香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

CSC4005 Distributed and Parallel Computing - Assignment 3 Report

WEN Zhenduo

118010323

zhenduowen@link.cuhk.edu.cn

Contents

1 Introduction **3**

1.1 Overview of N bodies simulation . . . 3

1.2 Softening . . . 4

1.3 Brief Introduction of MPI . . . 4

1.4 Brief Introduction of P-thread . . . 5

1.5 Brief Introduction of openMP . . . 5

1.6 Brief Introduction of CUDA . . . 7

2 Methodology **8**

2.1 Program Flow of MPICH Implementation . . . 8

2.1.1 Send structure in MPICH . . . 8

2.1.2 Control and Stop child processes . . . 10

2.2 Program Flow of P-Thread Implementation - Dynamic Scheduling . . . 10

2.2.1 Use pointers to avoid ”Lost local array” . . . 11

2.2.2 Prevent Data Racing . . . 12

2.3 Program Flow of OpenMP Implementation . . . 12

2.4 Program Flow of CUDA Implementation . . . 13

2.5 Program Flow of MPI-OPENMP Implementation . . . 13

3 Result Experiment and Analysis **15**

3.1 Experiment on Running Time . . . 15

3.2 Experiment on MPI and MPI-OpenMP running time . . . 17

3.3 Check Speed-Up and Efficiency . . . 18

4 Conclusion **19**

5 Appendix: How to run the submitted code? **21**

1 INTRODUCTION

This assignment report aims to realize and analysis N bodies simulation based on MPI(Message Passing Interface) Standards, P-thread Modules, openMP and CUDA. I also implement some goals including the openMP & MPI version and dynamic scheduling on P-thread, goals that, though not mentioned in the grading criteria nor in the assignment description, are very insightful to the parallel calculation of the N bodies simulation. In all, this report is both epagogic and heuristic in terms of parallel computing.

This introductory section provides a brief overview of N bodies simulation, MPI Standards, P-thread Modules, openMP and CUDA.

1.1 OVERVIEW OF N BODIES SIMULATION

In physics and astronomy, N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity (in our assignment).

N-body simulations are simple in principle, because they involve merely integrating the 6N ordinary differential equations defining the particle motions in Newtonian gravity. In practice, the number N of particles involved is usually very large (typical simulations include many millions, the Millennium simulation included ten billion, yet we only need 200 for final result shown) and the number of particle-particle interactions needing to be computed increases on the order of N2, and so direct integration of the differential equations can be prohibitively computationally expensive. Therefore, a number of refinements are commonly used.

Numerical integration is usually performed over small timesteps using a method such as leapfrog integration. However all numerical integration leads to errors. Smaller steps give lower errors but run more slowly. Leapfrog integration is roughly 2nd order on the timestep, other integrators such as Runge–Kutta methods can have 4th order accuracy or much higher.

1.2 SOFTENING

In our simulation, through, we need some method to deal with boundary cases: cases like collision with the walls or with other bodies. So we briefly talks about the refinement here.

Softening is a numerical trick used in N-body simulation to prevent divergence when bodies become too close together. This is obtained by modifying the regularized gravitational potential of each particle as:

$$\Phi = -\frac{1}{\sqrt{r^2 + \epsilon^2}}$$

rather than $\frac{1}{r}$ where ϵ is the softening parameter. The value of the softening parameter should be set small so as to keep the simulations not far from real.

In actual astrophysics, though, the distances between star clusters are so large and move in a relatively slow speed that astrophysicists rarely encounters the case that stars collide together.

1.3 BRIEF INTRODUCTION OF MPI

Message Passing Interface (MPI) is a series of standards designed to function on parallel computing architectures. Both point-to-point and collective communication are supported. Some basic concepts of MPI are:

- Communicator: Objects connect groups of processes in the MPI session. Each Communicator gives inner processes an independent identifier called rank and arranges its contained process in an ordered sequence.
- Point-to-point Communication: A series of significant MPI functions involve communication between two specific processes. This report mainly involves blocking send and receive **MPI_Send()** and **MPI_Recv()**.
- Collective Basics: Collective function involve communication among all processes in a communicator. This report utilizes Broadcasting **MPI_Bcast()**, Data Scattering **MPI_Scatter()**, and Data Gathering **MPI_Gather()**.

- Derived Data Types: Most MPI functions require specified data type defined in MPI. This is because MPI as an abstract standard aims to support heterogeneous environments where types might be represented differently. MPI predefines constants **MPI_INT, MPI_CHAR, MPI_DOUBLE** corresponding to **int,char,double**.

The sorting algorithm in this report is implemented on MPICH, a popular c++ realization of MPI Standards.

1.4 BRIEF INTRODUCTION OF P-THREAD

POSIX threads is an execution model that exists independently from any programming language. It allows a program to control multiple different flows of work and is implemented by manking calls to the POSIX Threads API. Some basic concepts of P-Threads are:

- Thread Management: creating children threads by calling **pthread_create()** and joining children back by calling **pthread_join()**.
- Synchronization: Different from MPI that has blocking communication designs, P-Thread achieves synchronization by setting mutual locks (mutex locks) on certain critical regions.

In this assignment, most data passing and communicating between threads are realized by passing pointers to the child process and set locks for safe data accessing.

1.5 BRIEF INTRODUCTION OF OPENMP

OpenMP(Open Multi-Processing) is an application programming interface that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. OpenMP is an implementation of pultithreading, which fundamentally follows the same rationale of P-Thread. The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed.

- omp for or omp do: used to split up loop iterations among the threads, also called loop constructs.
- sections: assigning consecutive but independent code blocks to different threads
- single: specifying a code block that is executed by only one thread, a barrier is implied in the end
- master: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

Since OpenMP is a shared memory programming model, most variavbles in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region, so data environment management is introduced as **data sharing attributes clauses** by appending them to the OpenMP directive.

- shared: the data declared outside a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- private: the data declared within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.
- default: allows the programmer to state that the default data scoping within a parallel region will be either shared, or none for C/C++, or shared, firstprivate, private, or none for Fortran. The none option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
- firstprivate: like private except initialized to original value.
- lastprivate: like private except original value is updated after construct.

- reduction: a safe way of joining work from all threads after construct.

1.6 BRIEF INTRODUCTION OF CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface that allows software to use certain types of graphics processing unit for computing.

