香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

CSC4005
Parallel Computing

Project 4

Author	**Student ID**
Luo Haoyan	119010221

November 30, 2022

Contents

1 Introduction

Heat distribution problem is a problem that focuses on the transformation of heat when there are heat sources from four edges and a heat source in the center. To simulate the process of heat distribution, we use finite difference method to calculate the result. After each iteration, the temperature of each point is (Using Jacobi iteration):

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1}, h_{i,j+1}}{4}$$

$(0 < i < n,\ 0 < j < n)$ for a fixed number of iterations or until the difference between iterations of a point is less than some very small prescribed amount.

I adopted 6 different approaches for the simulation, which are Sequential, MPI, Pthread, OpenMP, CUDA, and MPI + OpenMP versions. The implementation details will be illustrated in the following sections. Comprehensive experiments are conducted and performance under different configurations in four dimensions are analyzed: different number of threads/cores used; performance of Sequential program vs. MPI program vs. Pthread program vs. OpenMP program vs. CUDA program; performance under different problem size (number of bodies); performance of MPI program vs. OpenMP program vs. MPI+OpenMP program. You can check the appendix for the steps to run my program.

2 Method

2.1 Sequential Version

The sequential program for the heat distribution simulation is basically the implementation of updating temperature for the given data points. In my implementation, there are three separate functions taking in charge of the updating procedure. The update function updates the temperature based on Jacobi iteration with the data in odd iteration as input and the data in even iteration as output. The *maintain_fire* function facilitates the function of maintaining the temperature of the fire (keep it as 100 degree in my implementation). And last, the *maintain_wall* function helps to keep the wall temperature as 20 degree. Note that in my implementation, the wall indicates the boundary data points given the square area. A code snippet of maintaining temperature of the wall is shown below:

```
for (int i = 0; i < size; i++) {
    data[i] = wall_temp;
    data[i * size] = wall_temp;
    data[i * size + size - 1] = wall_temp;
    data[size * size - 1 - i] = wall_temp;
}
```

2.2 MPI Version

The program of MPI version can be divided into three parts:

- Broadcast the basic information and the states from root process to other process;
- Divide the problem of updating temperatures using stripe partition;
- Conquer the results back to the root process.

Following Figure shows the overall structure of my MPI program.

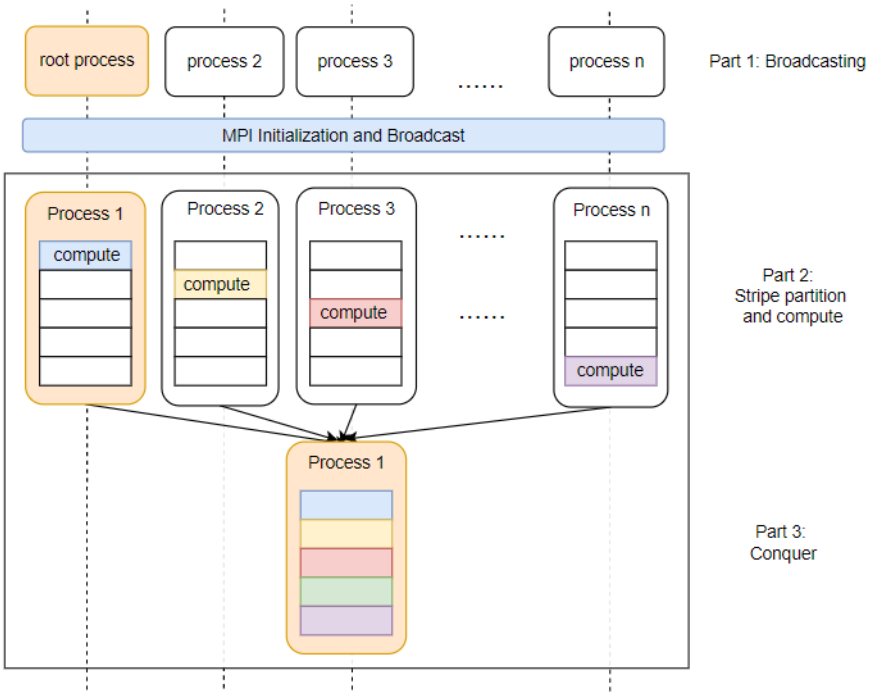

Figure 1: *The overall structure of MPI program*

2.2.1 Initialization and Broadcasting

In the first part, the root process first deals with the basic information of the input from user, including the size of the area, and then broadcast these meta-information to other processes. Then the root process will initialize the state of all data points and generate the initial fire distribution. Note that the information we broadcast here includes the id of rows (stripe partition) each process used. The following code snippet shows the corresponding broadcasted information:

```
// broadcast row information
MPI_Ibcast(row_count, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
```

```
        MPI_Ibcast(row_displacements, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        local_size = row_count[my_rank];
```

The broadcast information includes the number of rows that each process handle. At the same time, we also broadcast the number of data points that each process handle by the following code snippets. Note that the send buffer contains the number of points that are distributed to different processes:

```
        // broadcast data points information
        MPI_Ibcast(sendbuffer, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
        MPI_Ibcast(sendbuffer_displacements, world_size, MPI_INT, 0, MPI_COMM_WORLD,
↪   &request);
        MPI_Wait(&request, MPI_STATUS_IGNORE);
```

2.2.2 Problem Division: Strip Partitioning

Similar with the previous implementation of MPI program, we have to allocate the data to different processes to perform the divide-and-conquer strategy to increase the efficiency. Here, to further boost the performance, we use stripe partition but not the block partition. The reason is that there is less communication overhead when partitioning in strips than partition in blocks, which means it is easier to program and reduce the communication overhead when the startup time is relatively large. The following code snippet shows the problem division of the data with non-blocking operation:

```
    int begin = 1;
    int end = row_count[my_rank];
    // distribute data and calculate temperature
    MPI_Scatterv(global_data, sendbuffer, sendbuffer_displacements ,MPI_FLOAT,
↪   input_data, sendbuffer[my_rank], MPI_FLOAT, 0, MPI_COMM_WORLD);
    update(input_data, output_data, begin, end);
```

We set the begin indicator as 1 because that we have to send the neighborhood buffers (i.e. the neighbor rows that each process handle). Thus, for the specific computation, each process do not need to compute the first row that is the neighbor one. Then, the specific number of data will be distributed to each process.

