CSC4005: Parallel Computing
Project4: Heat Simulation

Name: Xinyu Xie
Student ID: 119020059

Dec 5th, 2022

Project4: Heat Simulation

Name: Xinyu Xie
Student ID: 119020059

Introduction

The heat simulation generates a model in which there are four walls and a fireplace. The temperature of the wall is 20 degree , and the temperature of the fireplace is 100 degree. In this assignment, we make use of Jabobi iteration to do heat simulation.

In this project, I implemented the seqential version, the parallel MPI version, the parallel Pthread version, the parallel OpenMP version, the parallel CUDA version, the parallel MPI + OpenMP version, of the heat simulation. The detailed implementation methods, results and conclusions are provided in the following sections.

Method

Sequential Heat Simulation

The following are the steps to perform sequential heat simulation:

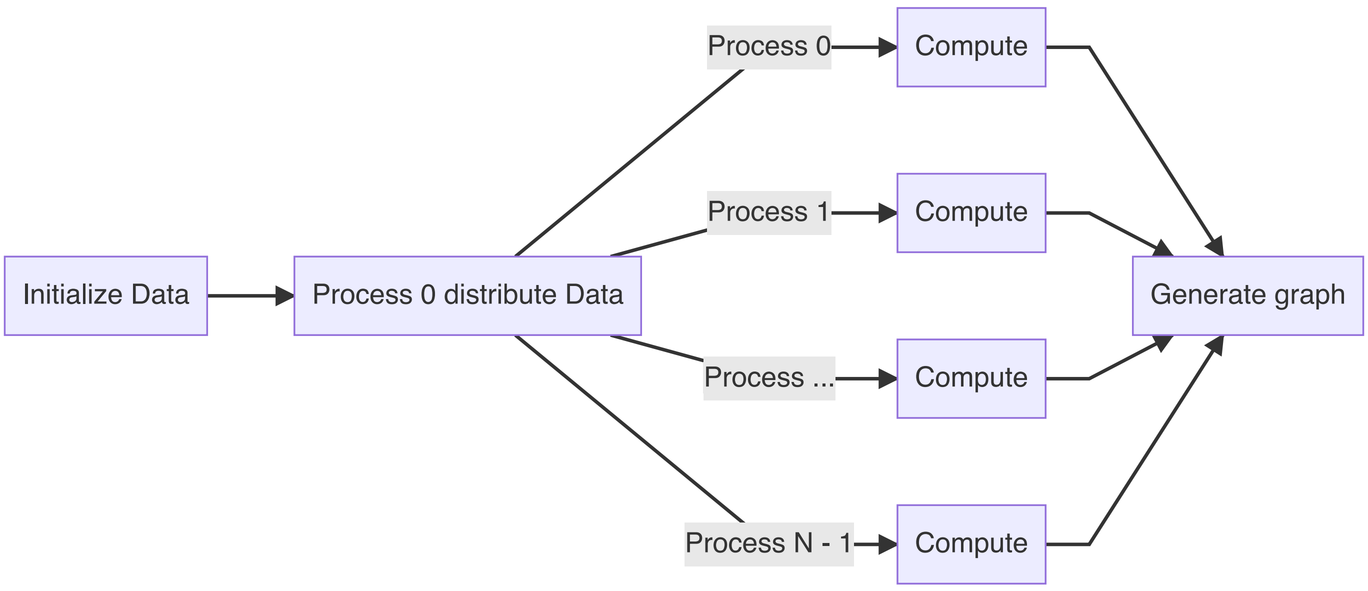
1. Initialize data and generate fire area.
2. In every iteration, for point in local data, update its temperature using Jabobi iteration, which equals to the average temperature of the four points around it (up, down, left, right).
3. Maintain fire area and wall area by setting the corresponding point temperature to predefined constant.
4. Draw the corresponding graph.

Parallel Heat Simulation using MPI

The following are the steps to perform parallel heat simulation using MPI:

1. Initialize data and generate fire area.
2. In total, N points are equally distributed to n processes, respectively. Master process distribute equal number of points to each slave process.
3. Inside each slave process, for point in local data, update its temperature using Jabobi iteration, which equals to the average temperature of the four points around it (up, down, left, right). Each slave process sends the updated last row to next rank and first neighbor row to previous rank. Maintain fire area and wall area by setting the corresponding point temperature to predefined constant. Wait for all slave processes to reach barrier to continue next time iteration.
4. Master process collect the result from each slave process and then draw the corresponding graph.

This flow chart is illustrated in the following figure:

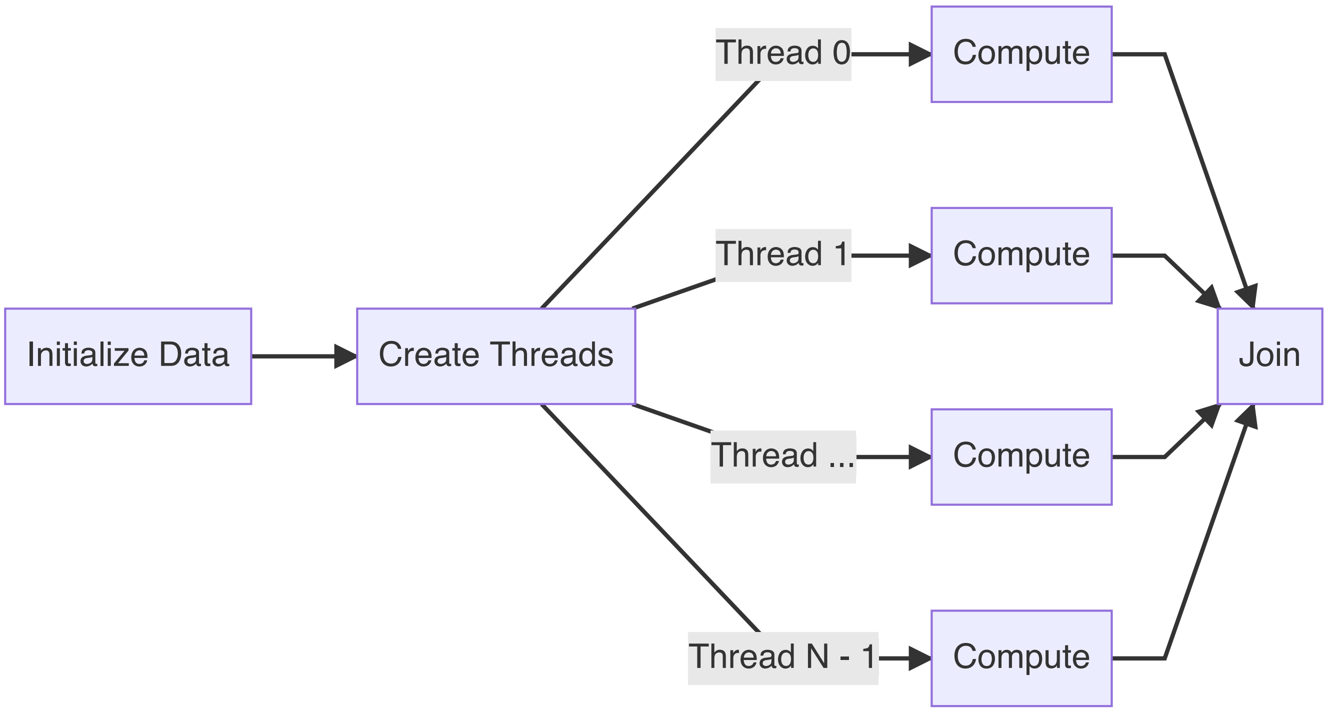


Parallel Heat Simulation using Pthread

The following are the steps to perform parallel heat simulation using Pthread:

1. Initialize data and generate fire area.
2. In total, N points are equally partitioned into n parts.
3. Create n child threads.
4. Inside each child thread, for each point in local data, update its temperature using Jabobi iteration, which equals to the average temperature of the four points around it (up, down, left, right). Maintain fire area and wall area by setting the corresponding point temperature to predefined constant. Wait for all threads to reach barrier to continue next time iteration.
5. Each child thread will join together.

