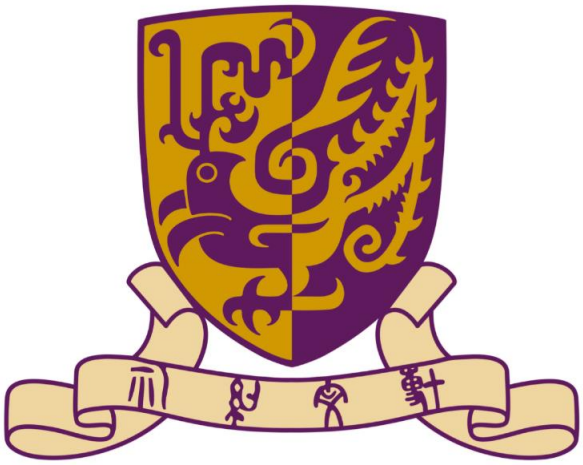

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

CSC4005 Distributed and Parallel Computing

Report 4:
A Comprehensive Implementation and Analysis of Heat Simulation by MPI, thread, OpenMP and CUDA

Submitted by:

Changwen LI

118010134

Submission date:

6th of December 2021

Table of Contents

A Comprehensive Implementation and Analysis of Heat Simulation by MPI, thread, OpenMP and CUDA

1. Introduction:

This assignment requires students to implement heat simulation by MPI (Message Passing Interface), thread, OpenMP and CUDA (Compute Unified Device Architecture). The report consists of 5 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. In the fourth part, the discussion part, we implemented the bonus. We compare the performance between using MPI+OpenMP and only using MPI.

1.1. Introduction to heat simulation

Heat simulation has very grand use in daily life. Physicists and engineers may need to do heat simulation very often. Here, we use Jacobi iteration to simulate heat radiation. The Jacobi iteration is computed by using two buffer arrays. One of them is used as read-only data, another one can be written and altered. The data in the array where alteration is allowed is independent from each other, and they are only influenced by the read-only array. After computation for all data in one array, the two buffers will be switched. Here we can exploit the parallelism since all written data are independent from each other.

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 6 ways (sequential, MPI, thread, OpenMP, CUDA, OpenMP and MPI).

For the MPI implementation, many data are sent to other processes by process 0 in an array. Rank 0 needs to send two arrays to other processes. The first one records the configuration data including room size, source coordinates, source temperature, border temperature. The second array we need to send is the data that need to be read by the corresponding process. The process will then create a new array dynamically, and then assign new value to it accoding to the reference information from the second array. Then it will send the new created array back to the first process. The other process also needs to send an array to indicate whether the data processed by it is stable. 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. The reason is that the routine will return once MPI_Send finishes its local action. 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.

```
// receive the data_back;
bool *stable_info = new bool[rank_num];
stable_info[0] = stable_or_not(stable, local_workload * room_size);
for (int i=1; i<rank_num; i++){
    int *local_stable = new int[1];
    MPI_Recv(local_stable, 1, MPI_INT, i, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // tag2
    if (local_stable[0] == 0) stable_info[i] = false;
    else stable_info[i] = true;
}

int starting_row_idx=local_workload;
for (int i=1; i<rank_num; i++){
    local_workload = workload_rows;
    if (remainder > 0 && i<remainder)  local_workload++;
    double *local_data = new double[local_workload*room_size];
    MPI_Recv(local_data, local_workload*room_size, MPI_DOUBLE, i, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    for (int j=0; j<local_workload; j++){
        for (int k=0; k<room_size; k++){
            data[(j+starting_row_idx)*room_size+k] = local_data[j*room_size+k];
        }
    }
    starting_row_idx += local_workload;
}

for (int i=1; i<rank_num; i++){
    ack_info[0]=1;
```

