

CSC4005 FA22 HW04

Haoran Sun (haoransun@link.cuhk.edu.cn)



1 Introduction

In this project, a heat distribution model using Jacob iteration (same as finite difference) is implemented. Despite a sequential version, the program is also accelerated by common parallelization libraries: MPICH, OpenMP, Pthread, and CUDA. The performance of each method is evaluated.

2 Method

2.1 System setup

The system contains a $n \times n$ square 2D mesh with temperature assigned where x ranging in $[-5, 5]$ and y ranging in $[-5, 5]$. Initially, fire regions Ω is defined as $\{(x, y) | x^2 + y^2 \leq 1\}$, and temperature will be assigned to 100°C . Other region will be assigned to 20°C . Note that all fire regions will keep 100°C and all boundary (edge) region $\{(x, y) | x = -5, 5; y = -5, 5\}$ will keep 100°C .

2.2 Program design and implementation

The programs are written in the C++ programming language. MPICH, Pthread, OpenMP, and CUDA libraries were used for parallelization. Besides, OpenGL is used for visualization purposes. Also, to improve the performance, the MPI version is further accelerated using OpenMP.

Despite MPI version written separately in `src/main.mpi.cpp`, the main program of other version are all wrapped in `src/main.cpp`. Particularly, CUDA functions are compiled in a separated library `build/lib/libcudalib.a`.

One can refer to A.1 to understand the program design.

2.3 Usage

Remark. For convenience, one can directly build the program by `scripts/build.sh` to compile all targets.

To simplify the compiling process, the CMake build system is used to compile programs and link libraries. One can execute the following lines to build executables.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release -DGUI=ON
cmake --build build
```

To disable the GUI feature, one can set `-DGUI=OFF` in the first line. The compiled programs and libraries are shown in the `build/bin` and `build/lib`. One can directly execute `build/bin/main*.gui` for a visualized demonstration.

```
./build/bin/main.seq.gui
./build/bin/main.omp.gui
./build/bin/main.pth.gui
./build/bin/main.mpi.gui
./build/bin/main.cu.gui
```

One can customize the running parameters such as the number of particles n and simulation steps according to the following lines.

```
./build/bin/main.seq          --dim 100 --nsteps 10000 --record 1
./build/bin/main.omp          -nt 10 --dim 100 --nsteps 10000 --record 1
./build/bin/main.omp          -nt 10 --dim 100 --nsteps 10000 --record 1
./build/bin/main.cu           --dim 100 --nsteps 10000 --record 1
mpirun -np 10 ./build/bin/main.mpi --dim 100 --nsteps 10000 --record 1
```

Remark. To execute **MPI + OpenMP** hybrid program, one can just append `-nt [n]` parameters when executing the MPI program. For example, the following line initializes a program with 10 MPI process, and each process has 2 OpenMP threads, which have $10 \times 2 = 20$ threads in total.

```
mpirun -np 10 ./build/bin/main.mpi -nt 2
```

2.4 Performance evaluation

The program was executed under different configurations to evaluate performance. With 40 different CPU core numbers (from 1 to 40 with increment 1, $p = 1, 2, \dots, 40$) and 40 different n (from 50 to 2000 with increment 50), 1600 cases in total were sampled for sequential, MPI, OpenMP, and Pthread programs. Test for CUDA program is implemented separately. Recorded runtime is analyzed through the Numpy package in Python. Figures were plotted through the Matplotlib and the Seaborn packages in Python. Analysis codes were written in `analysis/main.ipynb`.

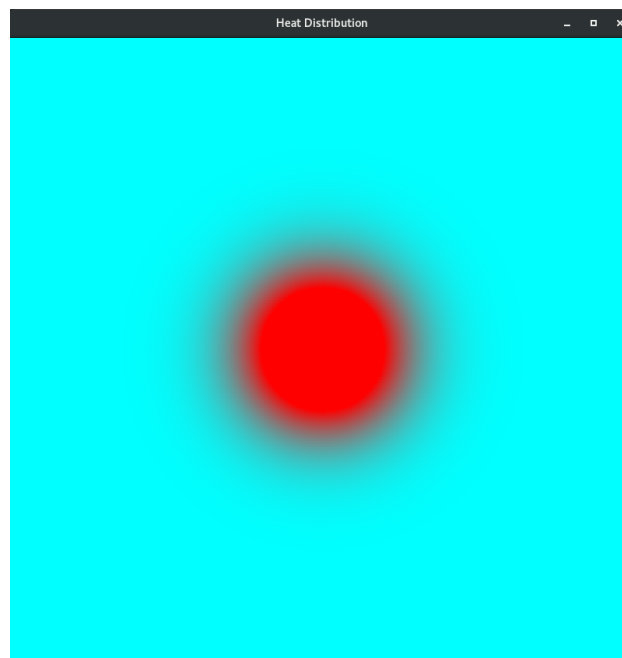


Figure 1: Sample GUI window

3 Result and discussion

3.1 CPU parallelization

From Figure A.2, we can know that when n ranging from 1000 to 2000, MPI, OpenMP, and Pthread have very different performance. For MPI program, the fps almost monotonically decreases with the increase of overall processes. The reason may caused from the communication time. It seems that the time cost of communication increase faster than the computation time when n increase. For OpenMP program, the fps increases linearly. This proves that multi-thread indeed improve the performance. For Pthread program, the fps increases with the number of threads until it reaches it maximum when the number of threads are around 15.

The heatmap which indicate the rate of acceleration plotted in the Figure A.4 provides some direct visualization of the performances of parallel variants. From this figure we can clearly know that MPI scheme has worst performance—it is even slower when overall threads is greater than 1. For Pthread, we can see it reaches its maximum speed-up rate when the number of thread is around 15. For OpenMP program, when n is large (≥ 1400), speed-up rate will be higher when the number of threads is greater.

3.2 GPU parallelization

GPU parallelization is much more massive than CPU parallelization. This allows one to implemented $n > 10^4$ with high fps, as Figure A.5 shows. Notably, the gpu shared memory is used to accelerate the read operations. (please refer to `__shared__` type and `__syncthreads` function in `cudaLib.cu`). According to NVIDIA, the memory access on shared memory is approximately 100× faster than the GPU memory(`__device__`) access.

For example, a naive vector addition in CUDA could be written as

```

1  __global__ void VecAdd(int *a, int *b, int *c, long int dim){
2      // thread partition
3      int start_idx = dim / (blockDim.x * gridDim.x) * threadIdx.x;
4      int end_idx   = dim / (blockDim.x * gridDim.x) * (threadIdx.x+1);
5      if (threadIdx.x+1==blockDim.x) end_idx = dim;
6      // vector add
7      for (int i = 0; i < dim; i++){
8          c[i] = a[i] + b[i];
9      }
10 }
```

During the calculation, each thread in GPU will require to access the memory independently. When the overall thread number is large, the memory miss could cost a huge amount of time. However, in CUDA, we can split those threads into different blocks: for example, if one call a kernel function kernel by `kernel<<<16,64>>>()`, then he is asking CUDA to generate 16 blocks where each block has 64 threads, overall $16 \times 64 = 1024$ threads. Similarly, `kernel<<<1,1024>>>()` also calls the function with 1024 threads. In principle, `VecAdd<<<16,64>>>(a, b, c, dim)` and `VecAdd<<<1,1024>>>(a, b, c, dim)` has no difference. Now consider, if we can let threads in each block, share a part of memory, then can it reduce the time cost by memory miss? Have a look at the following function