This flow chart is illustrated in the following figure:

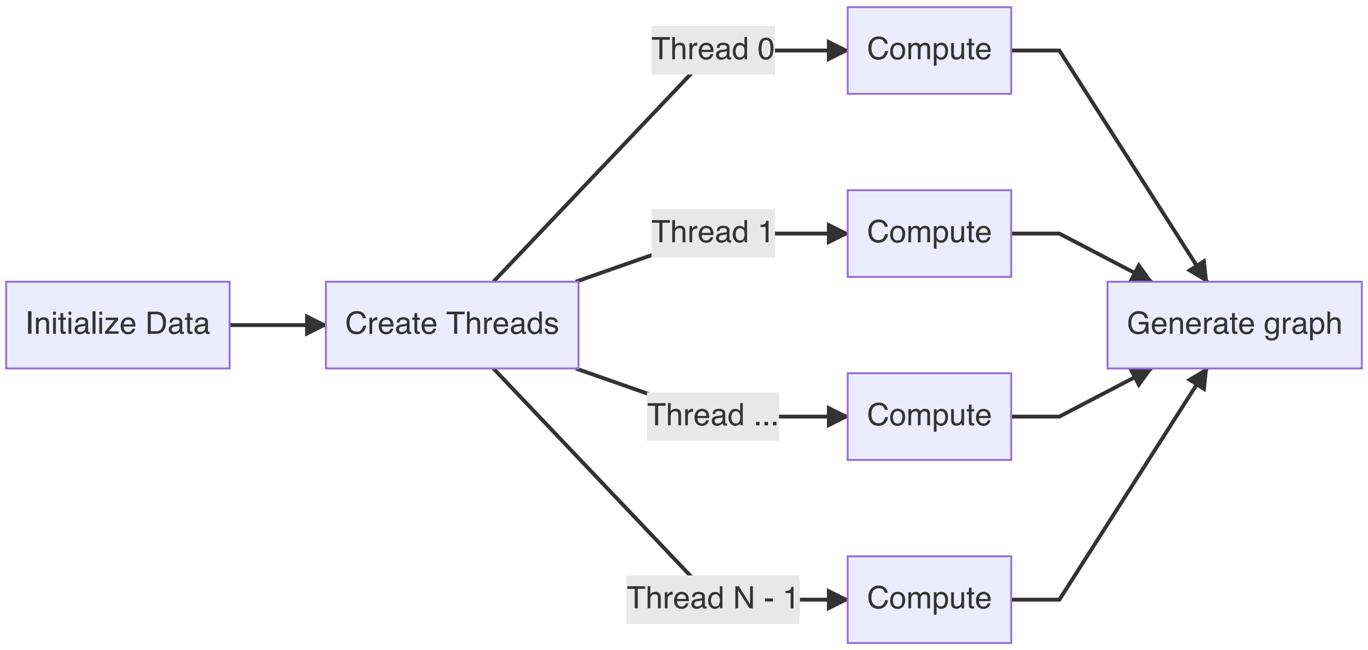


Parallel Heat Simulation using OpenMP

The following are the steps to perform parallel heat simulation using OpenMP:

1. Initialize data and generate fire area.
2. In every iteration, for each point, using parallely for to update its temperature using Jabobi iteration, which equals to the average temperature of the four points around it (up, down, left, right). Use parallely for to maintain fire area and wall area by setting the corresponding point temperature to predefined constant
3. Draw the corresponding graph.

This flow chart is illustrated in the following figure:

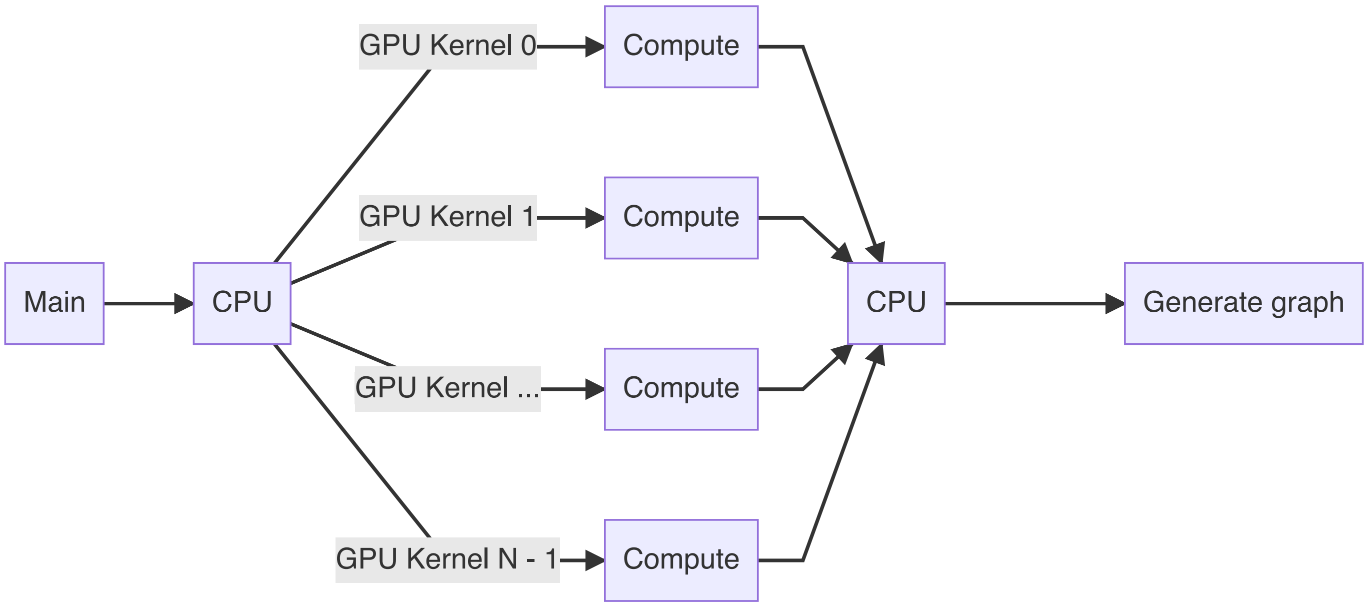


Parallel Heat Simulation using CUDA

The following are the steps to perform parallel heat simulation using CUDA:

1. Initialize data and generate fire area.
2. The main function fork n threads. In total, N points are equally partitioned into n parts, each thread will have 1 or 0 number of point.
3. Inside each thread having 1 point, update its temperature using Jabobi iteration, which equals to the average temperature of the four points around it (up, down, left, right).
4. Draw the corresponding graph.

