

THE CHINESE UNIVERSITY OF HONG KONG,
SHENZHEN

CSC4005 PARALLEL PROGRAMMING

Homework 4 Report

Name: Xiang Fei

Student ID: 120090414

Email: xiangfei@link.cuhk.edu.cn

Date: 2022.12

Table of Contents

List of Figures	ii
List of Tables	ii
1 Introduction	1
2 Design and Method	2
2.1 sequential version	2
2.1.1 initialize	2
2.1.2 generate_fire_area	3
2.1.3 update	4
2.1.4 maintain_fire	4
2.1.5 maintain_wall	5
2.2 pthread version	6
2.2.1 update	7
2.2.2 Args	8
2.2.3 worker	9
2.3 MPI version	10
2.4 openmp version	13
2.5 cuda version	14
2.6 bonus	15
3 Execution	16
3.1 Compile and Run	16
3.1.1 compile	16
3.1.2 run	17
3.2 makefile	18
3.3 Sbatch script	18
4 Result & Analysis	20

4.1	Comparison among different problem sizes	22
4.2	Comparison among different process/thread numbers	22
4.3	Comparison among the influence of multithreads and multiporcesses	24
5	Conclusion	26

List of Figures

1	The dynamic scheduling of pthread	6
2	The global variables of pthread	7
3	The Args structure of pthread	8
4	The worker function of pthread	9
5	The flow chart of pthread	10
6	The flow chart of MPI	13
7	The flow chart of openmp version	14
8	The flow chart of cuda version	15
9	The flow chart of my bonus version implementation	15
10	The cluster resource limit	16
11	The beginning GUI output of OpenMP version	20
12	The running GUI output of OpenMP version	21
13	The GUI output of OpenMP version after 5000 iterations	21

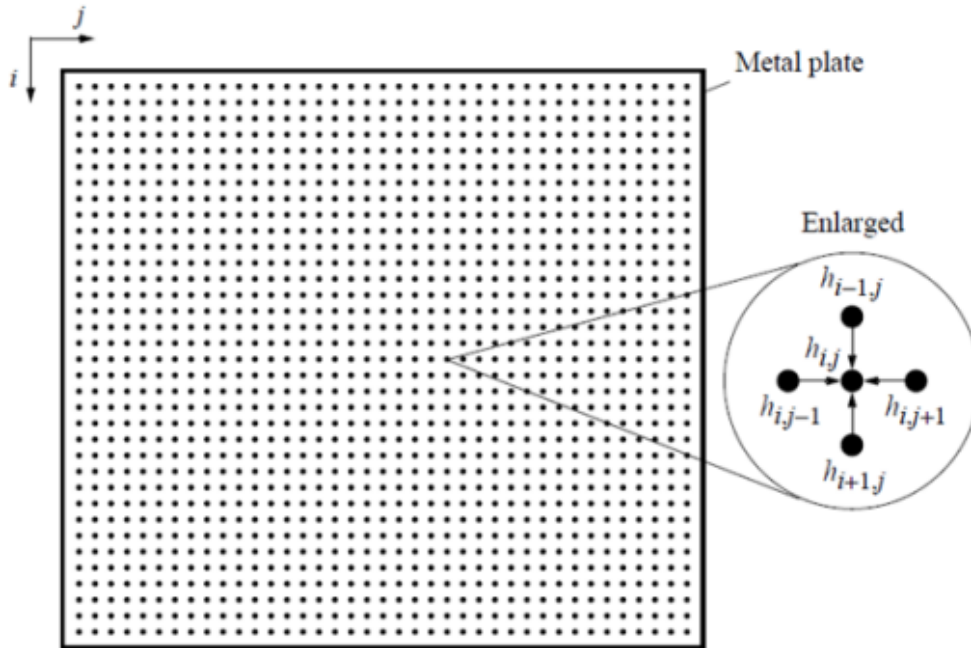
List of Tables

1 Introduction

Assignment 4 requires us to design a program for Heat distribution simulation and render the temperature of each point in the given room on screen with a type of GUI. In a two-dimension space, there are four walls and a fireplace. The walls have certain temperature and the fire source emits the highest heat to warm this room. Different temperature represents different color, with red and blue means high and low temperature respectively.

Over time, heat spreads into adjacent spaces. Here we use Jacobi iterations to simulate the heat distribution. In each pass, the temperature at the current point is simply the average of the four neighboring points. When the difference between the previous temperature value and the current temperature value reaches a critical value, the tolerance, the iteration stops and the room becomes stable.

Our task in the assignment is to continuously calculate the temperature at each point in the room for a certain amount of time and use a GUI method to render the thermal state on the screen and explore the relationship between the calculation speed and different parameters.



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

In this homework, we need to implement the sequential version code, parallel versions using MPI and Pthread and openMP and CUDA approaches. The combination of MPI & openMP version serves as the bonus.

MPI (Message Passing Interface) is the standard of message passing function library that includes many interfaces. It is a parallel technology based on the communication between

processes. A sample MPI interface is shown as follows. MPI_Send is one of the most popular functions in MPI.

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

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

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

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

2 Design and Method

In this part, I illustrate the design logic of each version of program. In addition, some details of the code implementation will also be elaborated. All of the implementations are based on the template given by TAs.

2.1 sequential version

In the implementation of sequential version program, several functions are needed to introduced, which implement the core logic of this program.

2.1.1 initialize

```
void initialize(float *data) {
    // intialize the temperature distribution
    int len = size * size;
    for (int i = 0; i < len; i++) {
        data[i] = wall_temp;
    }
}
```

In this function, I initialize the temperature distribution. The temperature of each point is stored into the input data array, whose length is `size * size`.

2.1.2 generate_fire_area

```
void generate_fire_area(bool *fire_area){
    // generate the fire area
    int len = size * size;
    for (int i = 0; i < len; i++) {
        fire_area[i] = 0;
    }

    float fire1_r2 = fire_size * fire_size;
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            int a = i - size / 2;
            int b = j - size / 2;
            int r2 = 0.5 * a * a + 0.8 * b * b - 0.5 * a * b;
            if (r2 < fire1_r2) fire_area[i * size + j] = 1;
        }
    }

    float fire2_r2 = (fire_size / 2) * (fire_size / 2);
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            int a = i - 1 * size / 3;
            int b = j - 1 * size / 3;
            int r2 = a * a + b * b;
            if (r2 < fire2_r2) fire_area[i * size + j] = 1;
        }
    }
}
```

In this function, I generate the fire area. I use 1 and 0 to identify whether a point is belong to fire area or not. The information is stored in the `fire_area` array, whose length is `size * size`.

2.1.3 update

```
void update(float *data, float *new_data) {  
    // update the temperature of each point, and store the res  
    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;  
        }  
    }  
}
```

In this function, I update the temperature of each point except the boundary points, and store the result in new_data array to avoid data racing.