```

1  #define BLOCKSIZE 64
2  __global__ void sharedMemVecAdd(int *a, int *b, int *c, long int dim){
3      // block partition
4      int block_start_idx = dim / gridDim.x * blockIdx.x;
```

```

5   int block_end_idx    = dim / gridDim.x * (blockIdx.x + 1);
6   if (blockIdx.x+1==gridDim.x) block_end_idx = dim;
7   int total_task       = block_end_idx - block_start_idx;
8   // shared memory partition
9   int num_iter = (total_task + BLOCK_SIZE - 1) / BLOCK_SIZE;
10  // block-wise shared memory
11  __shared__ int a_t[BLOCK_SIZE*2];
12  __shared__ int b_t[BLOCK_SIZE];
13  __shared__ int c_t[BLOCK_SIZE];
14  __syncthreads();
15
16  // main program
17  for (int i = 0; i < num_iter; i++){
18    if (threadIdx.x+i*BLOCK_SIZE < block_end_idx){
19      // thread
20      // copy data
21      a_t[threadIdx.x] = a[block_start_idx + threadIdx.x + BLOCK_SIZE*i];
22      b_t[threadIdx.x] = b[block_start_idx + threadIdx.x + BLOCK_SIZE*i];
23      __syncthreads();
24
25      // vector add
26      c_t[threadIdx.x] = a_t[threadIdx.x] + b_t[threadIdx.x];
27
28
29      // copy data back
30      c[block_start_idx + threadIdx.x + BLOCK_SIZE*i] = c_t[threadIdx.x];
31      __syncthreads();
32    }}
33 }

```