Figure 2.1.1: Screenshot of configuration and data sending in MPI

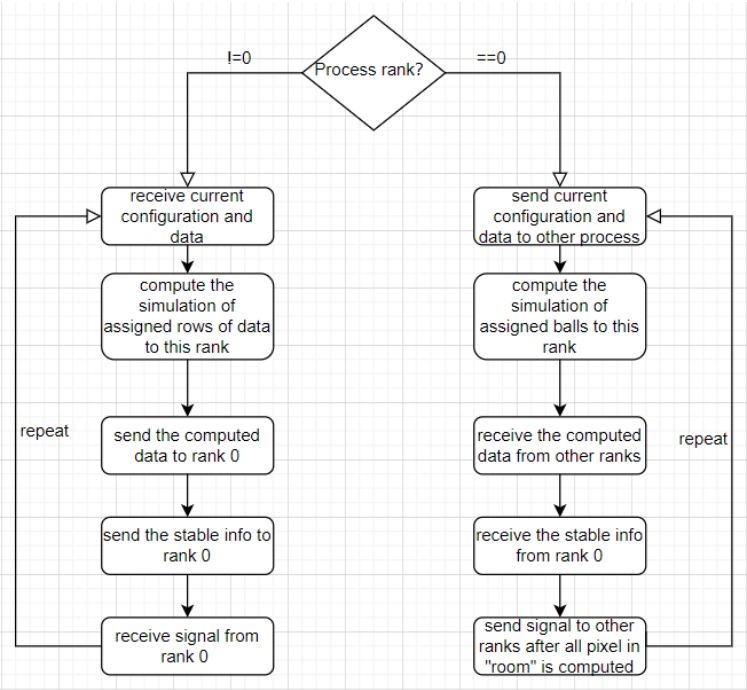

Figure 2.1.2: Brief architecture of MPI implementation

For the threading programming, we assign job to each threads in the codes. Each thread is assigned to calculate certain rows in the “room”. In the main program, there is an iteration to calculate all rows in the room. In each iteration, we assign a thread. An important thing to notice is that we need to make sure the number (index) of the rows assigned is correct. That is to say, the assigned rows should contain extra rows for reference. For instance, a thread calculate 3 rows might need to receive indexes with length 5, since it may need one additional row of data in the top and at the bottom. 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.

```
if (!finished) {
    for (int i=0; i<thread_num; i++){
        int current_workload = workload_rows;
        if (i<remainder) current_workload++;
        start_row_idx = current_row_idx;
        end_row_idx = start_row_idx + current_workload;
        thread_arr[i] = std::thread(calculate_thread, std::ref(current_state), std::ref(gr
        current_row_idx += current_workload;
    }

    for (int i = 0; i < thread_num; i++){
        thread_arr[i].join();
    }
    finished = stable_or_not(state_info, thread_num); // originally seq calculation here
    if (finished) end = std::chrono::high_resolution_clock::now();
```