Basically, the user need to allocate the memory space on devices by themselves, copy the host data to the device, and write device functions to apply computational steps to device data, then copy the result data back to host. The full processes can be described as following illustration shows:

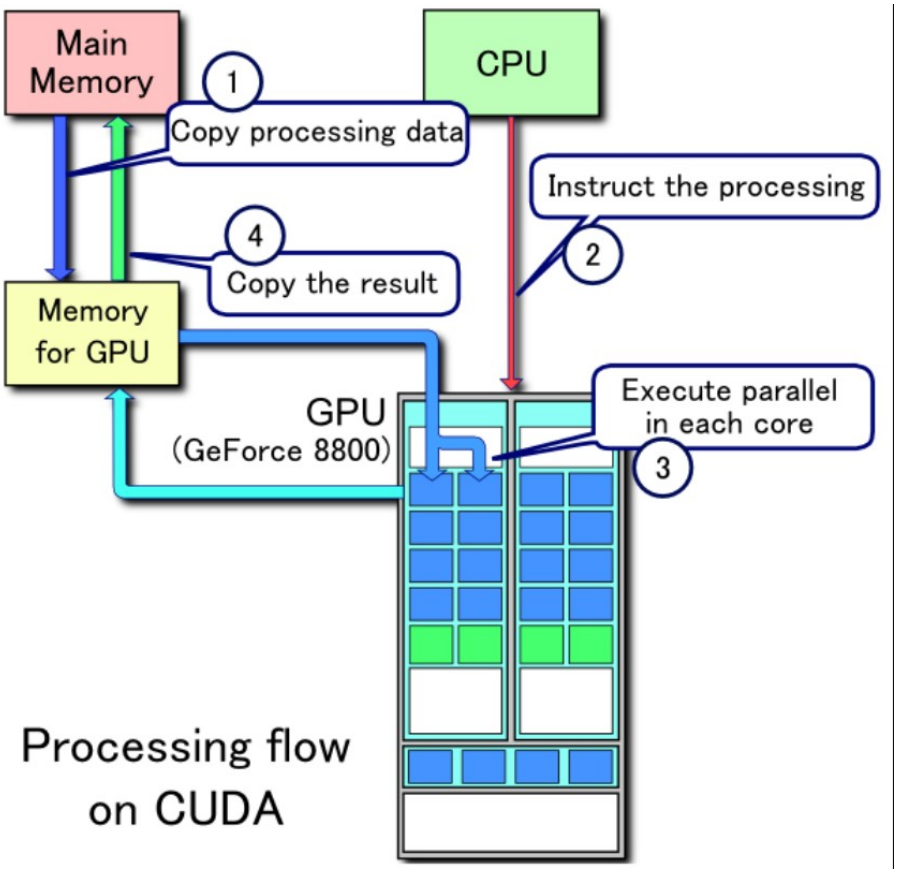

FIGURE 1: PROGRAM FLOW OF CUDA

2 METHODOLOGY

This section covers the technical implementation details of the assignment, including the general Program Flow, the specific implementation, and some Special Cases involve boundary specialization and exceptional case. Also, since dynamic scheduling is not mandatory in this assignment, the p-thread version implementation includes the dynamic scheduling realization.

2.1 PROGRAM FLOW OF MPICH IMPLEMENTATION

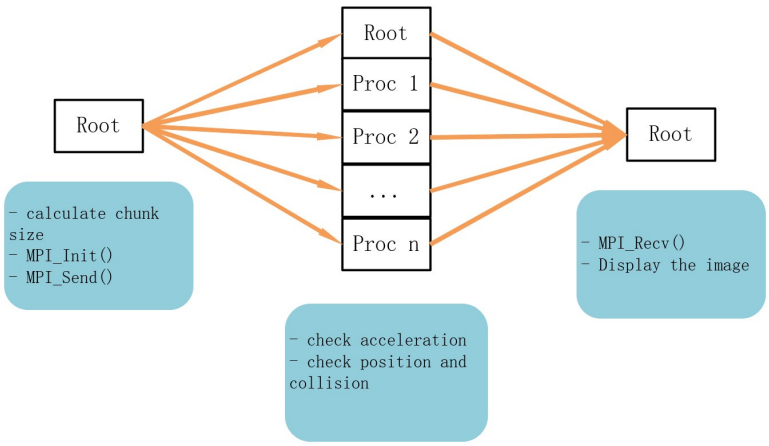

FIGURE 2: PROGRAM FLOW OF MPI IMPLEMENTATION

Figure 2 shows the program Flow of MPI Implementation. We first calculate the chunk size: the number of bodies need to be calculated by each process. Then we send the bodies pool to each of the processes and let them parallel-update the bodies pool. Then send the result pool back.

2.1.1 SEND STRUCTURE IN MPICH

We struct a buffer to send the pool information inter-process:

```
struct My_Buffer {
    double x[bodies];
```

```
    double y[bodies];
    double vx[bodies];
    double vy[bodies];
    double ax[bodies];
    double ay[bodies];
    double m[bodies];
};
```

This is simple. The hard time is to pass it through the MPI Interface. We declare a MPI_Data_Type according to the struct size:

The calculation on sub-processes is:

```
MPI_Recv(&buffer, 1, MPI_Pool, 0, mpi_tag, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
...
for (size_t i = start_index; i < start_index + sub_size; ++i) {
  for (size_t j = i + 1; j < pool.size(); ++j) {
    // update acceleration
    pool.check_and_update(pool.get_body(i), pool.get_body(j), radius,
                          gravity);
  }
}
for (size_t i = start_index; i < start_index + sub_size; ++i) {
  // update position and velocity according to acceleration
  pool.get_body(i).update_for_tick(elapse, space, radius);
}
...
MPI_Send(&buffer, 1, MPI_Pool, 0, mpi_tag, MPI_COMM_WORLD);
```

2.1.2 CONTROL AND STOP CHILD PROCESSES

Note that although we only want the calculation time of MPI-version implementation, it is still necessary to develop a scheduling system to control and stop child processes:

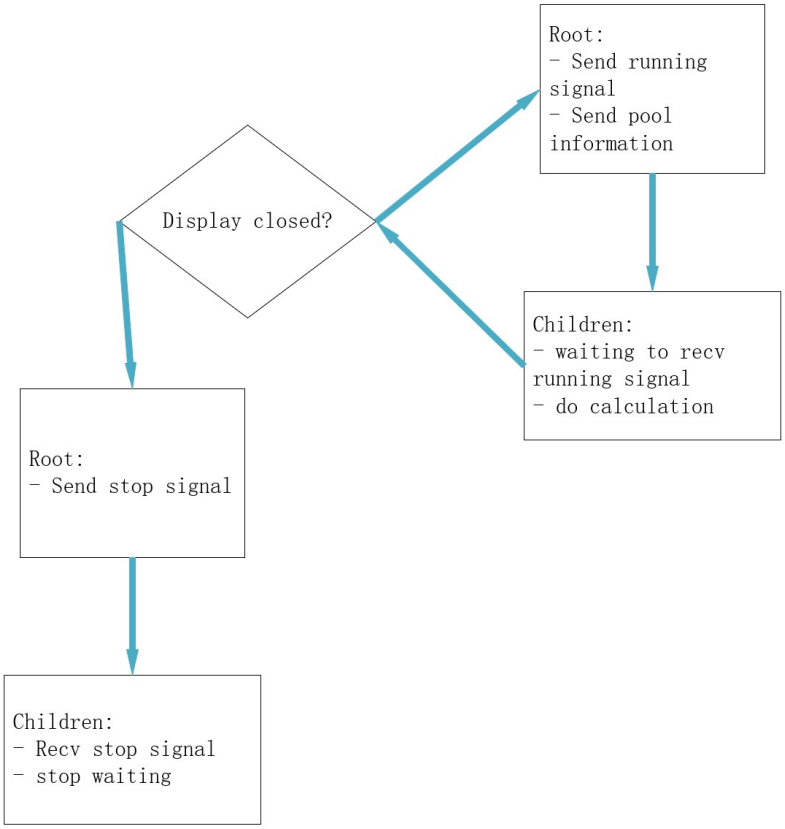

FIGURE 3: CONTROL AND STOP CHILD PROCESSES

2.2 PROGRAM FLOW OF P-THREAD IMPLEMENTATION - DYNAMIC SCHEDULING

To realize dynamic scheduling, we set a variable, job_remain = number of rows - number of threads, to store the number of uncalculated bodies. Now each thread in each iteration only calculate one row of the original canvas. And when this iteration ends, it will check whether $job\_remain > 0$, if so, it will start a new calculation iteration on row $job\_remain - 1$, otherwise it will exit and get joined by the parent thread. This design is shown in:

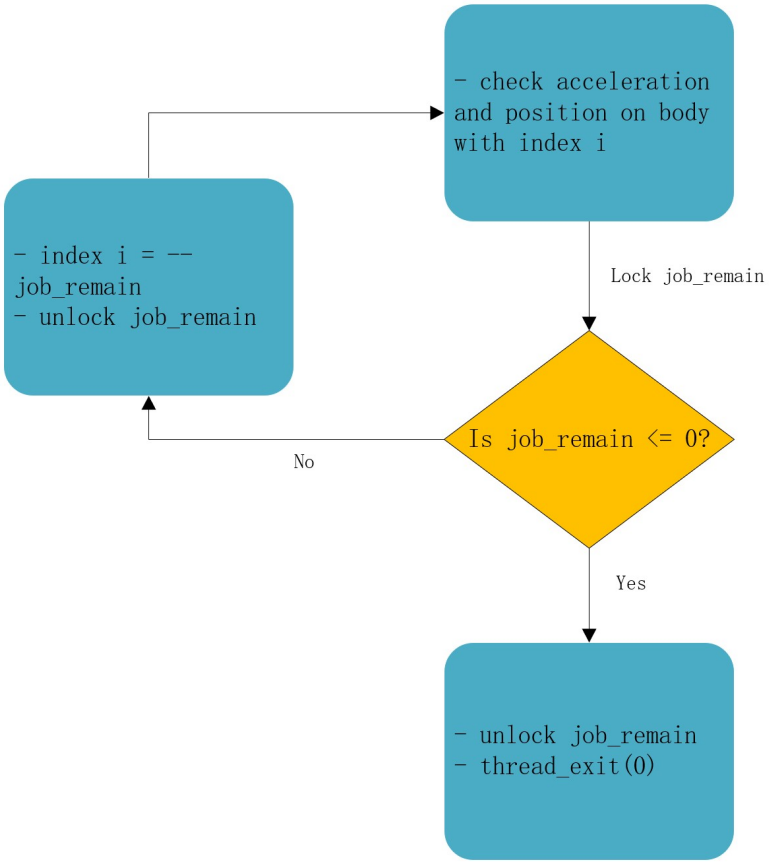

FIGURE 4: PROGRAM FLOW OF P-THREAD IMPLEMENTATION

2.2.1 USE POINTERS TO AVOID "LOST LOCAL ARRAY"

- When passing data from the parent thread to child thread, we only pass a pointer pointing to the start of part of pool that needs to be calculated by this child process.
- When retrieving data from the child thread to the parent thread, we do not need to pass anything since all calculation is written on the original canvas.

The reason to use pointers to directly written result onto the original canvas is the notorious problem of **"lost local array"**. Image we do initialize a local memory space to store the calculation and we pass it back (actually, We only passing back its head pointer). When the thread is joined back to the parent thread, the function stack collapses and all local variables are eliminated. **Although the parent process may successfully received the pointer pointing to the head of our local result array, that memory space is already freed.** So we only need to pass the pointer of original canvas, since the original canvas are safe when the child process is

joined.

2.2.2 PREVENT DATA RACING

To prevent data racing, we need to set lock on the variable **job_remain**. So as to make different threads always operating on different body indexes:

```
// dynamic scheduling lock
    pthread_mutex_lock(args->lock);
    if (*(args->task_remain) <= 0) {
      pthread_mutex_unlock(args->lock);
      pthread_exit(0);
    } else
      start_index = --*(args->task_remain);
    pthread_mutex_unlock(args->lock);
```

2.3 PROGRAM FLOW OF OPENMP IMPLEMENTATION

The OpenMP Implementation is designed to be similar to the MPI version, so as to make the interwined version easier to be realized. Instead of send pool information in buffers to different processes, in this implementation, we only do calculations on a single process. Yet use OpenMP pragma to run the loops in different threads.

The realization is simple:

```
#pragma omp parallel for collapse(2) num_threads(4)
      for (size_t i = 0; i < pool.size(); ++i) {
        for (size_t j = i + 1; j < pool.size(); ++j) {
          // update acceleration
          pool.check_and_update(pool.get_body(i), pool.get_body(j), radius,
                                gravity);
        }
```

```
}
```

2.4 PROGRAM FLOW OF CUDA IMPLEMENTATION

The CUDA Implementation works on the same communication rationale of MPI implementation: we store the body information in buffers, and send the buffers to devices, where body information will be checked and updated:

```
// call kernel
  cudaCheckBodies<<<(bodies + THREAD_NUMS_PER_BLOCK - 1) / THREAD_NUMS_PER_BLOCK, TH
  (x, y, vx, vy, ax, ay, m, args, bodies);


...
// kernel function (defined outside main but listed here)
__global__ void cudaCheckBodies(double *x, double *y, double *vx,
                                double *vy, double *ax, double *ay,
                                double *m, double *args, const int bodies)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  ...
  if (idx < bodies){
    ... [do checking on bodies]
  }
}
```

The program flow is shown in Figure 5.