This flow chart is illustrated in the following figure:

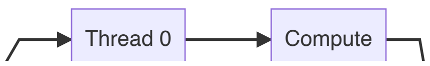


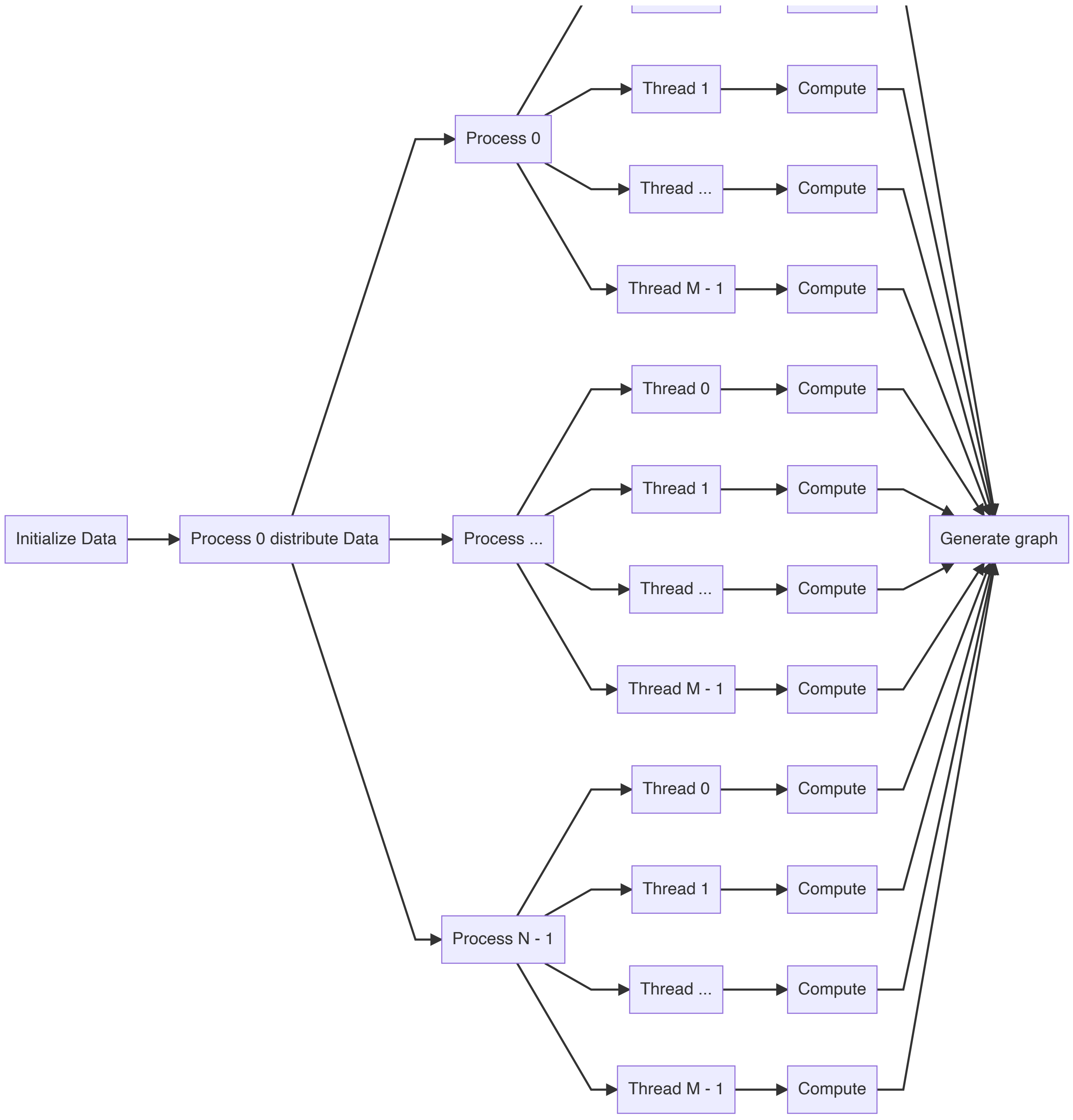
Bonus: Heat N-body Simulation using MPI + OpenMP

The following are the steps to perform parallel N-body simulation using MPI + OpenMP:

1. Initialize data and generate fire area.
2. In total, N points are equally distributed to n processes, respectively. Master process distribute equal number of points to each slave process.
3. Inside each slave process, use OpenMP to generate more threads. In every iteration, for point in local data, update its temperature using Jabobi iteration, which equals to the average temperature of the four points around it (up, down, left, right). Each slave process sends the updated last row to next rank and first neighbor row to previous rank. Use parallel for to maintain fire area and wall area by setting the corresponding point temperature to predefined constant. Wait for all slave processes to reach barrier to continue next time iteration.
4. Master process collect the result from each slave process and then draw the corresponding graph.

This flow chart is illustrated in the following figure:





Initialize Data
Process 0 distribute Data
Process 0
Process ...
Process N - 1
Thread 0
Thread 1
Thread ...
Thread M - 1
Compute
Generate graph

Implementation

Sequential Heat Simulation

The implementation of sequential heat simulation directly follows the method mentioned above.

The following is the core part of detailed implementation code:

```
void update(float *data, float *new_data, bool* fire_area) {
    // update the temperature of each point, and store the result in `new_data` to avoid data
racing
    for (int i = 1; i < (size - 1); i++){
        for (int j = 1; j < (size - 1); j++){
            int idx = i * size + j;
            if (fire_area[idx]) new_data[idx] = fire_temp;
            else {
                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;
            }
        }
    }
}
```

Parallel Heat Simulation using MPI

The implementation of parallel heat simulation using MPI is divided into following steps:

1. Do initialization and distribution of N points.
2. Master process distributes data points to each slave process.
3. Perform local computation in each slave process.
4. Master process gather results and generate results.

The following is the core part of detailed implementation code:

```
void update_once() {
    update(data, new_data);
    // send the last row to next rank
    if (my_rank != world_size - 1) {
        float* last_row = &new_data[displacement[my_rank + 1] * size] - size;
        MPI_Send(last_row, size, MPI_FLOAT, my_rank + 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&new_data[displacement[my_rank + 1] * size], size, MPI_FLOAT, my_rank + 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    // send the previous row the previous rank
    if (my_rank != 0) {
        float* first_neighbor_row = &new_data[displacement[my_rank] * size] - size;
        MPI_Recv(first_neighbor_row, size, MPI_FLOAT, my_rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Send(&new_data[displacement[my_rank] * size], size, MPI_FLOAT, my_rank - 1, 0,
MPI_COMM_WORLD);
    }
    maintain_wall(new_data);
    swap(data, new_data);
}

void slave() {
    while (count <= max_iteration) {
        update_once();
        count++;
        MPI_Barrier(MPI_COMM_WORLD);
    }
}

void master() {
    while (count <= max_iteration) {
        t1 = std::chrono::high_resolution_clock::now();
        update_once();
        MPI_Barrier(MPI_COMM_WORLD);
        t2 = std::chrono::high_resolution_clock::now();
        double this_time = std::chrono::duration<double>(t2 - t1).count();
        if (DEBUG) printf("Iteration %d, elapsed time: %.6f\n", count, this_time);
        total_time += this_time;
        count++;
    }
}
```

Parallel Heat Simulation using Pthread

The implementation of parallel heat simulation using Pthread is divided into following steps:

1. Do partition of N points.
2. Create a thread pool containing n child threads using pthread_create.
3. Perform local computation in each child thread.
4. Each child thread join together using pthread_join.