Figure 2.1.3: Screenshot of row indexes modification in thread

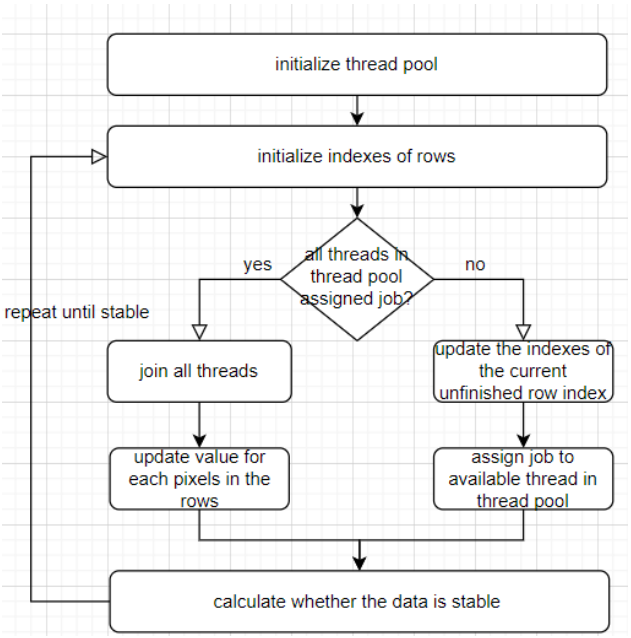

Figure 2.1.4: Brief architecture of threading implementation

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.

```
size_t j = 0;
switch (current_state.algo) {
    case hdist::Algorithm::Jacobi:
        #pragma omp parallel for private(j)
        for (size_t i = 0; i < current_state.room_size; ++i) {
            //debug use
            //std::cout<<std::to_string(omp_get_num_threads())+" "+std::
            for (j = 0; j < current_state.room_size; ++j) {
                auto result = update_single(i, j, grid, current_state);
                stabilized &= result.stable; //& one false, all false
                grid[{hdist::alt, i, j}] = result.temp;
            }
        }
        grid.switch_buffer();
        break;
```

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 configuration data and temperature data. After calculation by GPU, the changed value will be written in another array. This array will be later transformed to CPU to draw the room. In this program, we only use one SM since the SP (CUDA cores) is enough. And using only one SM can significantly reduce communication time. 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.

```
if (first) { // start timing
    first = false;
    finished = false;
    begin = std::chrono::high_resolution_clock::now();
}

if (!finished) {
    // need to return finished.
    cuda_cal<<<dimGrid, dimBlock>>>(buffer_num_g,state_g,stable_g,vec0_g,vec1_g);
    cudaMemcpy(vec0_c, vec0_g, sizeof(double)*room_size*room_size, cudaMemcpyDeviceToHost);
    cudaMemcpy(vec1_c, vec1_g, sizeof(double)*room_size*room_size, cudaMemcpyDeviceToHost);
    cudaMemcpy(stable_c, stable_g, sizeof(bool)*room_size*room_size, cudaMemcpyDeviceToHost);
```

Figure 2.1.6: use of kernel in CUDA implementation

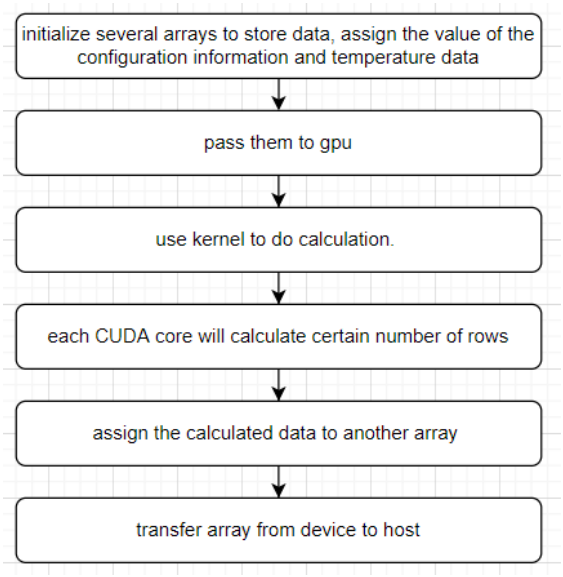

Figure 2.1.7: basic architecture of CUDA implementation

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 room_size $. The command to run the executables for thread, OpenMP version is $./csc4005_imgui thread_num room_size$. 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(s), the program can still be executed with a default number of threads and room size. The command to run the executables for sequential version is $./csc4005_imgui$. If you want to execute the CUDA program, the procedure is a little complicated. You need to run the CUDA program on the cluster. First, you need to type $salloc -N1 -t10$ to get the right to use the cluster. After that, type

$srun ./csc4005_imgui cuda_core_num room_size$. Then you may be able to run it without graphical exception. My CUDA program can successfully run without any bug. The video of running my CUDA program can be found in the /demo_video folder. 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 of CUDA

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 four sets 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 4, with data size for the room size is 200, 400, 800 and 1600. 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. The fourth set of experiment is about the bonus. It is discussed in the discussion part.

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.