2.5 PROGRAM FLOW OF MPI-OPENMP IMPLEMENTATION

The rationale is simple: MPI takes care of the communication between processes, where inside the process, OpenMP creates threads to do further parallel job. The Program Flow is:

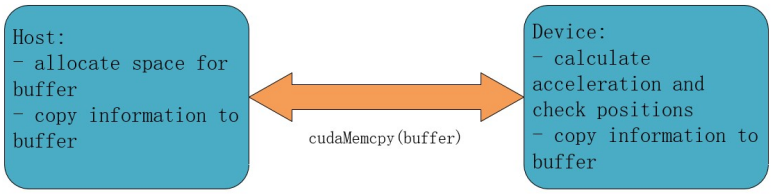

FIGURE 5: PROGRAM FLOW OF CUDA IMPLEMENTATION

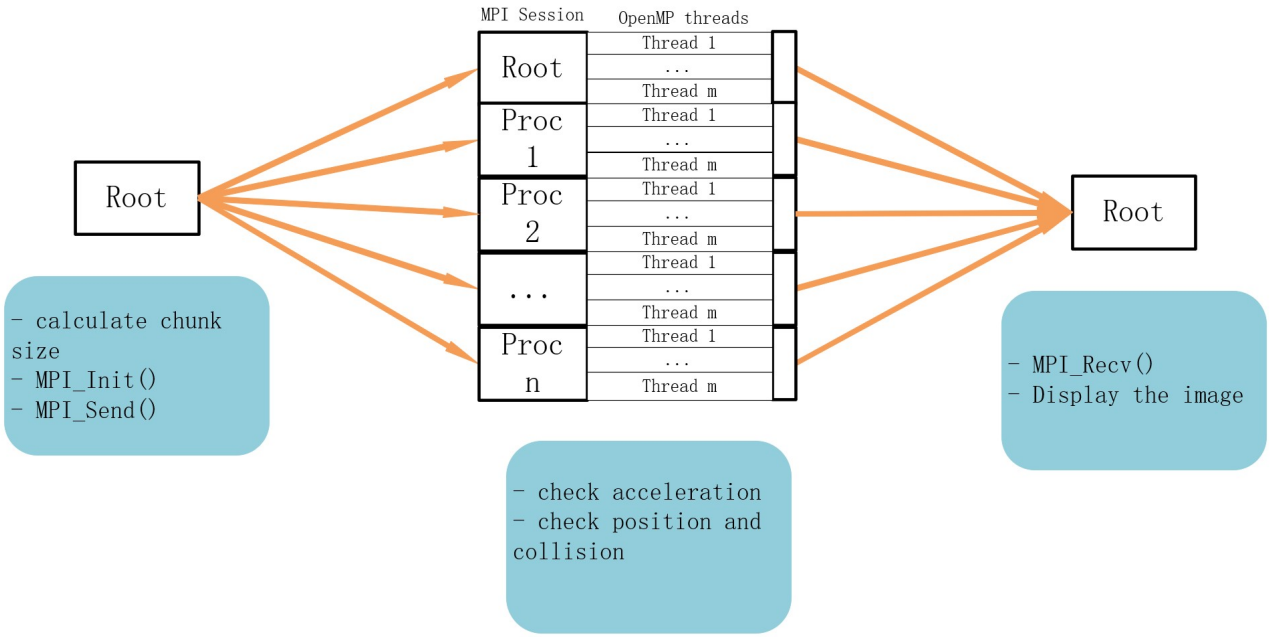

FIGURE 6: PROGRAM FLOW OF MPI-OPENMP IMPLEMENTATION

Now the calculation on sub-processes is:

```
    MPI_Recv(&buffer, 1, MPI_Pool, 0, mpi_tag, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    ...
#pragma omp parallel for collapse(2) num_threads(m)
    for (size_t i = start_index; i < start_index + sub_size; ++i) {
      for (size_t j = i + 1; j < pool.size(); ++j) {
        // update acceleration
```

```
            pool.check_and_update(pool.get_body(i), pool.get_body(j), radius,
                                  gravity);
          }
        }
#pragma omp parallel for num_threads(m)
        for (size_t i = start_index; i < start_index + sub_size; ++i) {
          // update position and velocity according to acceleration
          pool.get_body(i).update_for_tick(elapse, space, radius);
        }
        ...
        MPI_Send(&buffer, 1, MPI_Pool, 0, mpi_tag, MPI_COMM_WORLD);
```

3 RESULT EXPERIMENT AND ANALYSIS

This section covers a series of experiment on Running Time, Parallel Speed-Up and Efficiency. Note that the canvas size is fixed, so we only change the amount of interacting bodies.

If the size is large enough, we may apply the **tree method** to reduce the time complexity. The rationale of the tree method is to cut the canvas into segments and calculate only the interaction among bodies in each segment. It is okay to do so when the canvas size is large, since the gravity between two bodies with long distance is trivial. However, our canvas size is fixed and not very large, so it is illegal to do so.

3.1 EXPERIMENT ON RUNNING TIME

We let the iteration of calculating bodies interactions fixed to 50, set elapse=0.05 such that the bodies are fully moved, and test the running of the 100 iterations on average. We test the MPI/P-Thread/OpenMP/CUDA/MPI-OpenMP running time versus bodies numbers on different amount of processes/threads:

We see that 8 Processes runs the fastest and 16,32 processes run into overhead in MPI