The following is the core part of detailed implementation code:

```
void update(float *data, float *new_data, int start, int local_size) {
    // update the temperature of each point, and store the result in `new_data` to avoid data
racing
    for (int i = start; i < start + local_size; i++){
        for (int j = 1; j < (size - 1); j++){
            int idx = i * size + j;
            if (fire_area[idx]) new_data[idx] = fire_temp;
            else {
                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;
            }
        }
    }
}

void maintain_wall(float *data, int start, int local_size) {
    // maintain the temperature of the wall
    for (int i = start; i < start + local_size; i++){
        data[i] = wall_temp;
        data[i * size] = wall_temp;
        data[i * size + size - 1] = wall_temp;
        data[(size - 1) * size + i] = wall_temp;
    }
}

typedef struct {
    int start[3]; // array start position, 3 cutting methods
    int size[3]; // array size, 3 cutting methods
} Args;

void* worker(void* args) {
    Args* my_arg = (Args*) args;
    int* start = my_arg->start;
    int* size = my_arg->size;
    while (count <= max_iteration) {
        update(data, new_data, start[2], size[2]);
        pthread_barrier_wait(&worker_barrier);
        maintain_wall(new_data, start[1], size[1]);
        pthread_barrier_wait(&main_barrier);
        pthread_barrier_wait(&main_barrier);
    }
    pthread_exit(NULL);
    return nullptr;
}
```

Parallel Heat Simulation using OpenMP

The implementation of parallel heat simulation using OpenMP is divided into following steps:

1. Do initialization of N points.

2. Using parallel for for each thread to perform local computation.
3. Collect final results.

The following is the core part of detailed implementation code:

```
void update(float *data, float *new_data) {
#pragma omp parallel for
    // update the temperature of each point, and store the result in `new_data` to avoid data
racing
    for (int i = 1; i < (size - 1); i++){
        for (int j = 1; j < (size - 1); j++){
            int idx = i * size + j;
            if (fire_area[idx]) new_data[idx] = fire_temp;
            else {
                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;
            }
        }
    }
}

void maintain_wall(float *data) {
    // maintain the temperature of the wall
#pragma omp parallel for
    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 - 1) * size + i] = wall_temp;
    }
}
```

Parallel Heat Simulation using CUDA

The implementation of parallel heat simulation using CUDA is divided into following steps:

1. Do initialization of N points.
2. Create a thread pool containing n child threads and do partition of N points.
3. Perform local computation in each thread.
4. Each thread join together.

The following is the core part of detailed implementation code:

```
__global__ void update(float *data, float *new_data, int size, bool* fire_area) {
    // update the temperature of each point, and store the result in `new_data` to avoid data
racing
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx - size < 0 || idx + size >= size * size)
        return;
    if (fire_area[idx]) new_data[idx] = fire_temp;
    else {
        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;
    }
}

__global__ void maintain_wall(float *data, int size) {
    // maintain the temperature of the wall
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < size) {
        data[idx] = wall_temp;
        data[idx * size] = wall_temp;
        data[idx * size + size - 1] = wall_temp;
        data[(size - 1) * size + idx] = wall_temp;
    }
}
```

Bonus: Parallel Heat Simulation using MPI + OpenMP

The implementation of parallel heat simulation using MPI + OpenMP is divided into following steps:

1. Do initialization and distribution of N points.
2. Master process distributes data points to each slave process.
3. Perform local computation in each slave process using parallel for for each thread.
4. Master process gather results and generate results.

The following is the core part of detailed implementation code:

```
void update(float *data, float *new_data) {
#pragma omp parallel for
    // update the temperature of each point, and store the result in `new_data` to avoid data
racing
    for (int i = displacement[my_rank]; i < displacement[my_rank + 1]; i++){
        for (int j = 1; j < (size - 1); j++){
            int idx = (i + 1) * size + j;
            if (fire_area[idx]) new_data[i] = fire_temp;
            else {
                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;
            }
        }
    }
}

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

void maintain_wall(float *data) {
    // maintain the temperature of the wall
#pragma omp parallel for
    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 - 1) * size + i] = wall_temp;
    }
}

void slave() {
    while (count <= max_iteration) {
        update_once();
        count++;
    }
}

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

Execution

The following are the steps for execution:

1. To build, go to can root directory, run the following shell scripts to build:

   scripts/cmake_release.sh for non-GUI version

   scripts/cmake_release_gui.sh for GUI version

2. To run, go to the build directory:
3. Use "./seq [n size]" to run sequential version without GUI.
4. Use "./seqg [n size]" to run sequential version with GUI.
5. Use "mpirun -np [number of processes] ./mpi [n size]" to run parallel version using MPI without GUI.
6. Use "mpirun -np [number of processes] ./mpig [n size]" to run parallel version using MPI with GUI.
7. Use "./pthread [n size] [number of threads]" to run parallel version using Pthread without GUI.
8. Use "./pthreadg [n size] [number of threads]" to run parallel version using Pthread with GUI.
9. Use "./cuda [n size]" to run parallel version using CUDA without GUI.
10. Use "./cudag [n size]" to run parallel version using CUDA with GUI.
11. Use "openmp [n size] [number of omp threads]" to run parallel version using OpenMP without GUI.
12. Use "openmpg [n size] [number of omp threads]" to run parallel version using OpenMP with GUI.
13. Use "mpirun -np [number of processes] ./mpi [n size]" to run parallel version using MPI + OpenMP without GUI.
14. Use "mpirun -np [number of processes] ./mpig [n size]" to run parallel version using MPI + OpenMP with GUI.

The following are the steps to do experiments in hpc:

1. Go to project4 folder, compile all files:

   scripts/cmake_release.sh for non-GUI version

   scripts/cmake_release_gui.sh for GUI version

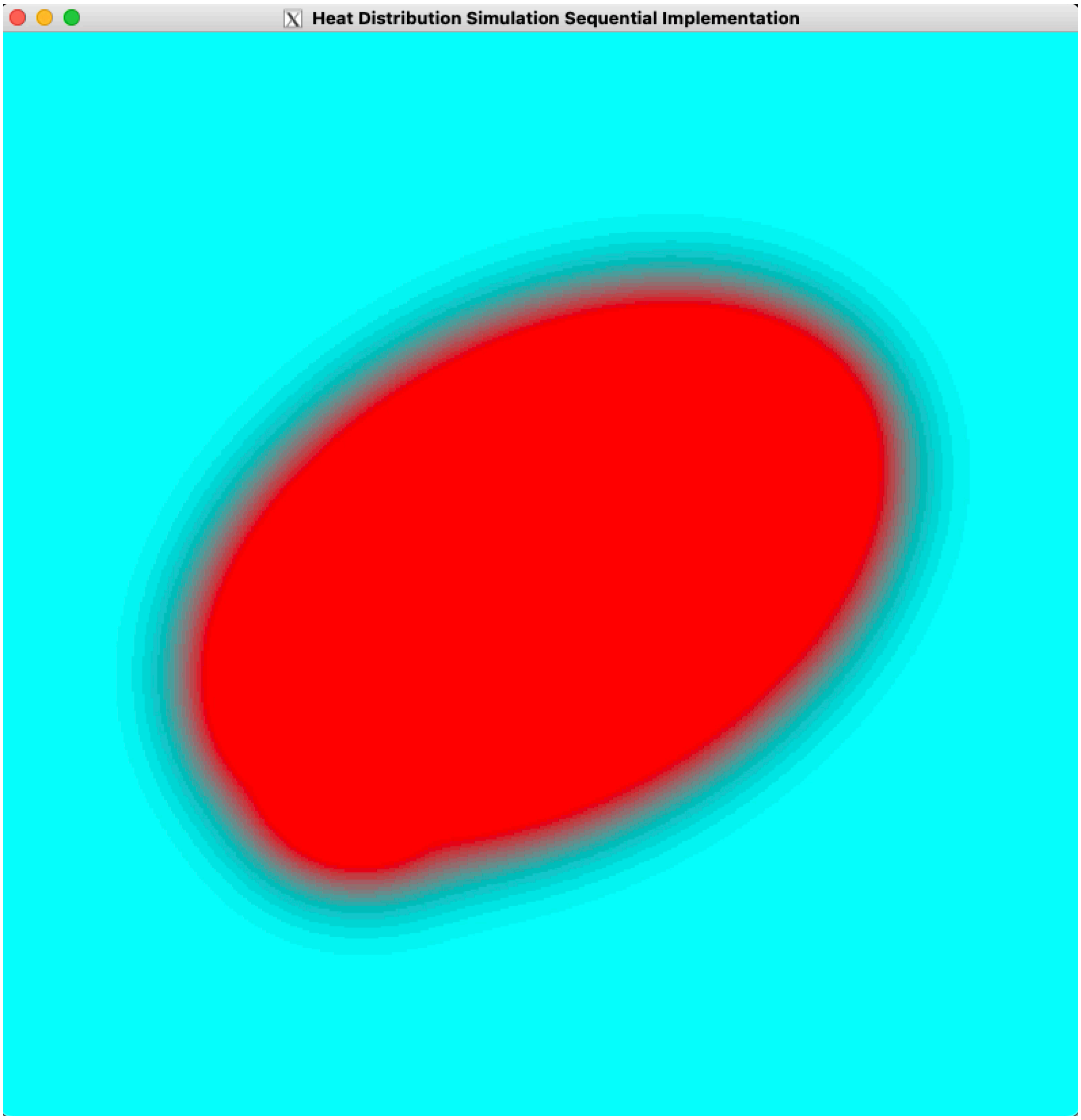
2. Config the task template batch file by specifying job name, nodes number, process number, number of CPU cores per process, total memory limit, time limit, and partition name shown as the following example.

   ```
   #!/bin/bash