2.2.3 Problem Gathering

After the problem is divided, each process would call the functions defined in sequential version (with a slight modification in IO) to perform their local computations. After all of them finishing the computation, the processes would be synchronized and the temperatures would be gathered together:

```
    // when gathering the data, the first row of each process is not needed
    float* output_data_shifted = &output_data[size];
    if(my_rank == 0) output_data_shifted = &output_data[0];
    MPI_Gather(output_data_shifted, local_size * size, MPI_FLOAT, global_data, local_size
↪   * size, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Note that since we also send the buffer rows to each process, when gathering the data, we do not need to send the buffer rows back to the root process. Thus, a shifted output is needed (actually just an extra index indicating it is in the second row) to be sent and gathered back to the root process. Meanwhile, each process would only update the states of the allocated data points in order to prevent from data race.

2.3 Pthread Version

For the pthread parallel program, there is no need to broadcast information from root process to other processes since Pthread can share the memory among threads. The overall structure of Pthread program is shown below:

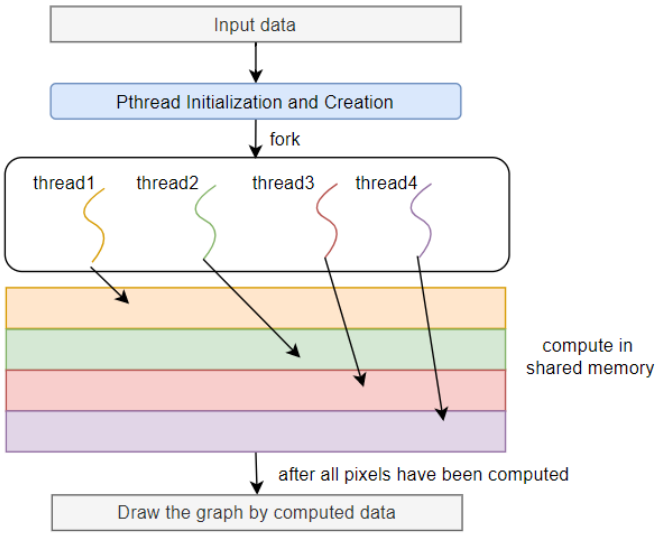

Figure 2: *The overall structure of Pthread program*

2.3.1 Threads Initialization

The basic computation data of problem is initialized and stored in share memory, so different threads have the ability to access the shared information. Besides, the allocated number of threads are created and they will execute the same function with different input data.

2.3.2 Problem Division

The problem would be partitioned into stripes as MPI version. Since in the stripe division distribution problem, each point gets a theoretically similar workload, it is better to directly use static scheduling instead of dynamic scheduling.

After specifying the tasks each thread would perform, the slave threads would be forked by the master thread and perform their local calculation. Note that for each iteration of updating the temperatures of data points, there is no data dependency between the elements. That's because there are two buffers involved and read and write operations are separated. Then after the computation, only master thread would perform pixel gathering function. One can refer to the following code snippet for the distribution of workload:

```
// pthread distribution
int num_my_row = size / num_thread;
int remain = size % num_thread;
if (id_thread < remain) num_my_row++;
int displacement = num_my_row * id_thread;
int begin = displacement;
int end = displacement + num_my_row;
if(begin == 0) begin++;
if(end == size) end--;
```

Note that at the last two rows, in order not to query the data point by index that is outside the boundary, we set two the begin indicator and the end indicator (correspond to the first thread and the last thread respectively) not to be 0 nor equal to size. This ensures the safety of our program.

2.3.3 Parallel Computing

Given the partition with the above mentioned mechanism, each thread would perform the function with their own workload. The detailed implementation is generally the same with the sequential computation, and you can refer the details to Figure 2. Specifically, we deploy similar program structure as sequential version and parse the argument into each thread (which contain basic information among different threads) as follows (odd round as an example):

```
if (count % 2 == 1) {
    for (int thd = 0; thd < n_thd; thd++) {
        args[thd].id_thread = thd;
        args[thd].num_thread = n_thd;
        args[thd].data = data_odd;
        args[thd].new_data = data_even;
        args[thd].fire_area = fire_area;
```

```
        }
    }
```

2.4 CUDA Version

As for the CUDA version, there are mainly four components to realize the parallel computing function:

1. Malloc memory in CPU for storing the states of data points in host, and malloc memory in GPU for storing the states of data points in device.
2. Generate states of data points in host and copy the information from host to device.
3. Launch the threads to perform computation in parallel.
4. Perform divide and conquer strategy at device, and copy the results back to host.

Below is a general structure figure that summarize the dataflow of CUDA program. Details are elaborated in the following sections.

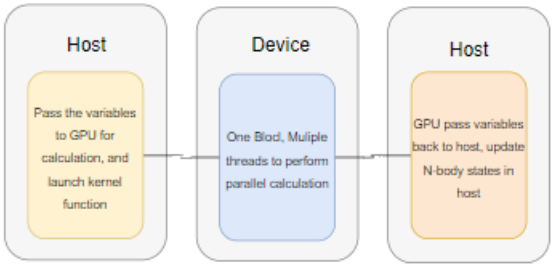

Figure 3: *The general description of CUDA program*

2.4.1 Malloc Memory and Launch Kernel Threads

First, the data would be initialized, including the state of the program, and the malloc of memory spaces in host and device. Then the second step is to launch kernel threads. Thread 0 would be in charge of initializing states of data points in the device memory (including assigning all the points with their initial temperature). Compared to other implementation of parallel computing, the memory structure of CUDA program is relatively more complicated. To ensure the operation efficiency and the correctness of data manipulation both in the host and the device, we first malloc two distinguished memory space in both host and device for storing the states of all data points.

2.4.2 Problem Partition and Parallel Computing

In this part, we need to apply stripe partition as before to the problem so that each kernel thread can perform its work individually. This step is basically the same as Pthread version. Then the main calculation routine

would be performed. The host would launch the kernel threads to perform the parallel computation. Similar to the previous parallel paradigm, the workload is divided with the same division strategy (stripe partition). In addition, one of the most important thing is to keep the data free from data race (since the memory declare with device could be accessed by all the kernel threads in one block). Following code snippet shows the call of functions:

```
if (count % 2 == 1) {
     update<<<n_block_size, block_size>>>(data_odd, data_even,size);
     maintain_fire<<<n_block_size, block_size>>>(data_even, fire_area,size);
     maintain_wall<<<1, 1>>>(data_even,size);
 } else {
     update<<<n_block_size, block_size>>>(data_even, data_odd,size);
     maintain_fire<<<n_block_size, block_size>>>(data_odd, fire_area,size);
     maintain_wall<<<1, 1>>>(data_odd,size);
 }
```

Details for the structure in the part as well as the overall detailed structure of CUDA program is shown in the following figure.

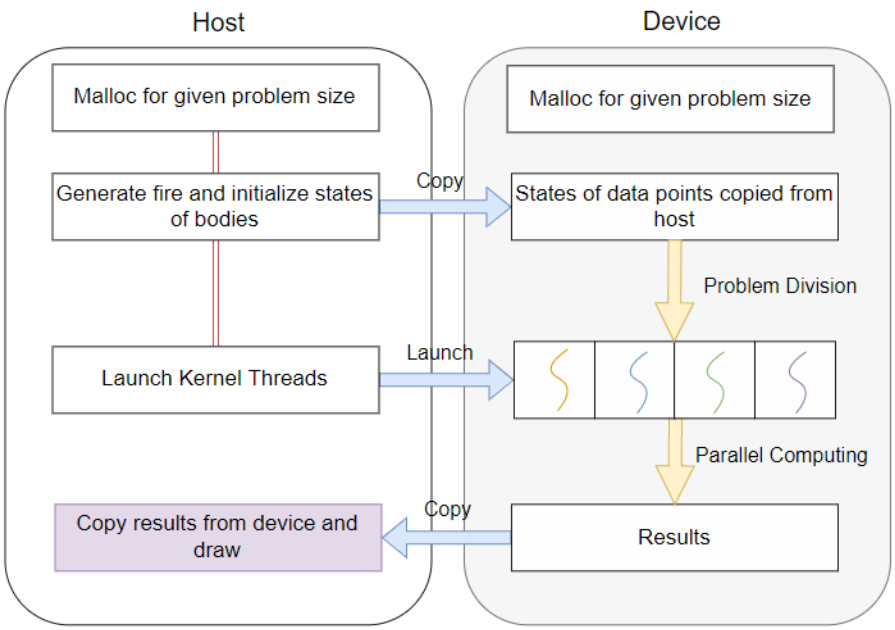

Figure 4: *The overall detailed structure of CUDA program*

2.4.3 Results Integration

After computing (updating temperature as well as maintaining temperature of fire and the walls) of each kernel threads. To keep the data consistency again, the results would be copied back to the host memory

which can be used by host to Paint the graph of simulation.

2.5 OpenMP Version

The OpenMP implementation routine is almost the same as the Pthread version, since they all share the core memory. And OpenMP version is very easy to implement since it would automatically allocate thread for the user. The overall structure of the OpenMP program is shown below.

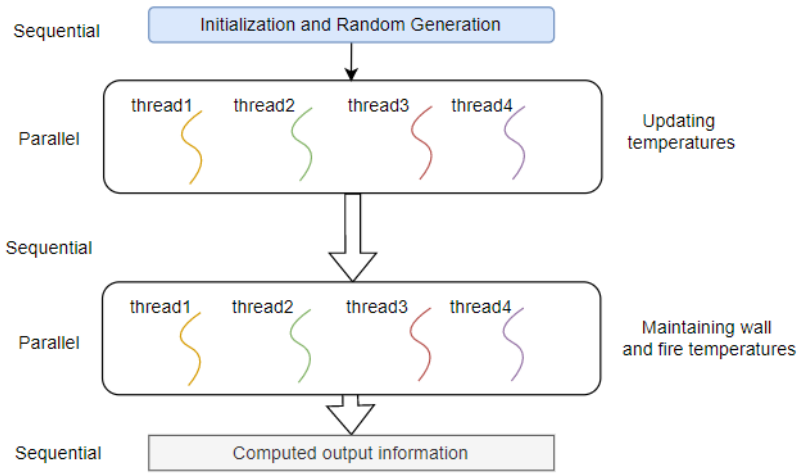

Figure 5: *The overall structure of OpenMP program*

You can consider the above paradigm as a fork-join strategy. In the heat distribution, I parallel the updating of temperature of each points, which means that each thread would handle certain number of points updating (see the code below using updating temperature as an example):

```
// OpenMP
omp_set_num_threads(n_omp_threads);
#pragma omp parallel for
for (int i = 1; i < size - 1; i++) {
    for (int j = 1; j < size - 1; j++) {
        int idx = i * size + j;
        float up = data[idx - size];
        float down = data[idx + size];
        float left = data[idx - 1];
        float right = data[idx + 1];
        float new_val = (up + down + left + right) / 4;
        new_data[idx] = new_val;
```

```
        }
    }
    }
```

Each thread will parallelly update the temperatures and wait other threads to finish. After all threads reach the barrier, they will update the distance information as well. As for other parts of the program, they are basically the same as sequential version.

2.6 Bonus: MPI + OpenMP Version

Since MPI allows communication among different cores, and each cores allow different number of threads to execute the same function with different workload, it is naturally a good idea to combine these two approaches together. The design of MPI + OpenMP program is just basically adding the OpenMP approach to the MPI design. Figure 6 shows the overall structure of MPI+OpenMP.

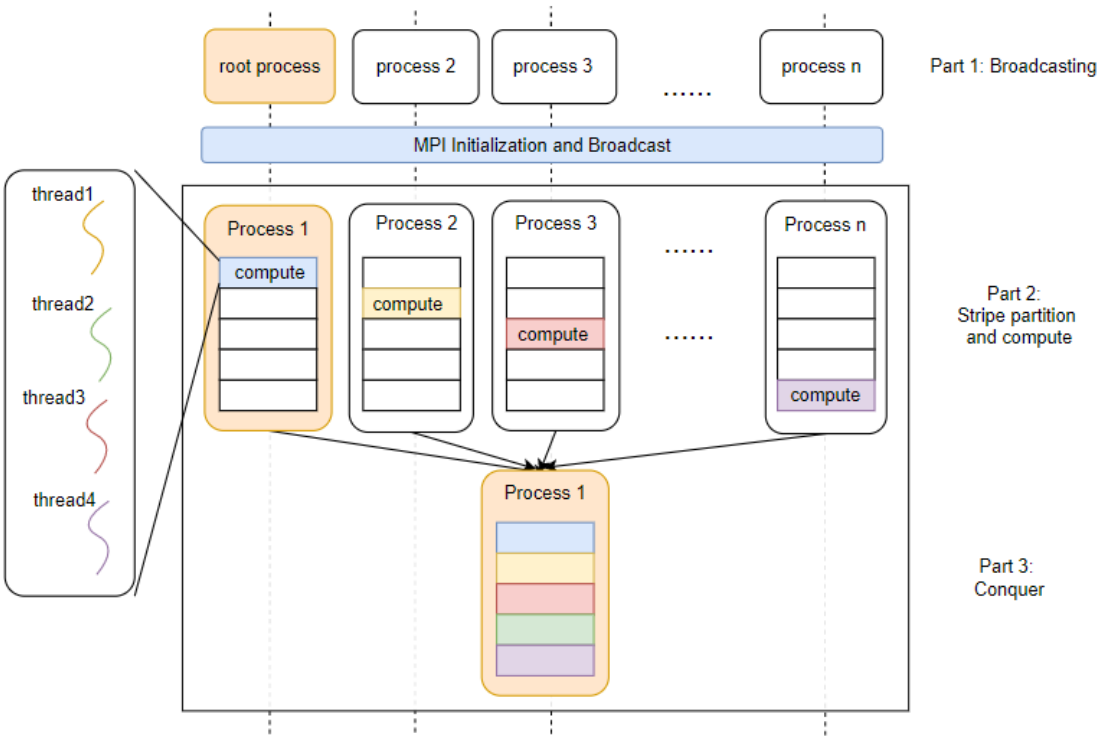

Figure 6: *The overall structure of MPI+OpenMP program*

For example, each process in MPI program originally use a for loop to iterate the points in the heat distribution assigned to it. Then, with the combination of OpenMP approach, the parallel for clause of OpenMP would increment the parallelization of the program. In other words, the workload in each process

is further sub-divided into different threads by OpenMP and then computed parallelly. After all the threads finish their computation (which means all the processes finish their computations as well), the data would be gathered and to the root process. In my implementation, I named the combination as *openmpi*.

3 Experiments

Comprehensive experiments are conducted to test and analyze the performance of parallel computing by MPI programming and multi-threaded programming. Various input data size are tested (range from size 100 to size 10000, here the size means the input resolution, thus the actually size of the data is $size^2$). Here, small size (100 - 1000), medium size (2000 - 4000), and large size (8000 - 10000) are respectively defined. All the test result ran by sbatch can be found in corresponding *.out* files.

3.1 Evaluation Metrics

In addition to the running time of the program, another evaluation metric: Speedup Factor, is introduced in the evaluatioin part. Parallel speedup is defined as the ratio of the time required to compute some function using a single processor (t_s) divided by the time required to compute it using P processors (t_p), the formula can be explicitly written as:

$$S(n) = \frac{Execution\ time\ using\ one\ processor(single\ processor\ system)}{Execution\ time\ using\ amultiprocessor\ with\ n\ processors} = \frac{t_s}{t_p}$$

You can refer other constants in my program in the physics file as follows:

```
#define fire_temp 100.0f
#define wall_temp 20.0f
#define fire_size 100
#define resolution 800
#define max_iter 2000
```

3.2 Results

3.2.1 Main results

The Speedup results (based on the equation above) of Sequential version, MPI version, Pthread version, OpenMP version, CUDA version, and the bonus (OpenMP + MPI) are presented in the following tables (and also in the appendix). The running time results for different configurations are also provided i(in the output file with the submitted code). Note that all the experiments in my implementation fix the number of iteration to 2000.

#process	100	200	400	800	1000	1600	3200
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	1.22068	1.22510	0.64645	0.56298	0.57316	0.53929	0.52253
4	1.31118	2.06207	1.05057	1.02199	1.06186	1.07642	1.12431
8	1.90317	3.15736	1.99269	2.00756	2.03576	2.11513	2.00048
12	2.17803	3.67498	2.97508	3.07338	2.55754	3.00069	3.00034
16	2.44960	3.81146	3.72498	3.40525	3.81565	3.84051	2.69578
24	2.15495	4.68153	3.22694	3.38523	3.62324	3.61350	3.60036
32	3.30497	4.70525	4.11083	4.55513	4.66761	4.74631	4.58865
40	0.68693	6.61052	5.33427	5.65263	5.84184	5.74713	5.52691

Figure 7: *The speedup results of MPI program*

#thread	100	200	400	800	1000	1600	3200
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.22692	0.64825	1.25580	1.80296	1.80841	1.88240	1.84986
4	0.25469	0.39840	1.57551	2.34999	2.50684	3.24457	2.69703
8	0.09278	0.29302	0.73807	2.14604	3.34471	5.41805	4.89902
12	0.05922	0.19364	0.84502	1.90152	2.18887	4.78935	5.54277
16	0.04349	0.14864	0.68292	1.96227	2.30341	3.36454	6.09727
24	0.02898	0.09706	0.49039	1.76998	2.36632	2.80990	3.92145
32	0.02192	0.07676	0.36826	1.46250	2.12063	3.17169	4.87681
40	0.01780	0.06258	0.29559	1.22445	1.84676	3.08207	5.53098

Figure 8: *The speedup results of Pthread program*

#thread	100	200	400	800	1000	1600	3200
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.77296	1.57746	1.71011	1.79177	1.89270	1.87266	1.90815
4	0.97677	1.95728	2.57853	3.46067	3.42809	3.67202	3.58182
8	1.06112	2.60383	4.14964	6.20404	6.81207	6.80139	6.77081
12	0.98642	2.54006	5.52094	8.89992	9.43066	9.43817	10.20485
16	0.88513	1.93322	6.79099	11.29881	12.01757	10.25984	11.62025
24	0.71048	2.21209	5.95995	10.34803	11.23686	13.08612	11.75814
32	0.51005	1.66463	5.29717	12.26292	15.01625	16.03719	13.54890
40	0.43033	1.55780	3.97040	12.32872	15.37018	18.13439	14.16550

Figure 9: *The speedup results of OpenMP program*

#thread	100	200	400	800	1000	1600	3200
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.77296	1.57746	1.71011	1.79177	1.89270	2.38295	2.15536
4	1.01263	1.95728	2.57853	3.46067	3.42809	3.67202	4.18206
8	1.06112	2.60383	4.14964	6.20404	6.81207	6.80139	6.77081
12	0.98642	2.54006	5.52094	8.89992	9.43066	9.43817	10.20485
16	0.88513	1.93322	6.79099	7.76729	12.01757	10.25984	11.62025
24	0.71048	2.21209	5.95995	10.34803	11.23686	13.08612	11.75814
32	0.51005	1.66463	5.29717	12.26292	15.01625	16.03719	13.54890
40	0.47660	1.55780	3.97040	25.09759	22.64308	21.25665	15.97958

Figure 10: *The speedup results of CUDA program*

3.2.2 Image Display

The GUI outputs for MPI version as an example when the input size is 800x800 are shown below. The parallel program uses 2 processes/threads and the number of iteration as as a demo example.

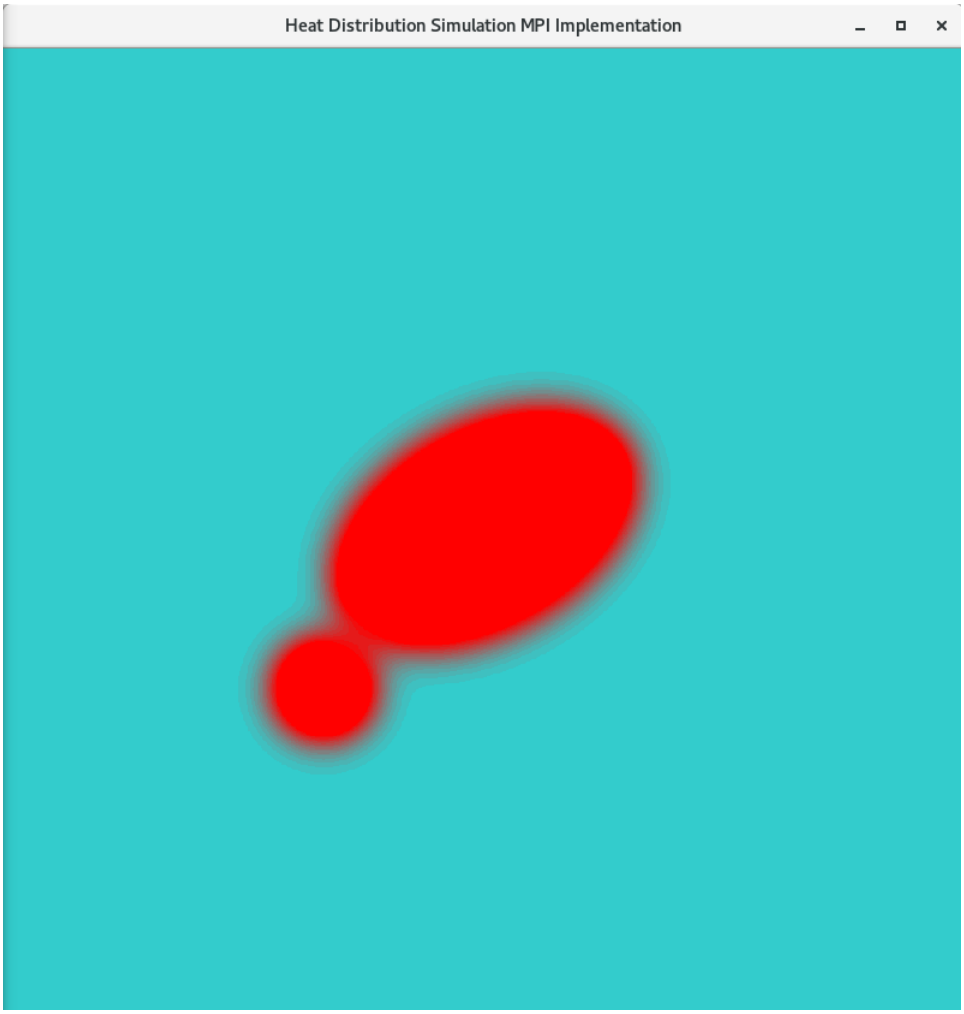

Figure 11: *The demo output of MPI program*

3.3 Performance Evaluation: Different Number of Threads or Cores

MPI Version: According to the results shown in the above Table, increases in speedup are both observed. To better understand the results , the visualization of the results for different number of threads is presented in the following figure:

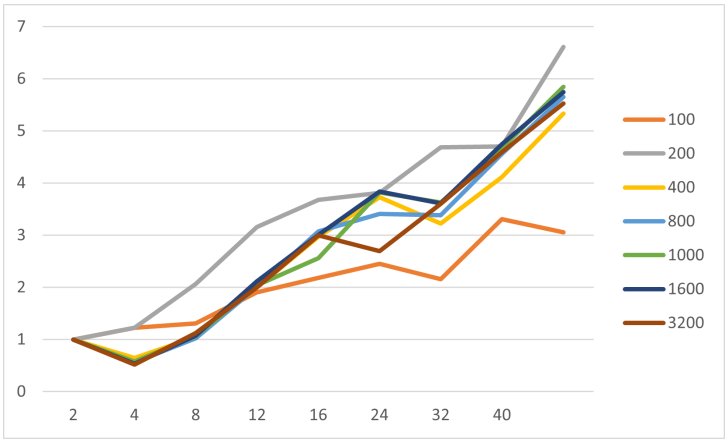

Figure 12: *The relationship between number of processes and speedup performance given different number of size (MPI)*

We can have the following three observations:

1. As shown in the figure, for relatively small problem size, as the number of cores increases, the overall performance of the parallel program increases (as long as it is not under a relatively small size).

2. From the above figure we can observe that the parallel approach will first enhance the performance as the number of cores increases in the small-medium input settings, and then the performance will fluctuate (see size=1000 as an example) but basically remain the same.