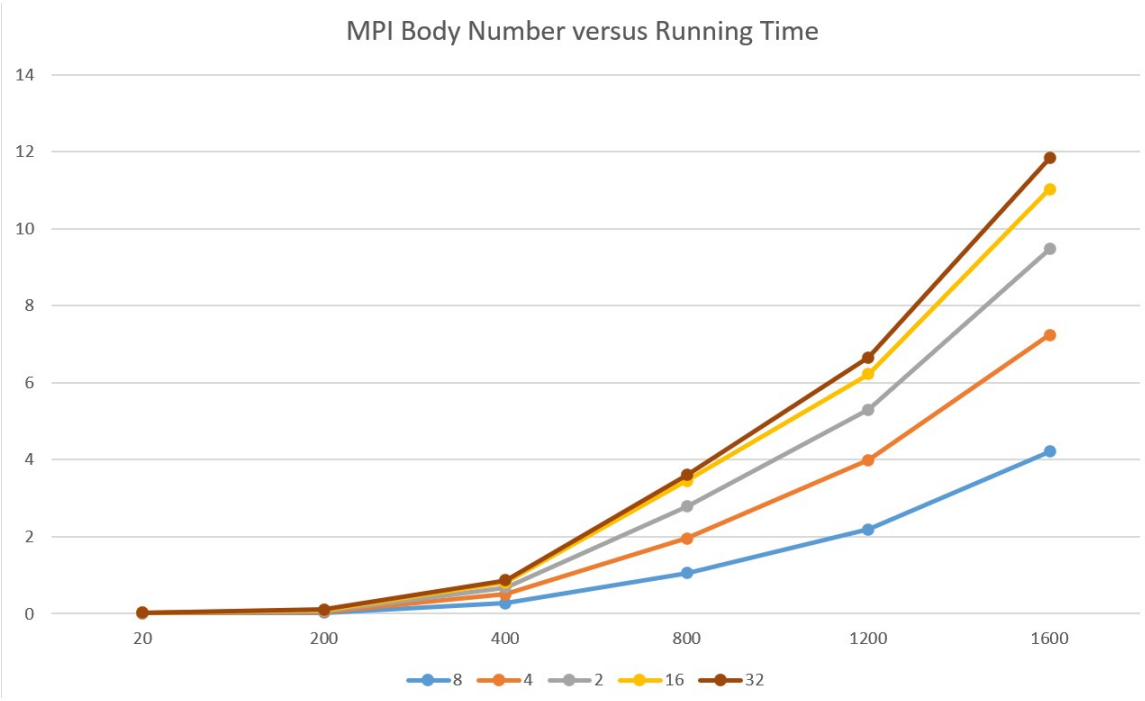

FIGURE 7: MPI BODY NUMBER VERSUS RUNNING TIME

implementation.

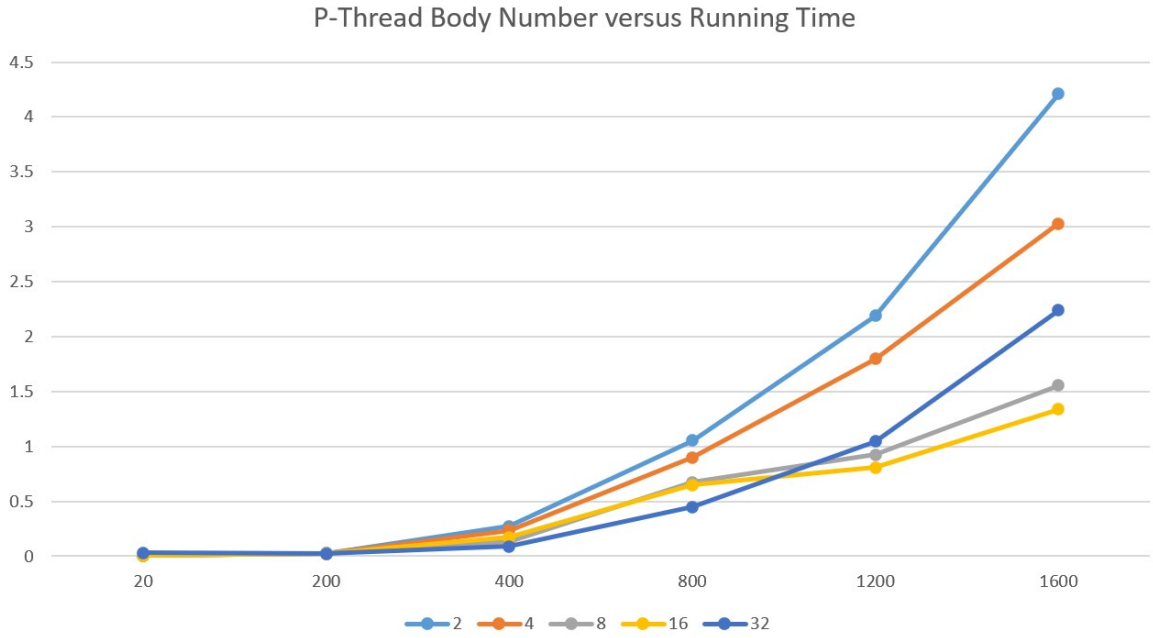

FIGURE 8: P-THREAD BODY NUMBER VERSUS RUNNING TIME

We see that 8,16 threads runs the fastest in pthread implementation.

We see that 8,16,32 threads runs the fastest in OpenMP implementation.

We see that 16,32 threads runs the fastest in CUDA implementation.