One should convince himself that `sharedMemVecAdd<<<16,BLOCKSIZE>>>>(a, b, c, dim)` do the exact same work as `VecAdd`. So what is the difference here? In each block, CUDA will create a shared memory, that is a fast memory accessible by ALL threads within this block. During the computation, the block will first read a memory block, then perform computation; after all threads finish the computation, the threads will write data back to the global memory.

4 Conclusion

In conclusion, four parallel computing schemes for n -body simulation are implemented and their performances are evaluated. For large, ignoring the precision, one may use GPU to accelerate the calculation.

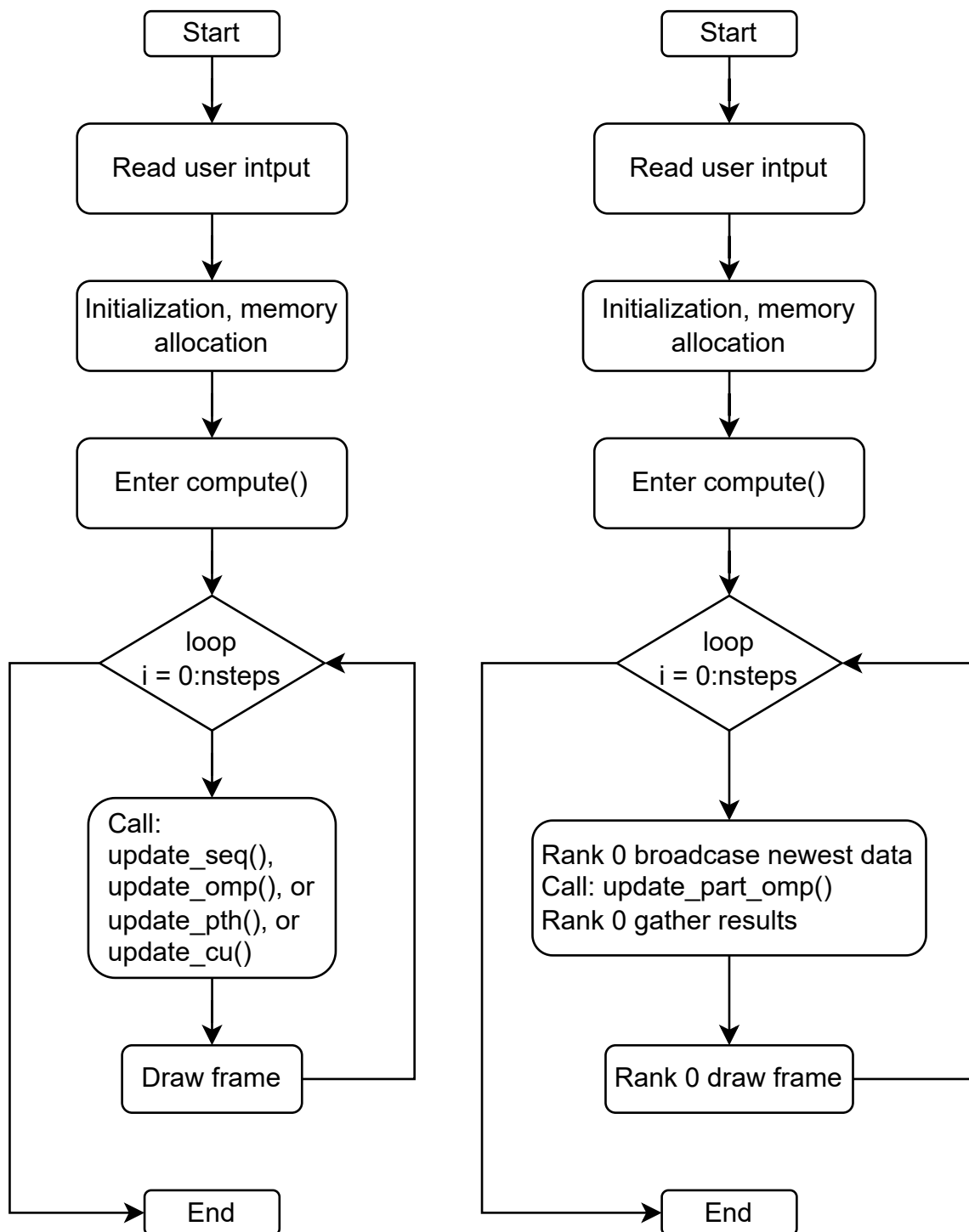
A Supplementary figures

Figure A.1: Program flowchart

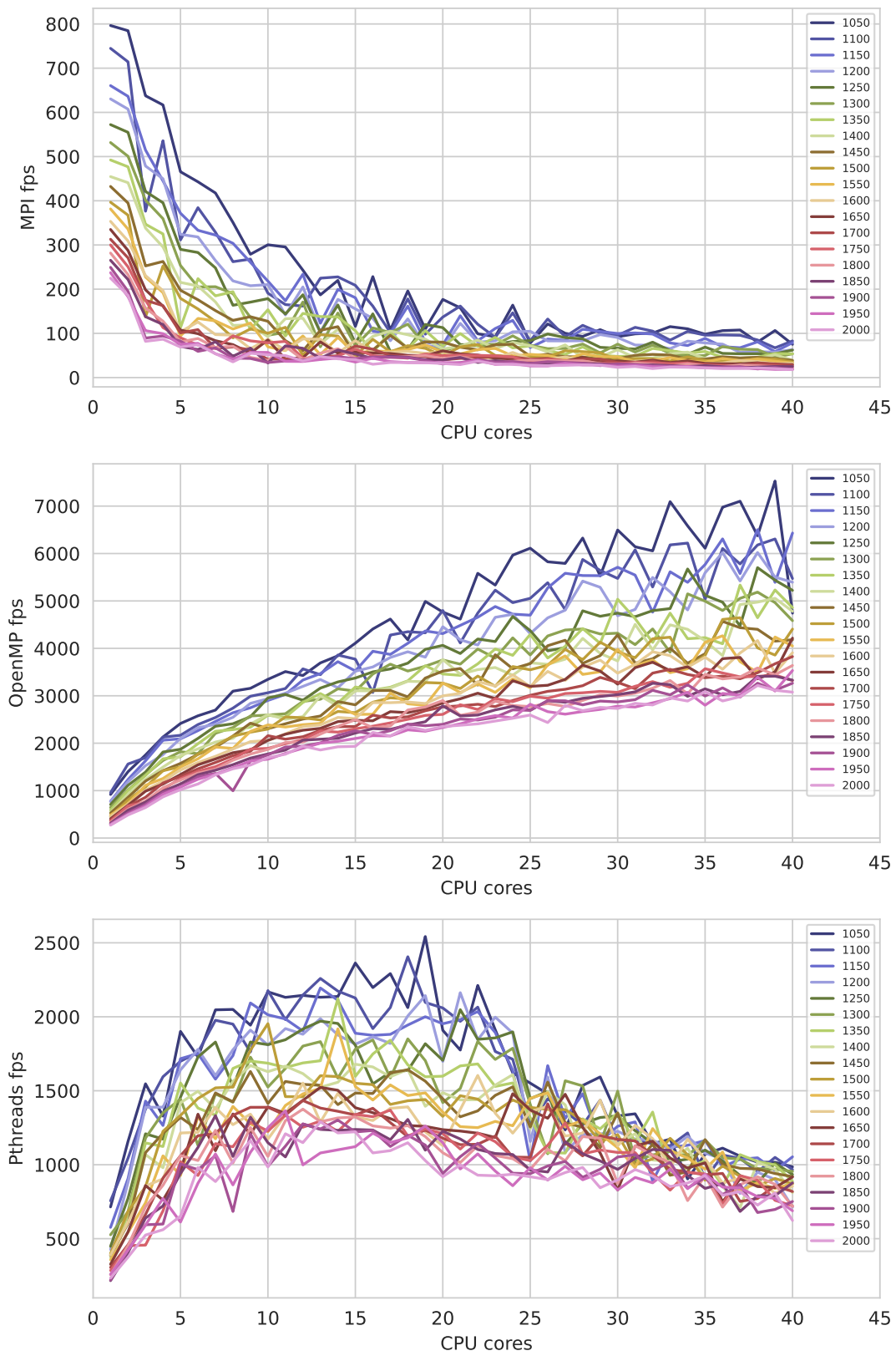


Figure A.2: fps vs the number of threads/processes plot.

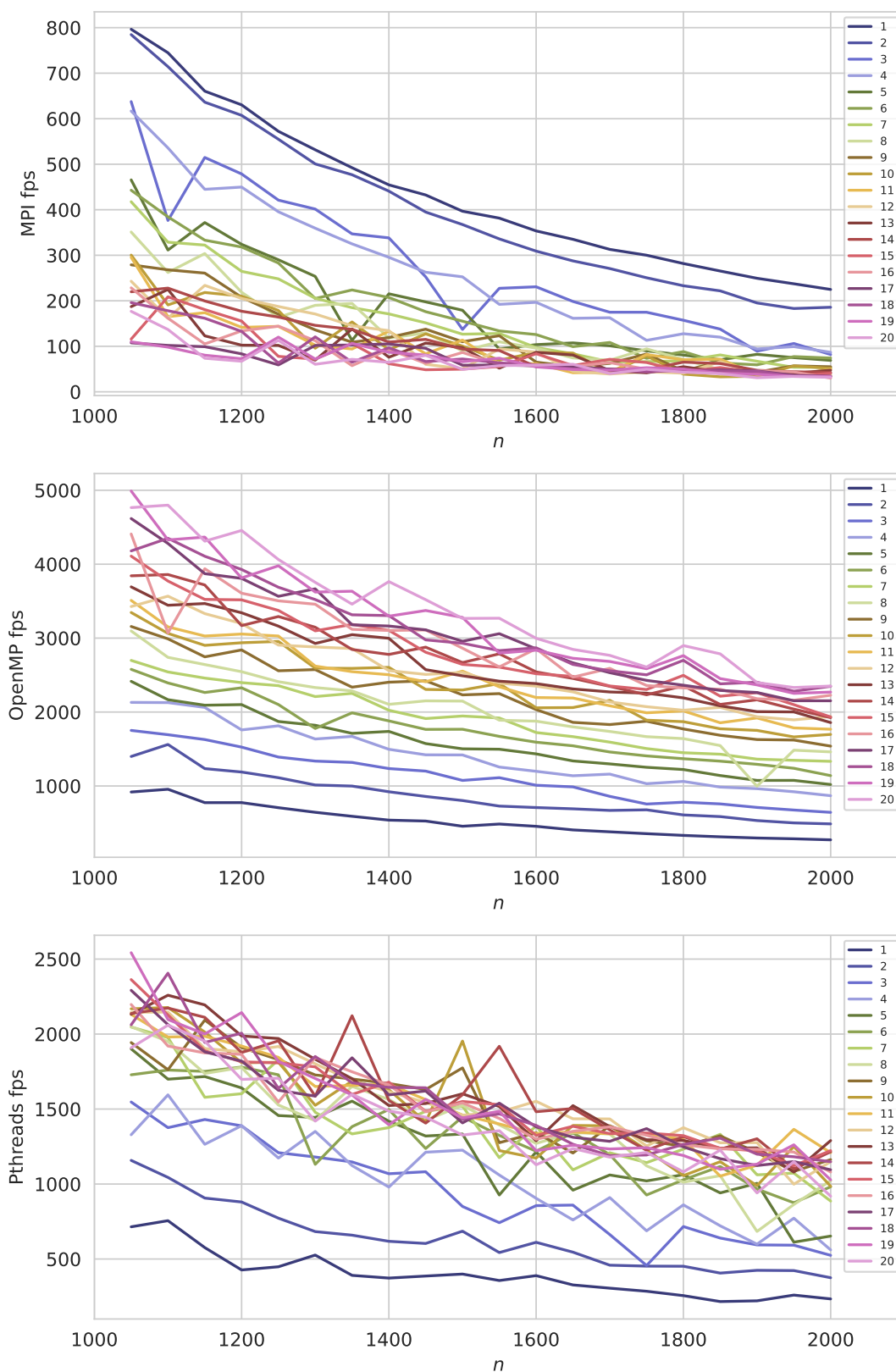


Figure A.3: fps vs the number of threads/processes plot.

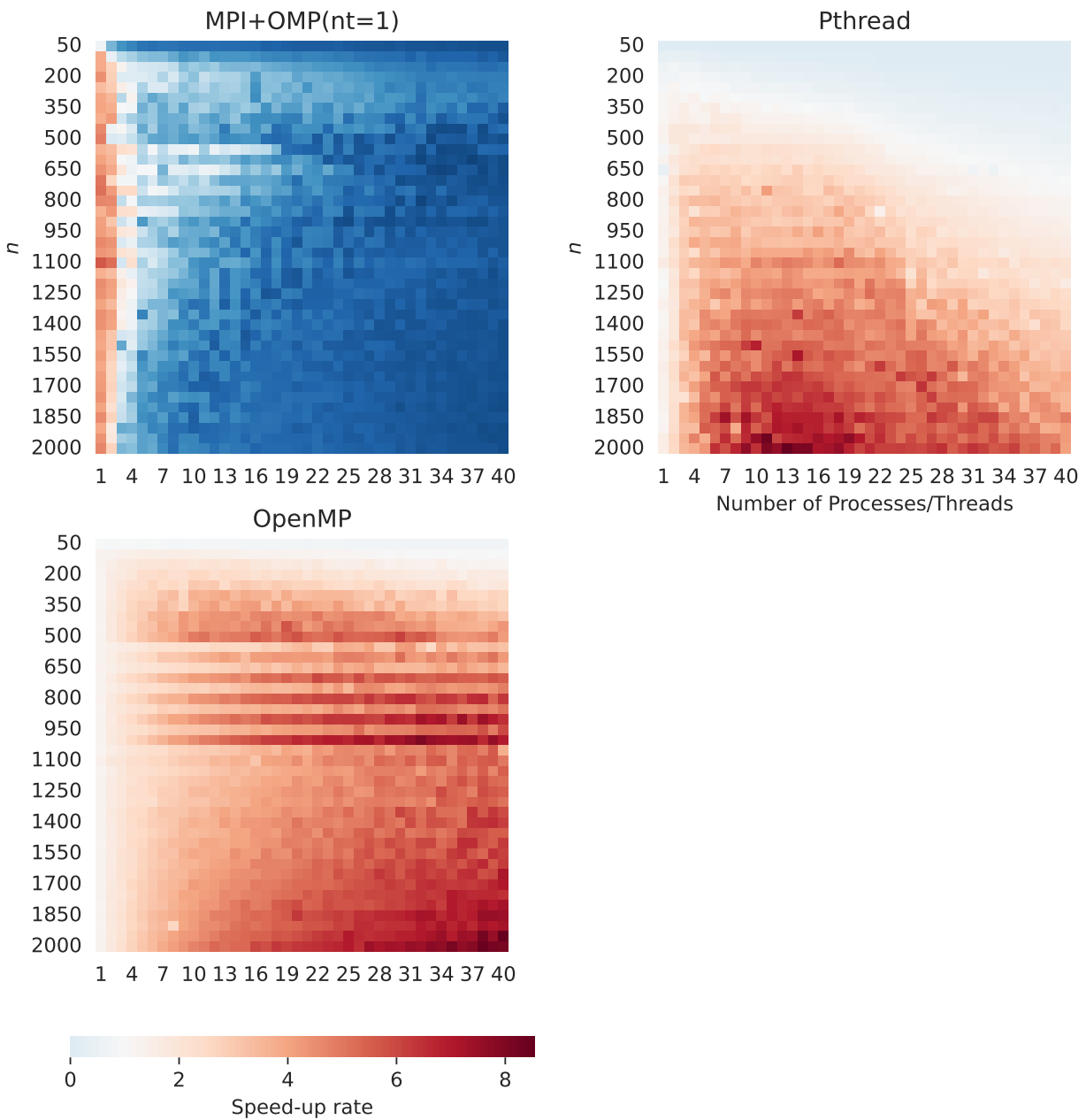
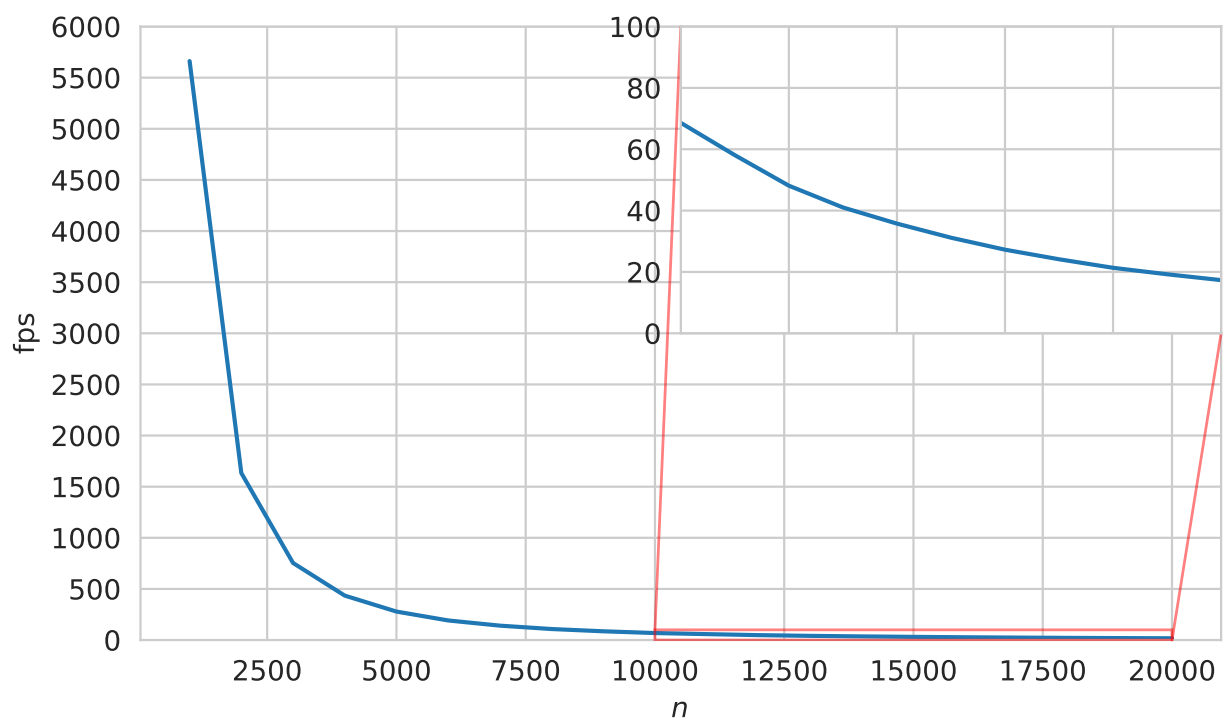


Figure A.4: Speed-up rate for multi-process/thread schemes.

Figure A.5: CUDA fps vs n plot.

B Source code

CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.20)
2 project(hw03 LANGUAGES CXX CUDA)
3
4 # set output path
5 set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
6 set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
7 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
8
9 # set include libraires
10 include_directories(src)
11
12 set(CMAKE_CXX_STANDARD 11)
13
14 # add src folder
15 add_subdirectory(src)