However, the communication time can take a large proportion especially in this project. In this project, the computation workload is quite light. In every iteration, computation only takes *O(N)* time complexity (assume *N* is the data size). Since every data only need to be calculated once by adding 4 data and dividing 4. The data transmission cost is very high in this project especially for MPI and CUDA. Since memory in MPI is not shared, there will be large data transmission because all the temperature data need to be transmitted. In the CUDA part, data need to be transmitted from host to device, then from device to host. These will cost a lot of time.

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 4. Then, test data of room size is 200, 400, 800 and 1600. 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 the larger the room size is, the more calculation is needed. Thus, more time is required. Another finding is that the time did not increase linearly but quadratically as the room size increases. The reason is that the computing time is $O(n^2)$ with the room size n.

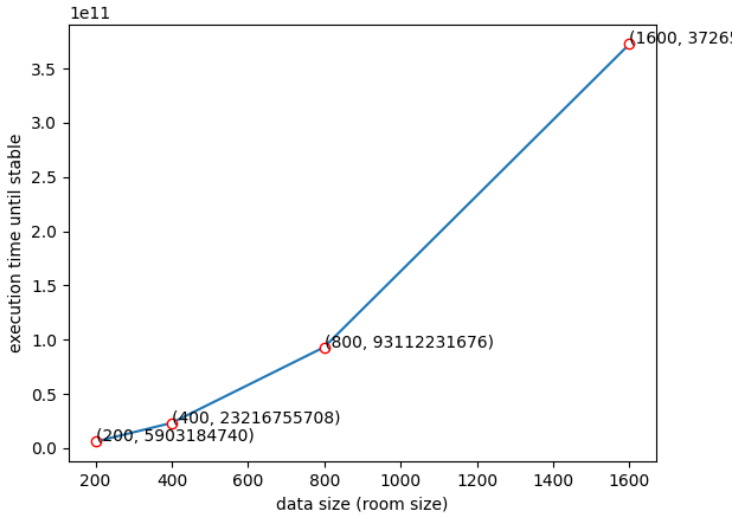

Figure 3.2.1.1: Execution time of different data size with 4 threads by thread

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 1600*1600 pixels (room size: 1600). 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. 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.
In the thread program, the program stabilizes faster as the number of threads increases. We can see that the slope of the graph is decreasing here because the overhead time increases when there are more threads since more threads need to be created and passed information.

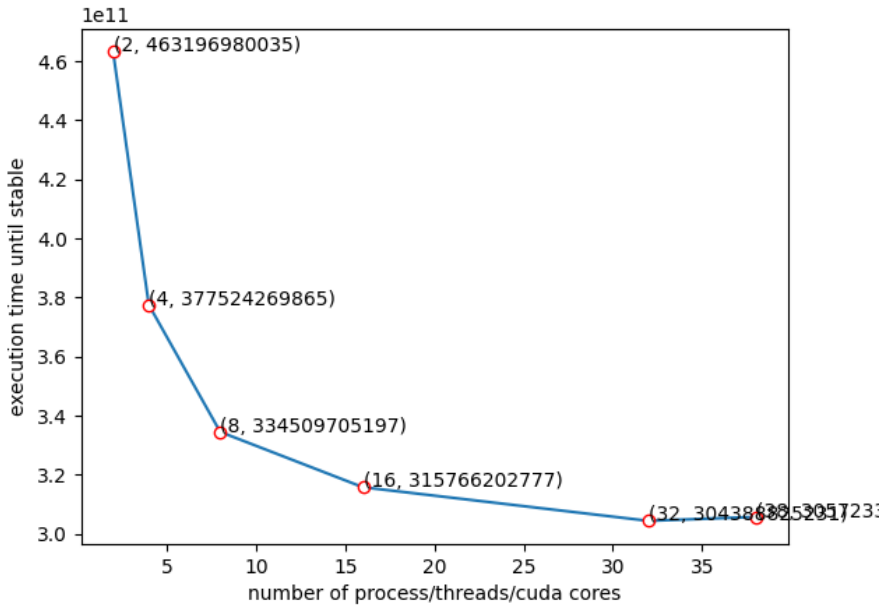

Figure 3.2.2.1: Execution time of different number of threads (Pthread method)