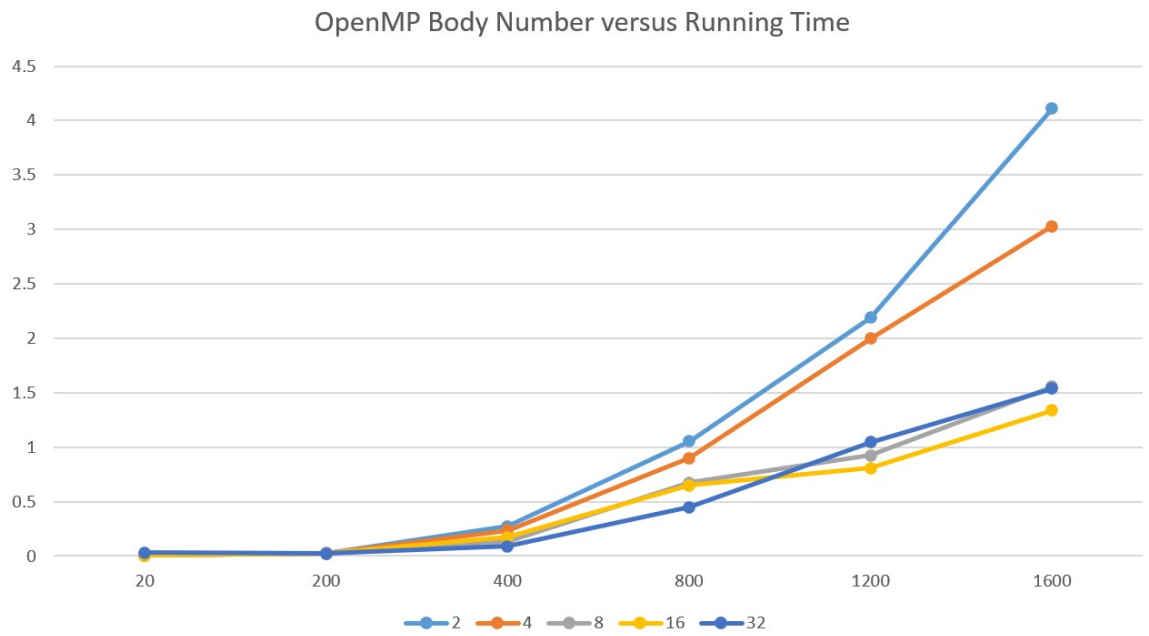

FIGURE 9: OPENMP BODY NUMBER VERSUS RUNNING TIME

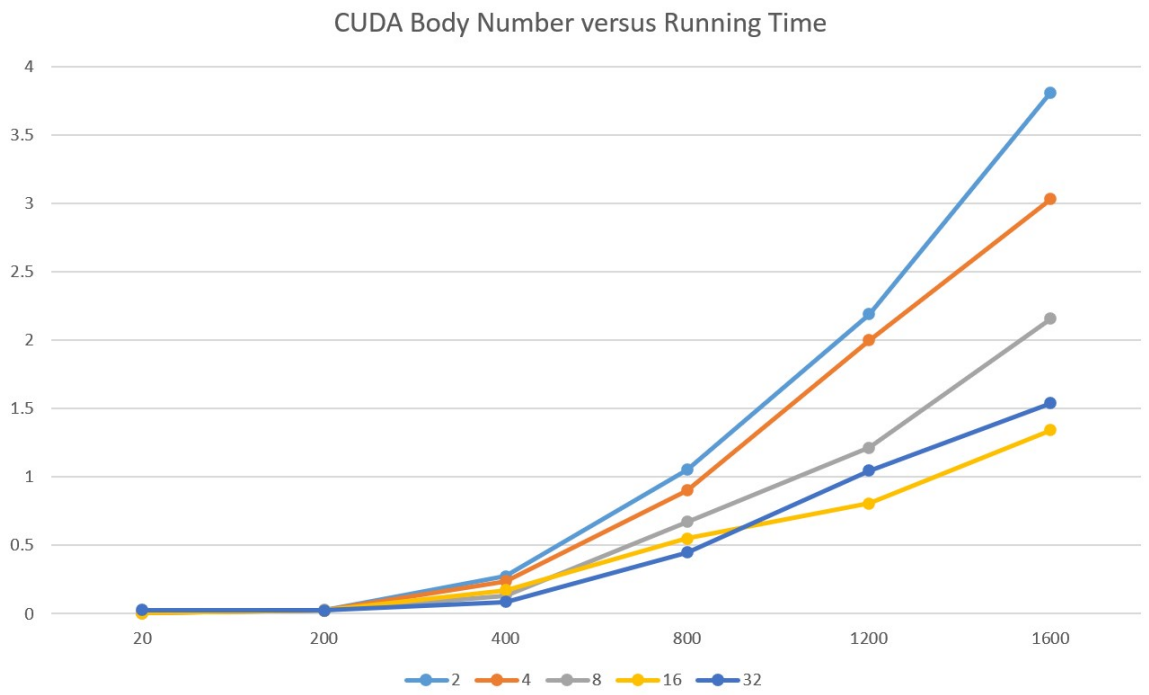

FIGURE 10: CUDA BODY NUMBER VERSUS RUNNING TIME

3.2 EXPERIMENT ON MPI AND MPI-OPENMP RUNNING TIME

The threads number of OpenMP is set to be 8 (the optimal number previously checked). We see that the running time become very similar when the processes number increases to 8. When the processes number equals to 4, OpenMP threads do decrease the runing time linearly. But

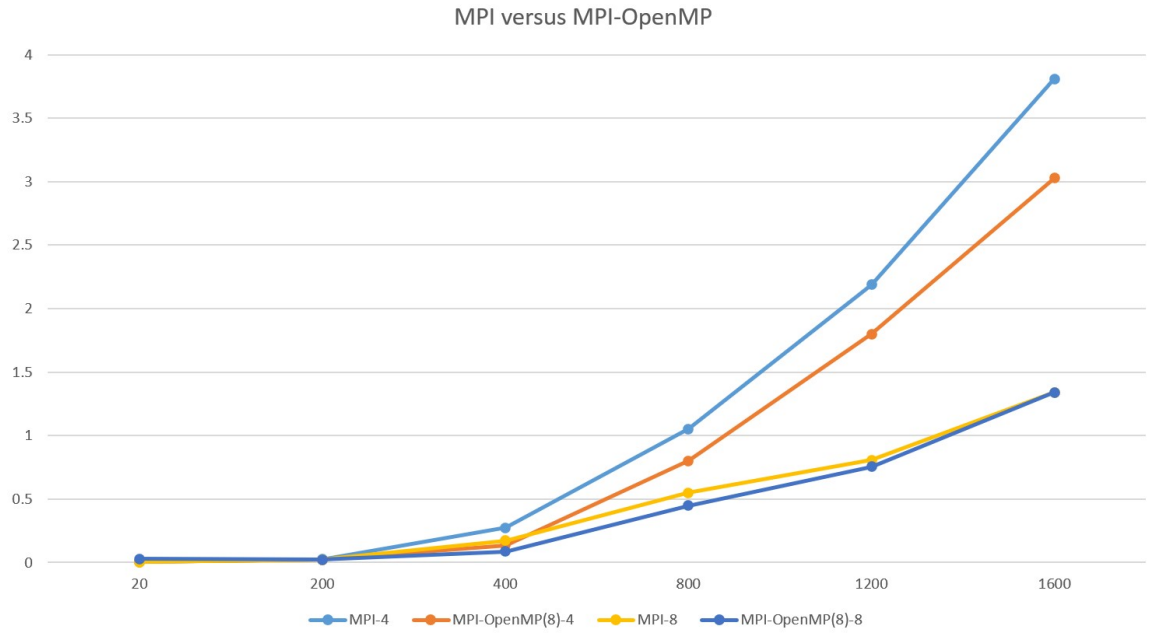

FIGURE 11: MPI VERSUS MPI-OPENMP

when processes number equals to 8, such improvement is trivial.