   #SBATCH --job-name=119020059_seq_{size} # Job name

   #SBATCH --nodes=1 # Run all processes on a single node

   #SBATCH --ntasks=1 # number of processes = 4

   #SBATCH --cpus-per-task=1 # Number of CPU cores per process

   #SBATCH --mem=1000mb # Total memory limit

   #SBATCH --time=00:60:00 # Time limit hrs:min:sec

   #SBATCH --partition=Debug # Partition name: Project or Debug (Debug is default)
   ```

3. SSH to hpc and run “sbatch [batch file path]” to submit the job.
4. Collect running output file “slurm-[num].out”.

Sample Outputs

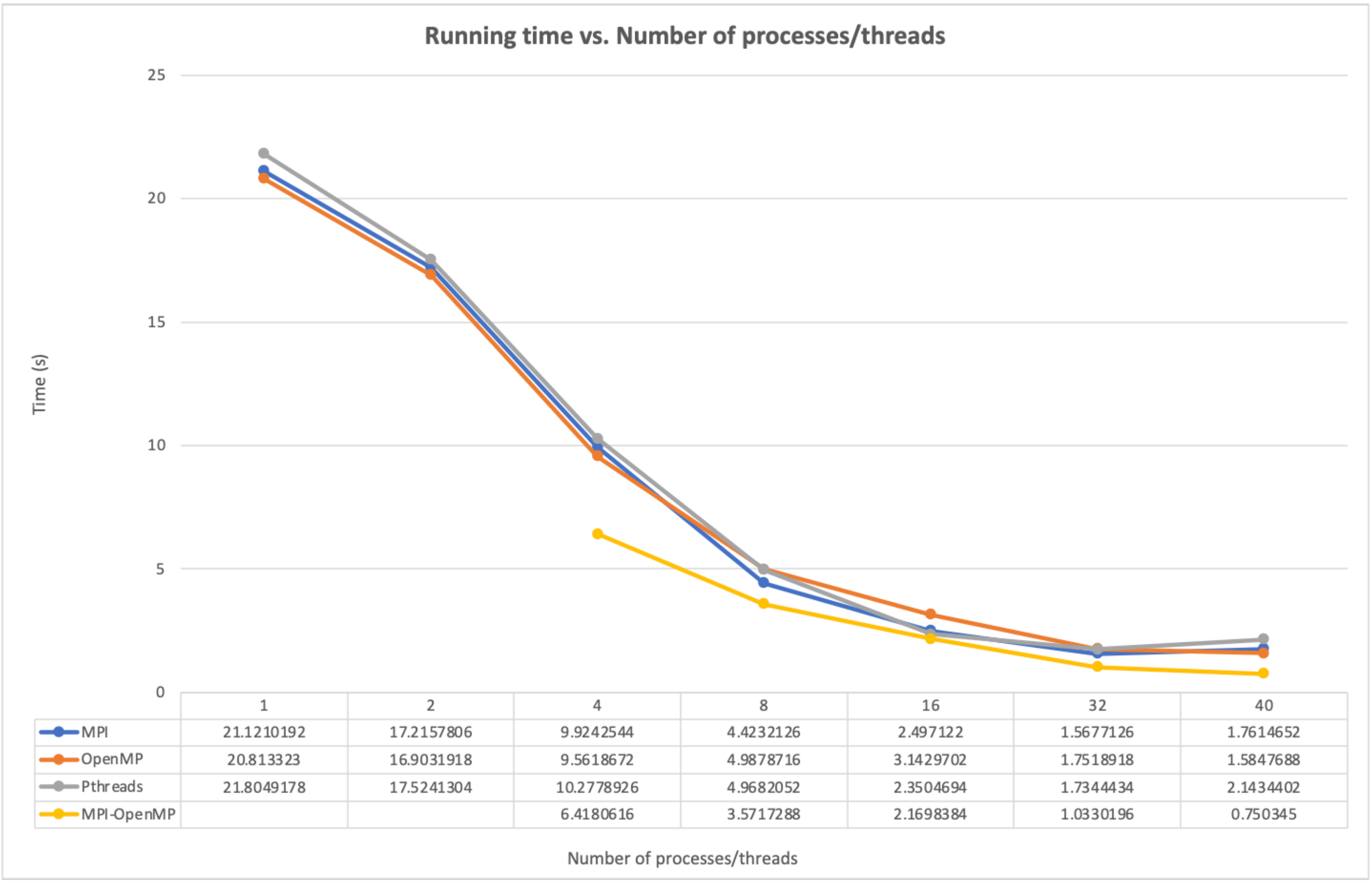
Window size = 800 × 800, Number of iterations = 2000. The following is the output:



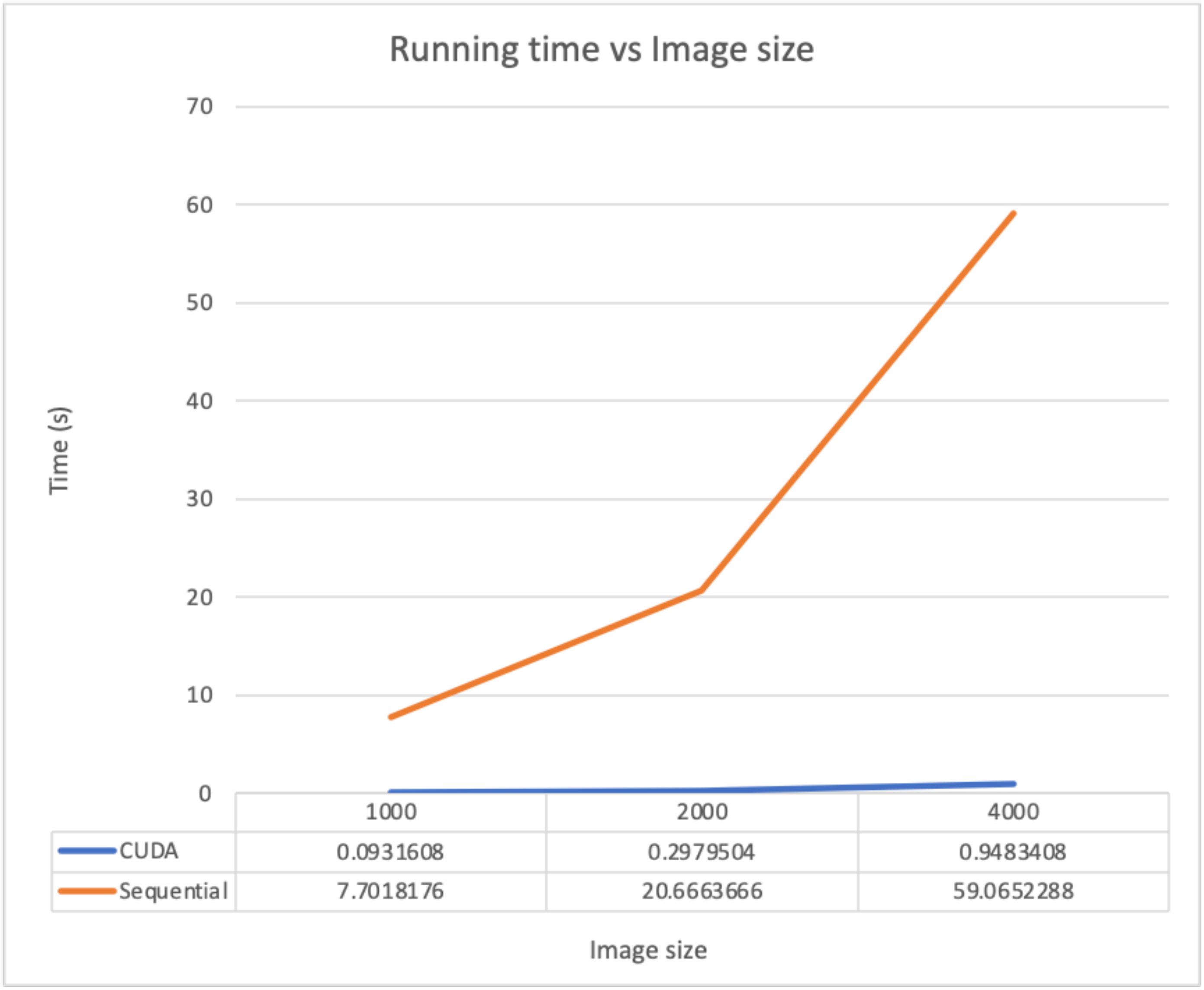
Analysis and Results

Running time vs. Number of processes/threads

Fix size = 2000, number of iterations = 2000. We compare parallel version of MPI, Pthreads, OpenMP with number of processes/threads = 1, 2, 4, 8, 16, 32, 40. For MPI + OpenMP version, I only adjust number of processes with fixed OpenMP threads number = 4, thus MPI + OpenMP has number of processes/threads = 4, 8, 16, 32, 40. For CUDA version, since the number of threads is the smallest multiple of 512 that is larger than N points, I analyze the relationship between running time and number of threads for CUDA version alone. The comparison is shown as follows:



	1	2	4	8	16	32	40
MPI	21.1210192	17.2157806	9.9242544	4.4232126	2.497122	1.5677126	1.7614652
OpenMP	20.813323	16.9031918	9.5618672	4.9878716	3.1429702	1.7518918	1.5847688
Pthreads	21.8049178	17.5241304	10.2778926	4.9682052	2.3504694	1.7344434	2.1434402
MPI-OpenMP			6.4180616	3.5717288	2.1698384	1.0330196	0.750345



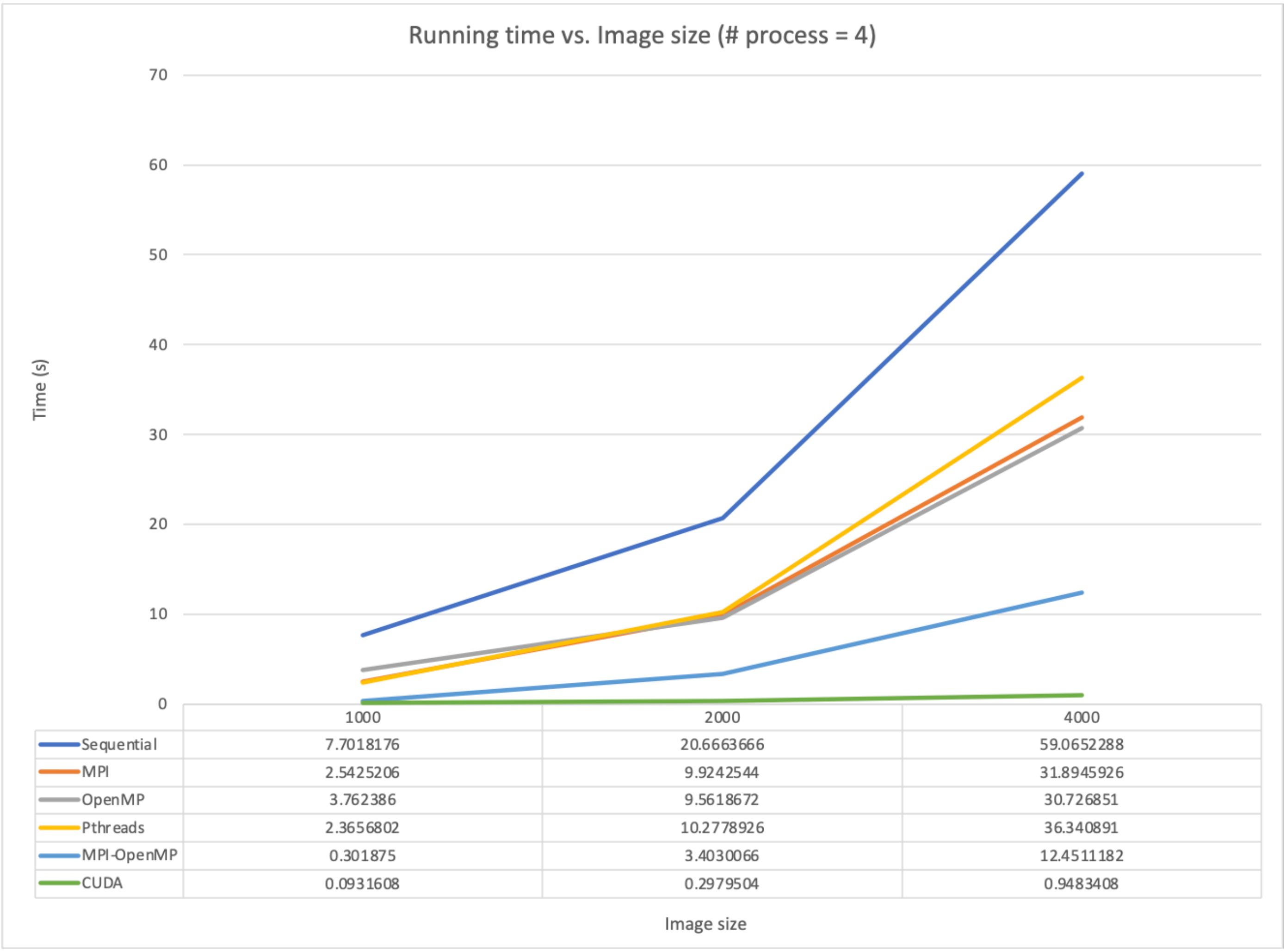
	1000	2000	4000
CUDA	0.0931608	0.2979504	0.9483408
Sequential	7.7018176	20.6663666	59.0652288