In the MPI program, the experiment result is a little abnormal. It is expected that the running time should decrease when the number of processes increases. However, in the graph shown below, the running time only decreases from 2 processes to 8 processes. Then the running time increases. When the number of processes increases from 32 to 38, the running time increase sharply. The reason for the increasing time is because of the mechanism of MPI. Memory cannot be shared in MPI, thus, there will be more communication in transferring data than Pthread. As mentioned before, the computation workload is light in this program. When the number of processes is very large, the communication time dominates the program, even longer than computation time. That is the reason for the increasing computation time. An interesting fact is that from all figures, we see no great increase of running time from 32 to 38. 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.

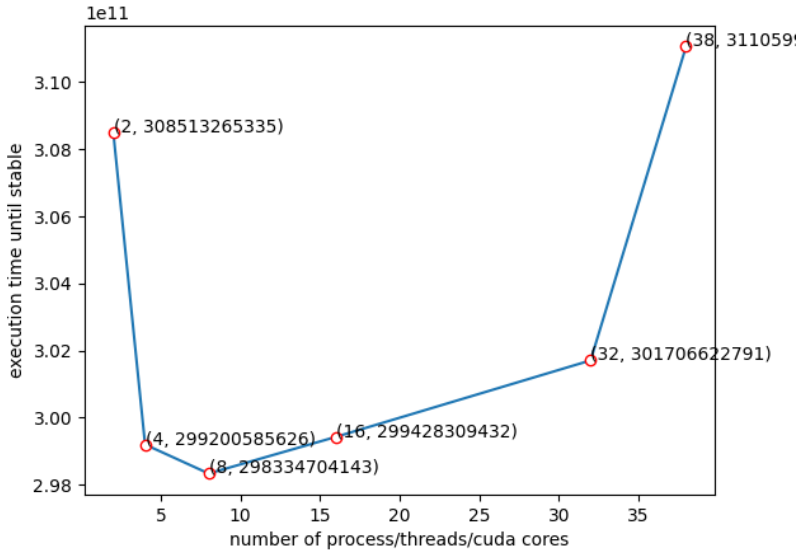

Figure 3.2.2.2: Execution time of different number of processes (MPI method)

In the OpenMP program, the experiment result is a little abnormal as well although it is more normal than the MPI version. It is expected that the running time should decrease when the number of threads increases. However, in the graph shown below, the running time only decreases from 2 threads to 16 threads. Then the running time increases. The reason for the increasing time is because of the heavy overhead when the number of threads increases. As mentioned before, the computation workload is light in this program. When the number of threads is very large, the communication time dominates the program, even longer than computation time.
Here, we may make a comparison between OpenMP, thread and MPI version. Only thread version is normal due to its lowest overhead. The MPI is the most abnormal one since it cannot share memory efficiently.