3.3 CHECK SPEED-UP AND EFFICIENCY

The speed-up factor is calculated by:

$$\text{Speed-Up} = 1 - \frac{1}{\text{SerialTime} + \frac{\text{ParallelTime}}{\text{p}}}$$

We fix body numbers be 1200, and visualize the speed up between these implementations:

The parallel efficiency is calculated by:

$$\text{Efficiency} = \frac{\text{SerialTime}}{\text{p} \times \text{ParallelTime}} \times 100\%$$

We fix body numbers be 1200, and visualize the efficiency between these implementations: The threads number of OpenMP is set to be 8 (the optimal number previously checked). Then we perform test in terms of thread numbers: We see that CUDA implementation maintains the highest efficiency

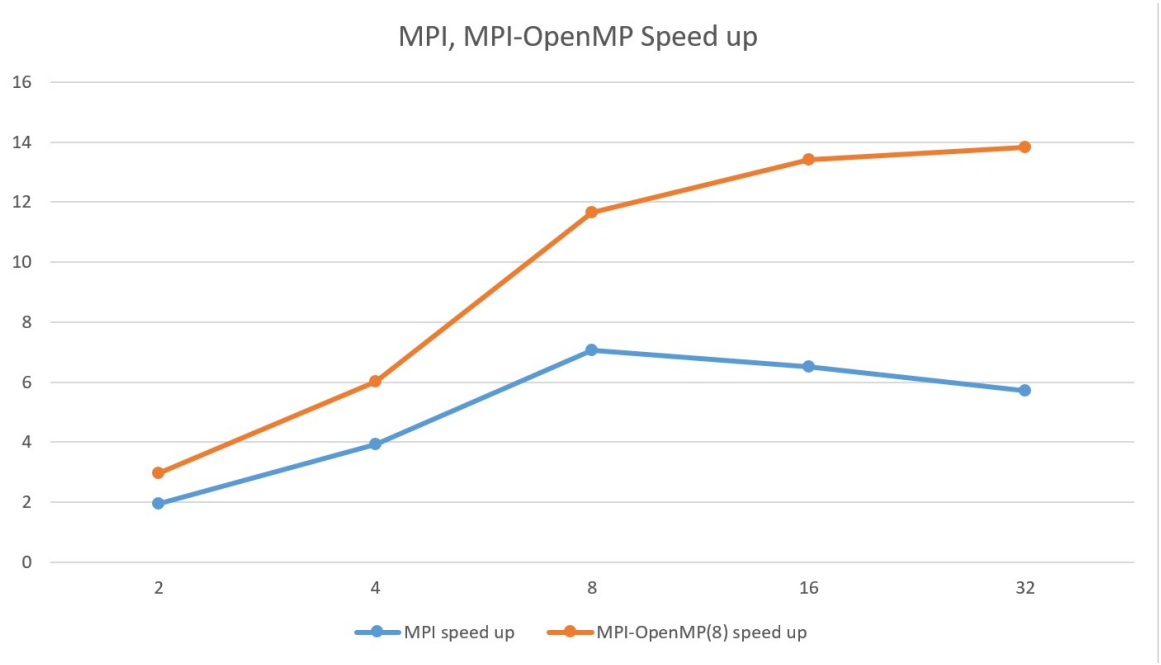

FIGURE 12: MPI SPEED-UP VERSUS MPI-OPENMP SPEED-UP

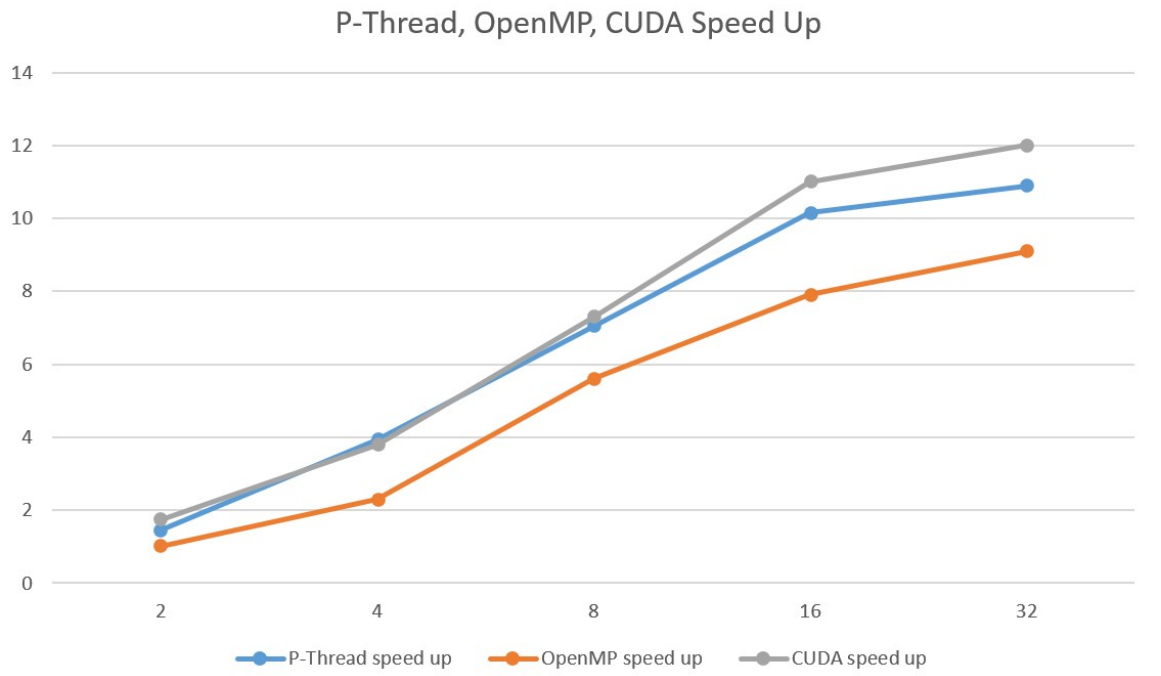

FIGURE 13: PTHREAD, CUDA, OPENMP SPEED-UP

4 CONCLUSION

This report utilizes the knowledge on MPI, P-Thread, OpenMP, and CUDA to implement an parallel version of N-Body Simulation. Some technical obstacles are encountered and overcome to achieve desirable performance: we design and show the program flow for MPI,