The above images show all parallel versions implementation of the relationship between running time and number of processes/threads. From the figure above, for MPI, OpenMP, Pthreads, MPI + OpenMP versions, I found that when the number of processes/threads increases, the running time decreases with slight fluctuations.

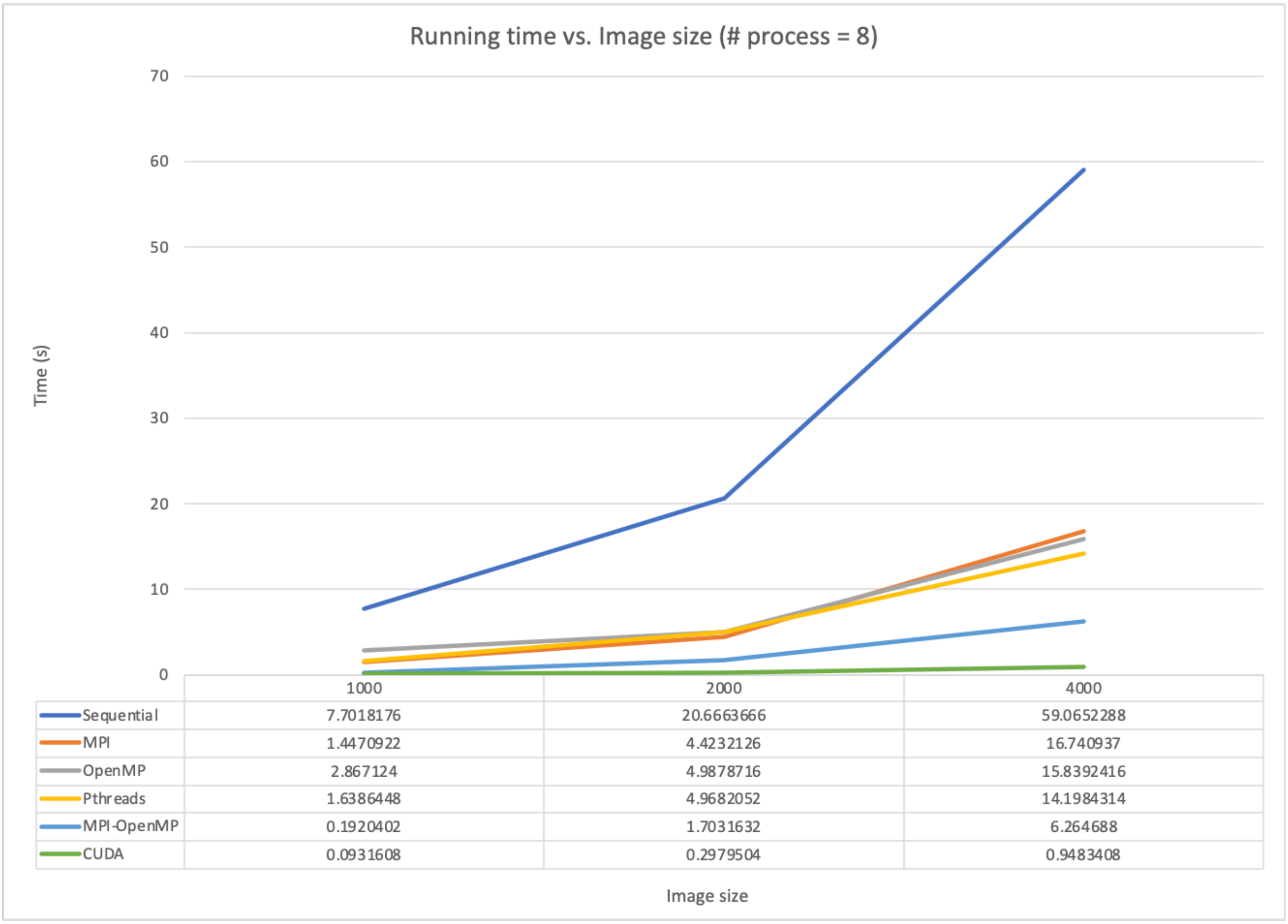
For MPI, OpenMP, Pthreads, MPI + OpenMP versions, when the number of processes/threads increases, the computation workload in each process/thread is less than before, thus the runnning time decreases, which illustrates that parallel versions of the program indeed reduce running time. However, when the number of processes/threads is very large, the costs for process communication and synchronization increases as the number of processes/threads increases. Since the costs for communication and synchronization dominate computation time when number of process/threads is very large, the running time may keep stable or even slightly increase. For CUDA version, number of threads is just image size. Since each thread is only responsible for updating one point in each iteration, the main cost lies in memory copy, which heavily dominate local computation time in each thread. Thus, as number of threads increases, the running time of CUDA version is not very small compared to its thread number. But due to its huge number of threads, the running time of CUDA is much smaller compared with other versions.

Sequential vs. MPI vs. Pthread vs. CUDA vs. OpenMP

Fix number of iterations = 2000, fix number of processes/threads = 4 or 8 for MPI, Pthreads, OpenMP, MPI + OpenMP versions. We compare sequential, MPI, Pthreads, OpenMP, MPI + OpenMP, CUDA versions with size = 1000, 2000, 4000. The comparison is shown as follows:



	1000	2000	4000
Sequential	7.7018176	20.6663666	59.0652288