2.1.4 maintain_fire

```
void maintain_fire(float *data, bool* fire_area) {  
    // maintain the temperature of fire  
    int len = size * size;  
    for (int i = 0; i < len; i++){  
        if (fire_area[i]) data[i] = fire_temp;  
    }  
}
```

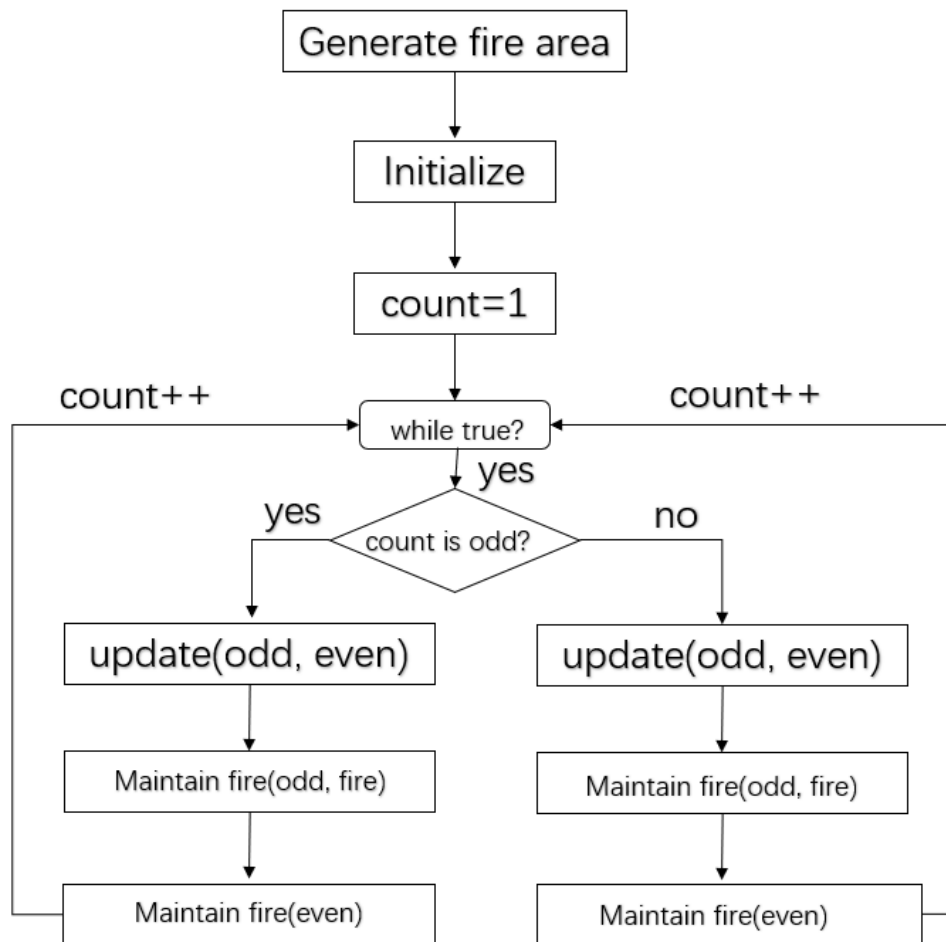
In this function, I maintain the temperature of fire area points, since the update step change their temperature.

2.1.5 maintain_wall

```
void maintain_fire(float *data, bool* fire_area) {  
    // maintain the temperature of fire  
    int len = size * size;  
    for (int i = 0; i < len; i++){  
        if (fire_area[i]) data[i] = fire_temp;  
    }  
}
```

This function is used to maintain the temperature of the wall. However, since the update function will not affect the boundary points, so this function is not needed. But in order to practice my parallel coding ability, I implement this function.

The flow chart of sequential version is as follows:



2.2 pthread version

For the pthread version, the first thing is to make sure that how to parallel, or we can say, how to partition the problem. In my program, for `update`, `maintain_fire` and `maintain_wall` functions, I use a dynamic scheduling considering batch size. At the beginning, each thread compute for one batch, which consists of a certain number of points (it can be adjusted). The purpose to consider batch is to reduce the time consuming of things which is not the core part of computing, and it can increase the efficiency. When one of the threads finish its computing, then the thread will compute the next batch. We can say "first finish, first work to another". When all the batches are computed, then we finished. The following graph shows the scheduling method of my pthread program (I take three threads as an example).

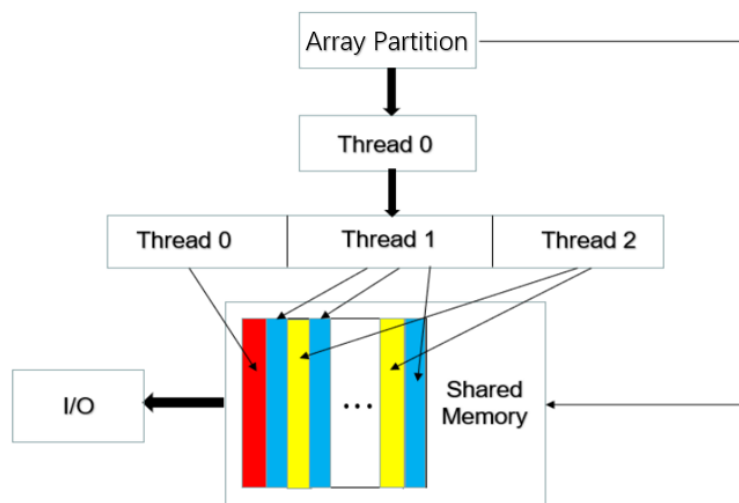


Figure 1: The dynamic scheduling of pthread

The array in the graph is the temperature of the points that should be computed. You can see the graph, at the beginning, thread 0, 1, 2 are assigned to the first three batches, then the thread 1 finishes its work, so this thread can go to work for the fourth batch, and so on, until all the batches are handled.

Then, I will introduce the code in detail.

First is the global variables I added.

```
int batch_size;
int current_iteration;
int max_iteration;
pthread_mutex_t mutex;
pthread_barrier_t barrier;
```

Figure 2: The global variables of pthread

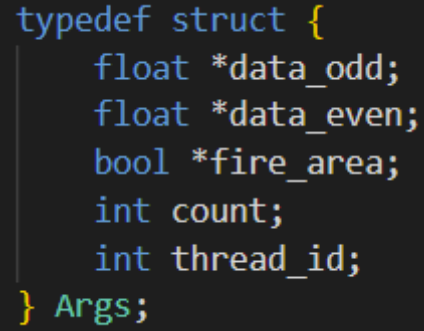
mutex is used to prevent conflicts. **barrier** is used to implement synchronization. **current_iteration** and **max_iteration** are used to implement dynamic scheduling, the details will be introduced in the later part, which is the introduction of important implementations (I only focus on the difference part with the sequential version, so do all the following part).

2.2.1 update

```
void update(float *data, float *new_data) {
    // update the temperature of each point, and store the result in `new_data`
    int local_iteration = current_iteration+1;
    current_iteration++;
    int len = size * size;
    while(true) {
        int start = local_iteration*batch_size;
        int end = local_iteration*batch_size + batch_size;
        for (int i = start; i < end; i++){
            if ((i<size)|| (i%size==0)|| (i%size==size-1)|| (i>len-size-1)){
                continue;
            }
            float up = data[i - size];
            float down = data[i + size];
            float left = data[i - 1];
            float right = data[i + 1];
            float new_val = (up + down + left + right) / 4;
            new_data[i] = new_val;
        }
        pthread_mutex_lock(&mutex);
        local_iteration = current_iteration + 1;
        current_iteration++;
        pthread_mutex_unlock(&mutex);
        if (local_iteration > max_iteration-1){
            break;
        }
    }
}
```

In this function, **current_iteration** is used to find the next free (has not been handled) particle. And **local_iteration** is the position of the thread handling batch. Here, **mutex** is used to avoid race, and to make sure that the next batch will be handled by only one thread. In fact, function **maintain_fire** and **maintain_wall** use the same method.