```

src/CMakeLists.txt

```

1 find_package(MPI REQUIRED)
2 find_package(CUDA REQUIRED)
3 find_package(Threads REQUIRED)
4 find_package(OpenMP REQUIRED)
5
6 # options
7 # gui option
8 option(GUI "OPENGL Rendering" OFF)
9
10 # omp flags
11 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
12
13 # libraries
14 add_library(cudalib cudalib.cu)
15 set(THREADS_PREFER_PTHREAD_FLAG ON)
16 include_directories(
17     ${MPI_INCLUDE_PATH}
18     ${CUDA_INCLUDE_DIRS}
19 )
20 link_libraries(
21     ${MPI_LIBRARIES}
22     ${CUDA_LIBRARIES}
23     cudalib
24 )
25
26
27 # targets & libs
28 add_executable(main.seq main.cpp)
29 add_executable(main.omp main.cpp)
30 add_executable(main.pth main.cpp)
31 add_executable(main.cu main.cpp)
32 add_executable(main.mpi main.mpi.cpp)
33 target_compile_definitions(main.omp PUBLIC OMP)
34 target_compile_definitions(main.pth PUBLIC PTH)
35 target_compile_definitions(main.cu PUBLIC CUDA)
36
37 # opengl & glut
38 if(GUI)
39     find_package(OpenGL REQUIRED)
40     find_package(GLUT REQUIRED)
41     include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})
42     link_libraries(${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
43     add_executable(main.seq.gui main.cpp)
44     add_executable(main.omp.gui main.cpp)