MPI	2.5425206	9.9242544	31.8945926
OpenMP	3.762386	9.5618672	30.726851
Pthreads	2.3656802	10.2778926	36.340891
MPI-OpenMP	0.301875	3.4030066	12.4511182
CUDA	0.0931608	0.2979504	0.9483408



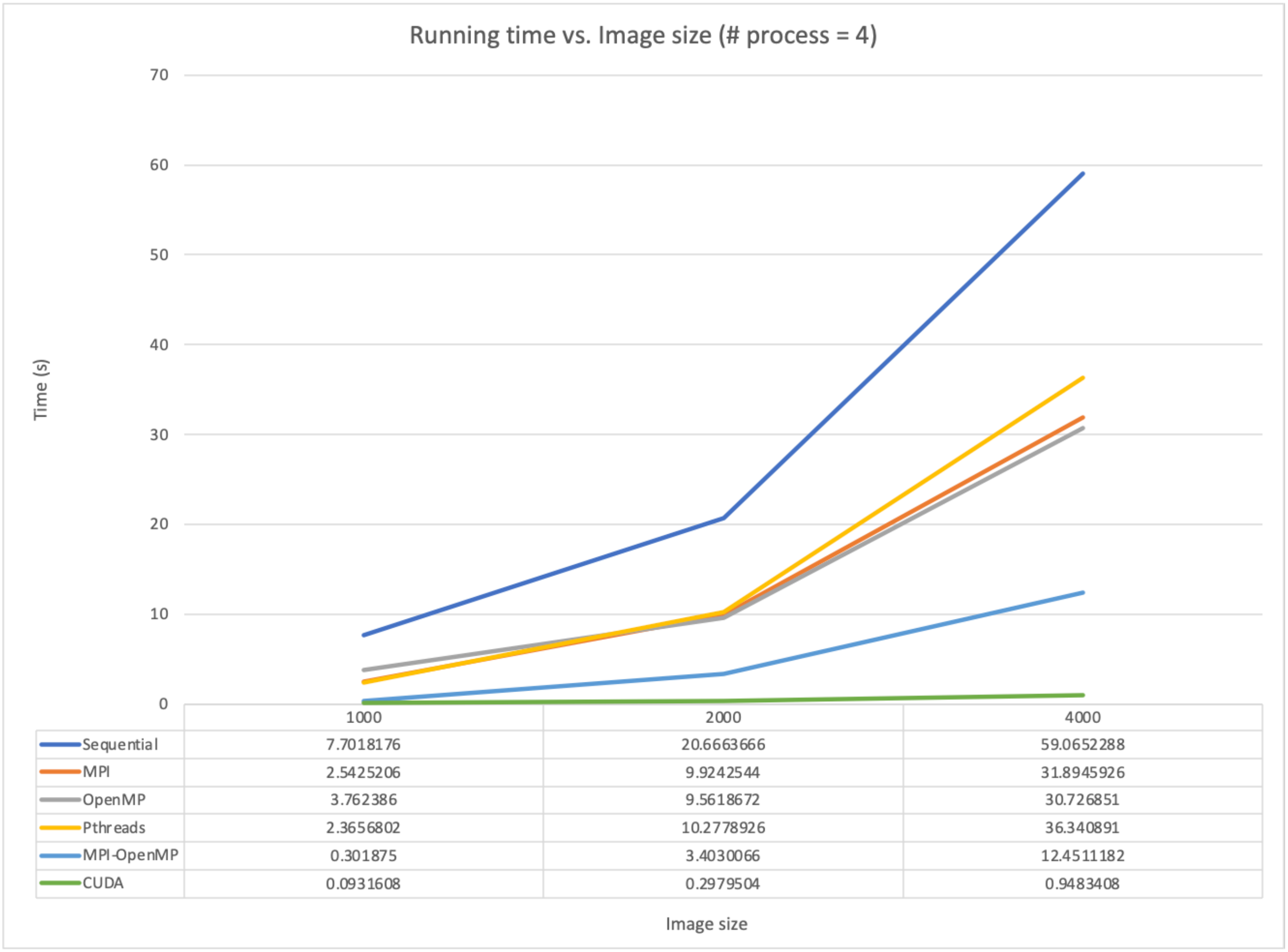
	1000	2000	4000
Sequential	7.7018176	20.6663666	59.0652288
MPI	1.4470922	4.4232126	16.740937
OpenMP	2.867124	4.9878716	15.8392416
Pthreads	1.6386448	4.9682052	14.1984314
MPI-OpenMP	0.1920402	1.7031632	6.264688
CUDA	0.0931608	0.2979504	0.9483408

The above images show the sequential, MPI, Pthreads, OpenMP, MPI + OpenMP, CUDA versions of the relationship between running time and image size with fixed number of processes/threads. From the figure above, I found that keeping number of processes/threads constant, when the image size is small, the running time is less. Since when image size is large, total number of points needed to be updated is larger, thus longer computation time is needed.

Moreover, I found that parallel versions have much less running time compared with sequential version. This illustrates that parallel computing could indeed reduce much computation time. The running time of MPI, Pthread, OpenMP, MPI + OpenMP versions is roughly the same when number of processes equals to number of threads. The running time is smaller for CUDA versions when the N body size is large enough. Since when the N body size is large enough, the cuda memory copy time is small, with much more number of threads than other parallel versions, the running time of CUDA should be much smaller. The slight fluntuations of running time maybe due to the costs of process communication and thread synchronization of different parallel versions of implementation.

Running time vs. Image size