2.2.2 Args



```
typedef struct {  
    float *data_odd;  
    float *data_even;  
    bool *fire_area;  
    int count;  
    int thread_id;  
} Args;
```

Figure 3: The Args structure of pthread

This structure stores the arguments for threads. **data_odd** is the odd count data array, **data_even** is the even count data array, **fire_area** is the fire area boolean array, **count** is the number of iteration, **thread_id** is the thread id.

2.2.3 worker

```
void *worker(void *args) {
    Args* arg = (Args*)args;
    if (arg->count%2==1) {
        update(arg->data_odd, arg->data_even);
        pthread_barrier_wait(&barrier);
        current_iteration = 0;
        maintain_fire(arg->data_even, arg->fire_area);
        pthread_barrier_wait(&barrier);
        current_iteration = 0;
        maintain_wall(arg->data_even);
    } else {
        update(arg->data_even, arg->data_odd);
        pthread_barrier_wait(&barrier);
        current_iteration = 0;
        maintain_fire(arg->data_odd, arg->fire_area);
        pthread_barrier_wait(&barrier);
        current_iteration = 0;
        maintain_wall(arg->data_odd);
    }
}
```

Figure 4: The worker function of pthread

In this function, the most importance thing is the use of **pthread_barrier_wait**, which is used to implement synchronization. By using the synchronization technique, we can make sure that we will update the temperature after the last step computing is finished

Finally, we can get the flow chart of my pthread program.

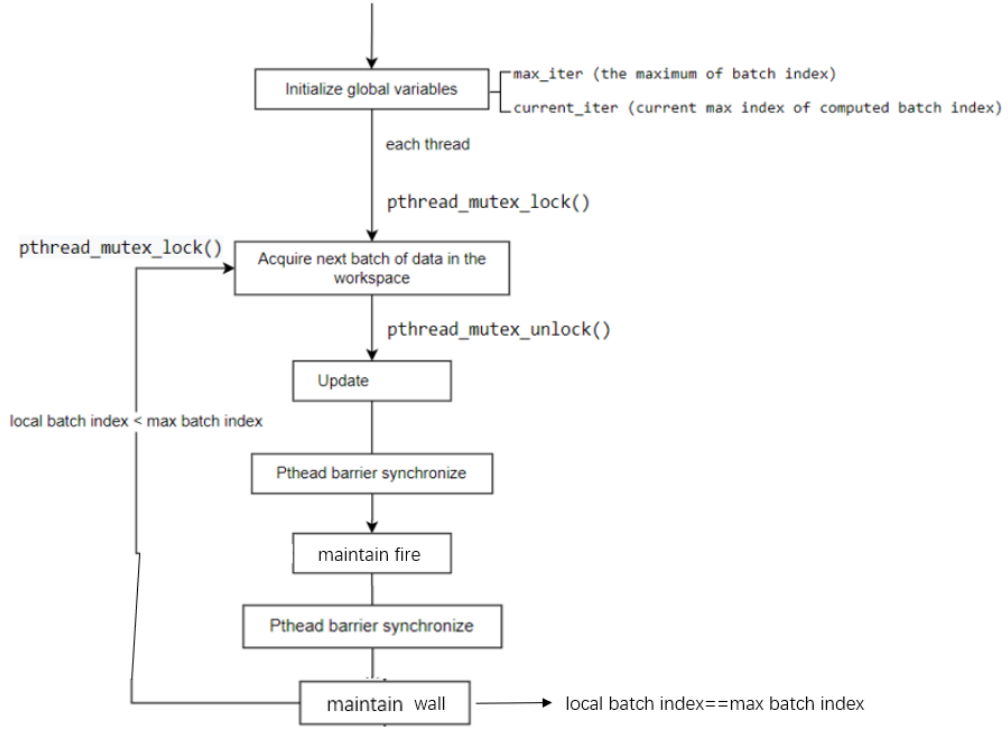


Figure 5: The flow chart of pthread

2.3 MPI version

In the MPI version, I just use a static scheduling, which means I assigned an equal amount of particles to each thread to calculate.

The implementation of my MPI version program is based on **MPI_Scatter**, **MPI_Gather**, **MPI_Send** and **MPI_Recv**.

```

MPI_Scatter(fire_area, (size*size/world_size), MPI_BYTE, slave_fire_area, (size*size/world_size), MPI_BYTE, 0, MPI_COMM_WORLD);
while (true){
    MPI_Scatter(data_odd, (size*size/world_size), MPI_FLOAT, slave_odd, (size*size/world_size), MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Scatter(data_even, (size*size/world_size), MPI_FLOAT, slave_even, (size*size/world_size), MPI_FLOAT, 0, MPI_COMM_WORLD);
    if (my_rank == 0){
        t1 = std::chrono::high_resolution_clock::now();
    }

    if (count%2 == 1) {
        update(slave_odd, slave_even, down_bound, up_bound);
        maintain_fire(slave_even, slave_fire_area);
        maintain_wall(slave_even);
    } else {
        update(slave_even, slave_odd, down_bound, up_bound);
        maintain_fire(slave_odd, slave_fire_area);
        maintain_wall(slave_odd);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (my_rank == 0){
        t2 = std::chrono::high_resolution_clock::now();
        this_time = std::chrono::duration<double>(t2 - t1).count();
        total_time += this_time;
        printf("Iteration %d, elapsed time: %.6f\n", count, this_time);
    }
    count++;

    MPI_Gather(slave_odd, (size*size/world_size), MPI_FLOAT, data_odd, (size*size/world_size), MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Gather(slave_even, (size*size/world_size), MPI_FLOAT, data_even, (size*size/world_size), MPI_FLOAT, 0, MPI_COMM_WORLD);
}

```

Here, **MPI_Scatter** is used to distribute the work to each process. By using this function, each process gets the information array it should handle. In addition, **MPI_Barrier** here is used to synchronize, so that the gather will be executed after the master process has finished the computing.

And after each process finish its work, we need to use **MPI_Gather** to gather the updated data, so that we can give a right output.

