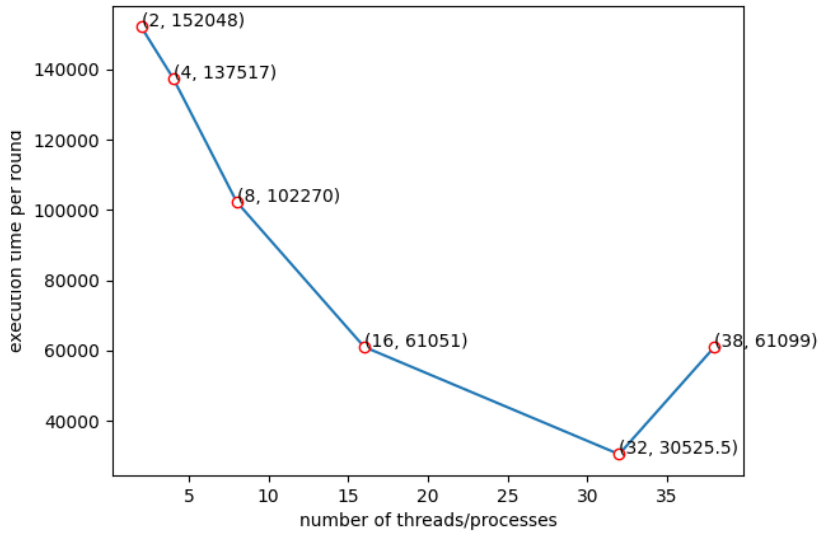


Figure 3.2.2.2: Execution time of different number of threads with 20 balls by MPI

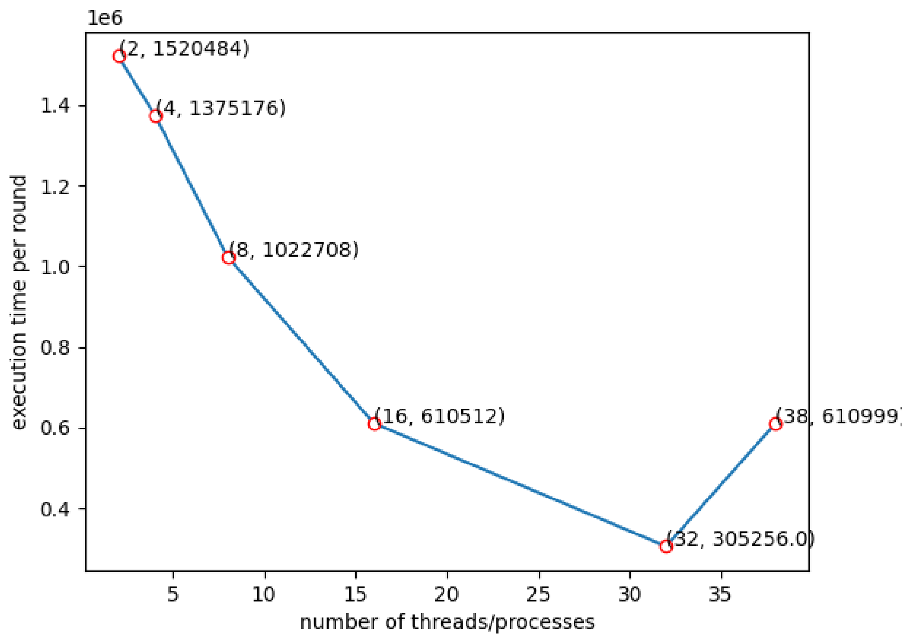


Figure 3.2.2.3: Execution time of different number of threads with 100 balls by OpneMP

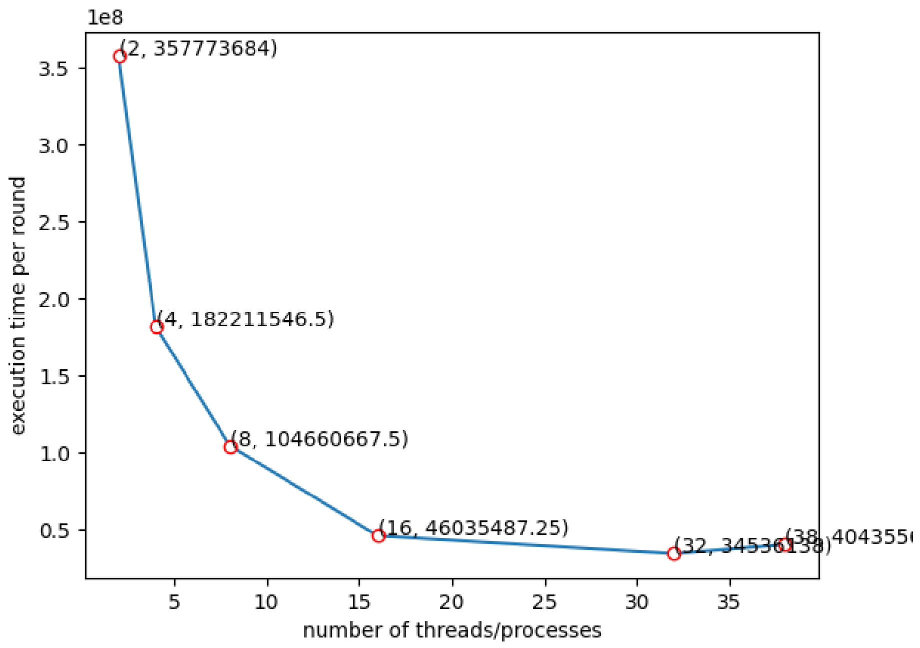


Figure 3.2.2.4: Execution time of different number of threads with 100 balls by CUDA

3.2.3. Performance comparison between sequential programming and parallel programming with MPI and thread

We set the data size as 100. And then measure the computation time of different data. The computation time for sequential version is 368283898ms. The time of the other implementation can be viewed from above graphs. Basically speaking, all parallel implementation is faster than the sequential one.

4. Conclusion

The project implements Simulation of N-bodies problem. The report has analyzed the performance of it with different configurations. Generally speaking, the experimental results fit the theoretical analysis.