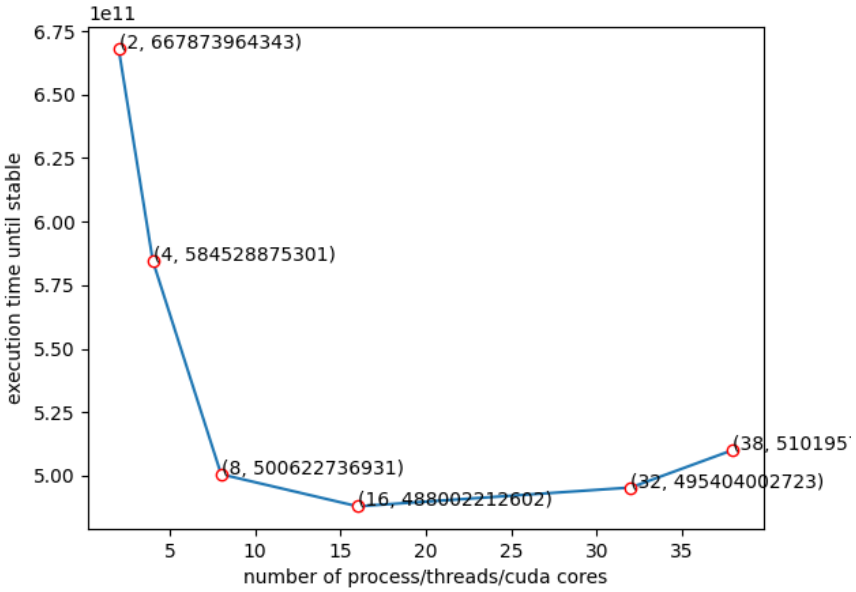

Figure 3.2.2.3: Execution time of different number of processes (OpenMP method)

In the CUDA program, the program stabilizes faster as the number of threads increases. We can see that the slope of the graph is decreasing here because the overhead time increases when there are more threads since more threads need to be created and passed information. Notice that there is no increase from 32 to 38 CUDA cores. The reason is that on one SM, there are more than 38 SP (CUDA cores). For instance, there are 64 CUDA cores on one SM in RTX 2080Ti. The architecture of GPU is different from CPU.

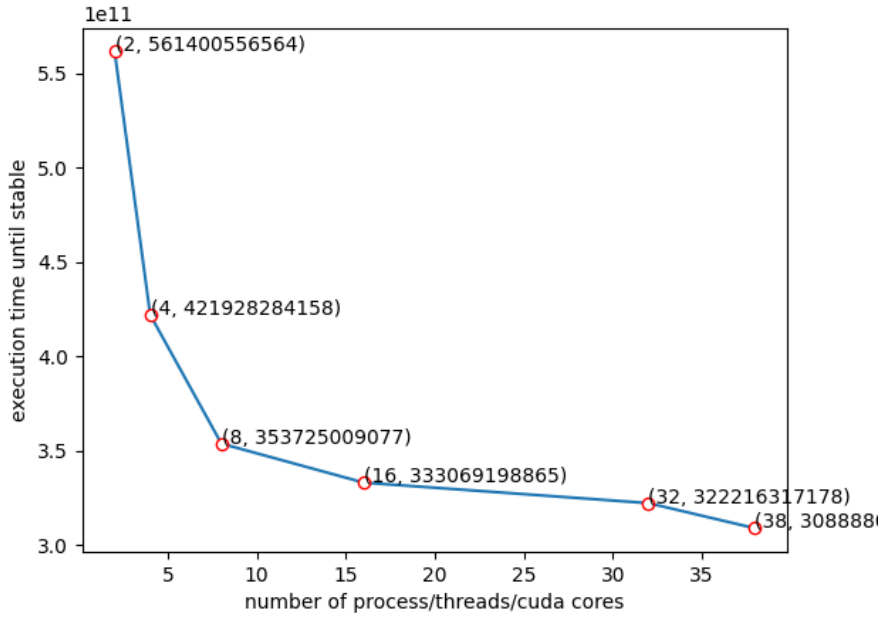

Figure 3.2.2.4: Execution time of different number of processes (CUDA method)

3.2.3. Performance comparison between MPI, Thread, OpenMP and CUDA

The graph for the comparison between MPI, Thread, OpenMP and CUDA implementation is shown in figure 3.2.3.1. The difference regarding performance of each implementation accounts for both the programming architecture and API mechanism. In this project, MPI is the fastest one and the OpenMP is the slowest one. The speed-up becomes negative for MPI and OpenMP when the process/thread number becomes large. For CUDA implementation, the speed-up is always positive. The reason is that although transferring data from CPU to GPU and then from GPU to CPU is quite time-consuming, the communication time for CUDA implementation is almost irrelevant from the number of CUDA cores. This makes the speed-up always positive. The GPU is suitable for parallel computing by its nature, and that is why it is suitable for deep learning. We can also make a prediction that if the number of CUDA cores or processes grows very large, CUDA implementation will be faster than MPI implementation since MPI implementation has negative speed-up when the number of parallelism is large.