```

```

45     add_executable(main.pth.gui main.cpp)
46     add_executable(main.cu.gui main.cpp)
47     add_executable(main.mpi.gui main.mpi.cpp)
48     target_compile_definitions(main.seq.gui PUBLIC GUI)
49     target_compile_definitions(main.omp.gui PUBLIC GUI OMP)
50     target_compile_definitions(main.pth.gui PUBLIC GUI PTH)
51     target_compile_definitions(main.cu.gui PUBLIC GUI CUDA)
52     target_compile_definitions(main.mpi.gui PUBLIC GUI)
53 endif()

```

src/main.cpp

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  #include <memory.h>
5  #include <chrono>
6  #include "utils.h"
7  #include "utils.cuh"
8  #ifdef GUI
9  #include "gui.h"
10 #endif
11 #include "const.h"
12 #include <thread>
13
14 void compute(){
15     // start timing
16     auto t0 = std::chrono::high_resolution_clock::now();
17     auto t1 = std::chrono::high_resolution_clock::now();
18     auto t2 = std::chrono::high_resolution_clock::now();
19     double t;
20     for (int s = 0; s < nsteps; s++){
21         // main compute program
22         #ifdef OMP
23             update_omp(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire);
24         #elif PTH
25             update_pth(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire,
26                 thread_arr, args_arr, nt);
27         #elif CUDA
28             update_cu(temp_arr0);
29         #ifdef GUI
30             copy_cu(temp_arr0);
31         #endif
32         #else
33             update_seq(&temp_arr, &temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire);
34         #endif
35
36         // calculating fps
37         int step = 60;
38         if (s%step==0 && s%(step*2)!=0) t1 = std::chrono::high_resolution_clock::now();
39         else if (s%(step*2)==0 && s!=0) {
40             t2 = std::chrono::high_resolution_clock::now();
41             t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count()
42                 ;
43             printf("fps: %f frame/s\n", step/t);
44         }
45
46         #ifdef GUI
47         #ifdef OMP
48             data2pix_omp(temp_arr0, pix, DIM, RES, T_bdy, T_fire);
49         #else
50             data2pix(temp_arr0, pix, DIM, RES, T_bdy, T_fire);
51         #endif
52         glClear(GL_COLOR_BUFFER_BIT);
53         glDrawPixels(RES, RES, GL_RGB, GL_UNSIGNED_BYTE, pix);
54         glFlush();
55         glutSwapBuffers();

```

```

55     // glFinish();
56     // std::this_thread::sleep_for(std::chrono::milliseconds(100));
57     #endif
58 }
59
60 // record data
61 if (record==1){
62     t2 = std::chrono::high_resolution_clock::now();
63     t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
64     double fps = nsteps / t;
65     runtime_record(type, DIM, size, fps);
66 }
67 }
68
69 int main(int argc, char *argv[]){
70     // parse argument
71     char buff[200];
72     for (int i = 0; i < argc; i++){
73         strcpy(buff, argv[i]);
74         if (strcmp(buff, "--dim")==0){
75             std::string num(argv[i+1]);
76             DIM = std::stoi(num);
77         }
78         if (strcmp(buff, "--nt")==0){
79             std::string num(argv[i+1]);
80             nt = std::stoi(num);
81         }
82         if (strcmp(buff, "--nsteps")==0){
83             std::string num(argv[i+1]);
84             nsteps = std::stof(num);
85         }
86         if (strcmp(buff, "--record")==0){
87             std::string num(argv[i+1]);
88             record = std::stoi(num);
89         }
90         if (strcmp(buff, "--Tx")==0){
91             std::string num(argv[i+1]);
92             Tx = std::stoi(num);
93         }
94         if (strcmp(buff, "--Ty")==0){
95             std::string num(argv[i+1]);
96             Ty = std::stoi(num);
97         }
98     }
99
100     // print info
101     print_info(DIM, nsteps);
102
103     // initialization
104     temp_arr = (float *)malloc(sizeof(float)*DIM*DIM);
105     temp_arr0 = (float *)malloc(sizeof(float)*DIM*DIM);
106     fire_arr = (bool *)malloc(sizeof(bool)*DIM*DIM);
107     x_arr = (float *)malloc(sizeof(float)*DIM);
108     y_arr = (float *)malloc(sizeof(float)*DIM);
109     #ifdef GUI
110     pix = (GLubyte *)malloc(sizeof(GLubyte)*RES*RES*3);
111     #endif
112
113     // assign mesh
114     for (int i = 0; i < DIM; i++){
115         x_arr[i] = (xmax-xmin) * i/DIM + xmin;
116         y_arr[i] = (ymax-ymin) * i/DIM + ymin;
117     }
118     // assign temperature
119     for (int i = 0; i < DIM; i++){
120         for (int j = 0; j < DIM; j++){
121             float x = x_arr[i];

```

```

122     float y = y_arr[j];
123     temp_arr[i*DIM+j] = T_bdy;
124     fire_arr[i*DIM+j] = false;
125     if (is_fire(x, y)){
126         temp_arr[i*DIM+j] = T_fire;
127         fire_arr[i*DIM+j] = true;
128     }
129 }
130 memcpy(temp_arr0, temp_arr, sizeof(float)*DIM*DIM);
131
132 #ifdef OMP
133 strcpy(type, "omp");
134 omp_set_num_threads(nt);
135 size = nt;
136 #elif PTH
137 strcpy(type, "pth");
138 thread_arr = (pthread_t *)malloc(sizeof(pthread_t)*nt);
139 args_arr = (PthArgs *)malloc(sizeof(PthArgs)*nt);
140 size = nt;
141 #elif CUDA
142 strcpy(type, "cuda");
143 initialize_cu(temp_arr, temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire,
144             Tx, Ty);
145 #else
146 strcpy(type, "seq");
147 size = 1;
148 #endif
149
150 // main program
151 #ifdef GUI
152 glutInit(&argc, argv);
153 glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
154 glutInitWindowPosition(0, 0);
155 glutInitWindowSize(RES, RES);
156 glutCreateWindow("Heat Distribution");
157 glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
158 // glutDisplayFunc(&compute);
159 gluOrtho2D(xmin, xmax, ymin, ymax);
160 glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
161 // glutMainLoop();
162 #endif
163
164 compute();
165
166 // finalization
167 free(temp_arr);
168 free(temp_arr0);
169 free(fire_arr);
170 free(x_arr);
171 free(y_arr);
172
173 #ifdef PTH
174 free(args_arr);
175 free(thread_arr);
176 #elif CUDA
177 finalize_cu();
178 #else
179 #endif
180
181 return 0;
182 }

```

src/main.mpi.cpp

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <iostream>