Then, I talked about the computing work of each process. For `maintain_fire` and `maintain_wall`, it is simple, just let each process do its own work like the following code.

```
void maintain_fire(float *data, bool* fire_area) {
    // TODO: maintain the temperature of fire
    int len = (size*size / world_size);
    for (int i = 0; i < len; i++){
        if (fire_area[i]) data[i] = fire_temp;
    }
}

void maintain_wall(float *data) {
    // TODO: maintain the temperature of the wall
    int len = (size*size / world_size);
    for(int i=my_rank*len;i<len;i++){
        if(i<size){ // up wall
            data[i] = wall_temp;
        }
        else if(i%size==0){ // left wall
            data[i] = wall_temp;
        }
        else if(i%size==size-1){ // right wall
            data[i] = wall_temp;
        }
        else if(i>len-size-1){ // down wall
            data[i] = wall_temp;
        }
    }
}
```

For update function, we need to use MPI.Send and MPI.Recv to pass data between different processes. For the processes except the process handling the upper bound and the process handling the lower bound, they should get the data from the above process and under process, so that that can update the temperature according to the neighbor points.

```

// TODO: update the temperature of each point, and store the result in `new_data` to avoid data
int len = size / world_size;
if (my_rank!=0){
    for (int i=0;i<size;i++){
        up_bound[i] = data[i];
    }
    MPI_Send(up_bound,size,MPI_FLOAT,my_rank-1,my_rank,MPI_COMM_WORLD);
}
if (my_rank!=world_size-1){
    MPI_Recv(up_bound,size,MPI_FLOAT,my_rank+1,my_rank+1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
if (my_rank!=world_size-1){
    for (int i=0;i<size;i++){
        down_bound[i] = data[(len-1)*size+i];
    }
    MPI_Send(down_bound,size,MPI_FLOAT,my_rank+1,my_rank,MPI_COMM_WORLD);
}
if (my_rank!=0){
    MPI_Recv(down_bound,size,MPI_FLOAT,my_rank-1,my_rank-1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}

```

Finally, we can also get the flow chart of my MPI version program.

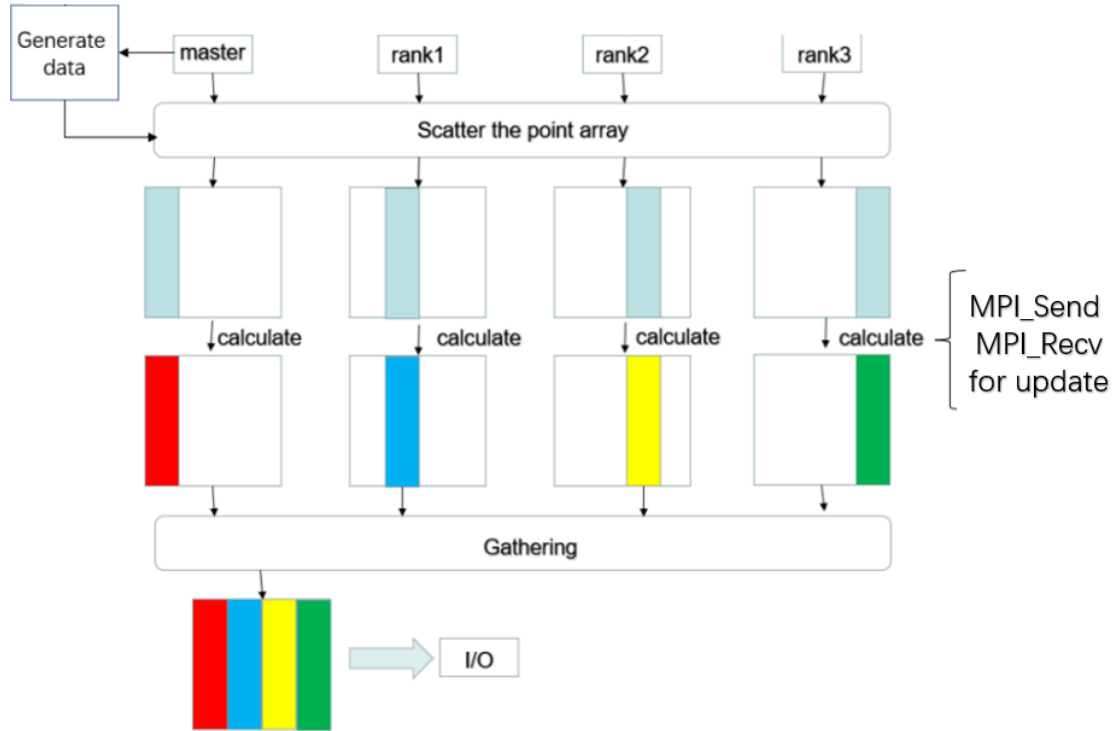


Figure 6: The flow chart of MPI

2.4 openmp version

For the openmp version implementation, we just need to make a small change with the sequential version. For initialize, generate_fire_area, maintain_fire, maintain_wall, data2pixels functions, I use `#pragma` in the for loop (and also outer for loop), so that we can use multi-

threads to solve the problem and implement parallelization.

please pay attention that here we should make the inner for loop used variables private.

Finally, we can get the flow chart of my openmp version program.

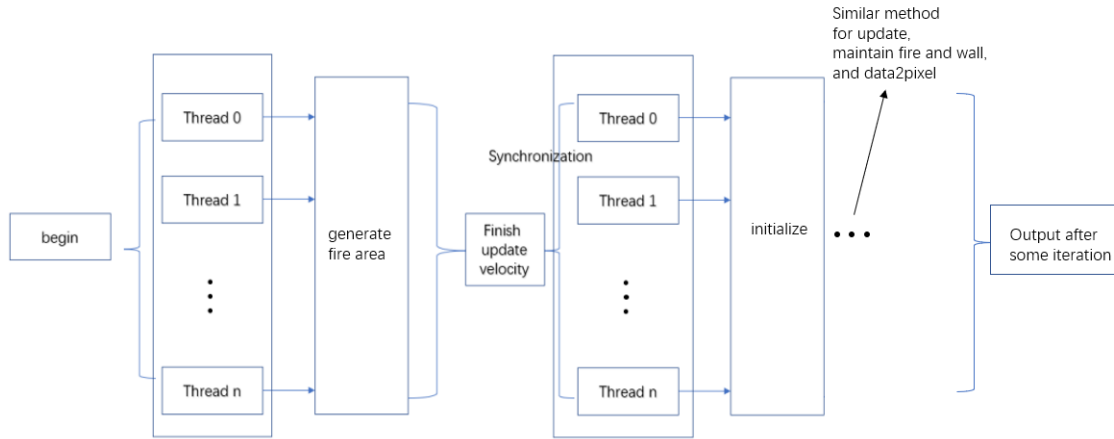


Figure 7: The flow chart of openmp version

2.5 cuda version

For my cuda version implementation, I also remove the outer for loop, since I can let each thread finishes one iteration of outer for loop work. and in order to maintain the data consistency, I use “_syncthreads()” function after the computation of update and maintain_fire each thread, therefore, the next computing will be handled only when the previous computing has finished.

In addition, I use **cudaMalloc** to allocate the memory in device, and I use **cudaMemcpy** to implement the memory copy from host to device or from device to host.

```
cudaMalloc(&device_data_odd, size * size * sizeof(float));
cudaMalloc(&device_data_even, size * size * sizeof(float));
cudaMalloc(&device_fire_area, size * size * sizeof(bool));

#ifdef GUI
GLubyte *pixels;
GLubyte *host_pixels;
host_pixels = new GLubyte[resolution * resolution * 3];
cudaMalloc(&pixels, resolution * resolution * 3 * sizeof(GLubyte));
#endif
```

Finally, the flow chart of my cuda version program can be obtained:

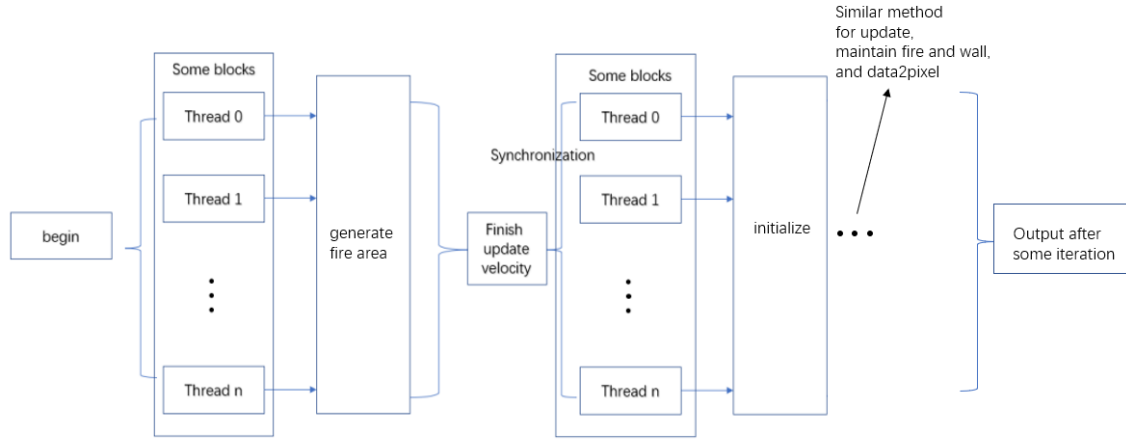


Figure 8: The flow chart of cuda version

2.6 bonus

For my bonus work, I use both MPI and openmp to parallelize the outer for loop, which means I can distribute the computation work to different threads of different process. In fact, we just need to add `#pragma` before the outer for loop in both `update`, `maintain_wall` and `maintain_fire` functions, and the remaining is the same with the MPI version (also use `MPI_Scatter`, `MPI_Gather`, `MPI_Send`, `MPI_Recv`)

Finally, we can get the flow chart of my bonus version program.

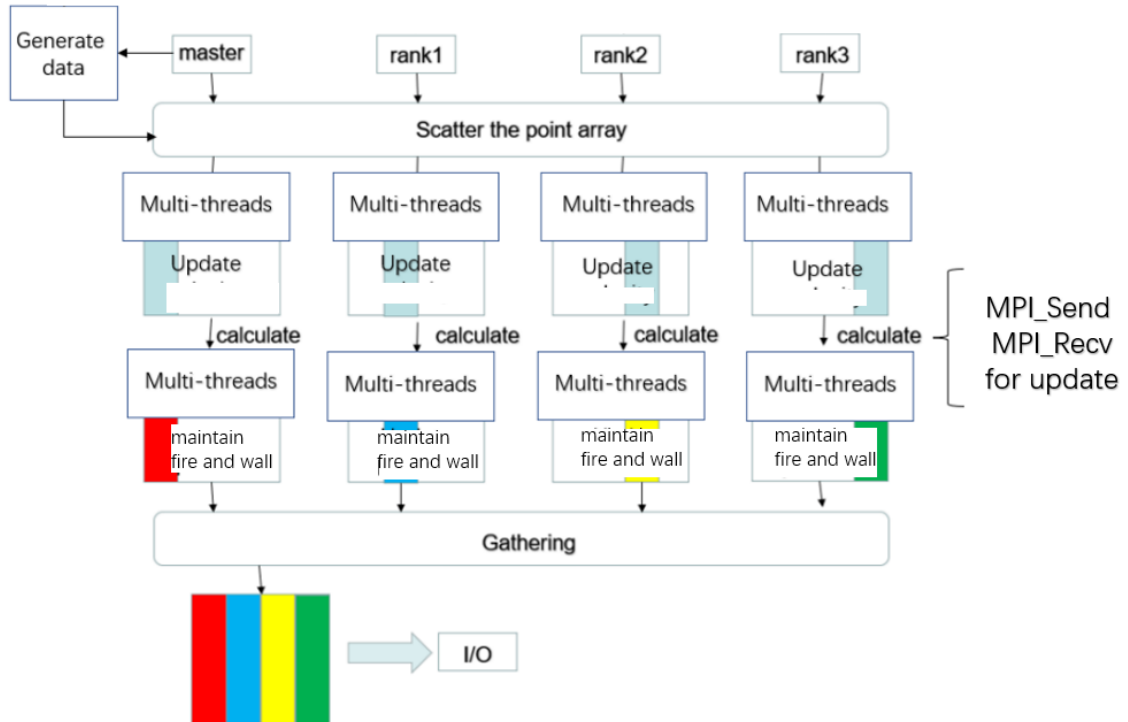


Figure 9: The flow chart of my bonus version implementation

3 Execution

My code executes on the remote cluster and my virtual machine (for GUI). The cluster resource limits are shown in the following graph.

Cluster Resource Limit

Oct 1, 2022

partition	Debug	Project
MaxNodes (max number of nodes allocated to a job)	1	1
MaxCPUsPerNode (max number of cpus on a node allocated to a job)	8	40
Max number of total cpu cores allocated to a job	1*8=8	1*40=40
MaxTime (max running time of a job)	60min	10min
MaxJobsPerUser (max number of running jobs of a user at a given time)	2	1
MaxSubmitJobsPerUser (max number of jobs submitted of a user at a given time)	10	100
Total number of nodes in this partition (we will add more if not enough)	9	18

For time consuming jobs with a few cores, you can use `Debug` partition.

For jobs requiring many cores, you can use `Project` partition.

Figure 10: The cluster resource limit

3.1 Compile and Run

In fact, my execution strictly follows the instruction in the readme file of the given template.

3.1.1 compile

- **Sequential (command line application):**

```
1 g++ ./src/sequential.cpp -o seq -O2 -std=c++11
```

- **Sequential (GUI application):**

```
1 g++ ./src/sequential.cpp -o seqg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
2 -lGLU -lGL -lm -DGUI -O2 -std=c++11
```

- **MPI (command line application):**

```
1 mpic++ ./src/mpi.cpp -o mpi -std=c++11
```

- **MPI (GUI application):**

```
1 mpic++ ./src/mpi.cpp -o mpig -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
2 -lGLU -lGL -lm -DGUI -std=c++11
```

- **Pthread (command line application):**

```
1 g++ ./src/pthread.cpp -o pthread -lpthread -O2 -std=c++11
```

- **Pthread (GUI application):**

```
1 g++ ./src/pthread.cpp -o pthreadg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut  
2 -IGLU -IGL -lm -lpthread -DGUI -O2 -std=c++11
```

- **CUDA (command line application):** notice that 'nvcc' is not available on VM, please use cluster.

```
1 nvcc ./src/cuda.cu -o cuda -O2 --std=c++11
```

- **CUDA (GUI application):** notice that 'nvcc' is not available on VM, please use cluster.

```
1 nvcc ./src/cuda.cu -o cudag -I/usr/include -L/usr/local/lib -L/usr/lib -lglut  
2 -IGLU -IGL -lm -O2 -DGUI --std=c++11
```

- **OpenMP (command line application):**

```
1 g++ ./src/openmp.cpp -o openmp -fopenmp -O2 -std=c++11
```

- **OpenMP (GUI application):**

```
1 g++ ./src/openmp.cpp -o openmpg -fopenmp -I/usr/include -L/usr/local/lib  
2 -L/usr/lib -lglut -IGLU -IGL -lm -O2 -DGUI -std=c++11
```

- **bonus (command line application):**

```
1 mpic++ ./src/bonus.cpp -o bonus -fopenmp -std=c++11
```

- **bonus (GUI application):**

```
1 mpic++ ./src/bonus.cpp -o bonusg -fopenmp -I/usr/include -L/usr/local/lib  
2 -L/usr/lib -lglut -IGLU -IGL -lm -O2 -DGUI -std=c++11
```

3.1.2 run

- **Sequential**

```
1 ./seq $problem_size
```

- **MPI**

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

- **Pthread**

```
1 ./pthread $problem_size
```

- **CUDA**

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

- **OpenMP**

```
1 openmp $problem_size $n_omp_threads
```

- **bonus**

```
1 mpirun -np $n_processes ./bonus $problem_size $n_omp_threads
```

3.2 makefile

Makefile helps you simplify compilation command.

```
1 make $command
```

where ‘command’ is one of ‘seq, seqg, mpi, mpig, pthread, pthreadg, cuda, cudag, openmp, openmpg’.