Fix number of iterations = 2000, fix number of processes/threads = 4 for MPI, Pthreads, OpenMP, MPI + OpenMP versions. We compare sequential, MPI, Pthreads, OpenMP, MPI + OpenMP, CUDA versions with size = 1000, 2000, 4000. The comparison is shown as follows:



	1000	2000	4000
Sequential	7.7018176	20.6663666	59.0652288
MPI	2.5425206	9.9242544	31.8945926
OpenMP	3.762386	9.5618672	30.726851
Pthreads	2.3656802	10.2778926	36.340891
MPI-OpenMP	0.301875	3.4030066	12.4511182
CUDA	0.0931608	0.2979504	0.9483408

The above images show the sequential, MPI, Pthreads, OpenMP, MPI + OpenMP, CUDA versions of the relationship between running time and image size with fixed number of processes/threads. From the figure above, I found that the running time is roughly quadratic with image size from the graph, since when the size doubles, number of data points is multiplied by 4. Thus, the computation time should be roughly 4 times.

Conclusion

In this project, the sequential version, parallel version using MPI, parallel version using Pthread, parallel version using OpenMP, parallel version using CUDA, and parallel version using MPI + OpenMP, of heat simulation are implemented and their proformances are also analyzed.

We can conclude from this project that when the number of processes/threads increases, the running time decreases with slight fluctuations. Moreover, keeping number of processes/threads constant, when the output image size is small, the running time is less. The running time is roughly a quadratic function of the output image size.

Through this project, I learned much knowledge regarding the parallel programming and also how to use the MPI, Pthread, OpenMP, CUDA, and MPI + OpenMP methods to implement parallel programs. I also enjoy a lot when conducting experiments for different number of bodies, different versions of implementation, and exploring the relationship between the running time and the number of processes/threads.