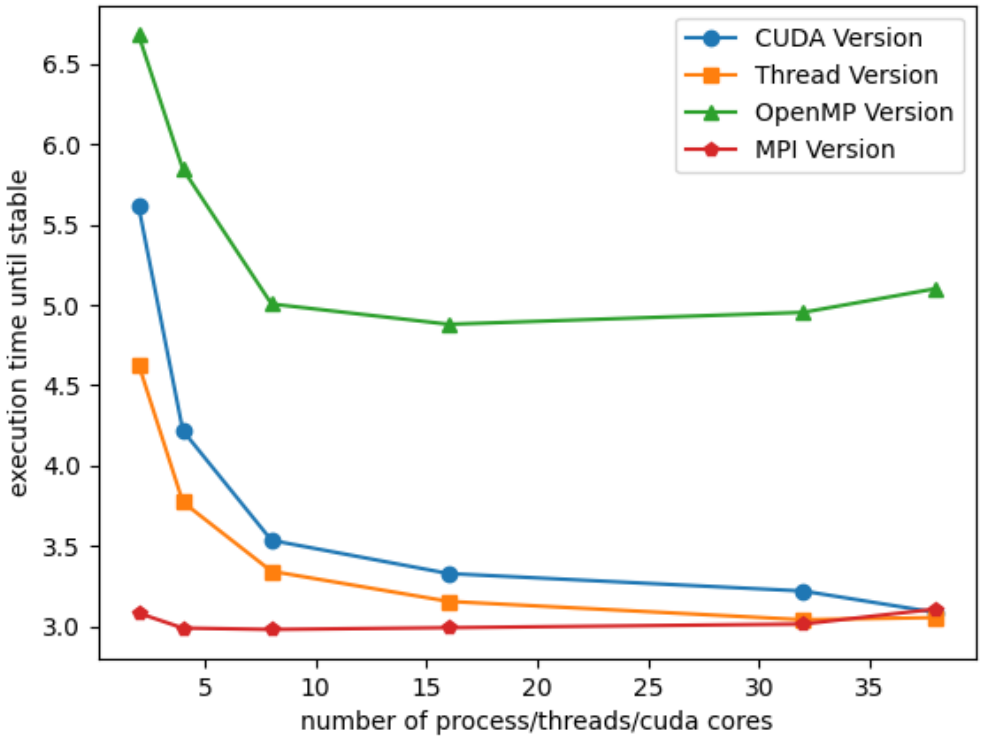

Figure 3.2.3.1: Comparison between MPI, Thread, OpenMP and CUDA Implementation

4. Discussion

This project implements the bonus as well. In the discussion part, we will discuss the effect of programming with both MPI and OpenMP. Since I finished the bonus part quite late, therefore I conduct the experiment of bonus mainly on my local CentOS virtual machine.

In this experiment, we use 4 cores to test 300*300 pixels. At first, we test the MPI version, the running time for the heat radiation to reach stable status is 2.26×10^{10} ns. For the MPI and OpenMP version with 4 processes and 4 threads, the execution time is 1.65×10^{11} ns. The program runs slower in this case. The reason is that there are only 4 cores available in this experiment. In the MPI+OpenMP version, we have 4 processes and 4 threads for each process. Therefore at most 4 threads execute at the same time because there are 4 cores only. Only 4 threads execute simultaneously while other 12 cores execute concurrently. There will certainly be more overhead time for switching threads.

Another experiment is done by using 2 processes with 2 threads. This version stabilizes in 3.63×10^{10} ns, which is much faster than using 4 processes with 4 threads. The reason is that there are less threads for the operating system to switch. Therefore, less overhead time is wasted.

5. Conclusion

This project implements and analyzes 6 versions of heat simulation. MPI, thread, OpenMP, CUDA, MPI plus OpenMP and sequential version are provided. The experiment result basically meets theoretical result. Two video demos are provided to show how to run the program. One of the demo demonstrates how to run CUDA program. Another one shows how to run MPI, thread, OpenMP, etc.