When you need to recompile, please first run ‘make clean’!

3.3 Sbatch script

MPI

For MPI program, you can use

```
1 #!/bin/bash
2 #SBATCH --job-name=your_job_name # Job name
3 #SBATCH --nodes=1                # Run all processes on a single node
4 #SBATCH --ntasks=20              # number of processes = 20
5 #SBATCH --cpus-per-task=1        # Number of CPU cores allocated to each process
6 #SBATCH --partition=Project      # Partition name: Project or Debug
7
8 cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
9 mpirun -np 4 ./mpi 800
10 mpirun -np 20 ./mpi 800
11 mpirun -np 40 ./mpi 800
```

Pthread

For pthread program, you can use

```
1 #!/bin/bash
2 #SBATCH --job-name=your_job_name # Job name
3 #SBATCH --nodes=1                # Run all processes on a single node
4 #SBATCH --ntasks=1               # number of processes = 1
5 #SBATCH --cpus-per-task=20 # Number of CPU cores allocated to each process
6 #SBATCH --partition=Project      # Partition name: Project or Debug
7
8 cd /nfsmnt/119010355/CSC4005_2022Fall.Demo/project3_template/
9 ./pthread 1000 4
10 ./pthread 1000 20
11 ...
```

here you can create as many threads as you want while the number of cpu cores are fixed.

CUDA

For CUDA program, you can use

```
1 #!/bin/bash
2
3 #SBATCH --job-name CSC3150CUDADemo ## Job name
4 #SBATCH --gres=gpu:1                ## Number of GPUs required for job execution.
5 #SBATCH --output result.out         ## filename of the output
6 #SBATCH --partition=Project         ## the partitions to run in (Debug or Project)
7 #SBATCH --ntasks=1                 ## number of tasks (analyses) to run
8 #SBATCH --gpus-per-task=1          ## number of gpus per task
9 #SBATCH --time=0-00:02:00          ## time for analysis (day-hour:min:sec)
10
11 ## Run the script
12 srun ./cuda 800
```

openmp

For openmp program, you can use

```
1 #!/bin/bash
2
3 #SBATCH --job-name job_name ## Job name
4 #SBATCH --output result.out  ## filename of the output
5 #SBATCH --partition=Project  ## the partitions to run in (Debug or Project)
6 #SBATCH --ntasks=1          ## number of tasks (analyses) to run
7 #SBATCH --gpus-per-task=1    ## number of gpus per task
8 #SBATCH --time=0-00:02:00    ## time for analysis (day-hour:min:sec)
```

```
9
10 ## Compile the cuda script using the nvcc compiler
11 ## You can compile your codes out of the script and simply srun the executable file .
12 ## Run the script
13 ./openmp 800 20
```

To submit your job, use

```
1 sbatch xxx.sh
```

4 Result & Analysis

First, I show the GUI output (take OpenMP case as an example).

Beginning:

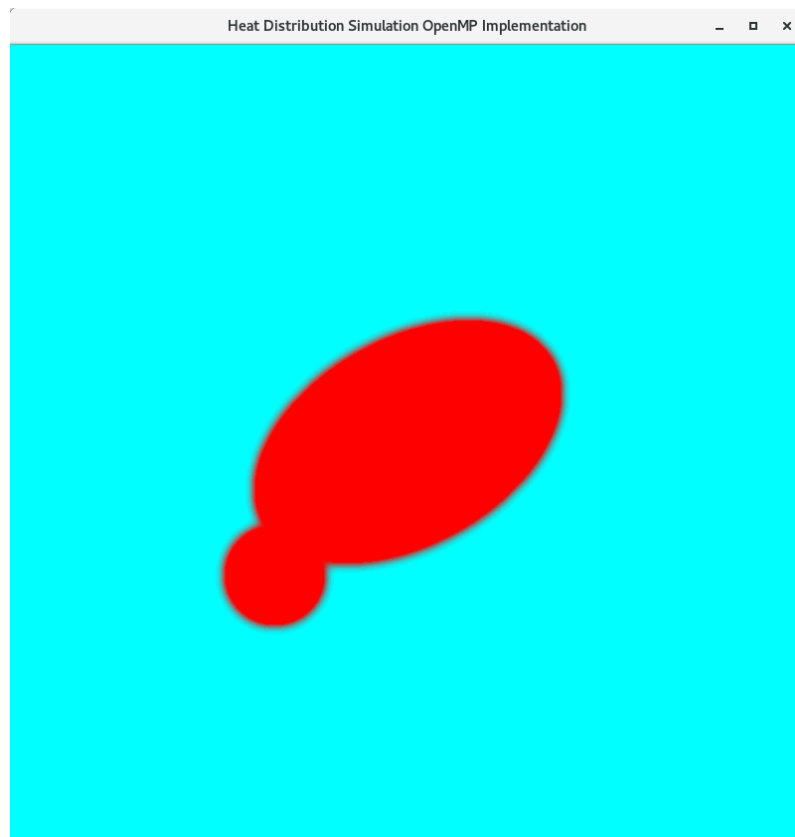


Figure 11: The beginning GUI output of OpenMP version

Running:



Figure 12: The running GUI output of OpenMP version

After 5000 iterations:

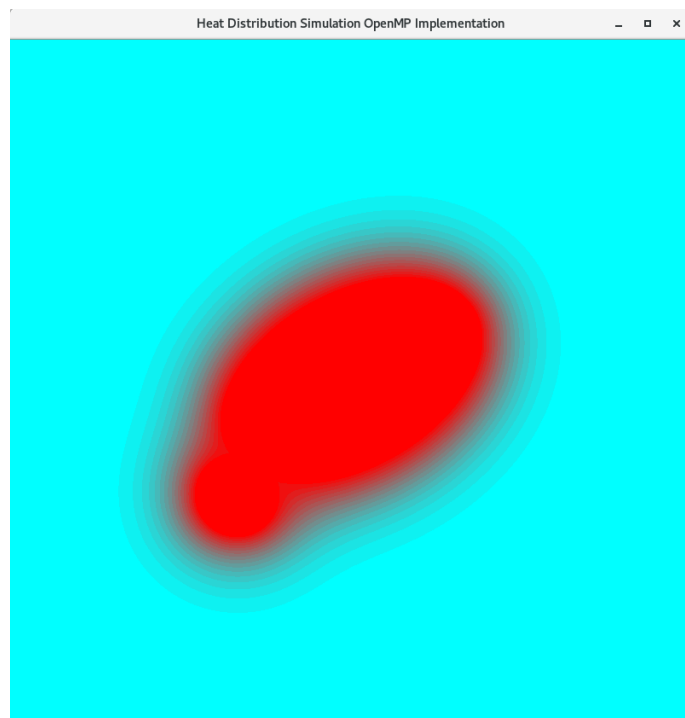


Figure 13: The GUI output of OpenMP version after 5000 iterations

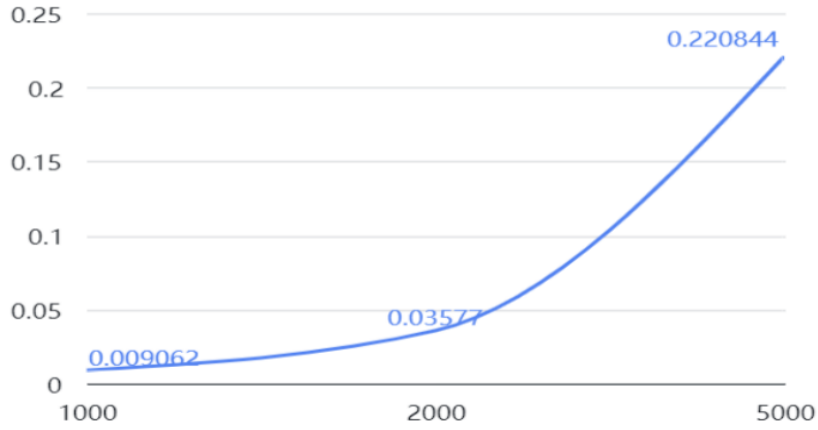
4.1 Comparison among different problem sizes

In this section, we simply consider the sequential version of Heat simulation. Note that, For images in this subsection, all y-axis represents the running time of one iteration (here we choose the mode of the running time in each iteration as the time to prevent the influence of abnormal data) while x-axis represents the problem size.

Assume the average computation time of temperature (including maintain wall and maintain fire) for each pixel is identical, if we double the problem size, the running time increases to four times. In this way, we can estimate the time complexity of Heat simulation in one iteration.

$$complexity = O(n^2)$$

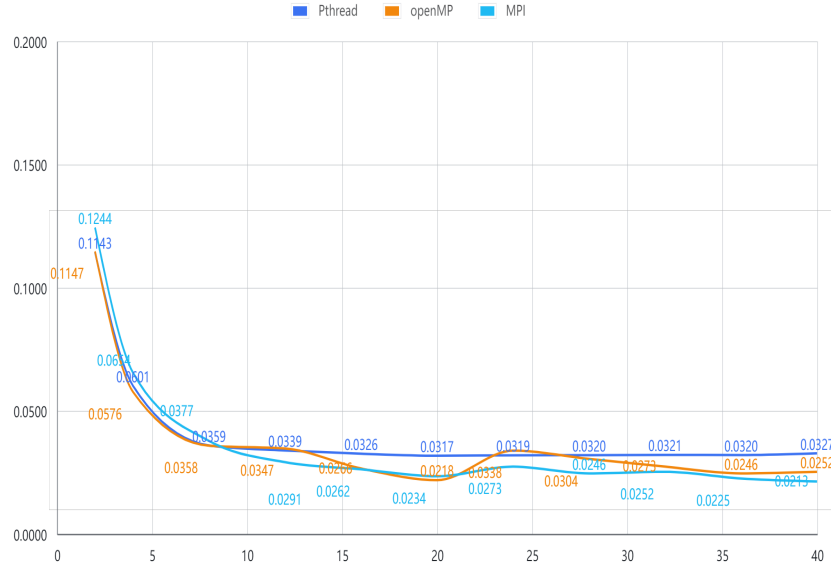
(time complexity)



As we can see from the result, the sequential version of Heat simulation satisfies the n-square time complexity.

4.2 Comparison among different process/thread numbers

In this section we consider all versions with different number of processes/threads. Here we fix the size number to 5000, and analyze the image where y-axis represents the running time of one iteration and x-axis represents the number of processes/threads. In the figure below, the deep blue line represents the Pthread version, the orange line represents the openMp version and the baby blue line represents the MPI version. As we can observe from the figure, as the number of processes/threads increases, we can significantly reduce the time cost of the same program task. This is because that regarding the same amount of data, we let multiple cores to process the data simultaneously. The larger the problem scale is, the more amount of time we can reduce by using multiple cores.



Then we seek to analyze the speedup of program w.r.t different number of processes/threads. Here we still fix the problem size to 5000 and consider the Pthread version, by the definition of the speedup, we can compute the speed up of the program in terms of different number of processes/threads.

$$S(n) = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \quad (\text{speedup definition})$$



As we can see in the figure, the speedup does not conform a linear relationship with the process/thread number. This phenomenon can be explained by the Amdahl's Law.

$$S(n) = \frac{W_s + W_p}{W_s + W_p/n} = \frac{f + (1-f)}{f + \frac{1-f}{n}} = \frac{n}{1 + f(n-1)} \quad (\text{Amdahl's Law})$$

n : amount of cores

W_s : sequential workload

W_p : parallel workload

f : W_s/W (the proportion of sequential workload)

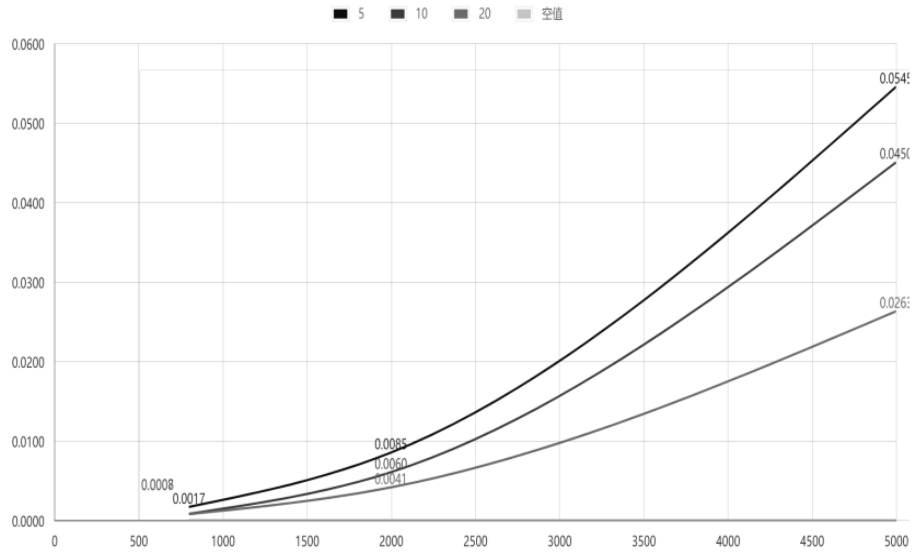
According to the Amdahl's Law, As n (which represents the process/thread number) approaches infinity, the value of speedup will become $1/f$, where f is the proportion of sequential workload. In other words, the speedup will not increase linearly along with the core amount. Instead, the asymptotic line of the speedup will be $S_n = \frac{1}{f}$

$$S(n) = \frac{W_s + W_p}{W_s + W_p/n} = \frac{f + (1-f)}{f + \frac{1-f}{n}} = \frac{n}{1 + f(n-1)} \Rightarrow \frac{1}{f} \quad \text{if } n \rightarrow \infty \quad (\text{Amdahl's Law})$$