3. With a relatively large problem size, the performance keeps increasing as the number of cores increases. There are some surge in some configuration settings but the trend is generally upward

This phenomenon can also be explained directly by Amdahl's law, which posits that the speedup is limited by the serial section. In heat distribution, the serial section is big because the steps of updating the temperature are easy, so the performance is limited. However, when the problem size is still relatively small, a clear increasing trend can still be seen.

Pthread: In the pthread version, the performance still increase and gradually cease as number of threads is over 16. The observations above enlighten that the optimal number of processes for a given configuration depends on many factors such as problem size. As the number of threads increase, workload are more evenly to be distributed to each core and perform the parallel computing, thus, each core's workload is decreased and total speed enhances.

Particularly, parallel computing can improve performance notably when the number of processes fits the problem size (i.e. the number of input elements can be fully divided by the number of processes). Moreover, it can be found that when the problem size is large (greater than 1000), the performance can reach its maximum and can even increase as the problem size gets larger.

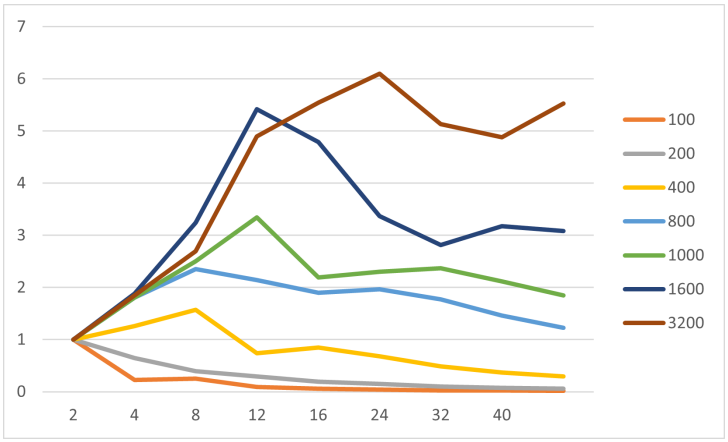

Figure 13: *The relationship between number of processes and speedup performance given different number of size (Pthread)*

The reason for the decreasing performance when problem size is relatively small (less than 400) results from the improvement of parallelly solving the small size of problem is smaller than the fork/join and synchronization overhead.

OpenMP: For cases of pthread and openmp, the treads are basically the same. As shown in the figure, for problem size ¿ 200, the performance would increase nearly linearly as the number of threads increases.

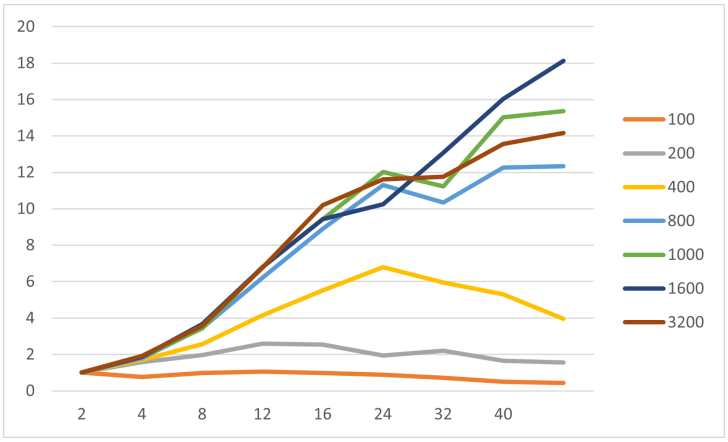

Figure 14: *The relationship between number of processes and speedup performance given different number of size (OpenMP)*

Then, the tendency of the increasing would cease when number of threads are greater than 24. We can have the following three observations:

1. As the number of threads increases, the performance (speedup) is almost linearly increasing when the input size is medium and large. For example, when the input size is 3200 and the number of threads

allocated is 40, the speedup factor can be up to 18.

2. However, when the input size is small, the performance improvement is not that obvious and might even harm the performance (see size 100 as an example).

3. Another interesting thing is the increase of each line is not smooth (which means the performance improvement will sometimes stop for a while as the number of threads increases).

The generally increasing tendency can be explained by Amdal's law:

$$S(n) = \frac{n}{1 + (n-1)f}$$

Since there is no data dependency in Mandelbrot set computation (all pixels would not affect each other), serial section (f in amdal's law) is quite low and can perfectly explain the performance gain as the number of threads increases. In addition, similar with MPI program, the performance gain is most obvious when the problem size is large or medium, while when the problem size is small, the overhead of allocating data may offset or exceed the performance gain given by parallel computing.
Moreover, the reason why the performance gain is small when the problem size is small is that the data allocation overhead can offset and even exceed the performance gain from parallel computing. The conclusion basically meets the statement of Amdahl's law, limited by the serial fraction (In program, like some preparation work done by root process), the improvement of performance that increasing threads brings is bounded.

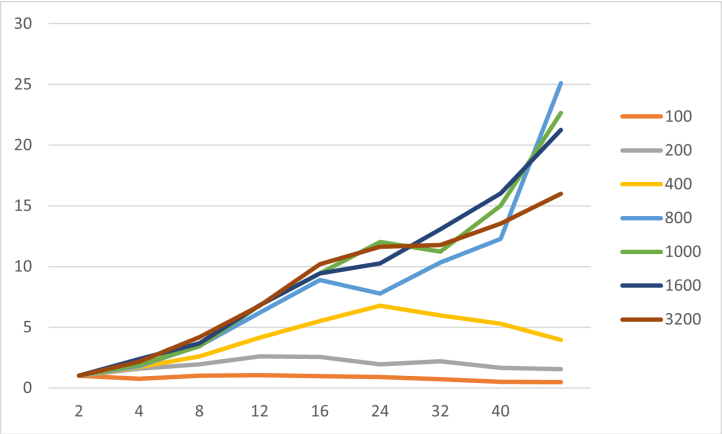

Figure 15: *The relationship between number of processes and speedup performance given different number of size (CUDA)*

CUDA: The above figure shows that as the number of processors increase, the performance of CUDA program will gradually slow down in most input sizes. While there are certain input size corresponds to increasing performance, the general tendency is similar to Pthread and MPI program. But at the same time,