```

```

4  #include <memory.h>
5  #include <chrono>
6  #include "utils.h"
7  #ifdef GUI
8  #include "gui.h"
9  #endif
10 #include "const.h"
11
12 void compute(){
13     // running type buffer
14     strcpy(type, "mpi");
15     // start timing
16     auto t0 = std::chrono::high_resolution_clock::now();
17     auto t1 = std::chrono::high_resolution_clock::now();
18     auto t2 = std::chrono::high_resolution_clock::now();
19     double t;
20     // mpi computing parameters
21     int start_idx, end_idx;
22     int jobsize = DIM / size;
23     partition(DIM, size, rank, &start_idx, &end_idx);
24     for (int s = 0; s < nsteps; s++){
25         // transfer data
26         MPI_Bcast(temp_arr0, DIM*DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
27         MPI_Barrier(MPI_COMM_WORLD);
28
29         // main compute program
30         update_omp_part(&temp_arr0, fire_arr, x_arr, y_arr, DIM, T_fire,
31             start_idx, end_idx);
32         MPI_Barrier(MPI_COMM_WORLD);
33
34         // transfer data
35         if (rank==0) MPI_Gather(MPI_IN_PLACE, jobsize*DIM, MPI_FLOAT, temp_arr0+
36             start_idx*DIM,
37             jobsize*DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
38         else MPI_Gather(temp_arr0+start_idx*DIM, jobsize*DIM, MPI_FLOAT, temp_arr0,
39             jobsize*DIM,
40             MPI_FLOAT, 0, MPI_COMM_WORLD);
41         // solve tail case
42         if (DIM%jobsize!=0) {
43             if (rank==0){
44                 MPI_Recv(temp_arr0+(DIM/size*size)*DIM, (DIM%jobsize)*DIM, MPI_FLOAT,
45                     size-1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
46             }
47             else if (rank+1==size){
48                 MPI_Send(temp_arr0+(DIM/size*size)*DIM, (DIM%jobsize)*DIM, MPI_FLOAT, 0,
49                     1, MPI_COMM_WORLD);
50             }
51         }
52         MPI_Barrier(MPI_COMM_WORLD);
53
54         // calculate fps
55         int step = 60;
56         if (s%step==0 && s%(step*2)!=0) t1 = std::chrono::high_resolution_clock::now();
57         else if (s%(step*2)==0 && s!=0) {
58             t2 = std::chrono::high_resolution_clock::now();
59             t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count()
60                 ;
61             if (rank==0) printf("fps: %f frame/s\n", step/t);
62         }
63     }
64
65     #ifdef GUI
66     if (rank==0){
67         data2pix_omp(temp_arr0, pix, DIM, RES, T_bdy, T_fire);
68         glClear(GL_COLOR_BUFFER_BIT);
69         glDrawPixels(RES, RES, GL_RGB, GL_UNSIGNED_BYTE, pix);
70         glFlush();
71         glutSwapBuffers();
72     }
73     #endif
74 }

```



```

66     }
67     #endif
68 }
69
70 // record data
71 if (rank==0 && record==1){
72     t2 = std::chrono::high_resolution_clock::now();
73     t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
74     double fps = nsteps / t;
75     runtime_record(type, DIM, size, fps);
76 }
77 }
78
79 int main(int argc, char *argv[]){
80     // mpi initialize
81     MPI_Init(NULL, NULL);
82     // fetch size and rank
83     MPI_Comm_size(MPI_COMM_WORLD, &size);
84     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
85
86     // parse argument
87     char buff[200];
88     for (int i = 0; i < argc; i++){
89         strcpy(buff, argv[i]);
90         if (strcmp(buff, "--dim")==0){
91             std::string num(argv[i+1]);
92             DIM = std::stoi(num);
93         }
94         if (strcmp(buff, "-nt")==0){
95             std::string num(argv[i+1]);
96             nt = std::stoi(num);
97         }
98         if (strcmp(buff, "--nsteps")==0){
99             std::string num(argv[i+1]);
100             nsteps = std::stof(num);
101         }
102         if (strcmp(buff, "--record")==0){
103             std::string num(argv[i+1]);
104             record = std::stoi(num);
105         }
106         if (strcmp(buff, "--Tx")==0){
107             std::string num(argv[i+1]);
108             Tx = std::stoi(num);
109         }
110         if (strcmp(buff, "--Ty")==0){
111             std::string num(argv[i+1]);
112             Ty = std::stoi(num);
113         }
114     }
115
116     // omp initialize
117     omp_set_num_threads(nt);
118
119
120     // print info
121     if (rank==0) print_info(DIM, nsteps);
122
123     // initialization
124     temp_arr = (float *)malloc(sizeof(float)*DIM*DIM);
125     temp_arr0 = (float *)malloc(sizeof(float)*DIM*DIM);
126     fire_arr = (bool *)malloc(sizeof(bool)*DIM*DIM);
127     x_arr = (float *)malloc(sizeof(float)*DIM);
128     y_arr = (float *)malloc(sizeof(float)*DIM);
129     #ifdef GUI
130     if (rank==0) pix = (GLubyte *)malloc(sizeof(GLubyte)*RES*RES*3);
131     #endif
132

```



```

133 // assign mesh
134 for (int i = 0; i < DIM; i++){
135     x_arr[i] = (xmax-xmin) * i/DIM + xmin;
136     y_arr[i] = (ymax-ymin) * i/DIM + ymin;
137 }
138 // assign temperature
139 for (int i = 0; i < DIM; i++){
140     for (int j = 0; j < DIM; j++){
141         float x = x_arr[i];
142         float y = y_arr[j];
143         temp_arr[i*DIM+j] = T_bdy;
144         fire_arr[i*DIM+j] = false;
145         if (is_fire(x, y)){
146             temp_arr[i*DIM+j] = T_fire;
147             fire_arr[i*DIM+j] = true;
148         }
149     }
150 }
151 memcpy(temp_arr0, temp_arr, sizeof(float)*DIM*DIM);
152 // main program
153 #ifdef GUI
154 if (rank==0){
155     glutInit(&argc, argv);
156     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
157     glutInitWindowPosition(0, 0);
158     glutInitWindowSize(RES, RES);
159     glutCreateWindow("Heat Distribution");
160     glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
161     gluOrtho2D(xmin, xmax, ymin, ymax);
162     glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
163 }
164 #endif
165
166 compute();
167
168 // finalization
169 free(temp_arr);
170 free(temp_arr0);
171 free(fire_arr);
172 free(x_arr);
173 free(y_arr);
174
175 #ifdef GUI
176 if (rank==0) free(pix);
177 #endif
178
179 return 0;
180 }

```

src/cudalib.cu

```

1 #include "utils.cuh"
2 #include "const.cuh"
3 #define BLOCK_SIZE 256
4
5 #define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
6 inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
7 {
8     if (code != cudaSuccess)
9     {
10         fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
11         if (abort) exit(code);
12     }
13 }
14
15 __device__ void partition_d(int nsteps, int size, int idx, int *start_ptr, int *end_ptr)
16 {