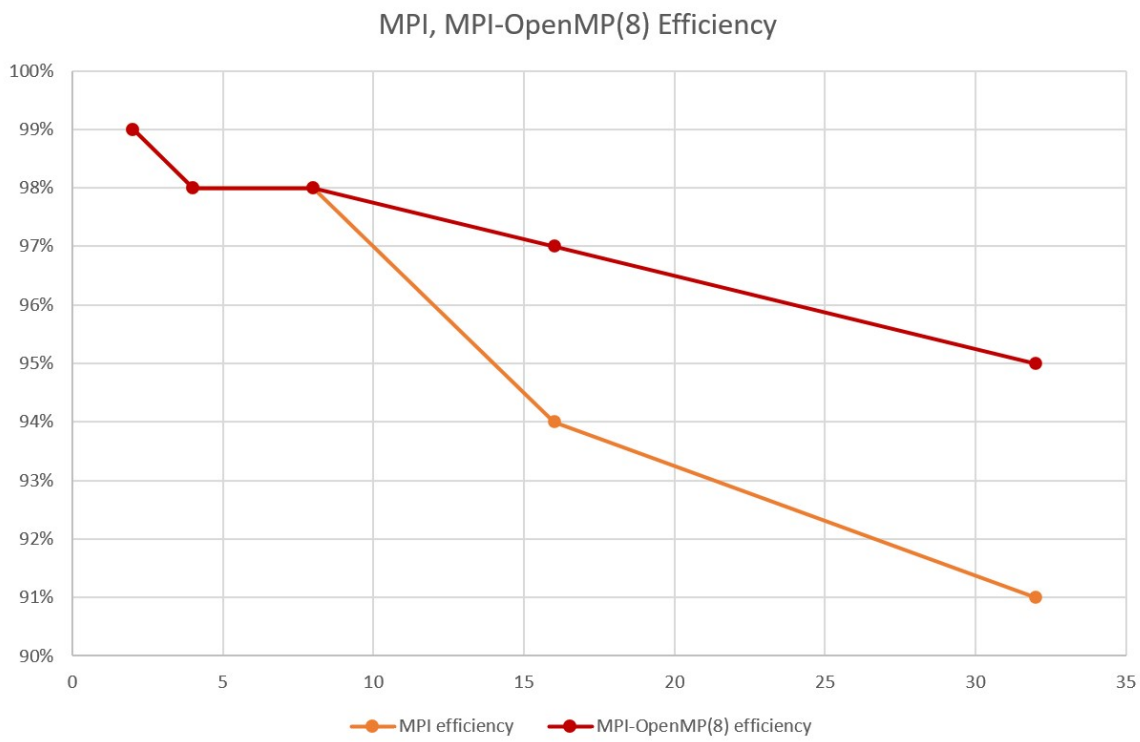

FIGURE 14: MPI EFFICIENCY VERSUS MPI-OPENMP(8) EFFICIENCY

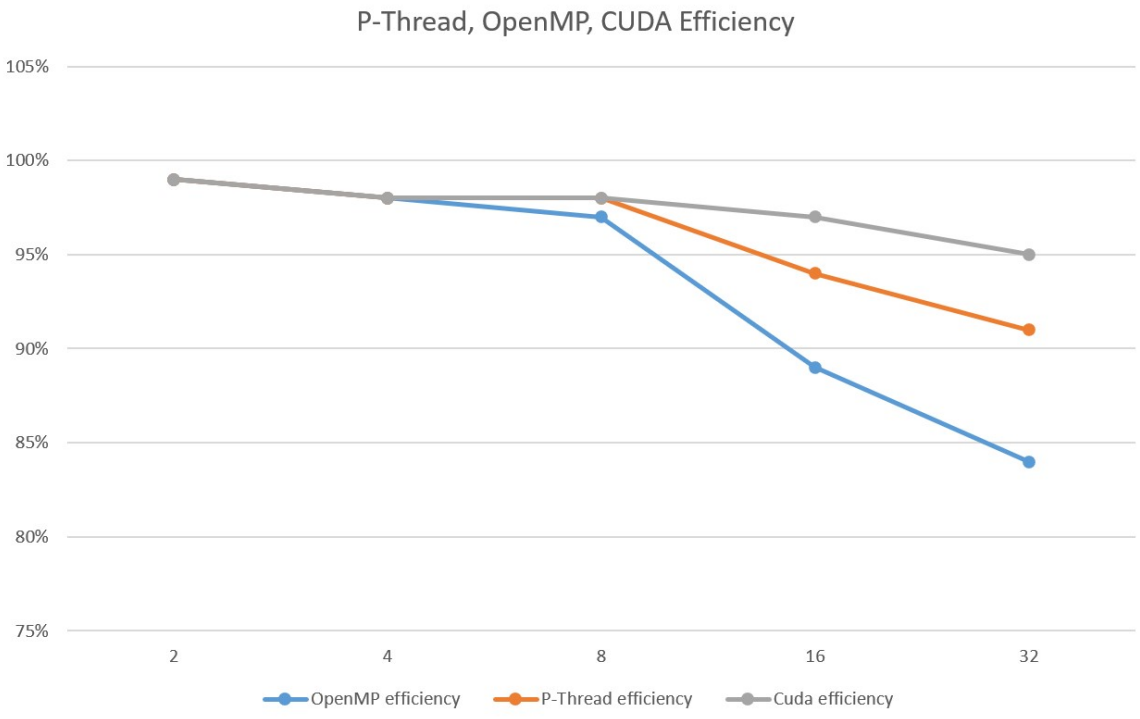

FIGURE 15: P-THREAD, OPENMP, CUDA EFFICIENCY

P-Thread, OpenMP, CUDA, and MPI-OpenMP Implementation. We also perform dynamic scheduling in p-Thread implementation. In the end, a series of experiments are conducted and verify the relationship between program performance and the number of processes

being used. We also compare the performance between MPI and MPI-OpenMP, OpenMP, Pthread, and CUDA. And the experiments support the relationship between the number of processes and the overall running time mentioned in the lecture. In all, the initial targets listed at the beginning of the report are majorly achieved.

5 APPENDIX: HOW TO RUN THE SUBMITTED CODE?

The source code is implemented on the given template. So it shouldn't be a problem in terms of Reproducibility. Compile the main project with:

```
mkdir build
cd build
cmake --build . -j4
```

under the build folder directory. It will generate the executable files of Sequential, MPI, P-Thread, OpenMP, MPI-OpenMP implementations.

The CUDA implementation is so different that it is listed in a different project. Compile the CUDA Project with:

```
mkdir build
cd build
source scl_source enable devtoolset-10
CC=gcc CXX=g++ cmake .. -DCMAKE_BUILD_TYPE=Release
make -j12
```