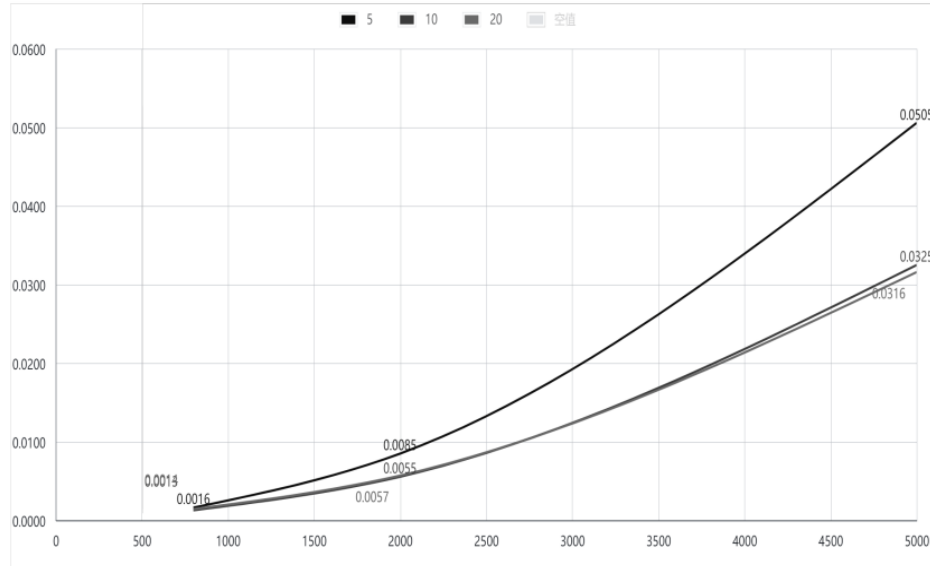
From the above figure, we can observe that the asymptotic line is $S(n) = 7$, thus we can have a guess that the sequential work proportion in problem size 5000 (i.e. the length and width of the output temperature contour image is 5000) is $\frac{1}{7}$.

4.3 Comparison among the influence of multithreads and multiprocesses

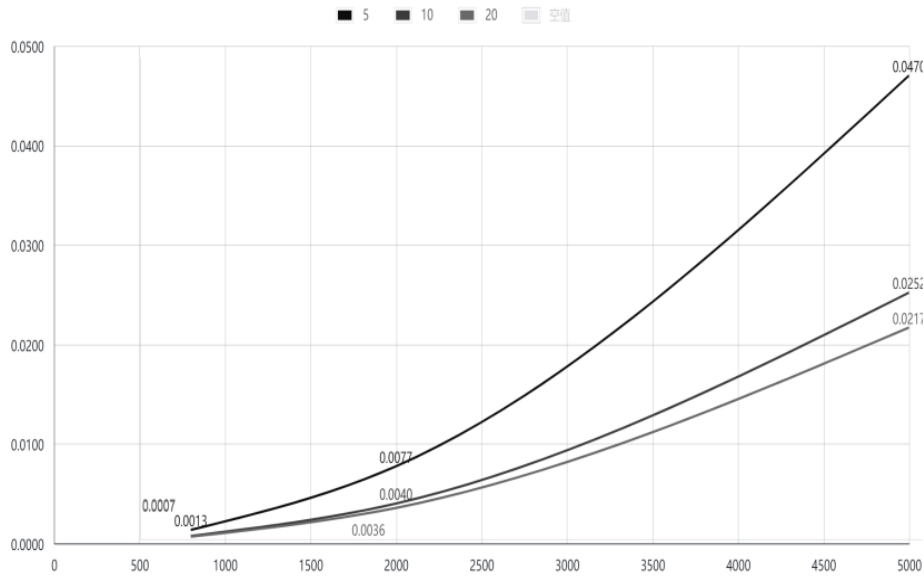
MPI:



Pthread:



OpenMp:



In this section we compare the influence of multiprocesses and multithreads to the program. Here we draw the image for the MPI, pthread, and openMP version of Heat simulation. In our images, the y-axis denotes the running time of one iteration and x-axis denotes the problem sizes. And we also plot 3 cases where the threads/processes number is set to be 5, 10, and 20, respectively. As we can observe from the figures above, as the problem size increases, the time consumptions of the program of all implementations increase. However, the increasing slope is the minimal for OpenMP version, and the maximal for MPI version. The different increase slope may come from different implementation of multiprocesses and multithreads, which is not the main focus of our discussion.

We mainly focus on the increase of running time by multiprocesses and multithreads. Note that, the MPI version uses multiprocesses and the other three version uses multithreads. We can observe from the image that, when the process/thread number increases from 5 to 10, the running time increase very rapidly. However, when the process/thread number increases from 10 to 20, the increase of running time is not much significant. For multiprocesses, the reason is already explained by the Amdahl's law as mentioned above. But for multithreads, why the decrease of running time is much slower if threads number is relatively large? Here we consider a simple case, if we run a program on a personal computer, can we decrease the running time to zero when we increase the thread number to a very large value? Obviously no. Each computer has a CPU containing many cores, each core has a maximum number of threads running on it which is determined by the hardware. If the thread number is larger than the maximum number, threads will race to get the position on CPU cores. In this case, the running time is not improved as the thread number increases. What's worse, because many threads use the resources of CPU alternatively, the running time for each thread may become lower (not necessary, depends on specific situations). Then we consider a program running on the HPC, compared to a personal computer, HPC cluster has many CPUs, thus more threads can run simultaneously on the cluster. However, when the thread number become hundreds, the case in which threads compete for CPU resources can still happen. This may be the reason why the decrease of running time is much slower if thread number is relatively large.

Another interesting observation is that OpenMP and pthread both use multithreads, but the time decrease with respect to the three versions is not identical. This may be resulted from many aspects. One aspect that worths mentioned may be the efficiency of threads. If each thread's workload is predefined (i.e each thread computes a defined number of pixels), the efficiency of threads may lose and time may be wasted because the computation time for each pixel may not be identical (although we assume it is identical in 3.1 for convenience), the stragglers will drag the running time. The time that each thread finishes their job is asynchronous and the final running time is determined by the thread which uses most time. In our pthread implementation, we adopt dynamic allocation, which means all threads keep working until all jobs are done. Therefore, a little increase in thread numbers may contribute to large running time improvements. This phenomenon can also be observed from the image. Consider the problem size equals to 5000, in the pthread version, when the thread number increases from 5 to 10, the running time in one iteration decreases from 0.05 to 0.0325.

5 Conclusion

Here I will summarize what I have learned when writing assignment 4.

- Learn about parallel computing. For example, different processes share different memory, and different threads can share the same global variables. The same pointer cannot be valid all the time. Both CPU and GPU in CUDA have visible regions.

-
- The problems to be noticed when writing parallel programs (process synchronization, process communication, thread creation and join, cost, data race)
 - MPI & Pthread & openmp & CUDA API usage.
 - I learned how to design experiments to compare different implementation in several dimensions.
 - I can manipulate processes and threads more flexibly. For example, using threads to implement dynamic scheduling, and deal with the remainder case for multiple processes.

That's all!!!