```

```

16     *start_ptr = nsteps / size * idx;
17     *end_ptr = nsteps / size * (idx+1);
18     if (idx+1==size) *end_ptr = nsteps;
19 }
20
21 __global__ void print_arr_cu(float *arr, int dim){
22     for (int i = 0; i < dim; i++){
23         printf("%f ", arr[i]);
24     }
25     printf("\n");
26 }
27
28 __device__ void print_arr_d(float *arr, int dim){
29     for (int i = 0; i < dim; i++){
30         printf("%f ", arr[i]);
31     }
32     printf("\n");
33 }
34
35 void initialize_cu(float *temp_arr, float *temp_arr0, bool *fire_arr,
36     float *x_arr, float *y_arr, int DIM, float T_fire, int Tx, int Ty){
37     printf("CUDA initialization\n");
38     // cuda parameters
39     DIM_d = DIM;
40     T_fire_d = T_fire;
41     Tx_d = Tx;
42     Ty_d = Ty;
43     // cuda memory allocation
44     gpuErrchk( cudaMalloc((void **)&temp_arr_d, sizeof(float)*DIM*DIM) );
45     gpuErrchk( cudaMalloc((void **)&temp_arr0_d, sizeof(float)*DIM*DIM) );
46     gpuErrchk( cudaMalloc((void **)&fire_arr_d, sizeof(bool)*DIM*DIM) );
47     // cuda memory copy
48     gpuErrchk( cudaMemcpy(temp_arr_d, temp_arr, sizeof(float)*DIM*DIM,
49         cudaMemcpyHostToDevice) );
50     gpuErrchk( cudaMemcpy(temp_arr0_d, temp_arr0, sizeof(float)*DIM*DIM,
51         cudaMemcpyHostToDevice) );
52     gpuErrchk( cudaMemcpy(fire_arr_d, fire_arr, sizeof(bool)*DIM*DIM,
53         cudaMemcpyHostToDevice) );
54     // synchronize
55     cudaDeviceSynchronize();
56 }
57
58 void finalize_cu(){
59     printf("CUDA finalization\n");
60     // cuda free
61     gpuErrchk( cudaFree(temp_arr_d) );
62     gpuErrchk( cudaFree(temp_arr0_d) );
63     gpuErrchk( cudaFree(fire_arr_d) );
64     // synchronize
65     cudaDeviceSynchronize();
66 }
67
68 __global__ void update_cu_callee(float *temp_arr, float *temp_arr0, bool *fire_arr,
69     float *x_arr, float *y_arr, int DIM, float T_fire){
70     int start_idx, end_idx;
71     int size = blockDim.x * gridDim.x;
72     int idx = blockIdx.x * blockDim.x + threadIdx.x;
73     partition_d(DIM-2, size, idx, &start_idx, &end_idx);
74     for (int i = start_idx+1; i < end_idx+1; i++){
75         for (int j = 1; j < DIM-1; j++){
76             float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
77             xw = temp_arr0[i*DIM+j+1];
78             xa = temp_arr0[(i-1)*DIM+j];
79             xs = temp_arr0[i*DIM+j-1];
80             xd = temp_arr0[(i+1)*DIM+j];
81             temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
82             if (fire_arr[i*DIM+j])

```

```

80         temp_arr[i*DIM+j] = T_fire;
81     }}
82 }
83
84 __global__ void update_cu_callee_shared(float *temp_arr, float *temp_arr0, bool *
fire_arr,
85     float *x_arr, float *y_arr, int DIM, float T_fire){
86     // block partition
87     int block_start_idx, block_end_idx;
88     int size = gridDim.x;
89     int idx = blockIdx.x;
90     partition_d(DIM, size, idx, &block_start_idx, &block_end_idx);
91     // block-size shared memory
92     __shared__ float temp_u[BLOCK_SIZE]; // upper
93     __shared__ float temp_c[BLOCK_SIZE]; // current
94     __shared__ float temp_l[BLOCK_SIZE]; // lower
95     __shared__ float temp_r[BLOCK_SIZE]; // record
96     __shared__ bool fire_c[BLOCK_SIZE]; // current fire
97     // tail case
98     float t_l, t_r;
99     // pre-initialize data
100    // main loop
101    for (int i = 1; i < DIM-1; i += BLOCK_SIZE){
102        for (int j = block_start_idx; j < block_end_idx; j++){
103            if (j!=0 && j!=DIM-1){
104                // load data
105                if (i+threadIdx.x < DIM){
106                    temp_u[threadIdx.x] = temp_arr0[(j+1)*DIM+i+threadIdx.x];
107                    temp_c[threadIdx.x] = temp_arr0[(j+0)*DIM+i+threadIdx.x];
108                    temp_l[threadIdx.x] = temp_arr0[(j-1)*DIM+i+threadIdx.x];
109                    fire_c[threadIdx.x] = fire_arr[(j+0)*DIM+i+threadIdx.x];
110                }
111                if (i+threadIdx.x<DIM-1 && i+threadIdx.x>0){
112                    if (threadIdx.x==0) t_l = temp_arr0[(j+0)*DIM+i+threadIdx.x-1];
113                    else if (threadIdx.x==BLOCK_SIZE-1) t_r = temp_arr0[(j+0)*DIM+i+threadIdx.x
+1];
114                }
115                __syncthreads();
116
117                // main compute program
118                float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
119                if (i+threadIdx.x<DIM-1 && i+threadIdx.x>0){
120                    xw = temp_u[threadIdx.x];
121                    xs = temp_l[threadIdx.x];
122                    if (threadIdx.x==0){
123                        xa = t_l;
124                    } else xa = temp_c[threadIdx.x-1];
125                    if (threadIdx.x==BLOCK_SIZE-1){
126                        xd = t_r;
127                    } else xd = temp_c[threadIdx.x+1];
128                    temp_r[threadIdx.x] = (xw + xa + xs + xd) / 4;
129                    // temp_r[threadIdx.x] = temp_c[threadIdx.x];
130                    if (fire_c[threadIdx.x]) temp_r[threadIdx.x] = T_fire;
131                }
132                __syncthreads();
133
134                // copy data back
135                if (i+threadIdx.x<DIM-1 && i+threadIdx.x>0)
136                    temp_arr[(j+0)*DIM+i+threadIdx.x] = temp_r[threadIdx.x];
137                __syncthreads();
138            }}}
139    }
140
141    __global__ void foo(float *arr, int DIM){
142        for (int i = 0; i < DIM; i++)
143            arr[i] = 0;
144    }

```

```

145
146 void update_cu(float *temp_arr){
147     // update_cu_callee<<<4,BLOCK_SIZE>>>(temp_arr_d, temp_arr0_d, fire_arr_d,
148     update_cu_callee_shared<<<16,BLOCK_SIZE>>>(temp_arr_d, temp_arr0_d, fire_arr_d,
149     NULL, NULL, DIM_d, T_fire_d);
150     cudaDeviceSynchronize();
151
152     // switch pointers
153     float *tmp = temp_arr_d;
154     temp_arr_d = temp_arr0_d;
155     temp_arr0_d = tmp;
156
157     // synchronize
158     cudaDeviceSynchronize();
159 }
160
161 void copy_cu(float *temp_arr){
162     // copy data to host
163     gpuErrchk( cudaMemcpy(temp_arr, temp_arr0_d, sizeof(float)*DIM_d*DIM_d,
164     cudaMemcpyDeviceToHost) );
165     cudaDeviceSynchronize();
166 }

```

src/utils.h

```

1 #pragma once
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <iostream>
5 #include <math.h>
6 #include <mpi.h>
7 #include <omp.h>
8 #include <pthread.h>
9 #include <sys/stat.h>
10 #include <sys/types.h>
11
12 void print_info(int DIM, int nsteps){
13     printf("Name: Haoran Sun\n");
14     printf("ID: 119010271\n");
15     printf("HW: Heat Distribution\n");
16     printf("Set DIM to %d, nsteps to %d\n", DIM, nsteps);
17 }
18
19 void partition(int nsteps, int size, int idx, int *start_ptr, int *end_ptr){
20     *start_ptr = nsteps / size * idx;
21     *end_ptr = nsteps / size * (idx+1);
22     if (idx+1==size) *end_ptr = nsteps;
23 }
24
25 void print_arr(float *arr, int n){
26     for (int i = 0; i < n; i++){
27         printf("%10.2f ", arr[i]);
28     }
29     printf("\n");
30 }
31
32 bool is_fire(float x, float y){
33     return (x*x + y*y <= 1);
34 }
35
36 void update_seq(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr, float *
37 x_arr, float *y_arr, int DIM,
38 float T_fire){
39     float *temp_arr = *temp_arr_ptr;
40     float *temp_arr0 = *temp_arr0_ptr;
41     for (int i = 1; i < DIM-1; i++){
42         for (int j = 1; j < DIM-1; j++){