the result illustrates the great scalability of CUDA program.

As for the reason behind, the threads in CUDA are very lightweight, which means that forking a new CUDA thread results in a very light cost, thus resulting in a greater scalability. And the second merits of CUDA program is that the maximum number of threads per block in GPU can support up to maximal 1024 threads. Therefore, the increasing threads can be parallel computed in GPU and decrease the time of parallel computing section according to Amdal's law.

3.4 Performance Evaluation: Different sizes

As it is shown in Figures: the performance would keep increasing as the problem size becomes greater. However, as the problem size keep increasing all the performance of all the approach would slow down and even cease.

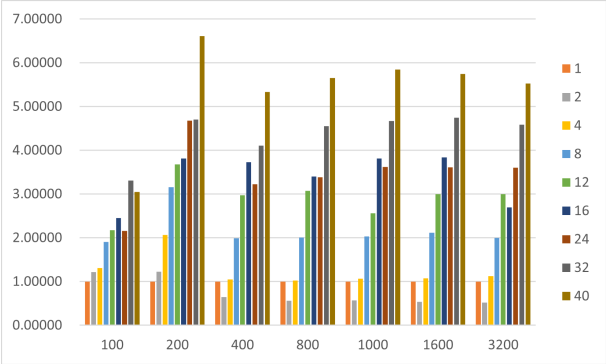

Figure 16: *Visualization of MPI program in different input sizes*

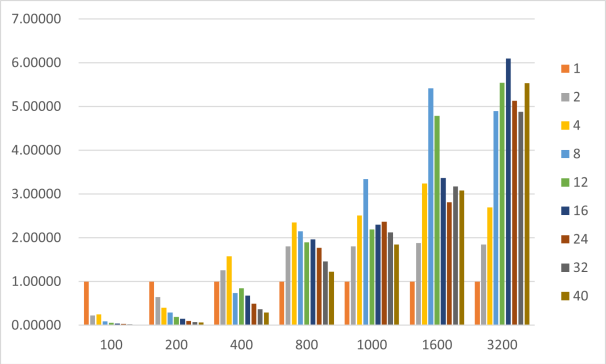

Figure 17: *Visualization of Pthread program in different input sizes*

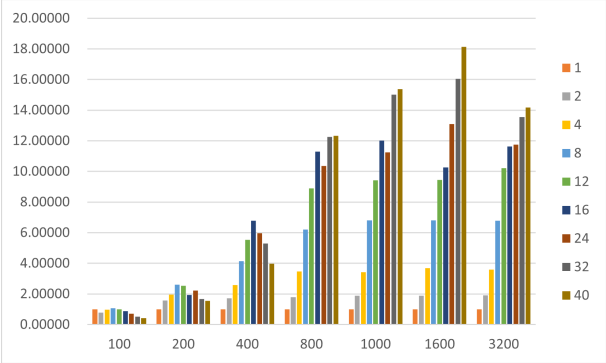

Figure 18: *Visualization of OpenMP program in different input sizes*

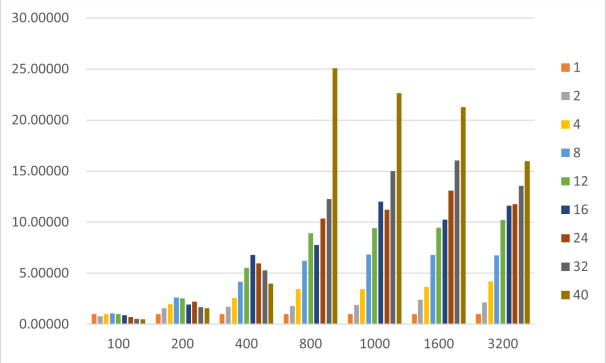

Figure 19: *Visualization of CUDA program in different input sizes*

From the figures above, we can observe the following:

1. When the problem size is small, no matter the number of threads/cores allocated is large or small, the performance will increase as the size increases.

2. When the problem size is medium, for smaller number of threads/cores, the performance will "stop" to increase as the size increases and keep flatten. This is more obvious in MPI program (since pthread program here use dynamic scheduling, which will stressed in following session).

3. Some fluctuations at MPI program are observed when the number of cores allocated is large (see 40 as an example). This observation is in line with the previous analysis about the allocation overhead.

Analysis The increased performance when the problem size is relatively small can be explained by Gustafson's law, which claims that as the problem size goes up, parallel computing will have a higher performance because:

$$speedup\ factor = S(n) = n + (1 - n)s$$

where s is the number of cores available and n is the proportion of sequential computing. As the problem size increases, the sequential proportion n will decrease, which makes the speedup more close to the number of cores. However, there also exists a reasonable boundary for this improvement as the problem size getting larger and larger. Thus, the performance improvement will slow down and cease when the problem size is too large (relative to the number of cores/threads). The trends are generally observed in all implementations.

3.5 Performance Evaluation: Different Parallel Approaches

MPI	OpenMP	Pthread	CUDA				
	100	200	400	800	1000	1600	3200
2	1.22068	1.57746	1.25800	0.31100	1.25300	1.85800	4.70200
4	1.31118	2.06207	1.57551	0.16900	0.65500	1.15600	2.45000
8	1.90317	3.15736	1.99269	0.10800	0.41000	0.48700	1.22400
12	2.17803	3.67498	2.97508	0.11500	0.23900	0.35200	1.19700
16	2.44960	3.81146	3.72498	0.08200	0.27200	0.28200	0.67600
24	2.15495	4.68153	3.22694	0.12600	0.24800	0.35600	0.79400
32	3.30497	4.70525	5.29717	0.15400	0.25100	0.30800	0.68000
40	3.05192	6.61052	3.97040	0.17100	0.30700	0.32700	0.59700

Figure 20: *The best speedup results summation*

We have the best configuration of Pthread colored in orange, openmp colored in red, cuda colored in green and mpi colored in blue. Based on comparison table, we can make the following conclusion:

1. MPI outperforms others when the problem size is small. The smaller the problem size, the more advantage that MPI take (especially when the problem size is under 800).

2. . Pthread generally outperforms others when the problem size is large and number of threads is small.

3. OpenMP takes advantage when the problem size is large and the number of threads is between 16 and 32.

4. CUDA takes advantage when the number of threads is large. It achive the best performance in most of the experiments with 40 number of threads.