```



```

42     float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
43     xw = temp_arr0[i*DIM+j+1];
44     xa = temp_arr0[(i-1)*DIM+j];
45     xs = temp_arr0[i*DIM+j-1];
46     xd = temp_arr0[(i+1)*DIM+j];
47     temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
48     if (fire_arr[i*DIM+j])
49         temp_arr[i*DIM+j] = T_fire;
50 }
51 // switch pointers
52 *temp_arr_ptr = temp_arr0;
53 *temp_arr0_ptr = temp_arr;
54 }
55
56 void update_omp(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr,
57 float *x_arr, float *y_arr, int DIM, float T_fire){
58     float *temp_arr = *temp_arr_ptr;
59     float *temp_arr0 = *temp_arr0_ptr;
60     #pragma omp parallel for
61     for (int i = 1; i < DIM-1; i++){
62         for (int j = 1; j < DIM-1; j++){
63             float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
64             xw = temp_arr0[i*DIM+j+1];
65             xa = temp_arr0[(i-1)*DIM+j];
66             xs = temp_arr0[i*DIM+j-1];
67             xd = temp_arr0[(i+1)*DIM+j];
68             temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
69             if (fire_arr[i*DIM+j])
70                 temp_arr[i*DIM+j] = T_fire;
71         }
72     }
73     // switch pointers
74     *temp_arr_ptr = temp_arr0;
75     *temp_arr0_ptr = temp_arr;
76 }
77
78 void update_omp_part(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr,
79 float *x_arr, float *y_arr, int DIM, float T_fire, int start_idx, int end_idx){
80     float *temp_arr = *temp_arr_ptr;
81     float *temp_arr0 = *temp_arr0_ptr;
82     #pragma omp parallel for
83     for (int i = start_idx; i < end_idx; i++){
84         for (int j = 1; j < DIM-1; j++){
85             if (i!=0 && i!=DIM-1){
86                 float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
87                 xw = temp_arr0[i*DIM+j+1];
88                 xa = temp_arr0[(i-1)*DIM+j];
89                 xs = temp_arr0[i*DIM+j-1];
90                 xd = temp_arr0[(i+1)*DIM+j];
91                 temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
92                 if (fire_arr[i*DIM+j])
93                     temp_arr[i*DIM+j] = T_fire;
94             }
95         }
96     }
97     // switch pointers
98     *temp_arr_ptr = temp_arr0;
99     *temp_arr0_ptr = temp_arr;
100 }
101
102 typedef struct pthArgs{
103     float *temp_arr;
104     float *temp_arr0;
105     bool *fire_arr;
106     float *x_arr;
107     float *y_arr;
108     int DIM;
109     float T_fire;
110     int start_idx;

```



```

109     int end_idx;
110     pthread_barrier_t *barr_ptr;
111 } PthArgs;
112
113 void *update_pth_callee(void *vargs){
114     PthArgs args = *(PthArgs *) vargs;
115     float *temp_arr = args.temp_arr;
116     float *temp_arr0 = args.temp_arr0;
117     bool *fire_arr = args.fire_arr;
118     float *x_arr = args.x_arr;
119     float *y_arr = args.y_arr;
120     int DIM = args.DIM;
121     float T_fire = args.T_fire;
122     int start_idx = args.start_idx;
123     int end_idx = args.end_idx;
124     for (int i = 1+start_idx; i < 1+end_idx; i++){
125         for (int j = 1; j < DIM-1; j++){
126             float xw, xa, xs, xd; // w: up; a: left; s: down; d: right
127             xw = temp_arr0[i*DIM+j+1];
128             xa = temp_arr0[(i-1)*DIM+j];
129             xs = temp_arr0[i*DIM+j-1];
130             xd = temp_arr0[(i+1)*DIM+j];
131             temp_arr[i*DIM+j] = (xw + xa + xs + xd) / 4;
132             if (fire_arr[i*DIM+j])
133                 temp_arr[i*DIM+j] = T_fire;
134         }
135     }
136     return NULL;
137 }
138
139 void update_pth(float **temp_arr_ptr, float **temp_arr0_ptr, bool *fire_arr, float *
x_arr, float *y_arr,
140 int DIM, float T_fire, pthread_t *thread_arr, PthArgs *args_arr, int nt){
141     float *temp_arr = *temp_arr_ptr;
142     float *temp_arr0 = *temp_arr0_ptr;
143
144     for (int i = 0; i < nt; i++){
145         int start_idx, end_idx;
146         partition(DIM-2, nt, i, &start_idx, &end_idx);
147         args_arr[i] = (PthArgs){.temp_arr=temp_arr, .temp_arr0=temp_arr0, .fire_arr=
fire_arr,
148             .x_arr=x_arr, .y_arr=y_arr, .DIM=DIM, .T_fire=T_fire, .start_idx=start_idx,
.end_idx=end_idx};
149         pthread_create(&thread_arr[i], NULL, update_pth_callee, (void *)&args_arr[i]);
150     }
151
152     for (int i = 0; i < nt; i++) pthread_join(thread_arr[i], NULL);
153
154     // switch array
155     *temp_arr_ptr = temp_arr0;
156     *temp_arr0_ptr = temp_arr;
157 }
158
159 void arr_check_if_identical(float *a, float *b, int dim){
160     for (int i = 0; i < dim; i++){
161         if (a[i]!=b[i]){
162             printf("fuck\n");
163             exit(1);
164         }
165     }
166 }
167
168 void runtime_record(char *jobtype, int N, int nt, double fps){
169     const char *folder = "data";
170     mkdir(folder, 0777);
171     FILE* outfile;
172     char filebuff[200];

```

```

173     snprintf(filebuff, sizeof(filebuff), "../s/runtime_%s.txt", folder, jobtype);
174     outfile = fopen(filebuff, "a");
175     fprintf(outfile, "%10d %5d %10.4f\n", N, nt, fps);
176     fclose(outfile);
177     printf("Runtime added in %s.\n", filebuff);
178 }

```

src/utils.cuh

```

1  #pragma once
2  #include <cuda.h>
3  #include <cuda_runtime.h>
4  #include <cuda_runtime_api.h>
5  #include <cuda_device_runtime_api.h>
6  #include <driver_types.h>
7
8
9  void initialize_cu(float *temp_arr, float *temp_arr0, bool *fire_arr,
10      float *x_arr, float *y_arr, int DIM, float T_fire, int Tx, int Ty);
11 void finalize_cu();
12 void update_cu(float *temp_arr);
13 void copy_cu(float *temp_arr);

```

src/const.h

```

1  #pragma once
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <iostream>
5
6  // global variables
7  int DIM = 200; // overall dimension
8  float T_bdy = 20; // boundary temperature
9  float T_fire = 100; // fire temperature
10 float xmin = -5;
11 float xmax = 5;
12 float ymin = -5;
13 float ymax = 5;
14 float *temp_arr = NULL;
15 float *temp_arr0 = NULL;
16 bool *fire_arr = NULL;
17 float *x_arr = NULL;
18 float *y_arr = NULL;
19
20 // computing-related constants
21 int nsteps = 100;
22
23 // IO & runtime options
24 int record = 0;
25 int nt = 1;
26 char type[1000];
27
28 // pthread parameters
29 pthread_t *thread_arr = NULL;
30 PthArgs *args_arr = NULL;
31
32 // mpi parameters
33 int size, rank;
34
35 // cuda parameters
36 int Tx = 16;
37 int Ty = 16;

```