Analysis It can be observed that MPI takes its advantage of seperate memory space. The advantage of MPI can be explained by the communication overhead is still not significant when the problem size is small. As for CUDA, since it has lightweight threads and high scalability, when the number of threads increase to large size, the performacne of CUDA will naturally outperform others. The results show high scalability of CUDA program when the problem size is between 800 and 3200.

In addition, CUDA and OpenMP's poor performance compared with sequential version can be explained that the problem size is still not large enough. Therefore, according to Gustafson's law, the positive affect of parallel computing is very limit. In addition, compared with MPI and Pthread, OpenMp and CUDA have many significant overheads even when the problem size is small (including fork/join, scheduling overhead for OpenMP, memory copy time of CUDA), which negatively affects the performance when the problem size is not large enough. Therefore, under these configurations, CUDA and OpenMP improve the performance little or even harm the performance compared with sequential version.

OpenMP shows the worst general performances. The reason behind may comes from its overhead caused by dynamic scheduling, synchronization and fork/join strategy. As for Pthread approach, when the problem size is large, it avoids the significant communication overhead of data transition. This trend can be seen in all three thread based methods, because these 3 approaches all rely on the shared memory. Therefore, Pthread outperforms when the problem size is lagre. However, when the number of threads increase (greater than 24), over-spawn occurs so that the performance of pthread would cease to increase and cannot outperform other methods.

3.6 Performance Evaluation: MPI vs. OpenMP vs. MPI+OpenMP

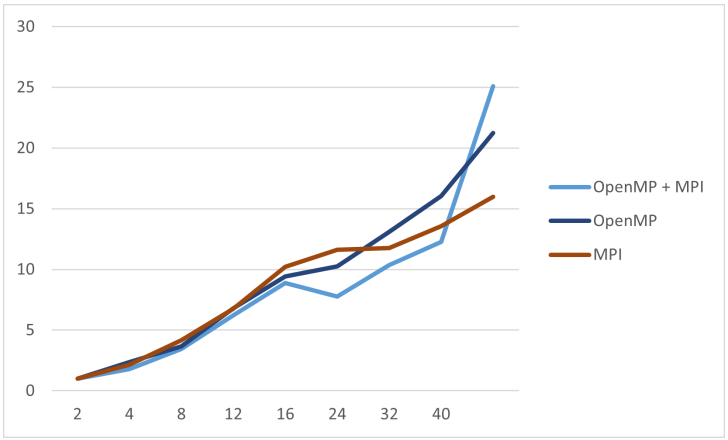

Figure 21: *Comparison between MPI and OpenMPI implementation, with 4 threads and 4 threads + 4 processes used in comparison*

Surprisingly, we found that the performance of openmpi is not generally better than the mpi and openmp. This phenomenon can be explained by the problem size is not large enough. For example, the biggest problem this experiment tested is 3200. If the cores is 32, it means that each threads of MPI deals with a sub-problem with 100. As it is shown in Figure, when computing such small problem size, the sequential version would outperform the parallel one. Therefore under this configuration, adding OpenMP to MPI will decrease the performance since the problem size is too small.

However, theoretically, when the problem size is large enough, the MPI+OpenMP version would get a better performance, to verify it, I test bonus program with problem size of 10000. And the result shows that the openmpi can take nearly 100 seconds to solve the problem while mpi need to take up to 240 seconds. This result verified the assumption. Also, we can see an increasing trend of the performance in the figure.

4 Conclusion

In general, parallel computation outperforms nearly every case with respect to the sequential one. And applying more threads or cores for the heat distribution would increase the parallel computing performance and the increasing trend is stable when the problem size is large enough. However, the parallel strategy is sometimes costly when the problem size is not large enough, results in more overhead that may outweigh the performance gain. For the four implemented parallel strategies, MPI generally outperform when the problem size is small. Pthread takes its advantage when the problem size is larger and the number of threads is not that large. When the number of threads keep increasing, the performance of CUDA program will gradually outperform others. In general, OpenMP program obtains worst performance. At the same time, the hybrid OpenMP + MPI progam will increase the performance when the problem size is larger and may not outperform the separated single method when the problem size is smaller.

Appendix

A Steps to Run My Program

Following are the steps to run my code (First please ensure that you are under the project folder). Note that the bonus implementation is in the file named of *openmpi.cpp*.

1. Compile: To compile all the program using Makefile, type the following commands in terminal:

```
make all
```

If you want to clear all the executable files and recompile, type

```
make clean
```

2. Run: To run different programs you can type the following commands respectively. The commands include both command line version and GUI version. For sequential version:

```
./seq $n_body $problem_size
./seqg $n_body $problem_size
```

MPI:

```
mpirun -np $n_processes ./mpi $problem_size
mpirun -np $n_processes ./mpig $problem_size
```

Pthread:

```
./pthread $n_body $problem_size $n_threads
./pthreadg $n_body $problem_size $n_threads
```

CUDA:

```
./cuda $problem_size
./cuda $problem_size
```

OpenMP:

```
openmp $n_body $problem_size $n_omp_threads
openmpg $n_body $problem_size $n_omp_threads
```

OpenMP+MPI:

```
mpirun -np $n_processes ./openmpi $n_body $problem_size $n_threads
mpirun -np $n_processes ./openmpig $n_body $problem_size $n_threads
```

You can also run them in a batch style if you are using a cluster:

```
sbatch seq.sh
sbatch mpi.sh
sbatch pthread.sh
sbatch openmp.sh
sbatch cuda.sh
sbatch openmpi.sh
```

You can check the results in the corresponding output file (under the slurm script folder).

B Running Time Results

#process	100	200	400	800	1000	1600	3200
1	0.04432	0.14768	0.69923	2.98210	4.78528	12.34607	49.91142
2	0.03631	0.12054	1.08164	5.29694	8.34889	22.89331	95.51909
4	0.03380	0.07162	0.66557	2.91792	4.50652	11.46953	44.39291
8	0.02329	0.04677	0.35090	1.48544	2.35061	5.83704	24.94973
12	0.02035	0.04019	0.23503	0.97030	1.87105	4.11442	16.63527
16	0.01809	0.03875	0.18771	0.87573	1.25412	3.21470	18.51466
24	0.02057	0.03155	0.21668	0.88091	1.32072	3.41665	13.86291
32	0.01341	0.03139	0.17009	0.65467	1.02521	2.60119	10.87715
40	0.06452	0.02234	0.13108	0.52756	0.81914	2.14822	9.03062

Figure 22: *The running time results for MPI program*

#thread	100	200	400	800	1000	1600	3200
1	0.04432	0.14768	0.69923	2.98210	4.78528	12.34607	49.91142
2	0.19532	0.22781	0.55680	1.65400	2.64612	6.55868	26.98115
4	0.17403	0.37068	0.44381	1.26898	1.90889	3.80515	18.50605
8	0.47774	0.50399	0.94738	1.38958	1.43070	2.27869	10.18805
12	0.74846	0.76264	0.82747	1.56827	2.18618	2.57782	9.00478
16	1.01909	0.99353	1.02388	1.51972	2.07747	3.66947	8.18586
24	1.52938	1.52145	1.42587	1.68482	2.02224	4.39377	12.72781
32	2.02196	1.92394	1.89872	2.03905	2.25653	3.89259	10.23443
40	2.49007	2.36002	2.36553	2.43547	2.59117	4.00578	9.02397

Figure 23: *The running time results for Pthread program*

#thread	100	200	400	800	1000	1600	3200
1	0.04432	0.14768	0.69923	2.98210	4.78528	12.34607	49.91142
2	0.05734	0.09362	0.40888	1.66433	2.52828	6.59281	26.15692
4	0.04538	0.07545	0.27117	0.86171	1.39590	3.36220	13.93466
8	0.04177	0.05672	0.16850	0.48067	0.70247	1.81523	7.37156
12	0.04493	0.05814	0.12665	0.33507	0.50742	1.30810	4.89095
16	0.05008	0.07639	0.10296	0.26393	0.39819	1.20334	4.29521
24	0.06239	0.06676	0.11732	0.28818	0.42586	0.94345	4.24484
32	0.08690	0.08872	0.13200	0.24318	0.31867	0.76984	3.68380
40	0.10300	0.09480	0.17611	0.24188	0.31134	0.68081	3.52345

Figure 24: *The running time results for OpenMP program*

speed							
#thread	**100**	**200**	**400**	**800**	**1000**	**1600**	**3200**
1	0.04432	0.14768	0.69923	2.98210	4.78528	12.34607	49.91142
2	0.05734	0.09362	0.40888	1.66433	2.52828	5.18100	23.15692
4	0.04377	0.07545	0.27117	0.86171	1.39590	3.36220	11.93466
8	0.04177	0.05672	0.16850	0.48067	0.70247	1.81523	7.37156
12	0.04493	0.05814	0.12665	0.33507	0.50742	1.30810	4.89095
16	0.05008	0.07639	0.10296	0.38393	0.39819	1.20334	4.29521
24	0.06239	0.06676	0.11732	0.28818	0.42586	0.94345	4.24484
32	0.08690	0.08872	0.13200	0.24318	0.31867	0.76984	3.68380
40	0.09300	0.09480	0.17611	0.11882	0.21134	0.58081	3.12345

Figure 25: *The running time results for CUDA program*